POLITECNICO DI MILANO

MANAGEMENT ENGINEERING COURSE STUDY ON
DIGITAL BUSINESS AND MARKET INNOVATION

MASTER THESIS

Exploiting the Semantic Web for the Automatic Extraction of Los Angeles City Data

ADVISOR:
Prof. Letizia Tanca
CO-ADVISOR:
Prof. Craig Knoblock

CANDIDATE:
Marianna Bucchi
STUDENT ID:
898422

Academic Year 2018-2019

Contents

List of Figures

List of Tables

Abstract

Data Integration is an essential step for developing data-driven decision in companies and for enhancing the awareness in managers towards the importance of information. However, nowadays is not easy to exert it properly due to the explosion of the Big Data phenomenon, of which consequences are affecting every corner of the enterprise, especially the processing time during projects' development. This research study proposes a solution to address this problem and promote effectiveness in companies, i.e. introduce a novel approach to automatically detect the content of a dataset, with a special attention on the information stored in one of its columns. This intelligent categorization is made possible by the exploitation of ontologies knowledge and structure, according to the principles of the Semantic Web, which is an extension of the World Wide Web. As a starting point, we tested several approaches using 41 datasets belonging to the city of Los Angeles. Hence, we tracked the improvements obtained in each step to design a more comprehensive methodology, i.e. the final model. Finally, we investigated the performance of this final model including also, when available, the exploitation of other information in the datasets, such as the location. Experimental results on datasets has shown that the accuracy and correctness of the outcome improved significantly with the development of the final design.

Sommario

L'Integrazione dei Dati è un passaggio fondamentale nello sviluppo di decisioni basate su di essi nelle aziende e nella maturazione da parte dei manager di una certa consapevolezza della loro importanza. Tuttavia, oggi non è così banale praticare tecniche di Integrazione dei Dati a causa dell'esplosione dal fenomeno dei Big Data, le cui conseguenze stanno sfidando il mondo aziendale in ogni ambito, specialmente il tempo di processo richiesto durante lo sviluppo di qualsiasi progetto. Questo lavoro di ricerca propone un tentativo di soluzione a questo problema e di incoraggiamento ad una più elevata efficacia nelle imprese, suggerendo un nuovo approccio per l'identificazione automatica del contenuto di un dataset, ponendo particolare attenzione sulle informazioni mostrate da una delle sue colonne. Questa categorizzazione intelligente è resa possibile grazie all'utilizzo delle informazioni e dalla struttura che contraddistinguono le ontologie, definite sulla base dei principi del Web Semantico, che consiste in una estensione del World Wide Web. Come punto di partenza del progetto, sono stati testati diversi approcci su 41 dataset appartenenti alla città di Los Angeles. Quindi, durante ogni fase di sviluppo, sono stati tracciati tutti i miglioramenti ottenuti fino a generare una metodologia più completa e ampia che si è identificata in un modello finale. Infine, le performance di quest'ultimo sono state esaminate includendo anche nell'analisi, quando possibile, l'utilizzo di ulteriori informazioni dal dataset, come ad esempio la colonna contenente la posizione. I risultati ottenuti dagli esperimenti sui dataset hanno mostrato un significativo miglioramento dell'accuratezza e correttezza delle risposte, dato dallo sviluppo graduale del modello finale.

Chapter 1

Introduction

The current transformation regarding data management and its recent developments constitutes one of the most discussed topics of the recent times. Indeed, the way information is exchanged embodies a new paradigm in terms of the impact on our society and on the environment around us. The consequences of this extraordinary innovation process are countless and affect many aspects of our daily life, resulting in the **proliferation of the amount of data generated and raising significant problems towards information storage and integration.**

Indeed, these fundamental processes, that already seem challenging, have turned out to be even more effort-demanding in the so-called **Big Data Era**. This widespread expression ("Big Data") refers to the extensive degree at which data is created, shared and utilized in recent times, including a set of methodologies that looks more complex than the usual procedures, and impacting many fields of activities of our society [41]. Big Data can be characterized along six different dimensions: volume, variety, velocity, value, veracity and variability [118]. **Volume** addresses the challenge in the relationship between the huge amount of information generated and the limited processing capacity of the current technical resources. **Variety** expresses the wide range of information that needs to be processed and analyzed, including also its structure and format. **Velocity** measures the temporary value of data, facing the problem of rapid information changes, on one hand, and the need of having it available in real-time, on the other. **Value** embodies the potential benefits that can be gained from Big Data and Big Data practices. **Veracity** shows the quality and the authenticity of data, addressing issues related to conflicting or impure information. Finally, **variability** conveys to what extent, and how fast, the structure of data is changing.

These issues are compelling for businesses and ambitious strategies are needed to look ahead and exploit the potential of new opportunities driven by advanced Big Data management. The tremendous amount of information available, in fact, is harshly testing the industries and the whole business world in the attempt to make valuable and data-driven decisions. According to a survey by [46], among several companies

from UK and US, business leaders say that *data informs their decisions an average of 16 times a day* and 46% of them stated that *they do not have the technology in place to take advantage of data.* Thus, in a society that is changing very fast it is becoming fundamental for enterprises to catch up with the global market, and, thus, to manage intelligently the information they already have.
Looking more thoroughly, all this data must be molded into an information foundation that is **integrated, consisted** and **trustworthy**. Indeed, these resulted to be the current leading priorities for top managers, as revealed by a research study pursued at MIT and published in the MIT Sloan Management Review [99] (Figure 1.1 shows the integral responses obtained by 500 managers in the US).

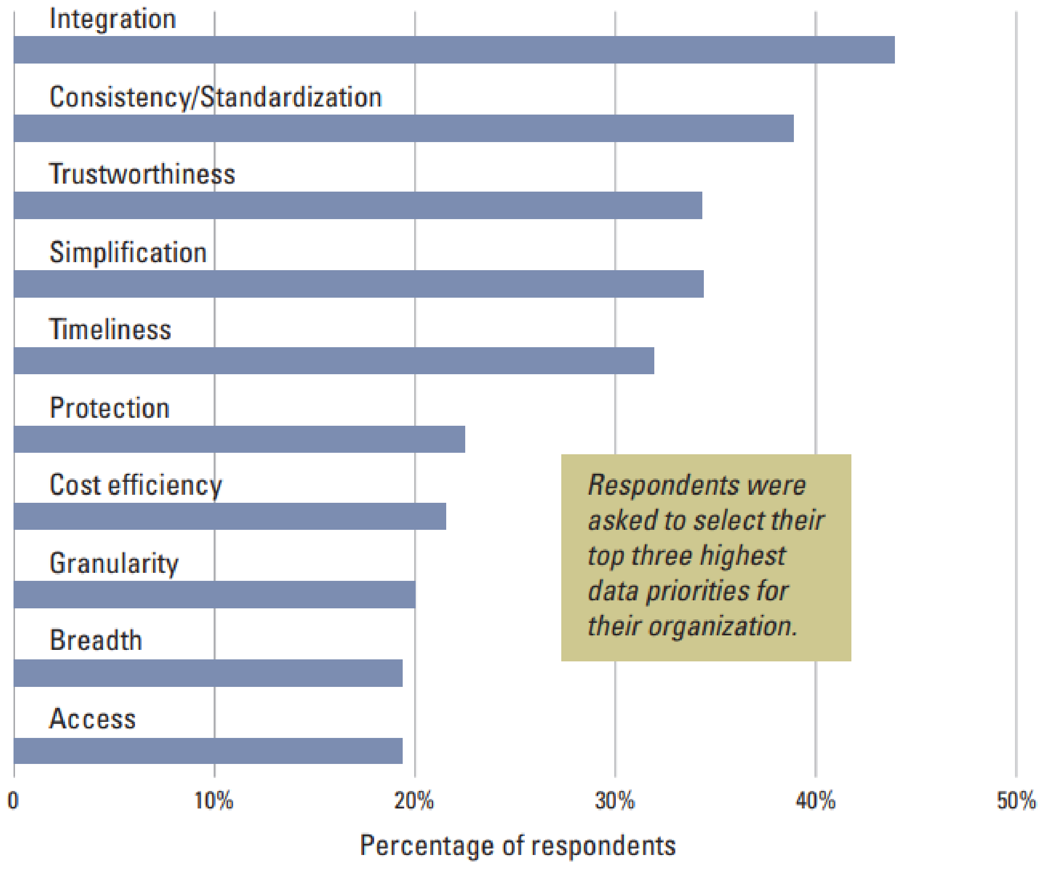


Figure 1.1: What Managers Want Most In Their Data [99]

Focusing on the first priority highlighted by this study, the **Data Integration** problem can be defined as the **combination of data coming from different sources**, providing the user with a unified vision of it [45]. It is essential, especially for companies, to benefit from integrated access to their information in such a way that both visualization and analysis of data across a wide range of sources become easy. Time and efficiency also represent key resources to gain competitive advantage against competitors. Thus, finding **a way to access the content of datasets in a fast and efficient way** could be an example of strategy to improve the competitive position of the organization.

Moreover, the Big Data phenomenon combined with Data Integration practices has caused processing time dilation in all Data Integration steps [82], especially during **record linkage**, which is an intermediate phase devoted to combine two or more sets of records referring to the same entity [27].

Considering this critical context, the main objective of this research project is to sustain the enterprises in the process of integrating and analysing the data they have, taking into account as a starting point a collection of available datasets containing a huge amount of various and heterogeneous information about the city of Los Angeles. The number of datasets is so large (more than a thousand) that departments within the city will often be unaware of the information they have. Thus, they are often unable to exploit and gain benefit from it. To address this problem, the design of a new integration model has been brought up, with the final aim to develop a strategy to access data more efficiently and effectively. In particular, the goal of this research is to find a **methodology for automatically extracting the content from each dataset**. In order to do so, the model should parse the information contained in each dataset and extract the category to which the dataset belongs. Categories refer to classes of items having particular shared characteristics and, in this case, the model should be able to capture these peculiarities and to detect the correct category from them. Categories (or classes) occur in a standardize form so that they result to be human understandable. For example a dataset can be labeled as containing information about `Companies` or `Hospitals`.

The methodology used in this project exploits the incredible knowledge spread in the World Wide Web and its arising technologies, including the **Semantic Web paradigm**, which has been introduced as a development of the World Wide Web and where data is structured and written in such a way that it becomes machine readable [126]. Leveraging on the Semantic Web means to have the opportunity to access a huge amount of data and compare it with the information stored in the datasets. This process recalls the characteristics of the record linkage step in Data Integration and it is made possible thanks to Knowledge Bases, which characterised the Semantic Web paradigm. A knowledge base consists in a store of information on which it is allowed automatic inference [66]. Particularly, the knowledge bases used for the purpose of this project are defined as **ontologies**, i.e. the explicit and structured specifications of the concepts represented in a knowledge base [26]. Ontology specifications are typically **Classes**, which are sets where similar items are grouped together, **Attributes**, which define the item properties and **Relationships**, which specifies the relations between Classes. For it is, in fact, ontologies benefit from a hierarchical structure where all the entities, i.e. the objects, are stored and represented [64].

As a result of this well organized structure, ontologies represent the perfect instrument to achieve the goals of this research study: to integrate datasets' information with Semantic Web data and to correctly categorize datasets on the basis of ontologies' structure. To address these issues, many approaches have been evaluated as a starting point, including the **Column Entity Annotation (CEA)** and the **Column Type Annotation (CTA)** models described by [123]. The former focused on comparing entities of one single column of a dataset with ontologies data, with the aim to find matches; the latter focused on identifying for that column what the correct class (i.e. the type) is, leveraging on the information gained from the ontologies. Subsequently, a more comprehensive approach has been considered to disclose the subject of the information stored in those datasets, including also the analysis of the column containing locations, when available, in the final design of the model. The results obtained has shown a correctness in the model of 80% (considering the evaluation of 41 datasets), highlighting what a difficult task this is.

The project is divided into seven Chapters which have been defined as follows.
An **Introduction** Chapter which is devoted to the presentation of the target problem and the explanation of the purpose of the research.
A second Chapter dedicated to the description of the **Preliminary Notions** needed to understand the topic and the technicalities behind the approaches applied, including the definition of the Semantic Web paradigm and it principles, and other important tools implemented in the project, like SPARQL and Elasticsearch..
The third Chapter is represented by the **State Of The Art** that encompasses the literature used as reference, such as Data Integration steps in a Big Data and Web-oriented environment, past work that show how to address the entity linking problem using ontologies and to automatically integrate data of organizations by creating a spatio-temporal index.
The fourth Chapter of this project is focused on the description of the **Methods and Materials**; thus, explaining the methodologies pursued, the tools applied in each step of the research and the reasons behind it.
In the fifth Chapter, a more detailed description of the problem and the way it has been faced is reported, including the technical explanation of the **Model Design** in all its facets and development steps.
Sixth Chapter is devoted to the illustration of the **Experiments** done according to the objective of the research and the exhibition of the results obtained, both quantitative and qualitative.
A **Concluding** Chapter shows the data-driven insights collected during the research and lays the foundation for future developments of the model.

1.1 Research Proposal

This research project provides an attempt to meet the companies' need to process data in a more efficient and effective way, since one of the main problems they have to deal with, rather than the amount of information available, consists in how the latter is handled and in the level of awareness they have acquired. Indeed, often the **companies are not sufficiently conscious of the data they have, not even of where and how it is stored**, and this may lead to serious issues. In addition, today's measures for integrating information from different sources are often too time-consuming and too costly [12].
There are **two driving factors** causing this unpleasant situation:

- databases are still seen as *silos* only controlled by experts,
- the way data is stored is heterogeneous as information architecture, its metadata and schema are not separated well from application logic, so that it is impossible to reuse it effectively.

In this perspective, this research study has the ambitious aim to sustain enterprises in the process of analysing and processing data by facilitating the access to it. The main objective of the presented project, in fact, is the generation of a model that should support the automatic organization of information regarding the city of Los Angeles, according to its **content**. In this way, departments and companies working in the city, and often awash in data, can easily find the information they need avoiding repetitions throughout different sources.
In order to pursue this objective, we have leveraged on **Wikidata** and **DBpedia** ontologies to analyse the information available in a dataset and detect the Class it belongs to. According to [75], Wikidata is a knowledge base born to be the central data management platform of Wikipedia and to supplement its project, while DBpedia extracts structured content from the information created in the Wikipedia projects and made it available on the World Wide Web in a standardised format [5]. This choice has been dictated by the fact that all the information stored in DBpedia is also incorporated in Wikidata. Wikidata hosts open knowledge where anyone can contribute and it is one of the most important centralized storage of structured data. Indeed, this information availability constitutes a favourable factor for the purpose of the research.
Moreover, the representation of the data in the DBpedia ontology adheres to the Semantic Web model [53], where each element is represented considering also its relationships with other elements according to different levels of expressiveness [60]. At the basis of the mentioned model, information is described through a triple *subject, predicate, object* in order to highlight the relationships between data. **Since good modelling is key of efficient reasoning [132], the structure of these**

ontologies has been considered appropriate for the objective of the research. Particularly, we considered the configuration of DBpedia ontology, built upon a structured, hierarchical system base on classes, as a very good model of reference.

The available data regarding the city of Los Angeles includes a significant number of datasets containing a wide range of information (from the trees in the streets to the characteristics of restaurants) and presented in a table form. In accordance with the aim of the project and the nature of the ontology chosen, the process of the attributes' selection during the analysis has been pursued considering columns containing **text**. In addition, we made sure that the final model was able to identify which was the column that best defined the dataset, in order to work on that specific column.
The integration has been done according to the information **content**, which meant leveraging on the analysis of the information contained in datasets single column to find matches with ontologies data and use the latter structure to retrieve a class for the dataset. In fact, from the exploitation of the knowledge stored in Wikidata and DBpedia, the model could enhance its knowledge about the content of the dataset. The class was derived by moving along DBpedia hierarchical structure and stopping according to certain conditions, evaluated during the development of the final model. We also tracked the processing time of the model to check its effectiveness in pursuing the objective. Since the time involved was little, even for datasets containing millions of rows, this solution enable users to gain time for analysing data and to enjoy an easy access to the dataset contents.

Chapter 2

Preliminary Notions

The scope of this Chapter is that of providing the background information which determined the knowledge base of the whole research project. In particular, this Chapter includes the explanation of the theories behind the Semantic Web and its structure, the Linked Open Data paradigm, the Knowledge Graph and the tools implemented in the project, such as SPARQL and Elasticsearch search engine. Through a high detailed elucidation of these preliminary notions, we aim to make the content of this paper even more understandable.

2.1 Semantic Web

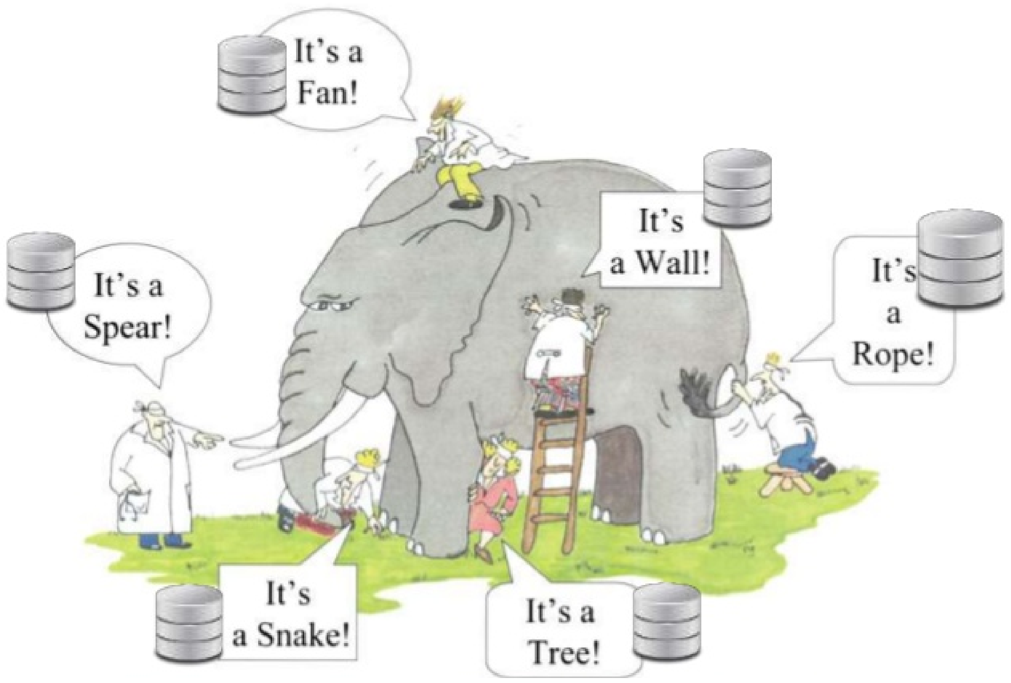


Figure 2.1: Semantic Heterogeneity [57]

In this paragraph we introduce the main theory on which the research project lays, i.e. the definition of the so-called **Semantic Web** paradigm, also known as **Web of Data**. The Semantic Web is basically an extension of the Web and of the Web-enabling database and Internet technology [60] and it was originally proposed by its inventor, Tim Berners-Lee, as the way to **solve the problem of semantic heterogeneity in the Web.** Indeed, when datastores for the same domain are developed by independent parties, this causes discrepancies in the information content and these differences are referred to as semantic heterogeneity. Semantic heterogeneity also appears in the presence of multiple XML documents, web services and ontologies - or more broadly, whenever there is more than one way to represent and organize a body of data [68]. Figure 2.1 shows in a nice and intuitive way what semantic heterogeneity is as a multiple perspective of reality which causes ambiguity, vagueness and inconsistency [57]. Therefore, in this scenario, the Web was designed as an information space with the goal to be useful for human-to-human communication, while the **Semantic Web approach develops methods and languages for expressing information in a machine-processable form** [19]. Thus, this solution has been introduced with the aim to overcome the limitations of the World Wide Web as that virtual environment where data form is for human consumption and formats available require specialized algorithms to access, search and reuse [91].
The question naturally arises: **how does the Semantic Web help solving the Web's drawbacks?** To answer this question it is useful to describe the architecture of the Semantic Web. Its design, in fact, stems from the idea of adding an extra abstraction layer, a so-called **semantic layer**, built on top of the Web, in order to make data not only human-processable but also machine-processable [60].
Figure 2.2 shows the technical structure that has been introduced within the Semantic Web, where data is organized in (at least) four levels of increased expressive power, each one corresponding to a specific representation need [60][91][19], namely:

1. **eXtensible Markup Language (XML)** [28][93][30]: allows the users to create customized *tags* - hidden labels, such as zip code, that annotate Web pages or sections of text - which are used by scripts or programs to exchange in a sophisticated manner a wide variety of information on the Web and elsewhere. XML grants users to add arbitrary structure to their documents but has no built-in mechanism to convey the meaning of the user's new tags to other users. An example of XML fragment is provided below [6].

```
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

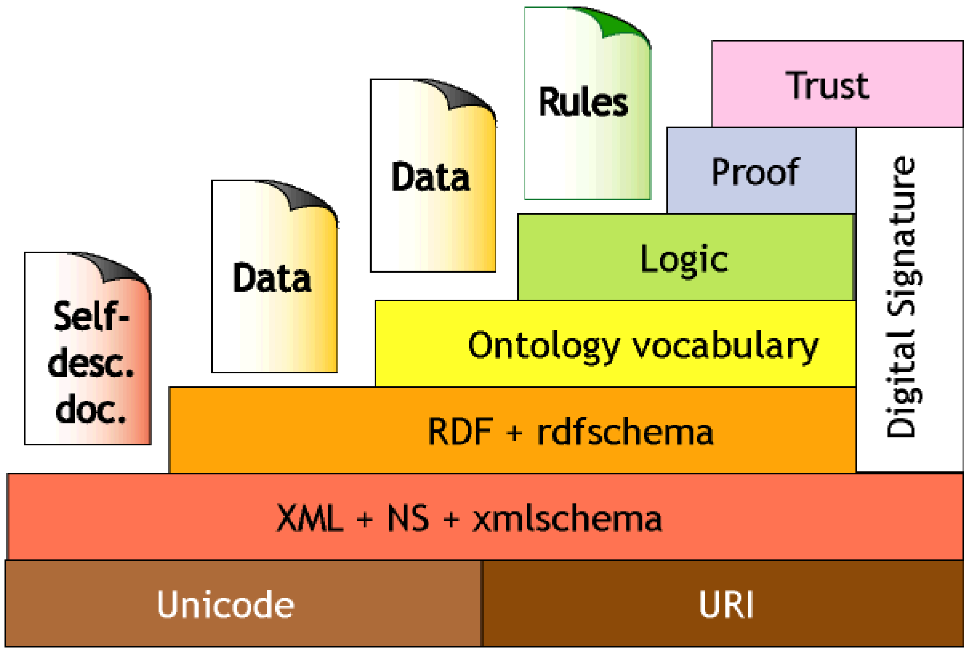


Figure 2.2: Standard Stack Architecture Of The Semantic Web

In addition to XML, **XML Schema** has been introduced with the purpose of defining a set of rules to which an XML document should conform [60] and to allow the exchange of information between interested parties who have agreed to adhere to those regulations and basic syntactical constraints [85] [124].

Generally speaking, there are several recommendations that users should follow to produce usable XML documents according to the World Wide Web Consortium [126]. However, there are no explicit constructs for defining classes and properties in XML Schema, therefore ambiguities may still arise when mapping an XML-based data model and that is why a semantic layer has been introduced [40].

2. **Resource Description Framework (RDF)**: a scheme that defines the Semantic Web data structure by providing the technology to express the meaning of terms and concepts in a form that computers can readily process and that intelligent agents can automatically parse [19]. It is a language for representing information about resources and the relationships among them in the Web [113], which are called *statements*. Items in RDF are identified by a **Universal Resource Identifier (URI)**, which is a compact string of characters for describing both abstract and physical resources, globally recognised through standards stated by Berners-Lee et al. [125]. All data in RDF is defined as an ordered set where its terms can be either a **URI** $u \in U$, a **blank node** $b \in B$, or a **literal** $l \in L$. In addition, RDF allows statements about resources to be expressed in a form of RDF *graphs*, where a RDF graph consists in a group of

RDF *triples*. Each triple *s, p, o* is characterized by a *subject s*, where $s \in U \cup B$, a *predicate p.* where $p \in U$ and an *object o*, where $o \in U \cup B \cup L$ [42] [49]. According to [86] the advantages of using URIs, to identify elements in the triples, especially subjects and predicates, are many, allowing:

- to avoid confusions among different elements with the same name,
- to precisely identify properties,
- to express concepts not just as words but to tie them to a unique definition so that everyone can find them on the Web.

There are various type of identifiers belonging to the URI category, such as the **Universal Resource Locator (URL)**, which specifies the location of an item, and the **Universal Resource Name (URN)** that is used to uniquely identify resources.

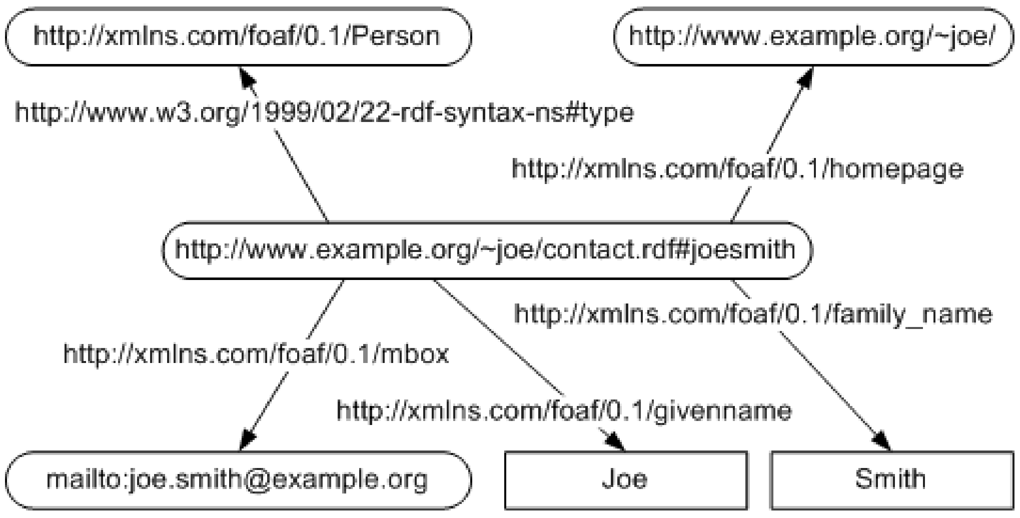

http://xmlns.com/foaf/0.1/Person
http://www.example.org/~joe/
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://xmlns.com/foaf/0.1/homepage
http://www.example.org/~joe/contact.rdf#joesmith
http://xmlns.com/foaf/0.1/family_name
http://xmlns.com/foaf/0.1/mbox
http://xmlns.com/foaf/0.1/givenname
mailto:joe.smith@example.org
Joe
Smith

Figure 2.3: RDF Graph Model Example

Figure 2.3 shows an example of RDF graph model describing statements about a person called "Joe Smith". The graph used in RDF model is a directed graph where each triple is represented as an edge with two connecting nodes (from subject to object) which form webs of information about related things. In this kind of RDF representations there could be **blank nodes**, i.e. empty resources for which a URI or literal is not given. In Figure 2.3 there are no blank nodes, but objects expressed through URIs or in **literal** form, such as "Joe".

RDF triples can be textually represented using XML tags to explain entities, concepts, properties and relations according to several specifications given in [16][32][76] [15][113][56] and they are used to describe the vast majority of the data processed by machines. RDF model better facilitates inter-operation than

XML because it provides a data model that can be extended to address other sophisticated representation techniques [43]; Figure 2.3 [104] shows an example.

Moreover, RDF is provided with a vocabulary named **RDFSchema** where all the elements are called *Resources* and with the sophisticated aim to specify the inference rules implied by the triples. RDFSchema is a form of semantic **Metadata** (broadly defined by [47] as "structured data about data") where information describes or supplements actual data by defining the semantic relationship between resources and properties[40][9]. The main RDFSchema constructs are based on the *Class* and the *Property* as resource types, i.e. characteristics of the entities represented, and *subClassOf* and *subPropertyOf* as relationship names. This terminology allows to declare resources as an instance of one or more classes and to specify hierarchies of both classes and properties through the usage of *subClassOf* and *subPropertyOf* [25]. Here we provide an example of RDFSchema written in XML.

```
<rdf:RDF
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:foaf="http://xmlns.com/foaf/0.1/"
        xmlns="http://www.example.org/~joe/contact.rdf#">
      <foaf:Person rdf:about= "http://www.example.org/~joe/
                      contact.rdf#joesmith">
        <foaf:mbox rdf:resource="mailto:joe.smith@example.org">
        <foaf:homepage rdf:resource="http://www.example.org/~joe/"/>
         <foaf:family_name>Smith</foaf:family_name>
         <foaf:givenname>Joe</foaf:givenname>
      </foaf:Person>
</rdf:RDF>
```

As it is possible to notice, properties can be defined by a prefix, such as `/rdfs` and `/foaf`, while each resource is uniquely described by a URI, like `http://www.example.org/joe/contact.rdf/joesmith`

3. **Ontology Vocabulary**: this is the **Web Ontology Language (OWL)**, the third basic component of the Semantic Web, defined according to [19] as a collection of statements that describes the contextual relations between concepts and specifies logical rules for reasoning about them on the Web. OWL facilitates greater machine interpretability of Web content than that supported by XML, RDF, and RDF Schema, by providing additional vocabulary along with a formal semantics [95]. OWL, in fact, represents an extension of RDFSchema as it classifies elements within the vocabulary according to their semantics and meanings.

   Generally speaking, OWL is a formal syntax for defining ontologies, which are the cornerstone of the knowledge domain and explicitly specify concepts by providing the syntax to define Classes and various operations on them[60][63]. Indeed, OWL ontology is characterized by a hierarchical structure of classes and subclasses named **taxonomy**. For example, the resources *student* and *professor* represent different entities belonging to the same class: *person.* Through classes, subclasses and relations among entities it is possible to express a large number of concepts by assigning properties to classes and allowing subclasses to inherit such properties. In addition, instances in OWL are called *individuals* and are related by two main properties:

- **object properties** which relate individuals of two different OWL classes,
- **datatype properties** which relate individuals with literal values. For example, the address of a house and its zipcode represent the datatypes of the property *location.*

As for the other layers in the Standard Stack, also for OWL the World Wide Web Consortium has suggested several specifications which have been described in detail in [95][133][13][106][71][36] and are constantly updated. With ontology pages on the Web, machines can now manipulate information in a more effective way that is useful and meaningful for the user [19][59] .

4. **Logic, Proof and Trust Level:** this level adopts technologies that are at a very early stage due to the absence of already defined standards and open architectural issues. According to [114] the layers that distinguish the upper level of this Standard Stack are:

   - **Logic Layer** to enable the writing of rules,
   - **Proof Layer** to execute the rules and produce new knowledge,
   - **Trust Layer** to decide whether to believe a given proof or not.

Last but not least, in accordance with [19], the real power of the Semantic Web is based on **software agents**, i.e. programs that collect Web content from diverse sources, process the information and exchange the results with other programs. As a results of the Semantic Web development, such agents are now able to establish a synergy and transfer data among themselves, enhancing their performances' effectiveness. Very likely, this phenomenon will increase exponentially as more machine-readable Web content and automated services become available. That is the reason why data providers should collaborate by properly publishing their data and linking it to the existing ones.

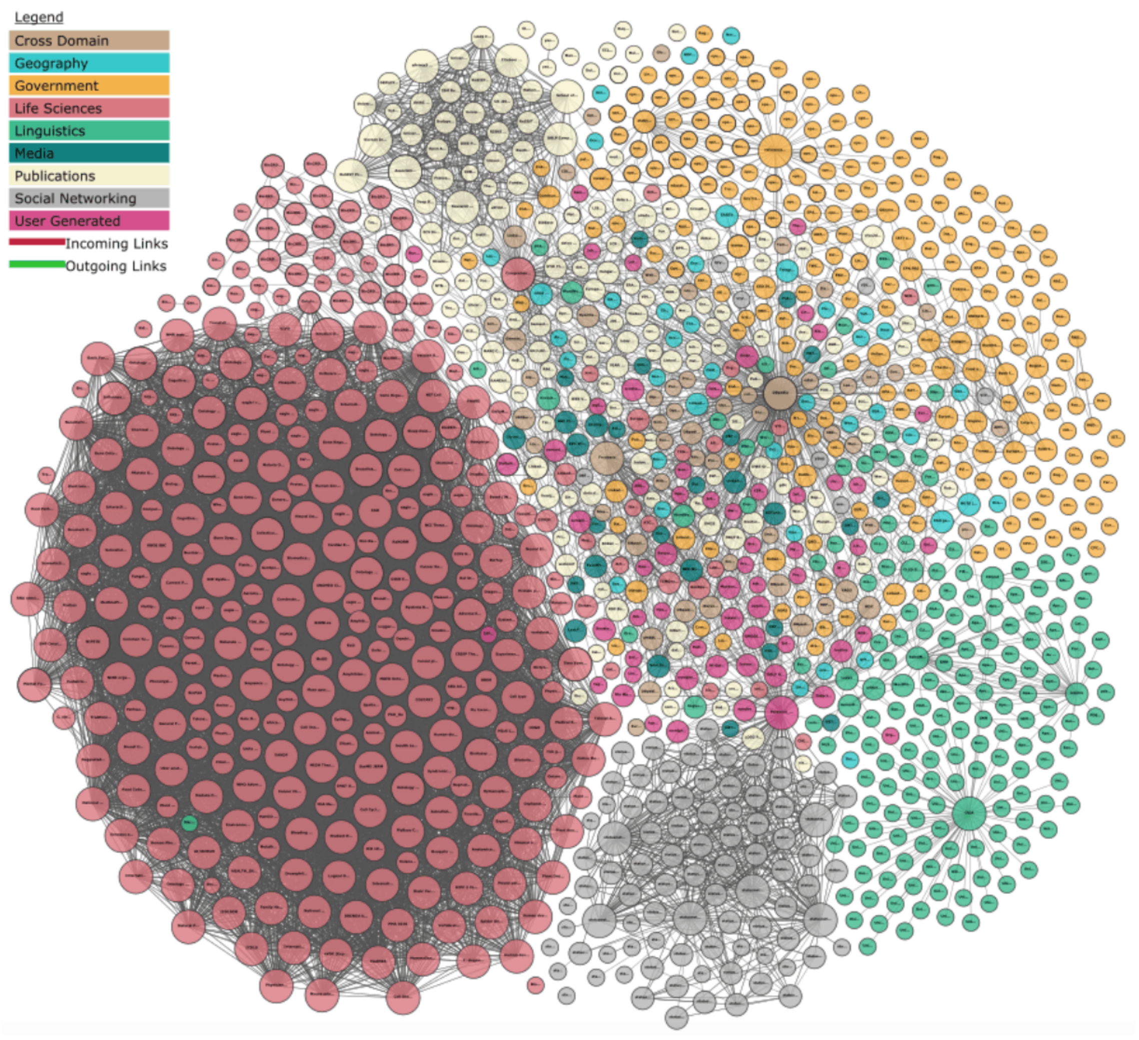


Figure 2.4: Linked Open Data Cloud [94]

2.2 Linked Data

The idea of Linked Data is a descendant of the Semantic Web paradigm as it refers to a set of **best practices for publishing and interlinking structured data on the Web by using RDF structure and URIs** [12][70]. Linked Data project goes along with a standardised, uniform and generic APIs[1], supporting information discovery, reliable access to data and metadata and distributed querying [134]. Thus, Linked Data represents a favourable environment for information exchanges and synergies between programs. These characteristics, in fact, allow Linked Data to have a wider adoption due to lower entry barriers for data providers. As information is not just published but also uniformly linked using RDF [21], it becomes significantly more discoverable and, therefore, more usable.

[1]Web Application Programming Interface is a set of instruction and standards for interacting with a web server or browser over HTTP protocol [98].

As it is possible to see from Figure 2.4, data sources are connected into a single global space through links set between different items, echoing the diversity and increasing the volume of information available to users and developers. The basic idea of Linked Data is to provide a general architecture in the process of sharing data on a global scale trough the application of four main principles [21]:

1. Use URIs to identify things
2. Use HTTP URIs [1] to allow people looking up things
3. Provide useful information in the URIs, trough standards such as RDF and SPARQL (see Section 2.4)
4. Include RDF links to other URIs to enable the discovery of realted information

According to T. Berners-Lee [18], breaking these principles does not destroy anything, but it means to miss the opportunity of making data interconnected. Thus, this will limit the ways information could later be reused in unexpected ways, which eventually represents the value added by the Web.
Furthermore, around the Linked Data paradigm, a whole community is moving with the World Wide Web Consortium to take existing (open) datasets and make them available on the Web [11], leveraging on the usage of open standards, like RDF, to describe metadata. This is the reason why sometimes we refer to Linked Data as **Linked Open Data**. The result of this global participation is shown in Figure 2.4 which illustrates all the knowledge bases engaged in the Linked Data project and where the expansion is unstoppable.

2.3 Knowledge Graph

The term **Knowledge Graph** has been associated with a variety of partially contradicting definitions and descriptions. The company which formulated this expression for the first time was Google in 2012, basically describing the Knowledge Graph as an **enhancement of their search engine with semantics** [119]. According to the Semantic Web Conference, instead, the Knowledge Graph could be envisaged as a **network of all-kind things which are relevant to a specific domain or to an organization**, not limited to abstract concepts and relations but also including instances of things like documents and datasets [23]. Many other different interpretations of the Knowledge Graph have been identified in the Literature, including definitions in [107][52][111]. Overall, the main aspect that is evident from these definitions is considering a Knowledge Graph as a large network; thus, a **graph-based**

[1]HTTP stands for Hypertext Transfer Protocol and it is used for distributed, collaborative information systems which defines communication means between endpoints [98]. It is the Web's universal access mechanism.

model embedding many different and heterogeneous elements. More recently, Nickel et al. [102] have defined the Knowledge Graph in such an interesting way that is worth reporting in this project. They described it as a **graph-structured knowledge base that stores factual information in form of relationships between entities.** What stems from this definition is the fact that the items stored are expressed as relational facts. Thus, in accordance with the definition given in Section 2.1, a Knowledge Graph could be described as an ontology, due to its structure based on interrelations between entities. Indeed, when a Knowledge Graph meets formal properties, it is an ontology [23]. Ontologies, in fact, were born to capture very complex relationships between classes and individuals than generic Knowledge Graphs[24]. Thus, formal specifications are needed to generate an ontology.
The presented research was born from the aim to exploit the concept of Knowledge Graph with formal properties, already available in the Web.

2.4 Wikidata

Wikidata is a free and **open knowledge base** that can be read and edited by both humans and machines and acts as central storage for the structured data of its Wikimedia sister projects including Wikipedia, Wikivoyage, Wiktionary, Wikisource, and others [4]. Wikidata was first introduced in October 2012 with the objective to overcome Wikipedia's shortcomings (the most significant one being the absence of a direct access to most of the data) creating new ways for Wikipedia to manage its information on a global scale [130]. The uniqueness of Wikidata approach is marked by several principles [4]:

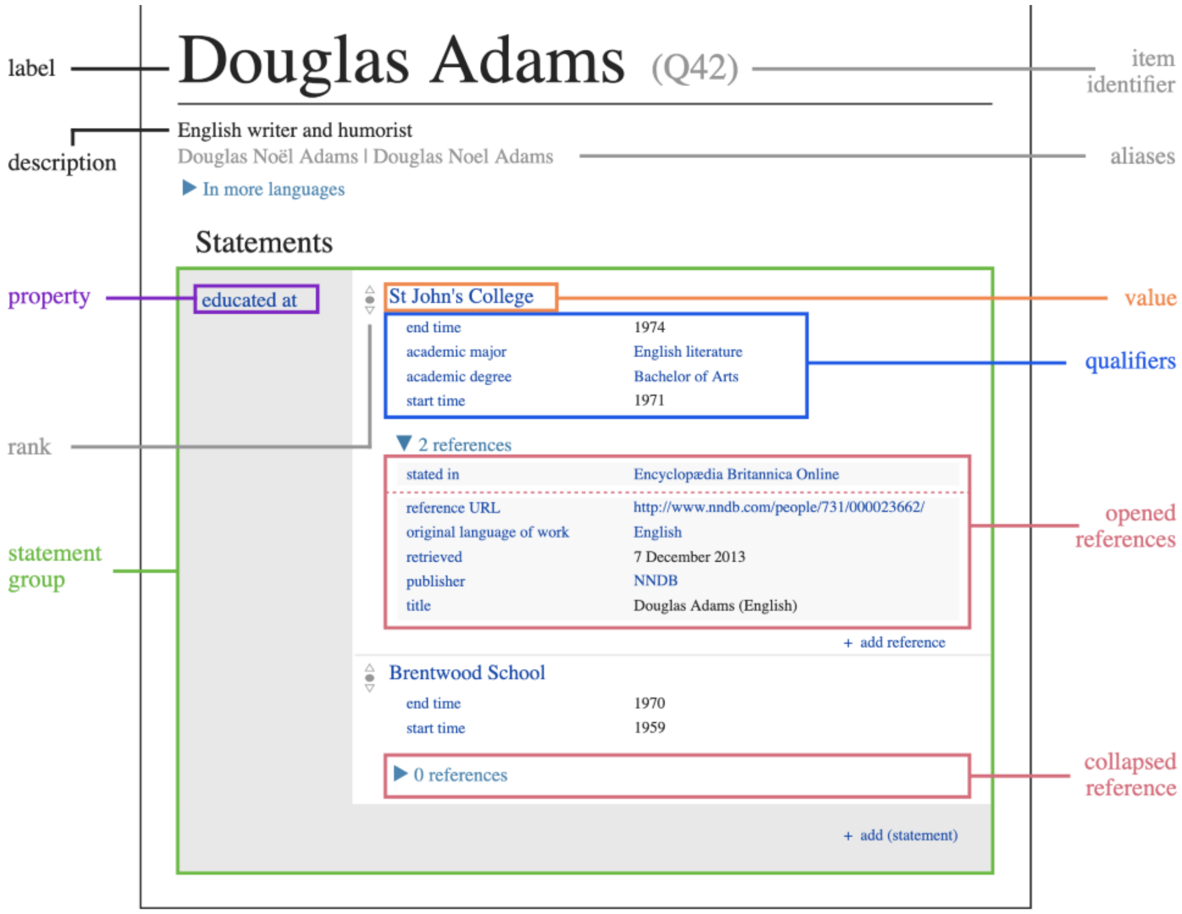
- be free and open to anyone in the world for storing and editing information,
- be collaborative in data and data schema management so that the whole community has the control over them,
- be multilingual accepting possible conflicts between information, i.e. mismatches between data that has been stored in different languages but that refers to the same entity, and providing mechanisms to organize it correctly,
- store data which belongs to different primary sources or databases,
- collect structured data,
- assist Wikipedia by increasing the quality of information management and easing the access to it,
- be in continuous evolution together with a growing community of editors.

Indeed, Wikidata key factor is the volunteer community's reuse and integration of external identifiers, i.e. unique attributes to identify entities, from existing databases

and authority controls which enable applications to integrate Wikidata with information from other sources that remain under the control of the original publisher. Being an open knowledge base where each one can give a personal contribute, the site has gathered data on more than 15 million entities, including over 34 million statements and over 80 million labels and descriptions in more than 350 languages [50]. The interconnection to external datasets from many different domains is the distinctive characteristic of the Linked Open Data paradigm and makes Wikidata part of the Semantic Web, supporting integration of other Semantic Web data sources. Furthermore, the collected information is exposed in various ways, mostly JSON[1], XML and several other formats, and it is available in the public domain. In order to retrieve information from Wikidata pages, and consequently from Wikipedia, users can leverage both on the so-called **Qnodes** as all the information is uniquely defined by them.
A **Qnode** is the exclusive number (preceded by a "Q") that describes the page of a particular item. Qnodes, in fact, are useful since Wikidata is a multi-lingual site and items need not to be identified by a label in a specific language, but by an opaque identifier, which is assigned automatically when the item is created and which can not be changed later on. For instance Q42, which is correlated with the URI `http://www.wikidata.org/entity/Q42`, represents the page about Douglas Adams, the famous English writer and humorist. Figure 2.5 shows an example of Q42 Wikidata page and the standard layout of the site.
As it is possible to see, every item page contains the following main parts [50]:

- a **label**: "Douglas Adams",
- a short **description**: "English writer and humorist",
- a list of **aliases** (e.g., "Douglas Noël Adams"),
- a set of **properties** which introduce a characteristic of the data (e.g. "educated at") and have a P prefix instead of Q. Properties, when paired with **values**, form a **statement** in Wikidata and they are described in dedicated pages connected to items, resulting in a linked data structure. Each property has a *data type* which defines the kind of values allowed in statements with that property,
- a list of **qualifiers** which provide additional contextual information to enrich a statement,
- a list of site **links** to pages about the item on Wikipedia and other projects.

[1]JavaScript Object Notation is a standardised text format for data interchange, easy for humans to read and write, and for machines to parse and generate [39]



Douglas Adams (Q42)

English writer and humorist
Douglas Noël Adams | Douglas Noel Adams
▶ In more languages

Statements

educated at

St John's College

end time	1974
academic major	English literature
academic degree	Bachelor of Arts
start time	1971

▼ 2 references

stated in	Encyclopædia Britannica Online
reference URL	http://www.nndb.com/people/731/000023662/
original language of work	English
retrieved	7 December 2013
publisher	NNDB
title	Douglas Adams (English)

+ add reference

Brentwood School

end time	1970
start time	1959

▶ 0 references

+ add (statement)

Figure 2.5: Douglas Adams Page In Wikidata [4]

Many Wikidata projects have shown the need to represent statements and qualifiers trough the Knowledge Graph [24], as an effective tool to improve knowledge extraction an analysis.

2.5 DBpedia: a free RDF repository

Synergistic research along different directions and extraction approaches which are able to embrace diverse knowledge domains simultaneously have become a matter of interest for developers and Web users. To accomplish such challenging tasks, in fact, a rich corpus of diverse data is needed. The DBpedia project has been introduced to answer this need and it focuses on **converting Wikipedia content into structured knowledge**, so that Semantic Web techniques can be employed against it [10]. Thus, the main objective of DBpedia is to extract structured information from Wikipedia and to make this information available on the emerging Web of Data [100], that is why it is also defined as “the Semantic Web mirror of Wikipedia” [1]. In order to do so, DBpedia uses the Resource Description Framework (RDF) triples as data model for representing retrieved information. Data is extracted from Wikipedia in a combined and linked cross-domain knowledge base and, then, published on the Web. This process allows to disclose information in a structured and standardised format. According to [10] the DBpedia datasets consist of 103 million RDF triples with information of about 2.18 million things), while considering also other open datasets, it reaches 2 billion triples [14]. All DBpedia information is the result of multi-domain ontology which has been derived from Wikipedia as well as localized versions of DBpedia in more than 100 languages. An example of DBpedia RDF statements is reported in Figure 2.6, which includes the definition of a subject (i.e. William Shakespeare) and its properties explained through predicates and objects. This statement expresses the triple which regards William Shakespeare and it can be also described through RDF graph model.

```
<http://dbpedia.org/resource/William_Shakespeare>
   <http://dbpedia.org/property/dateOfBirth> "April 1564" .
<http://dbpedia.org/resource/William_Shakespeare>
   <http://dbpedia.org/ontology/child>
   <http://dbpedia.org/resource/Judith_Quiney> .
```

Figure 2.6: RDF Statements [100]

To avoid ambiguity among entities, every element in DBpedia is denoted by a URI-based reference of the form `http://dbpedia.org/resource/Name` (or by a suffix like `dbr:Name`), where `Name` is derived from the URL of the source, tieing the DBpedia entity directly to a Wikipedia article [1]. The structure of the DBpedia knowledge base is maintained by the users' community, which creates mappings from Wikipedia instances to the **DBpedia ontology**, enhancing the overall clearness and the information structure [87].
To accomplish the goal of this project, the model, which will be described in Chapter 5, leverages on DBpedia ontology Classes. Indeed, it is important to underline that the mapping schema in DBpedia ontology follows a specific structure to call its classes: there is a URI to specify the page of the class, i.e. `http://dbpedia.org/ontology/Class`, and a suffix for the name of the class (for example `dbo:Place`), so that the final URI for the Class containing all places is `http://dbpedia.org/ontology/Place`.
Furthermore, from Figure 2.7 it is possible to see the hierarchical architecture which characterized DBpedia ontology, and its main classes, i.e. `dbo:Place`, `dbo:Agent` and `dbo:Species`, which share a common root node identified in `owl:Thing`. The complete list of DBpedia classes (320 in total [97]) can be found on the Web at `http://mappings.dbpedia.org/server/ontology/classes/`.
At this stage, the question raises: **how does DBpedia extract information from Wikipedia?**
Wikipedia articles consist mostly of free text, but also contain different types of structured information, like infobox templates [10], as reported in Figure 2.8. The

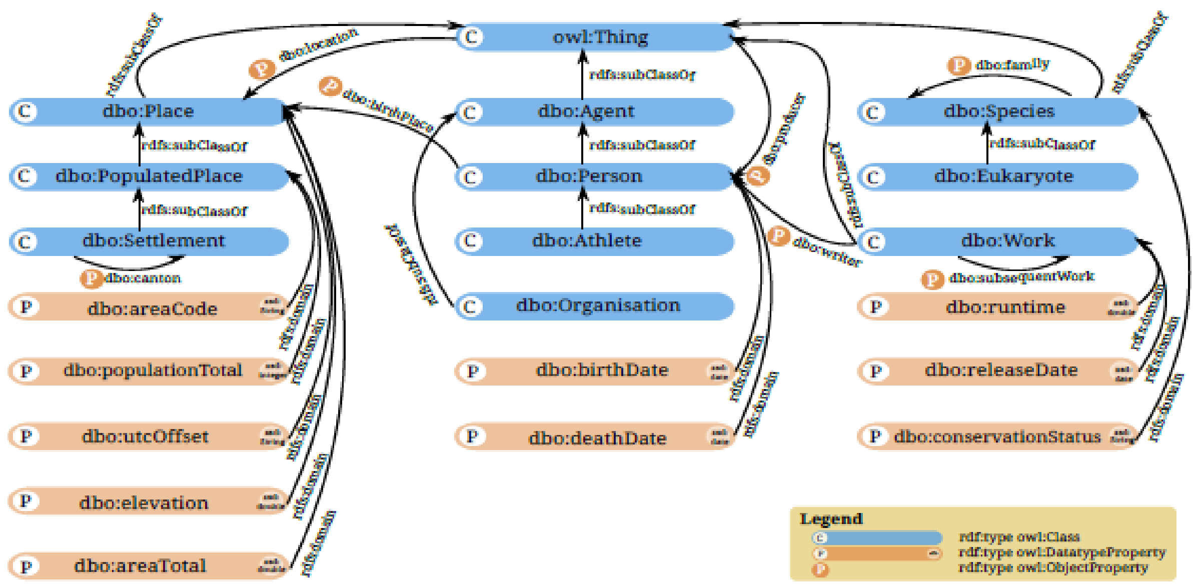


Figure 2.7: Snapshot Of The DBpedia Ontology [87]

DBpedia mapping strategies are based on retrieving data from those infoboxes and the ontology is built though various extractors which translate different parts of Wikipedia pages into RDF statements. According to Lehmann et al. [87] and [1] there are four categories of DBpedia extractors:

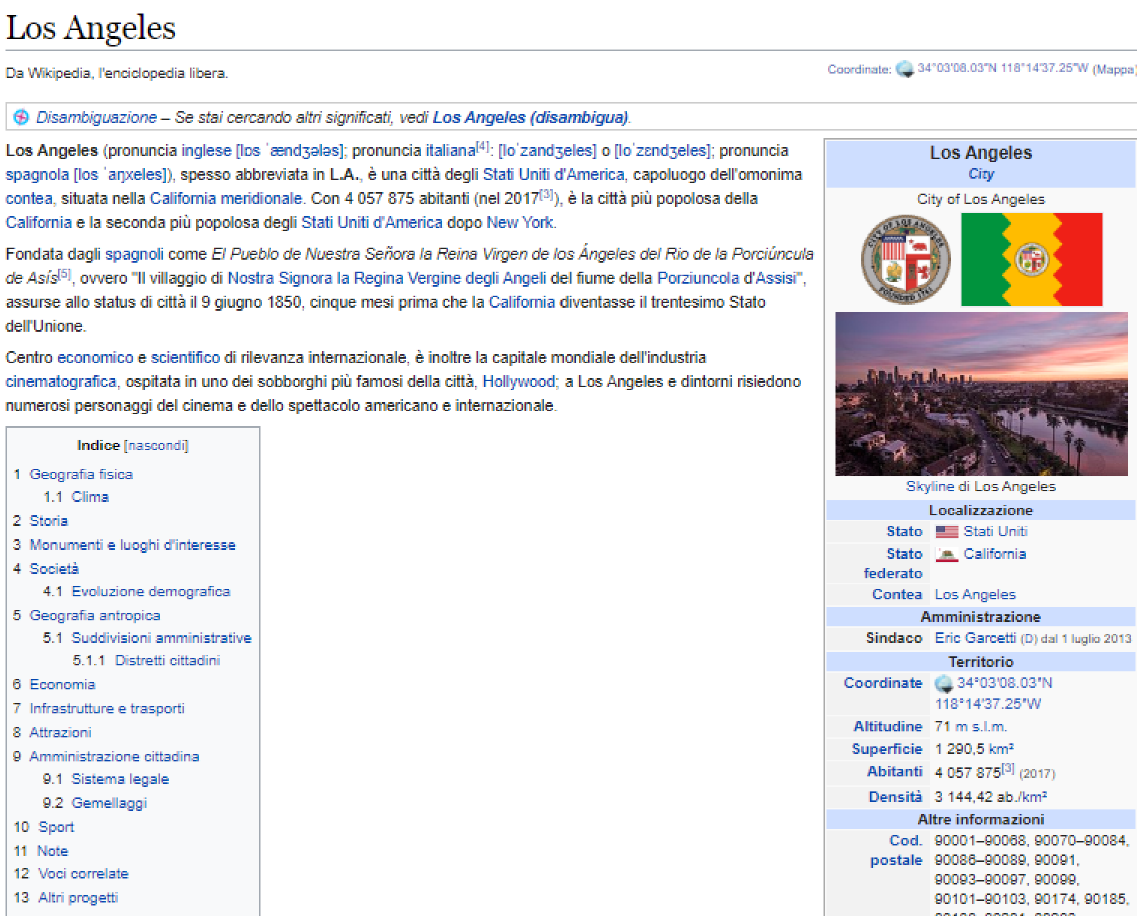
1. **Mapping-based Infobox Extraction**: applies manually written mappings to relate infoboxes (i.e. `rdf:types`) to terms in the DBpedia ontology, specifying a datatype for each infobox property and, thus, improving the quality of data extracted. DBpedia classes are extracted through this technique.

2. **Raw Infobox Extraction**: relies on not-explicit infobox extraction knowledge and, subsequently, the quality of data is lower than with the first method,

3. **Feature Extraction**: applies a number of extractors that are specialized in extracting a single feature from an article, such as geographic coordinates of a place,

4. **Statistical Extraction**: uses machine learning extractors based on Natural Language Processing methodologies to provide data that is based on statistical measures.

Once items are extracted, DBpedia includes them into its structure and periodically updates them.
Similarly to Wikidata, also in DBpedia the items are related by *properties* that are divided into *object* and *datatype* properties as described in Section 2.4. Properties are extracted through Raw Infobox Extraction and are expressed like classes, with



Los Angeles

Da Wikipedia, l'enciclopedia libera.

Coordinate: 34°03'08.03"N 118°14'37.25"W (Mappa)

Disambiguazione – Se stai cercando altri significati, vedi ***Los Angeles (disambigua)***.

Los Angeles (pronuncia inglese [los ˈændʒələs]; pronuncia italiana[4]: [loˈzandʒeles] o [loˈzɛndʒeles]; pronuncia spagnola [los ˈaŋxeles]), spesso abbreviata in **L.A.**, è una città degli Stati Uniti d'America, capoluogo dell'omonima contea, situata nella California meridionale. Con 4 057 875 abitanti (nel 2017[3]), è la città più popolosa della California e la seconda più popolosa degli Stati Uniti d'America dopo New York.

Fondata dagli spagnoli come *El Pueblo de Nuestra Señora la Reina Virgen de los Ángeles del Rio de la Porciúncula de Asís*[5], ovvero "Il villaggio di Nostra Signora la Regina Vergine degli Angeli del fiume della Porziuncola d'Assisi", assurse allo status di città il 9 giugno 1850, cinque mesi prima che la California diventasse il trentesimo Stato dell'Unione.

Centro economico e scientifico di rilevanza internazionale, è inoltre la capitale mondiale dell'industria cinematografica, ospitata in uno dei sobborghi più famosi della città, Hollywood; a Los Angeles e dintorni risiedono numerosi personaggi del cinema e dello spettacolo americano e internazionale.

Indice [nascondi]

Los Angeles
City

City of Los Angeles

Skyline di Los Angeles

Localizzazione	
Stato	Stati Uniti
Stato federato	California
Contea	Los Angeles
Amministrazione	
Sindaco	Eric Garcetti (D) dal 1 luglio 2013
Territorio	
Coordinate	34°03'08.03"N 118°14'37.25"W
Altitudine	71 m s.l.m.
Superficie	1 290,5 km²
Abitanti	4 057 875[3] (2017)
Densità	3 144,42 ab./km²
Altre informazioni	
Cod. postale	90001–90068, 90070–90084, 90086–90089, 90091, 90093–90097, 90099, 90101–90103, 90174, 90185,

Figure 2.8: Wikipedia Page Example: Los Angeles [5]

a URI and a suffix (`dbp`). An example is reported in Figure 2.7 where the property of population is expressed with `http://dbpedia.org/property/population` and `dbp:population`.

2.6 SPARQL: a way to access data on the Web

Both DBpedia and Wikidata are served via a public SPARQL endpoint[1] which, thanks to the documentation provided by [120] and [121], enables the access to DBpedia and Wikidata information from the Web. In particular, SPARQL is a recursive acronym for "SPARQL Protocol and RDF Query Language" and it is used to ask queries against RDF graphs [100], extracting the sets of triples. SPARQL works properly thanks to the fact that both Wikidata and DBpedia can be represented by a Knowledge Graph with formal properties. It is the tool that provides the correct syntax for the user to retrieve data from the ontologies and it works likewise SQL for tables of relational Databases.

[1]`https://dbpedia.org/sparql` and `https://query.wikidata.org/`

According to [54], SPARQL general form is defined by:

- a `PREFIX` where the *Namespaces*[2] are provided
- a `SELECT` section where the subject to return is specified
- a `WHERE` statement which defines the path in order to find the correct triple pattern in the RDF graph
- several modifiers, such as `ORDER BY` and `DISTINCT`

A SPARQL interface is appropriate when the user knows in advance exactly what information is needed. Moreover, to protect the service from overload, limits on query cost and result size are in place [10]. In this research SPARL queries have been used to retrieve information from Wikidata and DBpedia.

2.7 Elasticsearch: a fast and reliable data search engine

The users' main attitude in the Web is that of searching and exploring, particularly when the research process is comfortable, smart and fast. That is the reason why every business in the market strives to better satisfy customers with giving exactly what the customer is seeking [101]. Elasticsearch represents a solution to this problem as a **distributed, scalable and real-time open search engine** which works both with full-text information and for real-time analytics of structured data [62]. It was first developed by Shay Banon in 2010 to format, store and retrieve data in a place called *Index* which can be compared to a table in the relational database world, but where data is prepared for fast and efficient searching [83]. Furthermore, the main entity in the Elasticsearch platform is named **document** that in analogy to a chart-based dataset is the row of a column, but without the constraint of having a fixed structure because single documents are atomic and isolated.
Elasticsearch uses the concept of **relevance** and creates a *Search Index* to sort the resulting document set without the need of up-front schema definition [101]. This process is referred to as **scoring** and it is based on the calculation of a score which is directly proportional to the query match in the research phase. Thus, each document retrieved is associated with a score, i.e. a positive floating number, which indicates the level of relevance of that document with respect to the user research. Relevance is, eventually, computed through a **Practical Scoring Function** described in [48].

[2]A namespace is a group of related elements that have a unique name or identifier each[38]

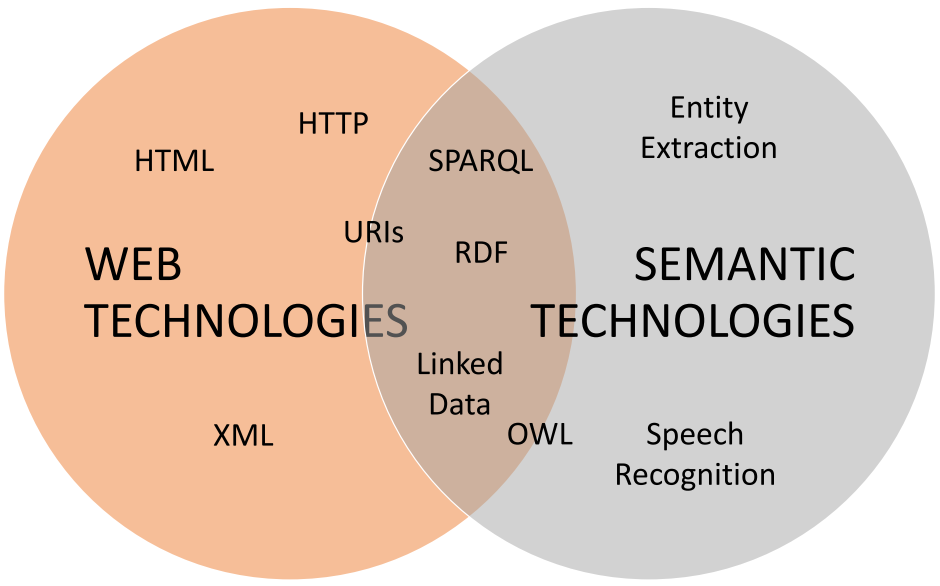


Figure 2.9: Wrap Up Of The Technologies

In conclusion of this preliminary section, we report a schema of the covered topics, distinguishing between technologies already existing for the World Wide Web and the new ones specifically for the use of Semantics. As it is possible to notice from Figure 2.9, there are some tools (which stand in between, or for which the line is blurred) that represent the technologies in development for the Semantic Web. This new paradigm, in fact, is the place where existing Web technologies are used or even improved to extract meaning from data.

Chapter 3

State Of The Art

This third Chapter is dedicated to the review of the literature in the Data Integration field, which encompasses traditional **Data Integration and Web Data Integration approaches using ontologies and unstructured data**. In addition, a focus on companies challenges in this environment has been pursued, considering the **impact of Big Data in management procedures**. Then, a section is devoted to show **More Related Work**, i.e. researches performed on the same datasets or regarding entity linkages approaches and columns' parsing methodologies.
The elucidations shown should be considered taking into account also that the project has the aim to address these two main tasks:

- map information stored in Los Angeles datasets to entities already existing in the ontologies, i.e. face the **Candidates Generation** problem through entity linking,
- extract the ontology Class of the whole dataset according to the information analysed in its columns, i.e. address the **Class Identification** issue.

3.1 Traditional Data Integration

Data can reside in different databases or change throughout time, thus, it has become fundamental to find an effective way to combine it. Data Integration was born to accomplish this task. The goal of an integrating system, in fact, is that of offering a uniform access to a set of autonomous and heterogeneous data sources [44].
In order to fuse the data from multiple sources, resolving the instance-level ambiguities and inconsistencies between two different sources is crucial. Traditional Data Integration addresses these challenges of semantic ambiguity, instance representation ambiguity, and data inconsistency by using a pipelined architecture, which includes three major steps [45]:

1. **Schema Alignment**: heterogeneity in schema means that in the sources to be integrated there are attributes which share the same semantics and others

which do not [45] [112]. In this first phase, a data integration system has the objective to solve those inconsistencies, by applying different strategies to align source schemas [17][44].

2. **Record Linkage**: once the schemas are aligned, the following step consists in deciding which records refer to the same entity and which refer to different ones.

   Pairing every group of records from databases is for sure infeasible, particularly when the volume is significant as it is explained in Section 3.2. As in Schema Alignment, also for Record Linkage many approaches have been adopted to establish which records refer to the same entity [69][20]. The most important one according to the objective of this research is: **Pairwise Matching** which consists in any process of comparing entities in pairs to make a local decision of whether or not they refer to the same entity [72][55]. As stated in [45], there are several approaches to pursue Pairwise Matching during the Record Linkage phase. Some of them are elucidated in the following paragraph.

   The first technique is *rule-based* and applies domain knowledge to make the final statement above the records compared. This approach can be very useful in complex matching scenarios, but it requires significant knowledge about data [72][51]. Furthermore, Fellegi and Sunter [55] proposed a *classification-based* technique based on training machine learning algorithms in order to classify whether a pair of records is a match or a not-match. As in the previous case, also here knowledge is required but with a focus on the number of examples to accurately train the classifier [116]. The third approach proposed for Pairwise Matching is a *distance-based* technique characterized by the computation of distance metrics to measure the dissimilarity between corresponding attribute values. Particularly, Elmagarmid et al. [117] elucidated the different methods to calculate the distance among values according to their nature, i.e. strings or numeric attributes. While these approaches require careful parameter tuning, they enjoy the advantage of being potentially reused for a large variety of entity domains as the knowledge domain needed is minimal. In Chapter 4 we will explain which distance metrics have been applied in this research project and why.

3. **Data Fusion**: the final step to address in Data Integration is that of determining the true value for each data item, which can be for example a cell in a dataset. Thus, Data Fusion step aims to understand which value to use in the integrated data when the sources provide conflicting values. Many times, in fact, information in different datasets can show conflicting values due to mistyping, sharing or out-of-date data [45] and this arises issues during integration. Several techniques have been introduced as solutions to Data Fusion

problem. There are, for example, traditional approaches to Data Fusion, which are typically *rule-based*, i.e. leveraging on rules to select the final value for that data item. Examples can be using the observed value from the most recently updated source or taking the average numerical values. However, when the amount of information is significant, these techniques still appear to be ineffective [22].

3.2 Big Data Challenges for Data Integration

Big Data embodies the paradigm of modern time as everyone nowadays is overwhelmed by information and is not fully aware of the power and challenge behind it. Moreover, Big Data not only refers to the unceasing proliferation of data, but also to the extraordinary diversity of data types, delivered at various speeds and frequencies [115]. This boils down to the definition of Big Data challenges along six different dimensions:

1. According to [79], *volume* refers to the tremendous amount of data existing that is difficult to be handled using the traditional systems, for example nowadays social networking sites alone are producing data in the order of terabytes. Figure 3.1 shows an overview of the social media usage based on monthly active users of the most active social media platforms in each country by territory.

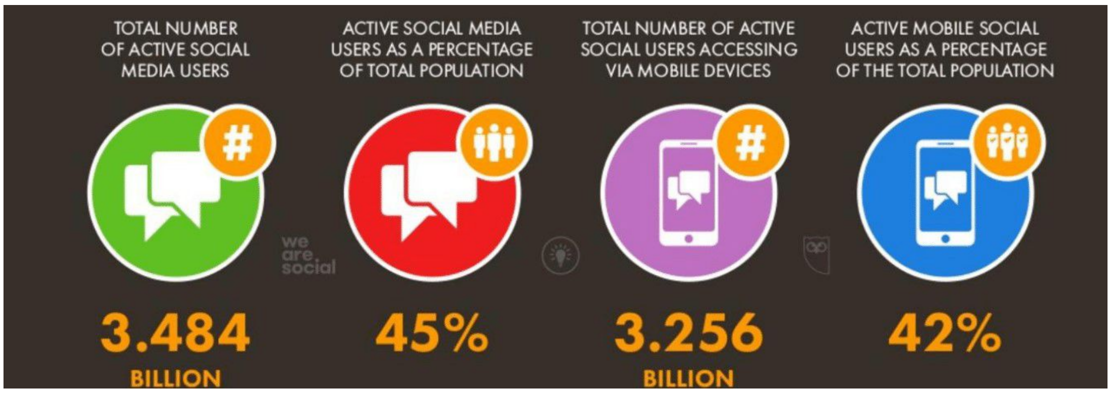


Figure 3.1: Social Media Overview January 2019 [73]

2. *Velocity* refers to the frequency of data generation or the frequency of data delivery [115], which has reached a pace never seen before.

3. *Variety* indicates that Big Data is not of single category as it includes not only data structured in tables or other traditional ways, but also semi-structured information from various resources like web Pages, social media sites, e-mail, documents, sensor devices data and so on. The level of heterogeneity of this kind of information requires more sophisticated data processing capabilities.

4. *Value* shows the way Big Data impact society with new opportunities and disruptive changes which contribute to a significant generation of new benefits across many domains. For instance, by adopting analytic-advanced technologies, organizations can leverage on Big Data to develop innovative insights, products, and services and enhance their profit, business growth, and competitive advantage [67].

5. As stated in [45], Big Data *veracity* relies on the fact that data sources are of widely differing qualities, with significant differences in the coverage, accuracy, and timeliness of data provided.

6. Finally, *Variability* addresses the challenge to maintain data loads and avoid flows inconsistencies with the continuous increase of information generated.

In accordance with the objectives of this research work, we focused on the challenges in the Big Data environment with regard to the second component of Data Integration, i.e. Record Linkage. According to [82], the increase in information volume has caused record linkage timing to dilate, reaching hours or even days. Many approaches have been adopted to address this issue, including the MapReduce programming model introduced by [81] and multiple blocking functions explained by [105]. The main results obtained in the first case consists in a reduction of computing time for Record Linkage thanks to a more even distributed workload across tasks, while Papadakis et al. [105] achieved a better efficiency and higher Recall (see Section 6.2).
Moreover, fast information updates which characterized Big Data have arisen the need of performing incremental record linkage approaches. This issue has been faced by Gruenheid et al. [65] introducing two incremental algorithms which apply correlation clustering on subsets of records and find the optimal solution. The experiments conclusion about incremental record linkage highlighted a significant improvement in efficiency, without sacrificing linkage quality.
On another hand, many projects have been developed in order to address the variety challenge which often culminates in text snippets to be linked with more structured records. One novel approach, for example, is a supervised learning technique which find the record from structured data that has the highest probability of match to the given unstructured text snippet. The matching function is also able to penalize mismatches and learn the relative importance among attributes, through the generation of a *similarity feature vector*. The results of this research study, proposed by Kannan et al. [78], show that the quality of matching improves when the weights of the features are learned and not fixed equally. In addition, their approach manifested great scalability with product offers, where they limited the matching to product specifications in the same category and select the candidates with at least one high- weighted feature. This methodology is highlighted here because a very similar approach has

been applied in this research work.
Furthermore, erroneous record linkage can be the result of out-of-date attribute values. To address the Big Data veracity challenge, Li et al. [88] proposed a model which is able to analyse entity evolution over time and identify obsolete records; while Chiang et al. [37] developed a probabilistic models and faster algorithms, including also clustering methods, to perform temporal record linkage. From the former studies, the results obtained show that performing linkage only on the base of high similarity of one of the words in the input leads to mistakes, however this is not applicable in records that refer to the same real-world entity which, in fact, observe continuity. In addition, Li et al. [88] discovered that applying approaches regarding very similar values and penalizing situations with low similarity between them, is not necessarily appropriate for temporal record linkage.

3.3 Data Integration In The Semantic Web

In this paragraph we have the aim to analyse the different approaches of using ontologies to solve Data Integration tasks and how their knowledge is reused in those situations. Ontologies, in fact, have been extensively used in data integration systems because they provide an explicit and machine-understandable conceptualization of a domain [40]. According to [131] there are three main ways to exploit knowledge from ontologies:

1. **Single ontology approach** where a shared global ontology is used to provide a uniform interface to the user for relate source schemas. Of course, this requires the sources to have the similar schemas and the same level of granularity. A typical example of a system using this approach is SIMS [8].

2. **Multiple ontology approach** where local distinct ontologies describe a data source and, then, are mapped to each other. The OBSERVER system [96] is an example of this approach.

3. **Hybrid ontology approach** where, at first, for each source schema a local ontology is built and, consequently, that is mapped to a global shared ontology so that new sources can be easily added with no need for modifying existing mappings. The framework described in [40] provides an example of this.

Moreover, from the literature it is possible to identify five different uses of ontologies in Data Integration [40]:

1. **Metadata Representation** which consists in using a single language to explicitly represent metadata by a local ontology

2. **Global Conceptualization** where a global ontology provides a conceptual view over the schematically-heterogeneous source schemas

3. **Support for High-level Queries** where a query is formulated without specific knowledge by the user and, then, it is rewritten into queries over sources, based on the semantic mappings between the global and local ontologies

4. **Declarative Mediation** in which hybrid peer-to-peer query processing system uses a global ontology as a declarative mediator for query rewriting between peers

5. **Mapping Support** where an ontology built for synonyms can be used to improve automation in mapping process

In conclusion, one of the key approaches to have unified and transparent access to data is the OBDA/OBDI (Ontology Based Data Access And Integration) model proposed by [110] and [34] where an *extension level* is built on the specifications (i.e. the triples in the ontology) to represent the data at the source with the structure of the schema in the ontology. Figure 3.2 provides a graphic description of the structure of the model.

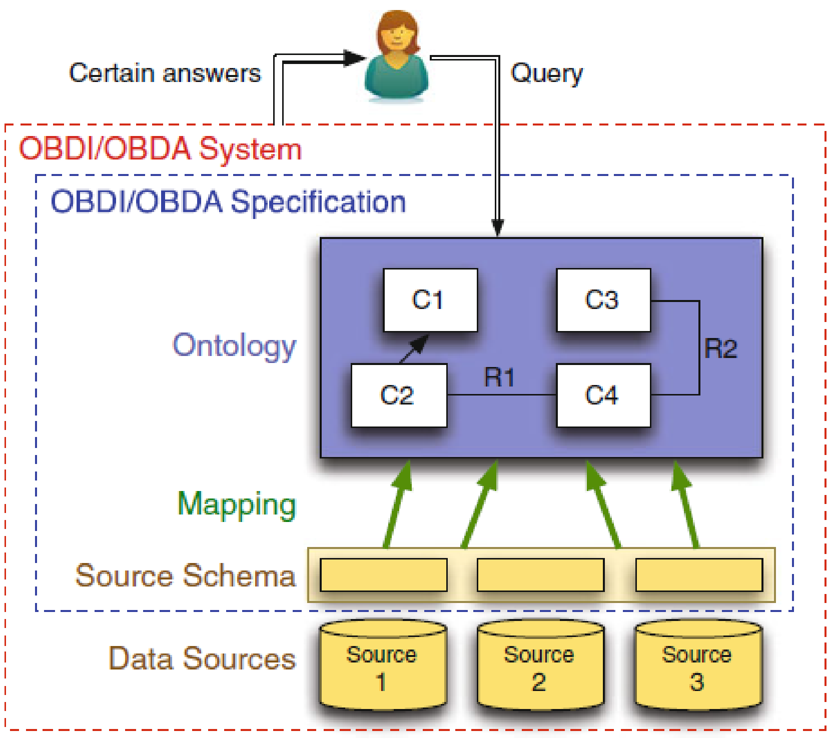


Figure 3.2: OBDI/OBDA Specification And System

3.4 The Voice Of Companies

Today, the discovery of Big Data opportunities for companies forces deep changes into most businesses, especially those that depend on mass consumers or related fields that produce an enormous quantity of information [115]. In many cases, in fact, that information is decentralized in different databases or, even worse, in heterogeneous formats. Combining data from multiple sources is fundamental for enterprises to obtain a competitive advantage and find opportunities for improvement.
Moreover, the Web represents the largest data source to exploit as it contains millions of databases with a wide range of information. Leveraging this incredible collection of data raises significant challenges [44]:

- **Schema Heterogeneity** on a much larger scale as information on the Web are countless and belong to independent authors,
- **Data Extraction** from tables embedded in Web pages is challenging because information is usually uncleaned and even contradictory,
- **Disparate Resources** makes it difficult to establish the right level of coordination among companies,
- **Adding new information** for greater context and more source material requires a lengthy manual process when data comes from the Web.

Thus, Data Integration represents one of the most serious issue for companies development. According to [35] most of semi-structured and unstructured data is not actually owned by organizations and being able to combine it with information collated by companies can generate greater opportunities than just trying to bring all data into one warehouse. In addition, due to the advent of Big Data, organizations are now awash in data and in many cases, they do not know what data exists within the organization and when they search for information which is not available it usually gets recreated from other sources.
Modern Data Integration solutions are emerging to face these challenges and offer a simplified and secure method of data collection, both from external datasets and from the Web, that easily scales with any business needs. Examples of these projects can be found in Amazon Redshift, Snowflake, Google BigQuery, Azure, or a number of other options [7].
In conclusion, Data Integration challenges stem also from the way companies access to data and how fast it can be accessed when real-time responsiveness and continuous availability is required. For real-time data management, in fact, clients need to rethink their data architecture, seeking for a more flexible and intelligent infrastructure [129].

3.5 More Related Work

In this final section of the State Of The Art, we decided to describe the projects most related to the one developed in this research. Particularly, a first paragraph is dedicated to the work by Knoblock et al. [80] , a second paragraph to the Semantic Annotation of Documents by [29] , and a last paragraph to a Domain-Independent Approach for Semantic Labeling by [109].

3.5.1 Automatic Spatio-temporal Indexing to Integrate and Analyze the Data of an Organization

In this research project they started from the idea that the data available, which belongs to the same datasets of the current work, has a strong spatial and temporal component. Thus, the objective of the research was that of **determining a spatio-temporal index** for each record through the development of an automatic algorithm trained to identify the fields that contain temporal or spatial information and then normalize it into standard formats. In order to identify temporal or spatial information the previously-developed (see Section 3.5.3) DSL approach for semantic labeling was used. DSL extracts a set of similarity features, such as Jaccard similarity [103] between the attributes name, TF-IDF cosine similarity [92] for overlapping values occurrences, Distribution similarity [77] and Histogram similarity [90], between unseen and sample data for each attribute and, subsequently, uses a trained Logistic Regression model to classify whether a new attribute is similar to an indexed type according to a probability threshold. In addition, a data normalization step was applied where an unsupervised method for automatic data cleaning was developed to standardized information formats. Finally, the last phase consisted into storing all of the data across all of the original sources in ElasticSearch, using a common format for representing all the data, so that information could become available for retrieval. The main results show that out of 177 datasets, only 23 were incorrectly mapped to the wrong spatial extent, while 154 were correctly mapped to the right temporal extent.

3.5.2 Semantic Annotation of Documents Based on Wikipedia Concepts

The main task of this research project consisted into identifying concepts from a selected ontology (Wikipedia was treated as such) that are relevant to a certain document or that are referred to it, as well as selecting specific passages in the document where the concepts are mentioned. This specific type of semantic annotation is called *wikification*. The main steps of *wikification* includesthe identification of phrases in the input document that refer to a Wikipedia concept, the determination of which concept a phrase refers to and the selection of relevant concepts. The approach described by [84] was applied to detect whether a phrase of an input document refers to a Wikipedia concept making use of Wikipedia hyperlinks. In addition, they faced the disambiguation problem by applying a global approach (disambiguate all the mentions as a group) to the set of concepts. This approach, in particular, includes the generation of a mention-concept graph [29] which collects all the semantic-related concepts (when a concept is relevant to a given document, the concept that is related semantically to the former is more likely to be relevant to that document, comparing to another concept not semantic-related) through an iterative process. Once all the concepts are accumulated, the model selects the concept with the highest pagerank as the most relevant one; thus, the one that explains the whole input document. The results of this research are very satisfactory, reaching a classification accuracy of 96.2%.

3.5.3 Semantic Labeling: A Domain-Independent Approach

In this project work the main objective was that of annotating source attributes with classes and properties of ontologies, i.e. apply semantic labeling, especially to situations where sources were already mapped to a common ontology and new sources needed to be mapped using the same ontology. The overall approach builds a feature vector $f[k]$ representing the similarity between two attribute according to the metric k, and trains a classifier to label each f_{oj} as True/False. If the label of f_{oj} is True, the semantic type of the unlabeled attribute is the semantic type that was recorded for an initial one. From that, it is possible to conclude the semantic type of unlabeled attribute. The main classifiers evaluated were Logistic Regression and Random Forests [31]. The results of the Domain-independent Semantic Labeler (DSL) show that training the classifier from the same domain, which provides more information about the characteristic of data, slightly improves the accuracy of the classifier.

Chapter 4

Methods And Materials

In this research project the approach adopted to investigate the problem and achieve the objectives is based on experimental-driven developments which have led to continuous optimizations and discoveries, contributing to the definition of a final model. In particular, each step has been fundamental to determine the improvements of the following ones. As we will explain in Chapter 5, the final model design includes various approaches that the user can follow in order to automatically extract the content from a dataset in the form of a category. In this Chapter we elucidate the basis on which the project has been developed, i.e. the methodology used for comparing available data with knowledge stored in ontologies, which consists in the Candidates Generation phase, the tools applied throughout the research and the reasons behind it. In conclusion, a section is devoted to illustrate the materials used as it regards the technicalities.

4.1 Column Entity Annotation for Candidates Generation

As a first step, we focused on the comparison between the information in the datasets with respect to ontologies' knowledge. This process includes the application of **Similarity Measures** between entities in the different sources. We provide here the definition of the Similarity Measure [61]:

Definition 1. *A* ***Similarity Measure*** *is a function $s : E \times F \to \mathbb{R}$ that takes a pair of items ($e \in E$ and $f \in F$ respectively) and returns a scalar $r \in \mathbb{R}$ measuring their similarity. E and F represents two different data sources, while the function s can be expressed through a distance metric.*

Leveraging on similarity measures was fundamental to address the **Candidates Generation** problem, defined as:

Definition 2. *Given a word* ***w*** *expressing a concept in the first data source E, the model, which is for instance a system or a program, searches in the second data* *source F and, using a similarity measure s, generates a set of relevant words called* ***Candidates*** *w_i', where i is the number of Candidates, that are closest to* ***w****. Candidates generation can be applied in situations with more than one word or even for entire phrases.*

Candidates Generation process has the aim of finding the exact match between the information taken as input and the available resources in knowledge bases. A generation of good and valuable candidates was the essential cornerstone of the project. Thus, we introduced an approach, called **Column Entity Annotation (CEA)** and defined by [123], that is based on parsing the cells in a single column of a dataset using a significant distance-based approach. The Column Entity Annotation model takes as input the cell of a column and compares it with information stored in the ontologies described in Section 2.4. To extract data from knowledge bases, the CEA leverages on the execution of queries in SPARQL (see Section 2.6) and on the computation of distance metrics between the label searched and the results obtained from the queries. As reported in Section 3.1, in fact, when the knowledge about data is low, distance-based methods are preferred, especially considering the opportunity to reuse them for a large variety of entity domains. Indeed, in this project the data available is heterogeneous with respect to its schema, format and domain, and there is not knowledge about its content. Section 4.4.2 illustrates an example of a dataset on which we ran the experimentations.
The choice of adopting CEA approach was carried out by the fact that it reached a very good result in terms of quality of candidates generation with a Recall (defined in Section 6.2) equals to 0.91. Another element which led to the adoption of this model consists in the relationship between its performance and the characteristics of the input data, in terms of the quantity of given entities. CEA performances, in fact, did not improve significantly when considering more than 100 rows as input data, each row containing from one to five words depending on the nature of the column. Having fewer input data but still obtaining good results is the key to develop efficient model. That is why also in this project, trials have been made mostly with datasets of 100 rows each.
Moreover, when applying the CEA model we evaluated different similarity measures for Candidates Generation:

- **Levenshtein Distance** is a methodology for measuring string edit distance, as well as the simplest one. It is defined as follows [58]:

  Definition 3. *Levenshtein Distance (LD) is a measure of the similarity between two strings, which we will refer to as the source string (s) and the target string (t). The distance is the minimum number of characters deletions, insertions, or substitutions required to transform s into t.*

  For example: if s is "test" and t is "tent", then LD(s, t) = 1, because one substitution (change "s" to "n") is sufficient to transform s into t.

The higher the LD, the greater the difference between the strings, thus the less a candidate is relevant in the comparison with the input data (a row of a column). Usually, applications of LD are in spelling correction and that is why this metric has been selected for this project. Considering the ontology as a vocabulary, in fact, the objective of candidate generation is to find the closest word, which likely culminates in matching exactly with the input word, from that vocabulary. For the accuracy of the results, we applied the **Normalized Levenshtein Distance**, which is computed as LD divided by the length of the longest string in the input.

- **Term Frequency, Inverse Document Frequency (TF-IDF)** described by [92] is considered one of the best feature extraction techniques as it is easy to compute and it represents a basic metric to extract the most descriptive terms in a document [127].

  Definition 4. *Term Frequency, also known as TF, measures the number of times (n) the i-term occurs in the j-document.*

  $$TF_{i,j} = n_{i,j} \tag{4.1}$$

  Definition 5. *Inverse Document Frequency, also known as IDF, weighs up the effects of less frequently occurring terms in a document by applying the formula:*

  $$IDF_i = 1 + log\frac{D}{|d : t_i \in d|} \tag{4.2}$$

  where D is the set of documents and d is the number of documents with the i-term t in it.

  Definition 6. *TF-IDF is the multiplication for each i-term of Term Frequency with its Inverse Document Frequency on each j-document.*

  $$TFIDF_i = IDF_i \times TF_{i,j} \tag{4.3}$$

  The TF-IDF metric is directly proportioned to the number of times a word appears due to the TF, but it is decreased by the number of documents that contain the word (IDF), which helps account for the fact that some words are more common in general (e.g. “the” or “a”); thus, they do not affect the results and this is relevant according to the input data reported in the Appendix. In the CEA context, an adaptation of the TF-IDF was applied, described as follows [123]:

  - TF of a given semantic feature is the number of cells for which the first candidate has this feature,

  - DF is defined as the number of total occurrences in all candidates of that feature.

- **Jaro Similarity** belongs to the group of *edit distances*, together with Levenhstein Distance, which are used to measure the similarity between two strings.The higher the Jaro Distance for two strings, the more similar the strings. According to [2], the definition is the following one:

Definition 7. *The Jaro Similarity sim_j of two given strings s_1 and s_2 is defined as:*

$$sim_j = \begin{cases} 0 & \textit{if } m = 0 \\ \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) & \textit{otherwise} \end{cases} \tag{4.4}$$

Where:

  - $|s_i|$ is the length of the string
  - m is the number of *matching characters*
  - t is half the number of *transpositions*

Two characters from s_1 and s_2 respectively, are considered matching only if they are the same.

In this research project we adopted a variant of Jaro Distance which gives more favourable ratings to strings that match from the beginning for a set prefix length l. This alternative is called **Jaro-Winkler Similarity** and is a string *edit distance* that was developed in the area of record linkage to best suit short strings such as names, and to detect typos [89]. The adoption of this similarity has been led by the interest in how the model would answer in situations where the input string as a whole does not have the same importance, i.e. the first group of characters is more relevant than the last ones in the string.

Definition 8. *Jaro–Winkler similarity uses a prefix scale* $\boldsymbol{p}$ *and a set prefix length* $\boldsymbol{l}$*. Given two strings s_1 and s_2, their Jaro–Winkler similarity sim_w is:*

$$sim_w = sim_j + lp(1 - sim_j) \tag{4.5}$$

Where:

  - sim_j is the Jaro similarity for strings s_1 and s_2
  - l is the length of common prefix at the start of the string up to a maximum of four characters

- p is a constant scaling factor for how much the score is adjusted upwards for having common prefixes (the standard value for this constant is 0.1)

The Jaro-Winkler distance d_w is defined as

$$d_w = 1 - sim_w \tag{4.6}$$

4.2 Warm Up: Wikidata and DBpedia

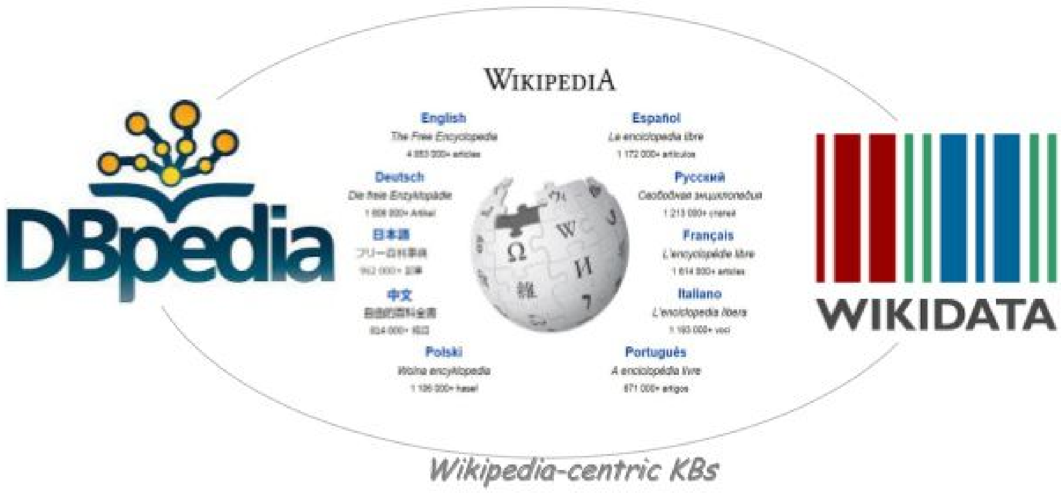


Figure 4.1: Wikidata and DBpedia [74]

This section is dedicated to explaining the choice behind using both Wikidata and DBpedia for accomplishing the objectives of the research project. The Candidates Generation phase, in fact, includes the Wikidata ontology together with DBpedia, but the final output is expressed through DBpedia URIs. The reasons that pushed us to make this decision were several. First of all, the goals of the project suggest to apply a *KG-agnostic* implementation, which means to not narrow it down with a specific ontology and a peculiar schema to use as reference. Thus, instead of using one single ontology, we leverage on two knowledge bases to enhance the performances of the model. Moreover, Wikidata represents the best choice in terms of looking for knowledge as it is considered the central data management platform of Wikipedia

[75]. Wikidata is also expanding continuously due to a direct editing interface where people can create, update or fix facts instantly, while DBpedia is a static, read-only dataset that is updated periodically. On another hand, DBpedia structure is to be preferred to the one of Wikidata, especially to define the final class, as it is better organized and more understandable. Just think about DBpedia URIs compared with Wikidata Qnodes. In addition, DBpedia starts with RDFS as a base data model which is universally recognised while Wikidata has its own data model. In conclusion, Wikidata categories have been judged as not suitable for the goal of this research, as Chapter 6 illustrate; thus, we leveraged on Wikidata broad knowledge, on one hand, and optimize the results with DBpedia structure.

4.3 OpenStreetMap

This final section of methodologies is devoted to introduce OpenStreetMap which is a collaborative project to create a free editable map of the world [3]. It is an open data platform which emphasizes local knowledge, as contains information about items on the basis of their location.

In this data, OpenStreetMap provides also the definition of a **Class** to which the item belongs and specify a **Type** for each Class. For this reason it has been included in the project as a final optimization step. Indeed, the data taken as input belongs to a circumcised space, which is the city of Los Angeles.

The request is made through the OpenStreetMap API and the answer looks like the example reported below:

```
'place_id': 182029085,
'lat': '34.11821875',
'lon': '-118.30029332196601',
'display_name': 'Griffith Observatory, 2800, East Observatory
Road, Thai Town, Hollywood, Los Angeles, Los Angeles County,
California, 90027, United States of America',
'class': 'tourism',
'type': 'museum'
```

As it is possible to notice, the item searched in OpenStreetMap is the Los Angeles famous Griffith Observatory, which is characterized by an identifier ('place_id'), spatial coordinates ('lon' and 'lat'), its full name ('display_name'), a generic category ('class') and a more specific one ('type'), which is what has been considered for this research study. In this way OpenStreetMap enables the user to exploit the information stored in the dataset and give useful insights for category discovery, as Chapter 6 elucidates.

4.4 Materials

In this section we briefly survey the tools used to develop the whole project.

4.4.1 Python

The Python language has been chosen for this research as it was designed to be highly extensible and able to be used for different tasks, rather than just data cleaning or data mining. Moreover, this design of a small core language with a large standard library and an easily extensible interpreter was preferred to a more complex and probably more sophisticated programming language like Java. In addition, Python is meant to be a highly readable language, which is very useful in this case as the model is designed to be improved and managed by a final user. Indeed, Python is designed to have an uncluttered visual layout, frequently using English keywords where other languages use punctuation. In conclusion, the existing methodologies applied in the starting phases of the model have already been developed in Python language, so it turned to be useless and time-wasting to use a different programming language.

4.4.2 Dataset Description

The project has been carried out with the collaboration of the city of Los Angeles to provide better access to the data that they already maintain. The datasets used for the definition of the model, in fact, belong to Los Angeles, which has published a great deal of their city datasets as open data available in two repositories: `https://data.lacity.org` and `http://geohub.lacity.org`. In these two repositories there are about 1,100 datasets totally that contain detailed information about everything related to the city from business licenses to which trees are planted in each park. For the purpose of this research, we focused on the datasets which were labeled as more understandable and meaningful, such as *Listing of Active Businesses*, *Restaurants* and *Department of Recreation and Parks Facilities*. This kind of datasets, in fact, resulted to be more suitable for the trials during the model development. Thus, 41 datasets were selected for the research, of which we reported an example in the attachments.

Chapter 5

Model Design

This chapter is focused on the explanation of the final model, including all the tools implemented in every step and the developments made. The improvements applied in each phase have been the outcome of the analysis of the results obtained throughout the work, in accordance with a *learning by doing* approach. For the purpose of this research, all the experiments have been done on a single column, taking the Class identified for that column as representative for the whole dataset, however working on multiple columns simultaneously should not be discarded as a possible improvement. The main phases of the project are described as follows:

- a **Warm Up Phase** to find a strategy to address the problem of identifying the best column of a dataset to use as input data,
- a **First Phase**, where we investigated a first approach to solve the Candidates Generation problem and exploit Wikidata ontology for the identification of the Class,
- a **Second Phase**, where a more sophisticated model has been introduced to optimize both Candidates Generation phase and Class identification task, taking into account also DBpedia knowledge base,
- a **Final Phase** that is focused on the development of the final model which encompasses the approaches previously investigated, considering the improvements applied for the identification of the column Class.

We tried to summarise the steps pursued in Figure 5.1 in order to give the reader an overall idea of the whole process. As it is possible to see, every approach we evaluated takes as input the column selected during the Warm Up phase and gives as output the identified class for every dataset. This methodology takes shape in the final model represented in Figure 5.6.

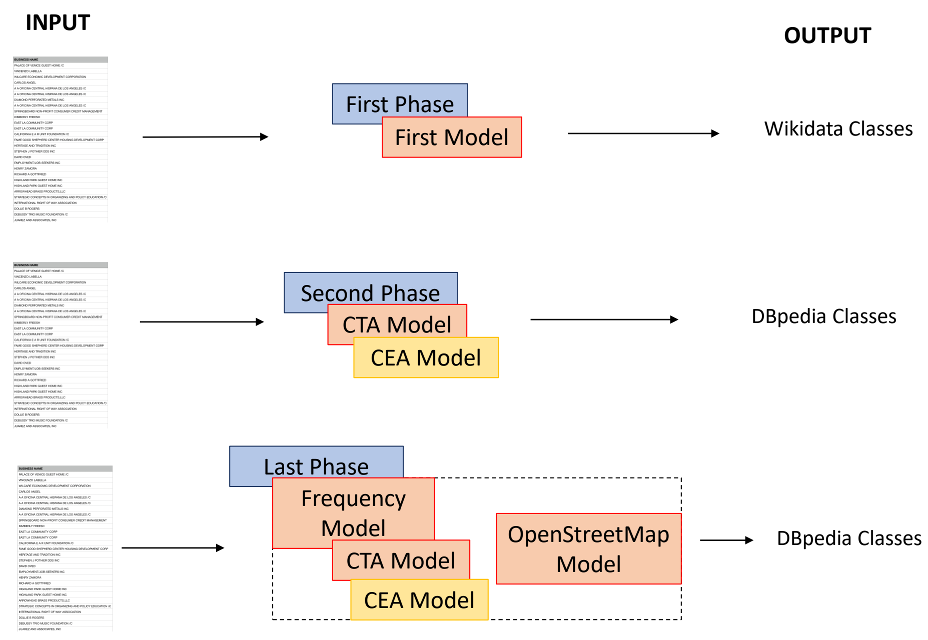


Figure 5.1: Flowchart Of Different Methodologies Used During The Research Study

5.1 Warm Up Phase: address the Column Selection problem

The majority of the Los Angeles city data is expressed in table form, i.e. a chart where the columns represent the attributes and the rows their respective values. As it is possible to notice from the Appendix, the available datasets do not have a standard format and contain information of every kind, such as strings, integers, times and dates. For the purpose of this research, the most useful column to select should be the one which most describes the content of the dataset, such as the one with names or general descriptions of the items represented. Thus, the objective of this Warm Up phase consists in finding a strategy to face the Column Selection issue and extracting the most useful column from each dataset. For the usability of the model, the column selection should be made autonomously by the algorithm in order to decrease the time to access information in the datasets. A heuristic approach has been pursued to address this problem, starting from the observation of the datasets and discarding columns showing certain characteristics, such as containing numbers, symbols, email addresses and so on. Los Angeles city datasets, in fact, are variegate and very different one another; thus, many aspects have been taken into account and others have been discarded during Column Selection phase. We described the criteria for evaluating the usability of a column in Section 6.1 as they were chosen on the bases of the trials made on the datasets when attempting to filter and isolate one relevant column. More sophisticated approach could be implemented to retrieve the most explanatory column in a dataset. However, the focus of the project has been more intense on the other phases. In addition, the variety of available datasets embodies a situation which suggest to be realistic as datasets and data formats are not similar.

5.2 First Phase: Candidates Generation and Class Identification in Wikidata

In the first phase of the project we investigated a starting approach for Candidate Generation and Class Identification problems, considering Wikidata as reference ontology and taken as input the column selected during the Warm Up Phase. To perform Candidates Generation, the first step has consisted of pursuing the **entity linking** task to map table cells of a column to entities in Wikidata Knowledge Base. To do so, a very simple approach has been followed: for each cell that must be annotated, the algorithm takes that cell as input, searches through Wikidata API for the given row string and obtain up to 100 candidates based on the instance's content. The candidates represent Wikidata entities which are retrieved through a SPARQL query request to Wikidata API. For the nature of the ontology, the ultimate result is a list of qnodes candidates for each label, as shown in Table 5.1.

BRANCH NAME	QNODES
Sylmar	Q7660594, Q3829489, Q2839471...
Woodland Hills	Q482390, Q2891280, Q38477...
Harbor City	Q61678450, Q8569060, Q846103...
Angeles Mesa	Q493378, Q4820395, Q489311...
Benjamin Franklin	Q817496, Q575822, Q458029...
Sunland - Tujunga	Q61678251, Q8102934, Q85762...
Northridge	Q1026939, Q857602, Q852301...
Fairfax	Q501785, Q6567309, Q74720...
Chatsworth	Q1068289, Q8585002, Q837205...

Table 5.1: Example of Qnodes candidates for a list of Library Branches in Los Angeles

Once the candidates have been generated, the next step has been to identify a first approach to pick the corresponding classes to which candidates belong in accordance with Wikidata structure and hierarchy.

items	Q13220204	Q13360155	Q13410400	Q1496967	Q15642541
Autauga	1.0	1.0	1.0	5.0	5.0
Baldwin	2.0	2.0	2.0	3.0	3.0
Barbour	2.0	2.0	1.0	2.0	2.0
Bullock	1.0	1.0	1.0	1.0	1.0
Calhoun	2.0	2.0	1.0	4.0	4.0
Chambers	2.0	2.0	1.0	2.0	2.0
Cherokee	2.0	2.0	0.0	3.0	3.0

Table 5.2: Example Of Class Selection In Wikidata

To address the Class Identification problem, we retrieved the class from each instance though queries in Wikidata and built a **comprehensive table** to select the correct class, leveraging on the definition of **transitive closure** in a graph [5]:

Definition 9. *If **X** is a set of items in a directed graph and **xRy** means "there is a direct relationship R between item **x** and item **y**" (x,y $\in$ X), then the **transitive closure** of **R** on **X** is the relation $\boldsymbol{R^+}$ such that $\boldsymbol{xR^+y}$ means "it is possible to go from item **x** to item **y** in one or more steps via the relation R".*

An example of the comprehensive table built by the model is reported in Table 5.2. As it is possible to notice, each row represents the input string, i.e. the cell of the selected column in the dataset, while the columns are the union of the **transitive closure** of the *SubClassOf* relationship over all the classes in Wikidata generated from the queries execution. The table is built such that all classes that are a *SuperClasseOf* any class in the closure have been included. In order to face the Class Identification problem, we leveraged on the concept of **disambiguation**, which definition is reported here [108]:

Definition 10. *A class is said to disambiguate a label well if it is related to one and only one candidate of that label.*

In this case, after having evaluated the transitive closure and having generated the comprehensive table, the class that best **disambiguates** the greatest number of labels is identified and, then, is selected to categorize the dataset. In the model above we leveraged on the fact that all the labels in a column must belong to some common Wikidata class. Thus, the generation of classes does not reach a unmanageable volume.
In case shown in Table 5.2 for example, value 1.0 in the first row and first column means that only one candidate for item named “Autauga” is *SubClassOf* type Q13220204. Moreover, Q13410400 seems to be the class that disambiguates the majority of labels, i.e. rows in the table, and, thus, the class that best describes the content of that column. As the results obtained were not satisfying (see Section 6.2), a more sophisticated approach has been applied for Class Identification task, i.e. the **Column Type Annotation (CTA)** model. This model, which is described in the

following section, introduces the exploitation of the DBpedia ontology due to the fact that Wikidata has revealed to be useful for the identification of the candidates, but too inaccurate for the recognition of meaningful classes.

5.3 Second Phase: a top-down approach for Class Identification in DBpedia

The second phase of the project has been devoted to investigate **Column Type Annotation (CTA)** model performance for Class Identification in DBpedia. CTA model has been introduced before by [123]. The choice of applying this methodology has been dictated by two main factors:

- CTA leverages on the Column Entity Annotation model (CEA), introduced in Section 4.1, for addressing the problem of Candidates Generation. Indeed, the candidates generated by CEA model are the input for the CTA approach. Integrating CEA with CTA is beneficial because CEA approach obtained good results in terms of accuracy as described in Section 4.1. Thus, this innovative approach to Candidates Generation has been pursued with the aim to increase the quality of the results.
- CTA algorithm is able to query information both in Wikidata and DBpedia, encompassing the knowledge of two different ontologies.

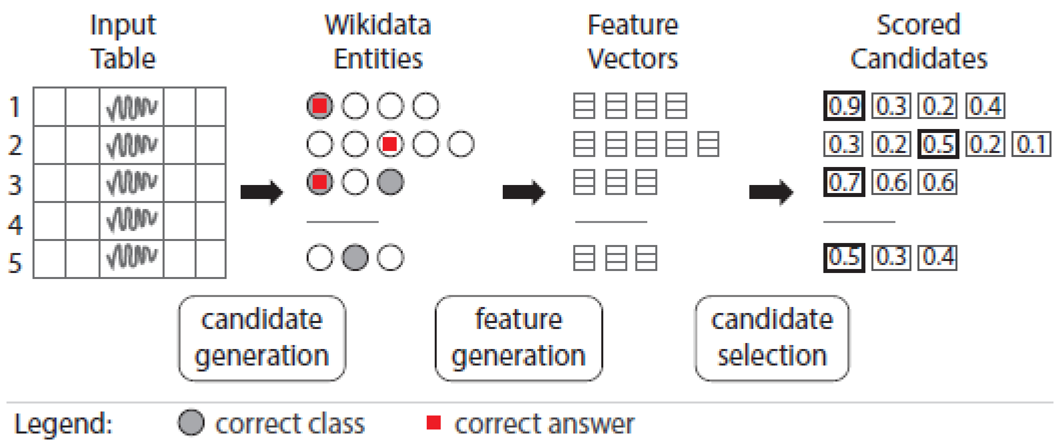


Figure 5.2: Overview Of CEA Approach [123]

5.3.1 Column Entities Annotation

Since we included this approach in the final model, we provide a deep description of CEA here, reporting as a first step an overview of this approach in Figure 5.2.

For **Candidates Generation**, the CEA model builds two Elasticsearch Indexes with the aim to increase the quality of the entities, leveraging on Elasticsearch scoring system based on relevance:

- **Wikidata Elasticsearch Index**, used to retrieve multilingual labels, aliases and descriptions with a focus on different fields, according to the input entities and the structure of Wikidata pages. The candidates generated are, then, combined with the ones obtained with the standard approach described in Section 5.2 with SPARQL queries in Wikidata API,

- **DBpedia Elasticsearch Index**, to collect candidates from DBpedia and, immediatly, mapping them from URIs to their corresponding Wikidata qnodes.

Moreover, the model generates another query in order to face the abbreviations problem, i.e. that situation when rows contain abbreviated words which result to be hard to understand and to match.
Finally, CEA applies a method called **Tokenization** to improve its performance in Candidates Generation phase.

Definition 11. ***Tokenization*** *is the act of breaking up a sequence of strings into pieces such as words, keywords, phrases, symbols and other elements called tokens [122].*

In our case, tokenization was used only with words and not with symbols or other elements, and was fundamental to apply the distance measures for Candidates Selection, shown in Section 4.1.
Thus, the model creates one last query to tokenize the strings taken as input.
Once Candidates Generation phase is concluded, for each Qnode we have the Elasticsearch Score (ES) from Wikidata or DBpedia and the number of the query it comes from (4 queries are generated totally by CEA model). Table 5.3 shows an example of how the results look like.

Query	**Qnode**	**ES**
4	Q10901384	73.49035
4	Q973020	69.01582
4	Q18614770	66.73978
3	Q19817567	57.05981
3	Q37256951	56.374943
2	Q18224405	51.187073
2	Q539581	51.14039

Table 5.3: Example Of Candidates Generation In CEA Model

Furthermore, to proceed in the investigation of the Class, we needed a strategy to select one single candidate for each row. In order to do so, it has been fundamental to combine and sort the candidates generated. To face this problem, we leveraged on the CEA model and the concept of **Normalized Reciprocal Rank**, which is defined as follows [5]:

Definition 12. *Considering rank position, K, of the first relevant results retrieved,*

$$ReciprocalRank = \frac{1}{K}$$

while Normalized Reciprocal Rank is the mean Reciprocal Rank across multiple queries:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i}$$

Indeed, after having generated the candidates from all the queries, the CEA model uses **Normalized Reciprocal Rank** as scoring system to sort them.
Once candidates have been sorted, the last step to achieve before addressing Class Identification problem was represented by **Candidate Selection**, i.e. obtaining one single candidate for each entity of the input data.
For the **Candidate Selection** phase, we leveraged again on the CEA model. In particular, this approach applies several techinques for Candidates Selection as described by [123]:

- *Top 1 Candidate*: select the top-scoring candidate from the candidate generation module, thus the one with the highest ES score;
- *Heuristic Linear Combination*: define a feature engineering methodology that combines different scores to produce an aggregated result. As it is shown in Figure 5.2, in fact, before candidate selection phase, there is an intermediate step called **features generation** where the list of candidates is used to compile a uniform-length feature vector v that contains the set of classes and properties that describe them. Once the feature vector is compiled, the scores are computed on the basis of:
  - **Lexical Features**: which is the lexical similarity between the original query string tokenized and the labels in the Knowledge Graph. Levenshtein Distance, introduced in Section 4.1, is used to measure this parameter;
  - **Semantic Features**: which is the semantic coherence among cells in a column. To express it, the modified TFIDF, shown in Section 4.1, is computed for each candidate using the feature vector v and the first candidate generated in the previous phase.

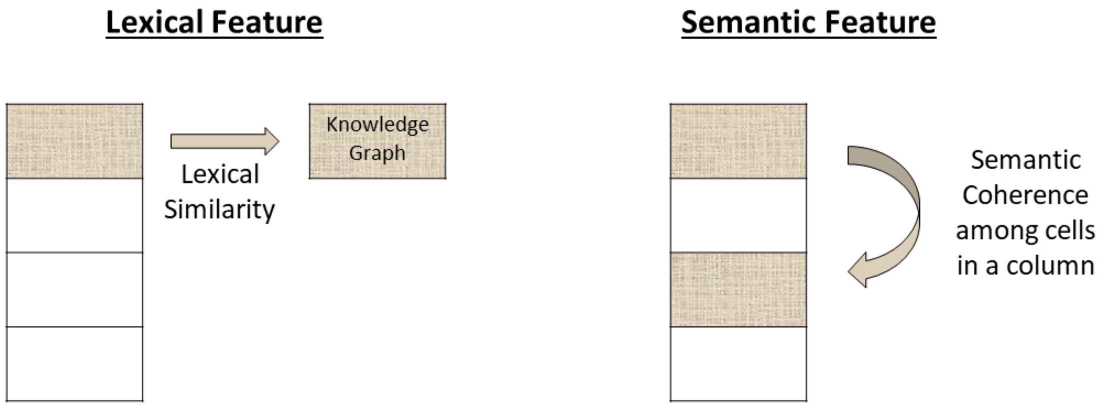


Figure 5.3: Lexical and Semantic Features

> In the Heuristic Linear Combination approach the calculation of the final score for each candidates is made through a simple multiplication between Levenshtein Distance and TFIDF. As in the first approach, also in this case the top-scoring candidate is selected.

For this project, we applied the Heuristic Linear Combination approach as it revealed to be more precise and complete than the first one (results shew F1-score and Precision equals to 0.826 and 0.852 respectively). Moreover, this methodology helped us in addressing the problems of *coverage* and *selectivity* of the features. Sometimes, in fact, features generated during the process were not equally informative. A feature has good coverage if for every cell there is a candidate for which it has value 1; while a feature is selective if few candidates possess it.
In conclusion, for the utilization of this approach, Qnodes candidates where mapped into their respective DBpedia URIs. The Python code which performs this task is shown in the attachments.

5.3.2 Column Type Annotation

After the candidates have been selected, we entered in the second phase of the project, which was the most important one as its main objective was to identify the class that best characterized the input column, i.e. address the Class Identification problem. As a starting point, we tried a first approach for the Class Identification and investigated its performance. This model is called **Column Type Annotation** and has been described the first time by Thawani et al. [123]. Figure 5.4 shows how the CTA proceeds in the identification of the class following a **top-down approach** in the hierarchy of DBpedia Knowledge Graph. The algorithm, indeed, starts from the highest level in the ontology graph, i.e. level 0 represented by `owl:Thing`, and calculates the percentage of candidates taken as input that belong to different classes

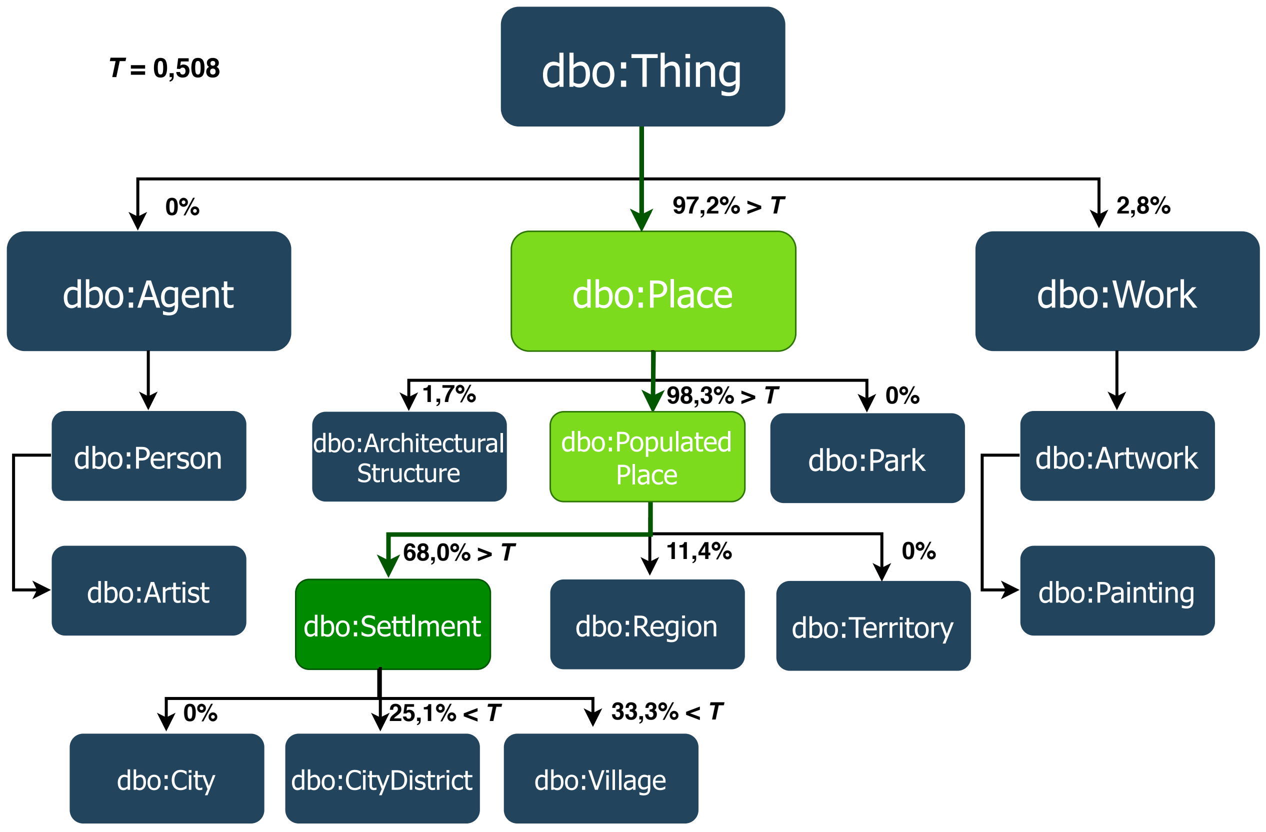


Figure 5.4: Overview Of CTA Approach [123]

along the graph. The CTA builds a query to retrieve the class for each candidate. In Figure 5.4 it is possible to see that, for example, on level 1 the percentage of cells under `dbo:Diploma`, `dbo:Agent` and `dbo:Place` classes are respectively 0%, 2.8% and 97.2%. As long as the percentage obtained is greater than a certain threshold T, the algorithm records that class in its search path and descends to another level. When the percentage is not above T anymore, the research stops and the algorithm gives as output the path below the root. In the example reported in Figure 5.4, the final output would be: `dbo:Place` $\rightarrow$ `dbo:PopulatedPlace` $\rightarrow$ `dbo:Settlement` and, therefore, `Settlement` would be the final class of the column selected and the category to which the dataset belongs.

In a first-round trial, an assessment of the CTA model performance has been pursued in order to understand whether the threshold percentage T, set equal to 0.508 for the algorithm to perform better[123], affects the accuracy of the Class Identification. Specifically, we applied two different values: 0.508 as initial value and 0.1. The choice of not increasing the value has been taken in accordance with the goal of this initial experiment, i.e. obtaining a response by the algorithm. Indeed, increasing the threshold would have negatively affected the possibility to have valuable results as a higher percentage of cells belonging to the same class would have been imposed to the model in producing an output. From the results obtained, the initial threshold ($T = 0.508$) has been selected as the most precise and effective one, as it is possible to see from Table 6.5.

A second evaluation has been pursued to select the best methodology for strings comparison, considering two different perspectives: on one hand, the type of measures selected for semantic labeling; on another hand, the way to combine it with TFIDF similarity. In particular, **four different scenarios** have been investigated:

- Multiplication between Levenshtein Distance and TFIDF, considered like *AS-IS* configuration of the model
- Weighted Average between Levenshtein Distance and TFIDF, considering the former less important ($w_1 = 0.4$) than the latter ($w_2 = 0.6$)
- Multiplication between Jaro-Winkler Similarity and TFIDF
- Weighted Average between Jaro-Winkler Similarity and TFIDF, considering the former less important ($w_1 = 0.4$) than the latter ($w_2 = 0.6$)

TF-IDF has been considered more relevant in the calculation of the Weighted Average with Levenshtein Distance since there were some words in the entities that should have been taken into account more by the model compared to others. In particular, we report an example of TFIDF approach in Candidate Selection phase in Tables 5.4 and 5.5.

LABEL	CANDIDATES	
Withney High School	dbr: Withney High School dbo: Agent dbo: Organisation dbo: EducationalInstitution	dbr: High School dbo: Agent dbo: Organisation dbo: EducationalInstitution
LA Elementary School	dbr: LA Elementary School dbo: Agent dbo: Organisation dbo: EducationalInstitution	dbr: Elementary School dbo: Agent dbo: Organisation dbo: EducationalInstitution
Lutheran Church School	dbr: Lutheran Church dbo: Agent dbo: Organisation dbo: ReligiousOrganisation	dbr: Lutheran Church School dbo: Agent dbo: Organisation dbo: EducationalInstitution
Sepulveda Elementary School	dbr: Sepulveda Elemetary School dbo: Agent dbo: Organisation dbo: EducationalInstitution	

Table 5.4: Semantic Feature Extraction

Semantic Feature	dbo:Agent	dbo:Organization	dbo:Educational Institution	dbo:Religious Organisation	
Term Frequency (TF)	**4**	**4**	**3**	**1**	
Document Frequency (DF)	**7**	**7**	**6**	**1**	
Inverse Document Frequency (IDF)	**0,05**	**0,18**	**0,18**	**0,65**	
TFIDF	**0,25**	**0,72**	**0,72**	**0,65**	**TFIDF Score**
***v_1* (dbr:Lutheran Church)**	**1**	**1**	**0**	**1**	**1,62**
***v_2* (dbr:Lutheran Church School)**	**1**	**1**	**1**	**0**	**1,69**

Table 5.5: TFIDF Approach To Candidate Selection

While the former shows the extraction of semantic features in the candidates from a dataset about schools, the latter reports the calculation of the TFIDF according to the approach described in Section 4.1. The candidates are generated with the hierarchy of classes they belong to and for each class the TFIDF is computed. As it is possible to see, thanks to the computation of the final TFIDF, the best score is the one identifying the answer which is actually correct, as Lutheran Church candidate is wrong and misleading; thus, having a lower TFIDF Score.
The results of the different approaches are shown in Table 6.6 from Chapter 6. As it is possible to see, CTA model worked best in its initial condition. In the attachments, we report the CTA model programming code.

5.4 Final Phase: the development of the Final Model

Once investigated the performance of Column Type Annotation and Column Entity Annotation models, another much simpler but effective approach has been evaluated to achieve better results in Class Identification issue. This last approach considers the Frequency of appearance of every class queried in the column of candidates. While the previously analyzed CTA method can be defined as a top-down approach, the **Frequency Model** consists in a **bottom-up solution** as it retrieves the class for each cell according to the lowest level in the DBpedia classes graph and picks the most frequent one. In Figure 5.5 we show an example of the application of the Frequency model, where the class selected is `dbo:Museum` with 47% of rows in the input column belonging to that class.
The evaluation of this approach, also considering the four different configurations introduced in Section 5.3.2, has brought positive results to the project, highlighting a

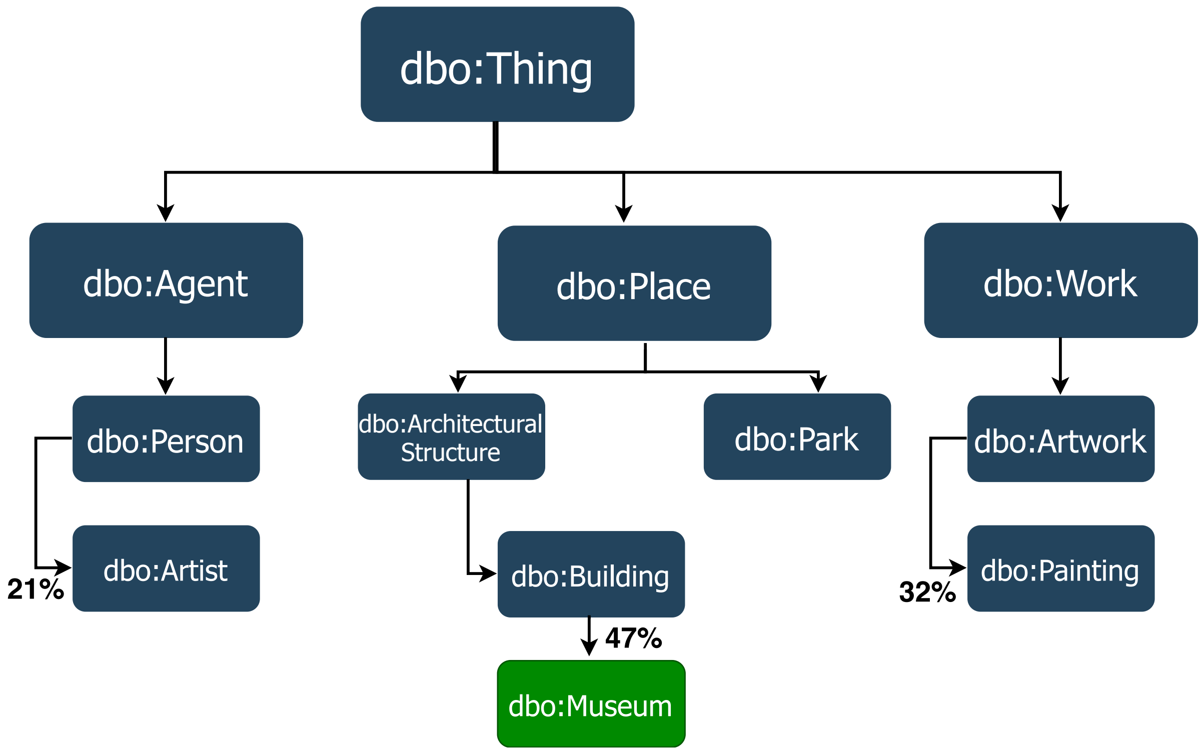


Figure 5.5: Overview Of Frequency Approach

good response in the case of Jaro-Winkler similarity. However, for the improvement of Class Identification performance the Levenshtein distance was preferred for semantic comparison, as elucidated in Chapter 6.

Since the results were still not satisfying for an optimized Class Identification, a final model has been designed and validated, integrating both the CTA and Frequency methodologies. We built the final design so that, when the candidates are properly selected, the model applies at first the Frequency approach and if a class is not available for the column, the algorithm tries with the CTA approach. When a class is not shown in the output, it means, in both cases, that the algorithm failed in identified the class. As it is described in Figure 5.5, if the CTA too is not able to detect the class (worst-case scenario), an alternative solution is applied, exploiting the data contained in the datasets.

Indeed, with the aim to enhance the contribution of the model in extracting information from a dataset, we tested an additional approach leveraging on the help of **OpenStreetMap Website**. In this perspective, the exploitation of the information available in OpenStreetMap about the city of Los Angeles could give back at least a hint on the content of the dataset. We included in the algorithm, in fact, the possibility to extract the location column from the dataset, quering the entities collected in OpenStreetMap and retrieve the category. This approach turned out to improve the final results by unveiling new insights for each dataset. Also in this case

the most frequent class extracted from OpenStreet Map has been considered as the one categorizing the whole dataset.
Figure 5.6 summarizes the flowchart of the final design of the model. As it is possible to notice, the model starts by selecting the column to parse. Then, it applies the Frequency model and check if there is an outcome. If the result shows no class, the algorithm applies the CTA model and parses again the column to find the class. If still an outcome is not provided, the model goes back to the input dataset, isolate the column with locations, if it exists, and exploits OpenStreetMap knowledge for extracting the class of the column. In every step, if the algorithm finds a class, it will stop.

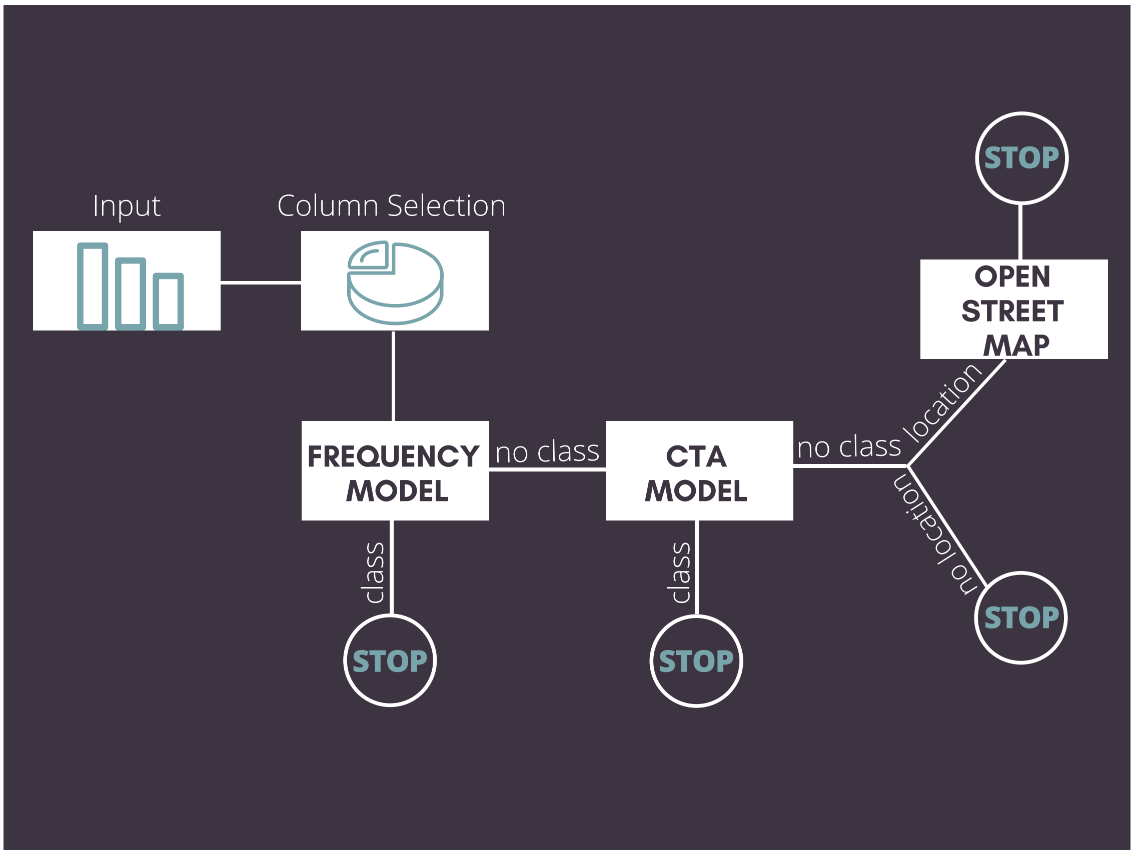


Figure 5.6: Flowchart Of The Final Model

The results of the application of this final approach are reported in Chapter 6, while the code is described in the attachments. Here we present the most important part of the final model code.

```
#wikify_column: this is a function used to implement
#all the features of the final model

 def wikify_column(self, i_df, column, case_sensitive=True,
 debug=False):
         #the algorithm collects in the list called
         #raw_labels all the input data, i.e. a column
         #of the dataset previously selected
         raw_labels = list()
         if isinstance(column, str):
             raw_labels = list(i_df[column].unique())
         elif isinstance(column, int):
             raw_labels = list(i_df.iloc[:, column].unique())


         #the algorithm creates a dataframe where the
         #input data is cleaned from Null values
         _new_i_list = []
         for label in raw_labels:
             _new_i_list.append({'label': label,
             '_clean_label': self.clean_labels(label)})
         df = pd.DataFrame(_new_i_list)

         #the algorithm starts the Candidates
         #Generation phase by running the queries and
         #collecting the candidates from them
         df['_candidates'] =
         df['_clean_label'].map(lambda x: self.run_query(x))
         df['_candidates_list'] =
         df['_candidates'].map(lambda x:
         self.create_list_from_candidate_string(x)[0])
         df['_candidates_freq'] =
         df['_candidates'].map(lambda x:
         self.create_list_from_candidate_string(x)[1])
         self.aqs = self.query_average_scores(df)
         all_qnodes = self.get_candidates_qnodes_set(df)
```

```
#candidates in DBpedia are transformed into
#their respective Wikidata Qnodes and all the
#candidats are added to the dataframe in a
#dedicated column
qnode_to_labels_dict =
self.create_qnode_to_labels_dict(list(all_qnodes))
qnode_dburi_map =
self.create_qnode_to_dburi_map(qnode_to_labels_dict)
qnode_typeof_map =
self.create_qnode_to_type_dict(qnode_to_labels_dict)


#the algorithm processes the candidates by
#applying TFIDF and Levenshtein distance
tfidf = TFIDF(qnode_to_labels_dict)
df = self.lev_similarity.add_lev_feature(df,
qnode_to_labels_dict, case_sensitive)

#the algorithm selects the best candidate for
#each row/label and creates a column with the
#most relevant candidates called 'answer'
cs = CandidateSelection(qnode_dburi_map, self.aqs,
qnode_typeof_map)
df = cs.select_high_precision_results(df)
df_high_precision = df.loc[df['answer'].notnull()]
label_lev_similarity_dict = self.create_lev_similarity_dict
(list(zip(df._clean_label, df.lev_feature)), cs)

#the algorithm translates the selected
#candidates from Wikidata Qnodes to their
#respective DBpedia URIs in order to create
#an another column named 'answer_dburi'
hp_qnodes = df_high_precision['answer'].tolist()
df['answer_Qnode'] = df['_clean_label'].map
(lambda x: tfidf_answer.get(x))
df['answer_dburi'] = df['answer_Qnode'].map
(lambda x: self.get_dburi_for_qnode(x, qnode_dburi_map))

#the algorithm takes the list of DBpedia URIs
#selected and applies the Frequency Model
cta_class = cta.process_frequency_match(df['answer_Qnode'].
tolist())

#if Frequency Model does not produce an
#output, i.e. is not able to address the
#Column Identification problem, the algorithm
#tries the CTA
if cta_class == "" or cta_class == "{}":
    cta_class = cta.process(hp_qnodes)
df['cta_class'] = cta_class.split('␣')[-1]
answer_dict = self.create_answer_dict(df)
i_df['{}_cta_class'.format(column)] =
i_df[column].map(lambda x: answer_dict[x][0])
i_df['{}_answer_Qnode'.format(column)] =
i_df[column].map(lambda x: answer_dict[x][1])
i_df['{}_answer_dburi'.format(column)] =
i_df[column].map(lambda x: answer_dict[x][2])
return i_df
```

In case CTA is not able to produce a class, the algorithm select the Location column and applies OpenStreetMap retrieval. For this last step the code is reported in the Appendix together with the entire piece of code that carries out tasks explained above.

Chapter 6

Experiments And Evaluations

In this Chapter we show the experiments undertaken for the development of the model and its validation. The first section is devoted to the **Preprocessing** phase where datasets are explored and prepared for the trials. In the second part of this Chapter we report the **Results** of the experiments, including all the assumptions made during the analysis. Finally, a last section is devoted to the **Qualitative Analysis** of the results, encompassing comments above the functioning of the model, its developments and how issues have been faced throughout the project.

6.1 Preprocessing

The Preprocessing phase has been characterized by a deep observation of the datasets in order to investigate data formats and columns features. As shown in the attachments, in fact, datasets of city of Los Angeles are of different types and contents, from the *trees* in the street, to the name of *restaurants*, or *businesses*. From this investigation, we found out that three main problems needed to be addressed before the development of the model:

1. available data needed to be cleaned from irregularities (such as symbols) or null values,

2. we needed a strategy to select the most explanatory column from a dataset,

3. we needed another strategy to isolate the column with spatial information from a dataset.

As it regards the first issue, several actions have been pursued in order to clean data, such as:

- removal of `null` values as misleading for the investigation of column class;

- removal of rows with numbers or very special symbols (e.g. $ or #);

- standardization of the capitalization in the columns in order to have information written in lowercase. This was made because having cells with uppercase strings led to inaccurate results, especially in the information retrieval phase in Wikidata.

Furthermore, we carried out a second data cleaning phase in order to address the Column Selection problem, i.e. to isolate the most useful column for the class investigation. In particular, from the observation of the datasets, we collected some criteria for a column to be excluded from the analysis. Such characteristics do not claim to be universal but have allowed us to make the right selection in the majority of the datasets available for this study (equal to 90.2%). The criteria identified for the exclusion of the "useless" columns are the following:

- Containing all equal values
- Including numbers for the majority of the rows (i.e. more than 50% of rows are integer or float type)
- Containing specific symbols for the majority of the rows (i.e. more than 50% of rows have symbols)
- Containing underlined text
- Containing e-mail addresses
- Including any information related to time (i.e. dates)

PARK NAME
Laurel Canyon Park
Westminster Park
Whitnall Park
Silver Lake Park
Peck Park
North Hollywood Park

DISTRICT #	**CONTACT**
13	(323) 663-7758
4	(323) 663-2555
7	(310) 549-4953
5	(818) 899-2200
13	(818) 291-9980
10	(818) 995-1170

Table 6.1: Example Of Useless Columns (District # and Contact) And A Useful Column (Park Name) In A Dataset About Parks In Los Angeles

Thus, after having applied these criteria, we were able to exclude the least explanatory columns from datasets and isolate the most useful one. Table 6.1 reports an example of a correct isolation of the most useful column compared to other two in a dataset about parks in Los Angeles.
Once selected the column, information is ready to be processed by the model for class evaluation. We excluded the possibility to parse more than one column of the same dataset as we focused on a simpler case study considering the difficulty of the task.

Finally, we addressed the problem of isolating the column containing location and spatial information from each dataset using the same heuristic approach for the Column Isolation problem. Indeed, we introduced one last criteria in the final model, i.e. selecting the column containing location information (such as address, city, zipcode, longitude and latitude) from each dataset. This step occurs in case there is no output given from Frequency and CTA models, and the algorithm tries to access OpenStreetMap knowledge, as explained in Section 5.4.
Once made these evaluations on data, we focused on the main task of the research, which was identifying the class for each dataset.

6.2 Results

In this section we present the main results that show the improvements of the model in pursuing the objective of the project. At first, we report an explanation about the metrics used in this evaluation phase. In order to measure the validity of the results, we leveraged on the most immediate evaluation metric: the **Accuracy**, which is defined as follows [128]:

Definition 13. *Accuracy measures the ability of the method being evaluated to make the correct prediction.*

It is a frequently-used criterion to assess the performance of a model and it can also be interpreted in different ways according to the field of application. In our case, we applied the following formula for measuring accuracy:

$$Accuracy_{\mathrm{i}} = \frac{n_{\mathrm{i}}}{D} \tag{6.1}$$

Where:

- n was the number of correct classes detected by the methodology i applied
- D was the total number of datasets evaluated ($D = 41$).

In addition, during the experimentation phase, we established a distinction between **permissive** and **restrictive** accuracy. This distinction was driven by the fact that in the model output there could be more or less precise answers. As it is possible to see from Table 6.2, in fact, the class detected by the model can be exactly the expected class (highlighted in green) or one of its supersub classes (highlighted in yellow). Thus, we introduced the permissive accuracy to include in the computation of n_i the number of answers which are more approximate (yellow cells) together with the less approximate ones (green cells), and the **restrictive** accuracy which considered only the latter.
In this research, we considered the accuracy to vary between 0 (the model was completely unable to pursue the task) and 1 (the model was completely able to pursue the task).

DATASET	MODEL OUTPUT	EXPECTED CLASS
Affordable Housing	{}	Building
Animals	Mammal	Animal
Apparel	Organisation	Organisation
LA Florists	Work	Organisation

Table 6.2: Example Of Results

Moreover, as secondary metric to assess the correctness of the model, we leveraged on the **F1-score**. The computation of this metric has been dictated by the need to strengthen the evaluation of the model and to use a more comprehensive measure. To define F1-score we need to introduce the **Confusion Matrix** [128], which is reported in Table 6.3.

	Predicted as Positive	**Predicted as Negative**
Are positive	TP	FN
Are negative	FP	TN

Table 6.3: Confusion Matrix [128]

Confusion Matrix shows the number of **True Positives (TP)**, **True Negatives (TN)**, **False Positives (FP)**, and **False Negatives (FN)** instances that are used to compare the predictions of method with the ground truth. TP represents the number of instances that were predicted as positive and were indeed positive, whereas FP is the number of instances incorrectly predicted as positive. TN and FN have a corresponding meaning for the negative class. We applied these definitions to describe **Precision** and **Recall** in our experiments:

$$Precision = \frac{TP}{TP + FN} \qquad Recall = \frac{TP}{TP + FN} \tag{6.2}$$

While the former is the fraction of relevant instances among all the retrieved instances, the latter represents the fraction of relevant instances retrieved over the total amount of relevant instances [33]. Thus, F1-score, indeed, encompasses the concepts of both **Precision** and **Recall** and it is theoretically defined with this formula [128]:

$$F1 = \beta \cdot \frac{precision \cdot recall}{precision + recall} \tag{6.3}$$

Where β is a positive real number, $\beta \in [0, \infty]$.
When F1-score is equal to 0 all the predictions are incorrect, when it is equal to 1, instead, all the predictions are correct.
For this research study we considered β to be equal to 2 following the standard value, while for F1-score we applied an adaptation, considering several assumptions:

- the TP were counted as the number of both less and more approximate answers, i.e. both green and yellow highlighted cells have been considered, as a more selective metric has been embodied by the restrictive accuracy;

- for FP we counted the number of incorrect classes, i.e. the red highlighted;
- for FN we considered the number of empty cells, thus the situation when the model is not able to detect a class.

Once defined the metrics and the assumptions taken during the experiments, we now report the results from the First Phase of the project, where we investigated a starting approach taking into account only Wikidata. Table 6.4 shows the results on 15 datsets from the city of Los Angeles.

D = 15	**First Phase Model**
TP	0
FP	4
FN	11
PRECISION	0
RECALL	0
F1 SCORE	0
ACCURACY	0

Table 6.4: First Phase Model Results

As it is possible to notice, no positive results have been obtained on the datasets tested in this phase. The classes identified by the model, in fact, appeared to be meaningless and suggested the inapplicability of Wikidata ontology mapping, as too broad and generic for the purpose of the project. Thus, we applied several improvements:

- introduce DBpedia for Candidates Generation and Class Identification tasks to enlarge the available knowledge base,
- improve the Candidates Generation phase considering other metrics for entity linkage,
- optimize the Class Identification phase with more sophisticated approaches.

The application of these improvements has meant to include both the CEA and the CTA models described by [123] in our approach as reported in Chapter 5. In addition, we increased the number of datasets (indicates as D) for the trials from 15 to 41. Since the scope of this research study consisted in categorizing datsets, we focused on testing the performance of CTA model.
In the first experiment we evaluated the CTA model comparing the actual acceptance threshold **T** for rows to belong to the same class with a lower one. We used as similarity measures the Levenshtein Distance and the TFIDF, and their multiplication for candidates selection. As Table 6.5 shows, in fact, the model was run on the 41 datasets considering the as-is configuration ($T = 0.508$) and the modification

(T = 0,1). On one hand, an increase in F1-score is observed (0,5456) with a lower threshold, probably caused by a betterment in the Recall; on another hand, the as-is CTA configuration resulted to be more accurate in general. Indeed, even though there is no difference in the value of permissive accuracy (0,36585), restrictive accuracy is higher when the threshold is higher (0,31707). What emerged from this results was that probably the CTA with a lower threshold was more capable in identifying classes, but not as accurate as the CTA considering a higher threshold.
Thus, the configuration considered more appropriate for the research was the Column Type Annotation with a threshold equal to 0,508. Moreover, from the experiments ran so far, it was possible also to see that the objective was not trivial and obtaining meaningful classes claimed the development of a very robust and smart model.

D = 41	**CTA (0,1 th)**	**CTA (0,508 th)**
TP	15	15
FP	22	9
FN	3	17
PRECISION	0,40541	0,625
RECALL	0,83333	0,46875
F1 SCORE	**0,54546**	0,53571
RESTRICTIVE ACCURACY	0,29268	**0,31707**
PERMISSIVE ACCURACY	0,36585	**0,36585**

Table 6.5: CTA Model first trial performance

Once established the best threshold for the CTA functioning, an evaluation was made on the different metrics to apply during Candidates Generation phase. Considering the four different configurations described in Section 5.3.2, we evaluated the performances of CTA related to them.

D = 41	**Lev-Mult**	**Lev-WA**	**Jar-Mult**	**Jar-WA**
TP	15	14	5	6
FP	9	9	3	3
FN	17	18	33	32
PRECISION	0,625	0,60869	0,625	0,66667
RECALL	0,46875	0,4375	0,13158	0,15789
F1 SCORE	0,53571	0,50909	0,21739	0,25532
RESTRICTIVE ACCURACY	**0,31707**	0,29268	0,09756	0,14634
PERMISSIVE ACCURACY	**0,36585**	0,34146	0,12195	0,12195

Table 6.6: CTA Model second trial performance: four configurations in comparison

Table 6.6 shows the results of the experiments ran on 41 datasets according to the distance measures chosen. The first two columns are devoted to the Levenshtein Distance multiplied, at first, with the TFIDF and, then, combined with it through the Weighted Average. The last two columns are dedicated to the application of the Jaro-Winkler similarity.

From these results it is clear that the CTA model performed better if associated with the Levenshtein Distance multiplied by the TFIDF for the Candidates Generation phase. Using Jaro-Winker, in fact, has decreased the performance significantly, restrictive and permissive accuracy have worsen by 69,2% and 66,7% respectively. As this model seemed not enough accurate, the Frequency approach was applied. For this set of experiments we combined CEA model for Candidates Generation phase with the Frequency Model and, then, evaluated the latter performances for Class Identification. Table 6.7 reports the results obtained, considering the four configurations previously defined (see Section 5.3.2).
As it is possible to notice, the results obtained with the Frequency model, much simpler but effective, are better than those from previous approaches. In particular, the Frequency model, combined with the CEA for the evaluation of candidates, seemed to be able to handle all the different configurations of metrics and still achieving significant results (for permissive accuracy all the values are above 50%). Thus, from these results it appeared clear that the Frequency approach is more effective than the CTA.

D = 41	**Lev-Mult**	**Lev-WA**	**Jar-Mult**	**Jar-WA**
TP	22	21	23	22
FP	14	19	17	17
FN	5	1	1	2
PRECISION	0,61111	0,525	0,575	0,56410
RECALL	0,81481	0,95455	0,95833	0,91667
F1 SCORE	0,69841	0,67742	0,71875	0,69841
RESTRICTIVE ACCURACY	0,39024	0,36585	0,4146	0,41463
PERMISSIVE ACCURACY	**0,53658**	0,512195	**0,56098**	**0,53658**

Table 6.7: Frequency Model first trial performance: four configurations in comparison

With the aim to enhance the performance, we evaluated the integration between the CTA and the Frequency Model to develop a unique model, as described in Chapter 5. During this experimentation, the question arose spontaneously: which combination of measure is the best one to apply in this case of an integrated model? Table 6.7, in fact, suggests to pick the Jaro-Winkler multiplied by the TFIDF, while Table 6.6 results recommend using the Levenshtein. Although Frequency model obtained better results than the CTA in the overall, we considered inconvenient to use Jaro-Winkler as the CTA model worsened significantly and we wanted to avoid this situation as integrating it with the Frequency model would have turned out to be useless. Thus, the choice made regarding the configuration suggested the Levenshtein Distance as Similarity Measure to apply during Candidates Generation phase. The average accuracy between CTA and Frequency models, in fact, was higher using Levenshtein than in the case of Jaro-Winkler (average computed as the sum of the permissive accuracy of Frequency and CTA divided by 2 in each case).

We carried out another experiment for testing the integrated model in 41 datasets. Table 6.8 shows that the development of the integrated model between CTA and Frequency provides good results in terms of Accuracy but no improvements regarding the F1-score. The reason of this phenomenon may have consisted in a higher number of corrected classes identified by the integrated model, but still not enough to balance those cases where the class was not even identified. In addition, restrictive accuracy has slightly improved with the integrated model, even though the final value reached was not significantly high (0,41463). Thus, this was a sign that there were classes which still appeared to be hard to be identified by the model. Overall, the values of Precision and Recall seemed to be more balanced than the previous cases, where Recall tended to be much higher. An explanation for this phenomenon might stem from the fact that previous approaches were able to categorize less datasets.

D = 41	**CTA + Frequency**	**Final Model**
TP	23	33
FP	7	8
FN	11	0
PRECISION	0,76667	0,80487
RECALL	0,67647	1
F1 SCORE	0,69841	**0,89189**
RESTRICTIVE ACCURACY	0,41463	**0,56098**
PERMISSIVE ACCURACY	**0,56098**	**0,80487**

Table 6.8: Final Model performance with and without OpenStreetMap support

The final turning point of the research project is also shown in Table 6.8, where, together with the results of Frequency model integrated with CTA, we reported the outcome of the trials on the final model which encompassed the retrieval of the class from OpenStreetMap. Being many datasets characterized by columns with location, in fact, we decided to exploit that information in favour of the project objectives. Thus, when the model is not able to detect the class from the column selected, it goes back to the dataset, isolate the column with locations data and search for that in OpenStreetMap, as explained in Chapter 5.

As it is possible to notice, this final improvement has optimized the results and enhanced the correctness of the model. At first, we have no cases where a class is not identified in the 41 datasets used for the experiments and that is why the recall is equal to 1. Moreover, both F1-score and permissive accuracy appear to have higher values, both over 80%. The exploitation of OpenStreetMap knowledge has benefited the performance of the model as the restrictive accuracy has reached significant value too (0,56098%). Thus, we considered this final results as satisfactory for the research study.

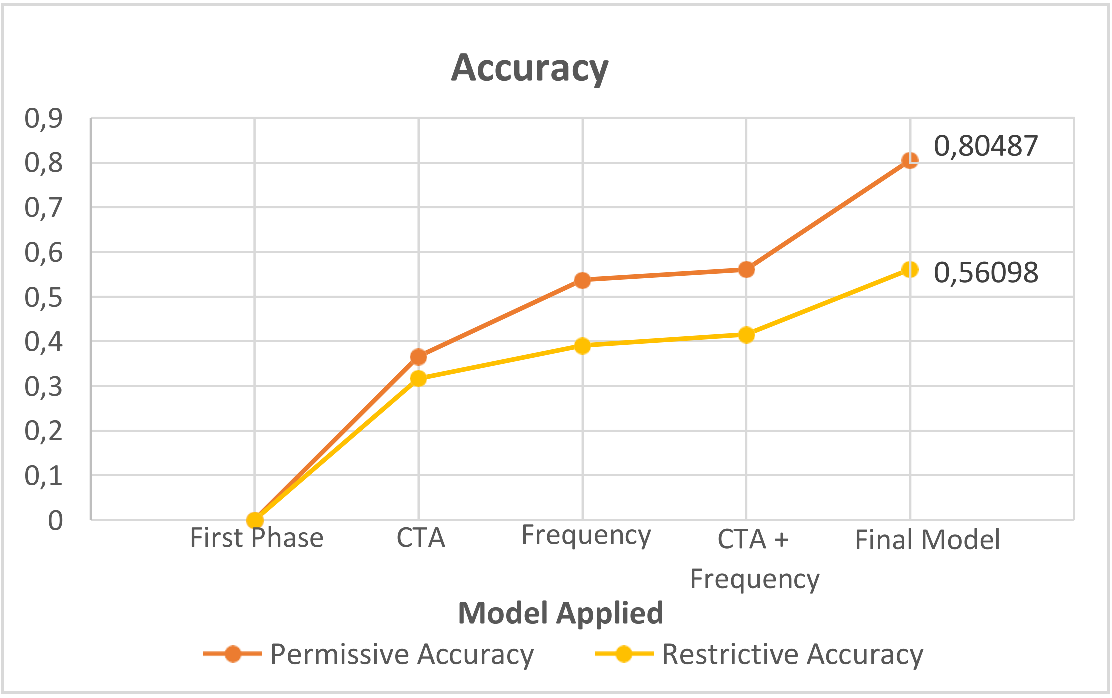


Figure 6.1: Accuracy Improvements

Finally, in Figure 6.1 we report the evolution of both permissive and restrictive accuracy throughout the experiments. The most important improvements adopted in the model have been the inclusion of the Frequency model and the OpenStreetMap approach, as the slope of the line (both for permissive and restrictive accuracy) is more pronounced.
In conclusion, there is still room for developments, especially in cases with very sophisticated datasets or other situations that are evaluated in the next section. In the attachments we inclyde the table with the final model results.

6.3 Qualitative Analysis

The identification of the class of a dataset is not an easy task, especially when the variables involved are many. In this case, the challenge has started from the Column Selection phase, has gone through the Candidates Generation step and has culminated in the Class Identification for the dataset. At first, the results obtained were not satisfying as the model could not produce useful results for the majority of the datasets. However, with the inclusion of several improvements and integrating all the components in a unique approach, the final result has reached acceptable and significant results. In addition, it is possible to justify the performance of the model since its purpose is not to propose a competitive classifier, but to lay the foundations for the exploitation of Open Linked Data and information available in the Web,

as a starting point for further implementation. Many other projects identified in the literature have shown interest in related fields, leveraging on other approaches, such as machine learning classification techniques [109]. However, in that case, for example, a significant number of labeled datasets were used to train the algorithm, while in our case the approach to datasets is totally blind.
Moreover, the amount of information needed for this project was not huge in terms of rows in datasets. Experiments, in fact, have been ran on 100 rows datasets as we noticed no difference in parsing a large number of rows, i.e. the model was able to give an output even with fewer rows. This is very important for the objectives of the research as the time to access information is usually limited, especially in contexts like enterprises. Indeed, the analysis of such small amount of rows takes less than 50 seconds, which means that, spent the time, the user could have a response about datasets, which sometimes are populated by millions of rows. In conclusion, this represents a key advantage in the effective identification of the category of a dataset.
On another hand, the model has been developed to analyse a single column and it is not able to take as input multiple columns. Working on a single column can be limiting, especially in those cases where the column selected is useless, i.e. it does not show meaningful entities. Indeed, this can represent a weak point as the methodology proposed is heuristic-based. Nevertheless, the objectives of the research project were more focused in categorizing the datasets in an efficient and effective way, thus more time has been spent on that. The datasets available, in fact, show various formats which appear to be exemplifying of a real situation and facing the column isolation with a heuristic seemed to be the most suitable solution for the project.
Another interesting point to notice is that the choice of DBpedia ontology seemed to be suitable for the reasearch goal. DBpedia ontology mapping, in fact, turned out to be neither too restricted nor too generic. The way categories and hierarchies are expressed proved to be useful and, above all, human understandable. Indeed, this is a fundamental aspect to take into account as the model has been developed with the aim to send it to a final user.
Furthermore, the selection of another measure for calculating the distances in the candidates selection phase has raised a weak point of the CTA model. Its performance, in fact, have deteriorated for more than the 60% between one trial and the other. One of the possible reason why that happened stems from the fact that the CTA model applies a threshold which has to be respected and this may have prevented in the identification of the class. Combining this model with the Frequency one has been the best way to integrate a top-down with a bottom-up approach, even though the result model by itself is not always correct in the extraction of the column in the dataset, that is why the support of OpenStreetMap has been introduced.

Chapter 7

Conclusions And Future Work

This conclusive Chapter has the aim to disclose significant evaluations on the research project and on future possible developments. In this paper we presented a novel approach to categorize datasets on the basis of a single column analysis and of the exploitation of Linked Open Data and Web ontologies. The methodology was simple, performance-driven, and embraces high interpretability. Several heuristic appraoches were developed to emulate the human discovery process when exposed to brand new information and to make the algorithm smart. In particular, the final model achieved a good reasoning capacity by integrating different appraoches on the basis of the data taken as input. Extensive experiments have shown that the method improved with the integration of the Frequency and with the consideration of the location, while qualitative analysis indicated that the model selection of the column ability was still limited.
One of the strength point of the final model was the capability to exploit knowledge bases and identify classes with an accuracy significantly high. In order to further improve the performances, several actions can be considered:

- according to [92] it is useful to normalize the TFIDF, as TF is often very high, while in this approach TFIDF was not normalized,
- as described in literature, many others approaches for record linkage could be applied and test, such as *classification-based* [55] or *ruled-base* [72],
- include other ontologies in the extraction of classes. A limit to the model, in fact, could be the absence of a class in the reference knowledge base. For example, DBpedia has no information about anything related with money,
- define a more sophisticated approach for Frequency model which is able to take into account also second or third most frequent class.

In addition, a more advanced methodology for column selection may be the subject of further research, especially the ones related with machine learning techniques. Examples of these opportunities are described by [109]. The fact that the model parses one column at a time, in fact, represents a limitation as additional columns can disclose important insights about the whole datasets. It would be also very interesting to study a way to integrate results obtained by different columns. In this perspective, the CPA model introduced by [123] could be supportive. Moreover, for the selection of location columns, the approach proposed by [80] could be considered for automatically identifying column with spatial information. All these methodologies could also affect positively the processing time, enhancing the value of the model.

Bibliography

[1] Dbpedia, global and unified access to knowledge.

[2] Jaro distance definition.

[3] Openstreetmap: powers map data on thousands of web sites, mobile apps, and hardware devices.

[4] Wikidata, the free knowledge base.

[5] Wikipedia, the free onlile encyclopedia.

[6] Xml example, w3schools. *https://www.w3schools.com/xml/xml$_e$xamples.asp.*

[7] Garrett Alley. The importance of data integration to today's business, 2018.

[8] Yigal Arens, Chun-Nan Hsu, and Craig A Knoblock. Query processing in the sims information mediator. *Advanced Planning Technology*, 32:78–93, 1996.

[9] Paolo Atzeni and Riccardo Torlone. Data and metadata management. *Semantic Web Information Management: A Model-Based Perspective*, 12 2010.

[10] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *The semantic web*, pages 722–735. Springer, 2007.

[11] Giovanni Bartolomeo and Tatiana Kovacikova. *Identification and Management of Distributed Data: NGN, Content-centric Networks and the Web*. CRC Press, 2013.

[12] Florian Bauer and Martin Kaltenböck. Linked open data: The essentials. 2011.

[13] Sean Bechhofer, Frank Van Harmelen, Jim Hendler, Ian Horrocks, Deborah L McGuinness, Peter F Patel-Schneider, Lynn Andrea Stein, et al. Owl web ontology language reference. *W3C recommendation*, 10(02), 2004.

[14] Christian Becker and Christian Bizer. Dbpedia mobile: A location-enabled linked data browser. *Ldow*, 369:2008, 2008.

[15] Dave Beckett. Rdf semantics. *https://www.w3.org/TR/rdf-mt/*, 2004.

[16] Dave Beckett. Rdf/xml syntax specification (revised). 2004.

[17] Zohra Bellahsene, Angela Bonifati, Fabien Duchateau, and Yannis Velegrakis. On evaluating schema matching and mapping. In *Schema matching and mapping*, pages 253–291. Springer, 2011.

[18] Tim Berners-Lee. Linked data, 2006.

[19] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific american*, 284(5):34–43, 2001.

[20] Dina Bitton, David J DeWitt, and Carolyn Turbyfill. Benchmarking database systems-a systematic approach. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1983.

[21] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data: The story so far. In *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227. IGI Global, 2011.

[22] Jens Bleiholder and Felix Naumann. Data fusion. *ACM computing surveys (CSUR)*, 41(1):1–41, 2009.

[23] A Blumauer. From taxonomies over ontologies to knowledge graphs–the semantic puzzle, 2014.

[24] Piero Andrea Bonatti, Stefan Decker, Axel Polleres, and Valentina Presutti. Knowledge graphs: new directions for knowledge representation on the semantic web (dagstuhl seminar 18371). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[25] Valerie Bonstrom, Annika Hinze, and Heinz Schweppe. Storing rdf as a graph. IEEE, 2003.

[26] Willem Nico Borst. Construction of engineering ontologies for knowledge sharing and reuse. 1999.

[27] Robert F Boruch and Ellen A Donnelly. Privacy of individuals in social research: Confidentiality. 2015.

[28] Jon Bosak and Tim Bray. Xml and the second-generation web. *Scientific American*, 280(5):89–93, 1999.

[29] Janez Brank, Gregor Leban, and Marko Grobelnik. Semantic annotation of documents based on wikipedia concepts. volume 42, 2018.

[30] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, Franois Yergeau, et al. Extensible markup language (xml) 1.0, 2000.

[31] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[32] Dan Brickley and Ramanathan Guha. Rdf vocabulary description language 1.0: Rdf schema. *W3C Recommendation*, 01 2004.

[33] Michael Buckland and Fredric Gey. The relationship between recall and precision. *Journal of the American society for information science*, 45(1):12–19, 1994.

[34] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Riccardo Rosati, and Gestionale Antonio Ruberti. Ontology-based data access and integration., 2018.

[35] Capgemini. Big fast data: The rise of insight-driven business. 2012.

[36] Roo J.D. Carroll, J.J. Owl web ontology language test cases. technical report. *W3C Recommendation*, 2004.

[37] Yueh-Hsuan Chiang, AnHai Doan, and Jeffrey F Naughton. Modeling entity evolution for temporal record matching. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1175–1186, 2014.

[38] P. Christensson. Techterms, 2013.

[39] Douglas Crockford. Introducing json. *Available: json. org*, 2009.

[40] Isabel F Cruz and Huiyong Xiao. The role of ontologies in data integration. *Engineering intelligent systems for electrical engineering and communications*, 13(4):245, 2005.

[41] Andrea De Mauro, Marco Greco, and Michele Grimaldi. A formal definition of big data based on its essential features. *Library Review*, 2016.

[42] Roberto De Virgilio. Smart rdf data storage in graph databases. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 872–881. IEEE, 2017.

[43] Stefan Decker, Sergey Melnik, Frank Van Harmelen, Dieter Fensel, Michel Klein, Jeen Broekstra, Michael Erdmann, and Ian Horrocks. The semantic web: The roles of xml and rdf. *IEEE Internet computing*, 4(5):63–73, 2000.

[44] A Doan, A Halevy, and Z Ives. Ontologies and knowledge representation. *Principles of Data Integration, Boston, Morgan Kaufmann*, pages 325–344, 2012.

[45] Xin Luna Dong and Divesh Srivastava. *Big data integration.* Morgan & Claypool Publishers, 2015.

[46] Dun and Bradstreet. Report: The past, present, and future of data. Technical report, 2019.

[47] Erik Duval, Wayne Hodgins, Stuart Sutton, and Stuart L Weibel. Metadata principles and practicalities. *D-lib Magazine*, 8(4):1082–9873, 2002.

[48] BV Elasticsearch. Lucene's practical scoring function, 2012.

[49] Basil Ell. *User Interfaces to the Web of Data based on Natural Language Generation.* KIT Scientific Publishing, 2017.

[50] Fredo Erxleben, Michael Günther, Markus Krötzsch, Julian Mendez, and Denny Vrandečić. Introducing wikidata to the linked data web. In *International Semantic Web Conference*, pages 50–65. Springer, 2014.

[51] Wenfei Fan, Hong Gao, Xibei Jia, Jianzhong Li, and Shuai Ma. Dynamic constraints for record matching. *The VLDB Journal*, 20(4):495–520, 2011.

[52] Michael Färber, Frederic Bartscherer, Carsten Menne, and Achim Rettinger. Linked data quality of dbpedia, freebase, opencyc, wikidata, and yago. *Semantic Web*, 9(1):77–129, 2018.

[53] Michael Färber, Basil Ell, Carsten Menne, and Achim Rettinger. A comparative survey of dbpedia, freebase, opencyc, wikidata, and yago. *Semantic Web Journal*, 1(1):1–5, 2015.

[54] Lee Feigenbaum. Sparql by example: the cheat sheet. *Cambridge semantics*, 2008.

[55] Ivan P Fellegi and Alan B Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.

[56] Eric Miller Frank Manola. Rdf primer. *https://www.w3.org/TR/rdf-primer/*, 2004.

[57] A. Freitas. Coping with data variety in the big data era: The semantic computing approach. 2014.

[58] Michael Gilleland et al. Levenshtein distance, in three flavors. *Merriam Park Software: http://www. merriampark. com/ld. htm*, 2009.

[59] Fausto Giunchiglia, Biswanath Dutta, and Vincenzo Maltese. Faceted lightweight ontologies. In *Conceptual modeling: foundations and applications*, pages 36–51. Springer, 2009.

[60] Fausto Giunchiglia, Feroz Farazi, Letizia Tanca, and Roberto Virgilio. The semantic web languages. *Semantic Web Information Management: A Model-Based Perspective*, 2010.

[61] Google. Candidates generation overview.

[62] Clinton Gormley and Zachary Tong. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine.* " O'Reilly Media, Inc.", 2015.

[63] Thomas R Gruber et al. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–221, 1993.

[64] Tom Gruber. Ontology, 2018.

[65] Anja Gruenheid, Xin Luna Dong, and Divesh Srivastava. Incremental record linkage. *Proceedings of the VLDB Endowment*, 7(9):697–708, 2014.

[66] Nicola Guarino and Pierdaniele Giaretta. Ontologies and knowledge bases. *Towards very large knowledge bases*, pages 1–2, 1995.

[67] Wendy Arianne Günther, Mohammad H Rezazade Mehrizi, Marleen Huysman, and Frans Feldberg. Debating big data: A literature review on realizing value from big data. *The Journal of Strategic Information Systems*, 26(3):191–209, 2017.

[68] Alon Halevy. Why your data won't mix. *Queue*, 3(8):50–58, 2005.

[69] Oktie Hassanzadeh, Fei Chiang, Hyun Chul Lee, and Renée J Miller. Framework for evaluating clustering algorithms in duplicate detection. *Proceedings of the VLDB Endowment*, 2(1):1282–1293, 2009.

[70] Tom Heath and Christian Bizer. Linked data: Evolving the web into a global data space. *Synthesis lectures on the semantic web: theory and technology*, 1(1):1–136, 2011.

[71] Jeff Heflin. Owl web ontology language-use cases and requirements. *W3C Recommendation*, 10(10):1–12, 2004.

[72] Mauricio A Hernández and Salvatore J Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data mining and knowledge discovery*, 2(1):9–37, 1998.

[73] Hootsuite. Digital 2019 reports. *https://datareportal.com/.*

[74] Andrea Wei-Ching Huang. A preliminary study on wikipedia, dbpedia and wikidata. *http://andrea-index.blogspot.tw/2015/06/wikipedia-dbpedia-wikidata.html*, 2015.

[75] Ali Ismayilov, Dimitris Kontokostas, Sören Auer, Jens Lehmann, Sebastian Hellmann, et al. Wikidata through the eyes of dbpedia. *Semantic Web*, 9(4):493–503, 2018.

[76] Dave Beckett Jan Grant. Rdf test cases. *https://www.w3.org/TR/rdf-testcases/*, 2004.

[77] Bin Jiang, Jian Pei, Yufei Tao, and Xuemin Lin. Clustering uncertain data based on probability distribution similarity. *IEEE Transactions on Knowledge and Data Engineering*, 25(4):751–763, 2011.

[78] Anitha Kannan, Inmar E Givoni, Rakesh Agrawal, and Ariel Fuxman. Matching unstructured product offers to structured product specifications. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 404–412, 2011.

[79] Avita Katal, Mohammad Wazid, and Rayan H Goudar. *Big data: issues, challenges, tools and good practices. 2013 Sixth international conference on contemporary computing (IC3).*

[80] Craig A Knoblock, Aparna R Joshi, Abhishek Megotia, Minh Pham, and Chelsea Ursaner. Automatic spatio-temporal indexing to integrate and analyze the data of an organization. In *Proceedings of the 3rd ACM SIGSPATIAL Workshop on Smart Cities and Urban Analytics*, pages 1–8, 2017.

[81] Lars Kolb, Andreas Thor, and Erhard Rahm. Multi-pass sorted neighborhood blocking with mapreduce. *Computer Science-Research and Development*, 27(1):45–63, 2012.

[82] Hanna Köpcke, Andreas Thor, and Erhard Rahm. Evaluation of entity resolution approaches on real-world match problems. *Proceedings of the VLDB Endowment*, 3(1-2):484–493, 2010.

[83] Rafal Kuc and Marek Rogozinski. *Elasticsearch server*. Packt Publishing Ltd, 2013.

[84] A. Rettinger. L. Zhang. Final ontological word-sense-disambiguation prototype. *Deliverable D3.2.3, xLike Project*, 2014.

[85] Dongwon Lee and Wesley W Chu. Comparative analysis of six xml schema languages. *ACM Sigmod Record*, 29(3):76–87, 2000.

[86] Roger Lee and Haeng-Kon Kim. *Computer and Information Science*, volume 493. Springer, 2013.

[87] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, et al. Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[88] Pei Li, Xin Luna Dong, Andrea Maurino, and Divesh Srivastava. Linking temporal records. *Proceedings of the VLDB Endowment*, 4(11):956–967, 2011.

[89] Zhou Yang Luo. A library implementing different string similarity and distance measures.

[90] Yu Ma, Xiaodong Gu, and Yuanyuan Wang. Histogram similarity measure using variable bin size distance. *Computer Vision and Image Understanding*, 114(8):981–989, 2010.

[91] Valerio Maggio. Semantic python: Mastering linked data with python.

[92] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval.* Cambridge university press, 2008.

[93] Benoit Marchal. *XML by Example.* Que Publishing, 2002.

[94] John P. McCrae. The link open data cloud.

[95] Deborah L McGuinness, Frank Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.

[96] Eduardo Mena, Arantza Illarramendi, Vipul Kashyap, and Amit P Sheth. Observer: An approach for query processing in global information systems based on interoperation across pre-existing ontologies. *Distributed and parallel Databases*, 8(2):223–271, 2000.

[97] Pablo N Mendes, Max Jakob, and Christian Bizer. Dbpedia: A multilingual cross-domain knowledge base. In *LREC*, pages 1813–1817. Citeseer, 2012.

[98] Microsoft. *Distribution List Creation and Usage Web Service Protocol.* 2018.

[99] MIT. Mit sloan management review. Technical report, 2011.

[100] Mohamed Morsey, Jens Lehmann, Sören Auer, Claus Stadler, and Sebastian Hellmann. Dbpedia and the live extraction of structured data from wikipedia. *Program*, 2012.

[101] Sravanthi Naraharisetti. Customize relevance with elastic search. *Medium Website*, April,2018.

[102] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33, 2015.

[103] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. Using of jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists*, volume 1, pages 380–384, 2013.

[104] Marek Obitko. Translations between ontologies in multi-agent systems. *Unpublished Ph. D. dissertation, Faculty of Electrical Engineering, Czech Technical University in Prague*, 2007.

[105] George Papadakis, Jonathan Svirsky, Avigdor Gal, and Themis Palpanas. Comparative analysis of approximate blocking techniques for entity resolution. *Proceedings of the VLDB Endowment*, 9(9):684–695, 2016.

[106] Peter. F. Patel-Schneider. Owl web ontology language semantics and abstract syntax, w3c recommendation. *http://www.w3.org/TR/2004/REC-owl-semantics-20040210/*, 2004.

[107] Heiko Paulheim. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic Web*, 8:489–508, 2016.

[108] Li Shiuau Peh. Domain-specific semantic class disambiguation using wordnet. In *Fifth Workshop on Very Large Corpora*, 1997.

[109] Minh Pham, Suresh Alse, Craig A Knoblock, and Pedro Szekely. Semantic labeling: a domain-independent approach. In *International Semantic Web Conference*, pages 446–462. Springer, 2016.

[110] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. In *Journal on data semantics X*, pages 133–173. Springer, 2008.

[111] Jay Pujara, Hui Miao, Lise Getoor, and William Cohen. Knowledge graph identification. In *International Semantic Web Conference*, pages 542–557. Springer, 2013.

[112] Erhard Rahm and Philip A Bernstein. A survey of approaches to automatic schema matching. *the VLDB Journal*, 10(4):334–350, 2001.

[113] Markus Lanthaler Richard Cyganiak, David Wood. Rdf 1.1 concepts and abstract syntax. *https://www.w3.org/TR/rdf11-concepts/*, 2014.

[114] Riccardo Rosati. The upper layers of the semantic web. Corso di Laurea Magistrale in Ingegneria Informatica Sapienza Università di Roma, 2014.

[115] Philip Russom et al. Big data analytics. *TDWI best practices report, fourth quarter*, 19(4):1–34, 2011.

[116] Sunita Sarawagi and Anuradha Bhamidipaty. Interactive deduplication using active learning. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 269–278, 2002.

[117] Monica Scannapieco, Ilya Figotin, Elisa Bertino, and Ahmed K Elmagarmid. Privacy preserving schema and data matching. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 653–664, 2007.

[118] Seth Schaafsma. Big data: The 6 vs you need to look at for important insights. *https://www.motivaction.nl/en/news/blog/big-data-the-6-vs-you-need-to-look-at-for-important-insights.*

[119] Amit Singhal. Introducing the knowledge graph: things, not strings. *Official google blog*, 16, 2012.

[120] OpenLink Software. Openlink virtuoso universal server documentation.

[121] Andy Seaborne Steve Harris, Garlik. Sparql 1.1 query language.

[122] Techopedia. What does tokenization mean?

[123] Avijit Thawani, Minda Hu, Erdong Hu, Husain Zafar, Naren Teja Divvala, Amandeep Singh, Ehsan Qasemi, Pedro Szekely, and Jay Pujara. Entity linking to knowledge graphs to infer column types and properties. 2019.

[124] Henry S Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. Xml schema part 1: structures second edition. *W3C recommendation*, 39, 2004.

[125] R. Fielding Tim Berners-Lee and L. Masinter. *RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax.*

[126] C.M.Sperberg-McQueen Eve Maler François Yergeau Tim Bray, Jean Paoli. Extensible markup language (xml) 1.0 (fifth edition).

[127] Jana Vembunarayanan. Tf-idf and cosine similarity. *https://janav.wordpress.com/2013/10/27/tf-idf-and-cosine-similarity/*, 2013.

[128] Carlo Vercellis. *Business intelligence: data mining and optimization for decision making.* Wiley Online Library, 2009.

[129] IBM Storage System Vincent (Yu-Cheng). How fast can a company get access to its data?

[130] Denny Vrandečić and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.

[131] Holger Wache, Thomas Voegele, Ubbo Visser, Heiner Stuckenschmidt, Gerhard Schuster, Holger Neumann, and Sebastian Hübner. Ontology-based integration of information-a survey of existing approaches. In *Ois@ ijcai*, 2001.

[132] Bonnie Lynn Webber and Nils J Nilsson. *Readings in artificial intelligence*. Morgan Kaufmann, 2014.

[133] Chris Welty, Deborah L McGuinness, and Michael K Smith. Owl web ontology language guide. *W3C recommendation, W3C (February 2004) http://www. w3. org/TR/2004/REC-owl-guide-20040210*, 2004.

[134] David Wood, Marsha Zaidman, Luke Ruth, and Michael Hausenblas. *Linked Data*. Manning Publications Co., 2014.

Appendix

.1 Programming Documentation

The project has been developed in the following GitHub directory
`https://github.com/usc-isi-i2/wikidata-wikifier`.

.1.1 Column Selection

```
import pandas as pd
import numpy as np
import re

df = pd.read_csv("")
df = pd.DataFrame(df)
col_to_erase = []
col_to_save = []

# DROP COLUMNS WITH FLOAT OR INTEGERS
df_float = df.select_dtypes(exclude=object)
df_float_col = list(df_float.columns.values)
col_to_erase.extend(df_float_col)

# DROP LOCATION COLUMNS

def is_word_in_text(word, text):
    pattern = r"(^|[^\w]){}([^\w]|\$)".format(word)
    pattern = re.compile(pattern, re.IGNORECASE)
    matches = re.search(pattern, text)
    return bool(matches)

for col in df.columns.values:
    location_list = ['Telephone', 'TELEPHONE', 'telephone',
    "CITY", "city", "City","LOCATION",
    "location", "Location", "ZIP",
    "zip", "Zip", "ADDRESS", "Address",
    "address", "DATE", "Date", "date"]
    c = col
    for loc in location_list:
        if is_word_in_text(loc, col) is True:
            col_to_erase.append(c)

# DROP COLUMNS WITH DATE OR TIME
for col in df.columns:
    if df[col].dtype == 'object':
        try:
            df[col] = pd.to_datetime(df[col])
        except ValueError:
            continue
    if not df[col].dtype == 'object':
        col_to_erase.append(col)
    else:
        break


# DROP COLUMNS WITH HYPERLINKS, EMAILS OR GEO COORDINATES
for col in df.columns:
    patterns = ["http","@", "   "]
    for j in range(len(df)):
        for pattern in patterns:
            try:
                if re.search(pattern,str(df[col][j])):
                    col_to_erase.append(col)
            except ValueError:
                break

col_to_erase = list(dict.fromkeys(col_to_erase))
initial = list(df.columns.values)
almost_final_col = [x for x in initial if x not in col_to_erase]
col_to_save = list(dict.fromkeys(col_to_save))
col_to_erase = list(dict.fromkeys(col_to_erase))

#OBTAIN THE FINAL COLUMN
initial = list(df.columns.values)
almost_final_col = [x for x in initial if x not in col_to_erase]
```

```
final_col = [x for x in almost_final_col if x in col_to_save]
df_final = df[final_col]
th = int(len(df_final)/2)
df_final.dropna(axis=1, thresh = th)
```

.1.2 CTA Model

```
#COMPUTE THE LEVENSHTEIN DISTANCE
#from similarity.normalized_levenshtein import NormalizedLevenshtein
from similarity.jarowinkler import JaroWinkler
label_fields = ['wd_labels', 'wd_aliases', 'person_abbr',
'db_anchor_texts']

class AddLevenshteinSimilarity(object):
    def __init__(self):
        self.lev = JaroWinkler()

    def lev_mapper(self, label, wikidata_json, case_sensitive=True):
        qnode = wikidata_json['id']
        max_lev = -1
        max_label = None
        label_lower = label.lower()
        for label_field in label_fields:
            _labels = wikidata_json.get(label_field)
            for l in _labels:
                if case_sensitive:
                    lev_similarity = self.lev.similarity(label, l)
                else:
                    lev_similarity = self.lev.similarity(label_lower,
                    l.lower())
                if lev_similarity > max_lev:
                    max_lev = lev_similarity
                    max_label = l
        if 'db_labels' in wikidata_json and
        'en' in wikidata_json['db_labels']:
            en_labels = wikidata_json['db_labels']['en']
            if not isinstance(en_labels, list):
                en_labels = [en_labels]
            for l in en_labels:
                if case_sensitive:
                    lev_similarity = self.lev.similarity(label, l)
                else:
                    lev_similarity = self.lev.similarity
                    (label_lower, l.lower())
                if lev_similarity > max_lev:
                    max_lev = lev_similarity
                    max_label = l

        anchors = wikidata_json.get('db_anchor_texts', [])
        if not isinstance(anchors, list):
            anchors = [anchors]
        for anchor in anchors:
            if case_sensitive:
                lev_similarity = self.lev.similarity(label, anchor)
            else:
                lev_similarity = self.lev.similarity
                (label_lower, anchor.lower())
            if lev_similarity > max_lev:
                max_lev = lev_similarity
                max_label = anchor

        if max_label is not None:
            return (qnode, max_lev, max_label)
        return (None, None, None)

    @staticmethod
    def candidates_from_candidate_string(candidate_string):
        qnode_set = set()
        db_group_set = set()
        if candidate_string is not None
        and isinstance(candidate_string, str):
            c_tuples = candidate_string.split('@')
            for c_tuple in c_tuples:
                if c_tuple is not None
                and isinstance(c_tuple, str) and c_tuple != 'nan':
                    try:
                        vals = c_tuple.split(':')
                        if vals[0] != '5':
                            qnode_set.add(vals[1])
                        else:
                            _ = vals[1].split('$')
                            db_group_set.add(_[0])
                        if len(_) > 1:
                            qnode_set.add(_[1])
                except:
                    print(c_tuple)

        return list(qnode_set), list(db_group_set)

    def compute_lev(self, label_cand_str,
    wikidata_index_dict, case_sensitive):
        clean_label = label_cand_str[0]
        candidate_string = label_cand_str[1]
        qnodes, db_groups =
        self.candidates_from_candidate_string(candidate_string)
        wikidata_jsons = [wikidata_index_dict[qnode] for qnode
        in qnodes if qnode in wikidata_index_dict]

        results = []
        for wikidata_json in wikidata_jsons:
            r = self.lev_mapper(clean_label, wikidata_json,
          case_sensitive=case_sensitive)
            if r[0] is not None:
                results.append('{}:{}'.format(r[0], r[1]))

        return '@'.join(results)

    def add_lev_feature(self, df, wikidata_index_dict, case_sensitive):
        df['_dummy'] = list(zip(df._clean_label, df._candidates))
        df['lev_feature'] = df['_dummy'].map(
            lambda x: self.compute_lev(x, wikidata_index_dict,
            case_sensitive))
        return df

#COMPUTE TFIDF
class TFIDF(object):
    def __init__(self, qnodes_dict):
        self.qnodes_dict = qnodes_dict
        self.properties_classes_map =
        self.create_all_properties_classes_map()

    @staticmethod
    def get_properties_classes_for_qnode(qnode_dict):
        properties_classes_set = set()
        wd_properties = qnode_dict.get('wd_properties', [])
        properties_classes_set.update(wd_properties)

        wd_prop_vals = qnode_dict.get('wd_prop_vals', [])
        for wd_prop_val in wd_prop_vals:
            _ = wd_prop_val.split('#')
            _p = _[0]
            _v = _[1]
            if _p == 'P31':
                properties_classes_set.add(_v)
        dbpedia_instances = qnode_dict.get('db_instance_types', [])
        for dbpedia_instance in dbpedia_instances:
            if dbpedia_instance.
            startswith('http://dbpedia.org/ontology/'):
                properties_classes_set.add(dbpedia_instance)
        return properties_classes_set

    def create_all_properties_classes_map(self):
        properties_classes_set = set()
        for qnode in self.qnodes_dict:
            v = self.qnodes_dict[qnode]
            properties_classes_set.update
            (self.get_properties_classes_for_qnode(v))
        return {p: idx for idx, p in enumerate(properties_classes_set)}

    def create_feature_vector_dict(self, label_candidates_tuples):
        # creates input for tfidf computation
        feature_vector_dict = {}
        _p_c_len = len(self.properties_classes_map)
        for label_candidates_tuple in label_candidates_tuples:
            label = label_candidates_tuple[0]
            candidates = label_candidates_tuple[1]

            feature_vector_dict[label] = {}
            for candidate in candidates:
                feature_vector = [0] * _p_c_len
                if candidate in self.qnodes_dict:
                    prop_class_list = self.
                    get_properties_classes_for_qnode
                    (self.qnodes_dict[candidate])
                    for _p_c in prop_class_list:
                        if _p_c in self.properties_classes_map:
                            feature_vector
                            [self.properties_classes_map[_p_c]] = 1
                feature_vector_dict[label][candidate] = feature_vector
        return feature_vector_dict

    def compute_tfidf(self, label_candidates_tuples,
    label_lev_similarity_dict, high_precision_candidates=None, ):
        candidates = self.create_feature_vector_dict
        (label_candidates_tuples)
        feature_count = len(self.properties_classes_map)
        tfidf_values = [{'tf': 0, 'df': 0, 'idf': 0} for _
        in range(feature_count)]
        corpus_num = sum(len(qs) for _, qs in candidates.items())

        # compute tf
        for f_idx in range(feature_count):
            for e in candidates:
                for q, v in candidates[e].items():
                    if high_precision_candidates:
                        if q == high_precision_candidates.get(e):
                            if v[f_idx] == 1:
                                tfidf_values[f_idx]['tf'] += 1
                    else:
                        tfidf_values[f_idx]['tf'] += 1

        # compute df
        for f_idx in range(feature_count):
            for e in candidates:
                for q, v in candidates[e].items():
                    if v[f_idx] == 1:
                        tfidf_values[f_idx]['df'] += 1

        # compute idf
        for f_idx in range(len(tfidf_values)):
            if tfidf_values[f_idx]['df'] == 0:
                tfidf_values[f_idx]['idf'] = 0
            else:
                tfidf_values[f_idx]['idf'] =
                math.log(float(corpus_num) / tfidf_values[f_idx]
                ['df'], 10)

        # compute final score
        ret = {}
        for e in candidates:
            ret[e] = {}
            for q, v in candidates[e].items():
                ret[e][q] = 0
                for f_idx in range(feature_count):
                    ret[e][q] += tfidf_values[f_idx]['tf'] *
                    tfidf_values[f_idx]['idf'] * v[f_idx]
        lev_ret = {}
        for label in ret:
            _dict = ret[label]
            _lev_similarity_dict = label_lev_similarity_dict.get(label,
            None)

            if _lev_similarity_dict:
                for qnode, tfidf_score in _dict.items():
                    _lev_score =
                    float(_lev_similarity_dict.get(qnode, 0.0))
                   _dict[qnode] = _lev_score *tfidf_score
            lev_ret[label] = _dict

        answer_dict = {}
        for label in lev_ret:
            _dict = lev_ret[label]
            max_v = -1
            max_q = None
            for k, v in _dict.items():
                if v > max_v:
                    max_v = v
                    max_q = k
            answer_dict[label] = max_q
        return answer_dict
```

.1.3 Column Type Annotation

```
import json
from SPARQLWrapper import SPARQLWrapper, JSON
```

```
#URILIST: lists of the answers, thus the DPbedia uris
#for each instance/row
#CLASSURI: class to which the single instance uri belongs
#CLASSLIST: list of super classes --> the ones directly
#correlated with Thing


class CTA(object):
    def __init__(self, dburi_typeof):
        self.dburi_typeof = dburi_typeof
        function that identifies the class from the uri of each
        instance and check if it belong to superclasses

        self.super_classes =
        pd.read_csv('wikifier/caches/SuperClasses.csv',
                                        header=None)[0].tolist()
        self.db_classes =
        json.load(open('wikifier/caches/DBClasses.json'))
        self.db_classes_closure =
        json.load(open('wikifier/caches/DBClassesClosure.json'))
        self.sparqldb = SPARQLWrapper("http://dbpedia.org/sparql")

    def is_instance_of(self, uri):
        self.sparqldb.setQuery(

            select distinct ?x where
            {{ <{}> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
            ?x .
            ?x <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
            <http://www.w3.org/2002/07/owl#Class> .}}

            .format(uri))
        self.sparqldb.setReturnFormat(JSON)
        results = self.sparqldb.query().convert()
        instances = set()
        for result in results["results"]["bindings"]:
            dbp = result['x']['value']
            instances.add(dbp)
        return instances

def evaluate_class_closure(self, urilist, classuri):
        matches = 0
        classuriclosure = set()
        if classuri in self.db_classes_closure:
            classuriclosure = set(self.db_classes_closure[classuri])
        validuri = []
        for uri in urilist:
            if uri in self.dburi_typeof:
                instances = self.dburi_typeof[uri]
            else:
                instances = self.is_instance_of(uri)
                self.dburi_typeof[uri] = list(instances)

            for instance in instances:
                if instance in classuriclosure:
                    validuri.append(uri)
                    matches += 1
                    break

        score = matches / len(urilist)
        return [score, validuri]

def find_class(self, urilist, classlist, currentans, ans_list,
threshold):
        ans_list.append(currentans)
        if len(classlist) == 0:
            return #ritorna None

        max_score = -1
        max_validuri = []
        max_class = ''
        for superclass in classlist:
            [score, validuri] = self.evaluate_class_closure(urilist,
            superclass)
            if max_score < score:
                max_score = score
                max_validuri = validuri
                max_class = superclass

        if max_score >= threshold:
            subclasses = self.db_classes[max_class]
            self.find_class(max_validuri, subclasses,
            max_class, ans_list, threshold)
            return
  def process(self, df, threshold=0.508, class_level=0):

        urilist = df['answer'].tolist()
        if len(urilist) == 0:
            return ""
        ans_list = []
        self.find_class(urilist, self.super_classes, '',
        ans_list, threshold)
        ans_list = ans_list[1:]
        if len(ans_list) <= class_level:
            return ""
        return "␣".join(ans_list)

#CANDIDATE GENERATION AND SELECTION
class CandidateSelection(object):
    def __init__(self, qnode_to_dburi_map, aqs, qnode_typeof_map):
        self.qnode_to_dburi_map = qnode_to_dburi_map
        self.aqs = aqs
        self.qnode_typeof_map = qnode_typeof_map
        self.country_dict =
        json.load(open('wikifier/caches/country_qnode_dict.json'))
        self.states_dict =
        json.load(open('wikifier/caches/us_states_qnode_dict.json'))
        self.person_classes =
        json.load(open('wikifier/caches/dbpedia_person_classes.json'))
        self.places_classes =
        json.load(open('wikifier/caches/dbpedia_place_classes.json'))
        self.normalized_lev = NormalizedLevenshtein()
        db_classes = json.load(open('wikifier/caches/db_classes.json'))
        self.db_classes_parent =
        self.create_class_parent_hierarchy(db_classes)

    def sort_lev_features(self, lev_feature_s, threshold=0.9):
        qnode_dict = {}
        try:
            qnodes_lev = sorted([z.split(':') for z
            in lev_feature_s.split('@')], key=itemgetter(1),
            reverse=True)

            # mette in ordine i dati
            qnodes_lev = list(filter(lambda x:
            float(x[1]) >= threshold, qnodes_lev))
            for idx in range(len(qnodes_lev)):
                qnode = qnodes_lev[idx][0]
                levscore = qnodes_lev[idx][1]
                qnode_dict[qnode] = levscore
        except:
            return {}

        return qnode_dict

    def top_ranked(self, df_tuple):
        chosencand = None
        maxscore = 0
        sorted_candidates = df_tuple[0]
        lev_candidates = df_tuple[1]
        candidates_with_one = list()

        for candidate in lev_candidates:
            if lev_candidates[candidate] == '1.0':
                candidates_with_one.append(candidate)

        if len(candidates_with_one) == 1:
            return candidates_with_one[0], 1

        if len(candidates_with_one) > 1:
            return None, 2

        for candidate in sorted_candidates:
            if candidate in lev_candidates:
                score = float(lev_candidates[candidate]) +
                sorted_candidates[candidate][1]
                if score > maxscore:
                    chosencand = candidate
                    maxscore = score

        return (chosencand, 3) if chosencand is not None
        else (None, 4)

    def sort_qnodes(self, qnode_string):
        sorted_qnodes = list()
        if qnode_string is not None and isinstance(qnode_string, str)
        and
        qnode_string.strip() != '':
            q_scores = qnode_string.split('@')
            for q_score in q_scores:
                if q_score != 'nan':
                    q_score_split = q_score.split(':')
                    score = q_score_split[2]
                    id = str(q_score_split[0])
                    qnode = q_score_split[1]

                    if id == '42' or id == '10':
                        sorted_qnodes.append((qnode,
                        float(score) ** 0.25))
                    else:
                        sorted_qnodes.append(
                            (qnode,
                            self.aqs[id]['lambda'] * (float(score) /
                            self.aqs[id]['avg'])))

        results = list()
        for sorted_qnode in sorted_qnodes:
            db_uri = self.qnode_to_dburi_map.get(sorted_qnode[0], None)
        results.append((sorted_qnode[0], db_uri, sorted_qnode[1]))

        results.sort(key=itemgetter(2), reverse=True)
        _dict = {}
        for result in results:
            _dict[result[0]] = [result[1], result[2]]

        return _dict

    def sort_qnodes_2(self, qnode_string):
        result_dict = dict()
        result_dict['wd'] = list()
        result_dict['es'] = list()
        if qnode_string is not None and
        isinstance(qnode_string, str) and qnode_string.strip() != '':
            q_scores = qnode_string.split('@')
            for q_score in q_scores:
                if q_score != 'nan':
                    q_score_split = q_score.split(':')
                    score = q_score_split[2]
                    id = str(q_score_split[0])
                    qnode = q_score_split[1]

                    if id == '42' or id == '10':
                        if id == '42':
                            result_dict['wd'].append((qnode,
                            float(score) ** 0.25))
                        else:
                            result_dict['es'].append((qnode,
                            float(score) ** 0.25))
                    else:
                        result_dict['es'].append((qnode,
                        self.aqs[id]['lambda'] (float(score) /
                        self.aqs[id]['avg'])))

        wd_list = result_dict['wd']
        _new_wd_list = list()
        for q_s in wd_list:
            qnode = q_s[0]
            _new_wd_list.append((qnode, float(q_s[1])))
        result_dict['wd'] = sorted(_new_wd_list, key=itemgetter(1),
        reverse=True)
        es_list = result_dict['es']
        _new_es_list = list()
        for q_s in es_list:
            qnode = q_s[0]
            _new_es_list.append((qnode, float(q_s[1])))

        result_dict['es'] = sorted(_new_es_list, key=itemgetter(1),
        reverse=True)
        return result_dict

    def lev_similarity(self, label, db_uris, threshold=0.9):
        max_score = -1.0
        best_match = None
        for db_uri in db_uris:
            dburi_part = db_uri.split('/')[-1]
            dburi_part = dburi_part.replace('_', '␣')
```

```
                similarity = self.normalized_lev.similarity(label,
                dburi_part)
                if similarity > max_score:
                    max_score = similarity
                    best_match = db_uri
            if max_score >= threshold:
                return best_match
            return None

        def choose_candidate_with_cta(self, df_tuple):
            sorted_lev = df_tuple[0]
            sorted_candidates = df_tuple[1]
            cta_class = df_tuple[2]
            answer = df_tuple[3]
            label = df_tuple[4]
            wiki_candidates = sorted_candidates['wd']
            es_candidates = sorted_candidates['es']
            sorted_lev_tuples = list()
            for k, v in sorted_lev.items():
                sorted_lev_tuples.append((k, v))
            if not isinstance(answer, float) and answer is not None
            and answer.strip() != '' and answer != 'nan':
                return answer, 'high_p', 'uniq'
            if cta_class is None or cta_class.strip() == '' or
            cta_class in self.places_classes:
                if label in self.states_dict:
                    return self.states_dict[label], 'state_dict', 'uniq'
                if label in self.country_dict:
                    return self.country_dict[label], 'country_dict', 'uniq'
                lev_cands = sorted(sorted_lev_tuples, key=itemgetter(1),
                reverse=True)
                lev_cands_groups = itertools.groupby(lev_cands,
                itemgetter(1))
                for k, v in lev_cands_groups:

                    _l_s = [(x[0], x[1]) for x in v]
                    _l = [x[0] for x in _l_s]
                    lev_sim = self.lev_similarity(label, _l)
                    if lev_sim:
                        return lev_sim, 'label_lev_uri_noclass',
                        json.dumps(_l_s)
                    for wiki_c in wiki_candidates:
                        if wiki_c[0] in _l:
                        return wiki_c[0], 'wiki_noclass', json.dumps(_l_s)
                    for es_c in es_candidates:
                        if es_c[0] in _l:
                        return es_c[0], 'es_noclass', json.dumps(_l_s)

                return None, 'novaliddburi', 'nojoy'

            cta_class_cands = list()

            for qnode, score in sorted_lev.items():
                qnode_class_list = self.qnode_typeof_map.get(qnode)

                if qnode_class_list:
                    if cta_class in qnode_class_list:
                        cta_class_cands.append((qnode, score, label))

            if len(cta_class_cands) == 1:
                return cta_class_cands[0][0], 'unambiguous␣lev', 'uniq'
            if len(cta_class_cands) > 1:
                cta_class_cands = sorted(cta_class_cands,
                key=itemgetter(1), reverse=True)
                cta_cands_groups = itertools.groupby(cta_class_cands,
                itemgetter(1))
                for k, v in cta_cands_groups:
                    if cta_class in self.person_classes and False:
                        return None, 'person', 'ambiguous'
                    else:
                        _l_s = [(x[0], x[1]) for x in v]
                    _l = [x[0] for x in _l_s]

                    lev_sim = self.lev_similarity(label, _l)

                    for wiki_c in wiki_candidates:
                        if wiki_c[0] in _l:
                            return wiki_c[0], 'wiki', json.dumps(_l_s)
                    for es_c in es_candidates:
                        if es_c[0] in _l:
                            return es_c[0], 'es', json.dumps(_l_s)
                    if lev_sim:
                            return lev_sim, 'label_lev_uri', json.dumps(_l_s)
                    if len(cta_class_cands) == 0:
                        return self.choose_candidate_with_cta(
                            (sorted_lev, sorted_candidates,
                            self.db_classes_parent.get(cta_class), answer, label))

                    return None, 33, 'empty'

                @staticmethod
                def create_class_parent_hierarchy(db_classes):
                    dbclasses_parent = dict()
                    for db_class, children in db_classes.items():
                        for child in children:
                            dbclasses_parent[child] = db_class
                    return dbclasses_parent

                def select_high_precision_results(self, df):
                    df['sorted_lev'] =
                    df['lev_feature'].map(lambda x: self.sort_lev_features(x))
                    df['sorted_qnodes'] =
                    df['_candidates'].map(lambda x: self.sort_qnodes(x))
                    df['_dummy_2'] = list(zip(df.sorted_qnodes, df.sorted_lev))
                    df['top_ranked'] =
                    df['_dummy_2'].map(lambda x: self.top_ranked(x))
                    df['answer'] = df['top_ranked'].map(lambda x: x[0])
                    df['high_confidence'] = df['top_ranked'].map(lambda x: x[1])
                    return df

                def select_candidates_hard(self, df):
                    df['sorted_lev_2'] = df['lev_feature'].map(lambda x:
                    self.sort_lev_features(x, threshold=0.3))
                    df['sorted_qnodes_2'] =
                    df['_candidates'].map(lambda x: self.sort_qnodes_2(x))
                    df['_dummy_3'] = list(zip(df.sorted_lev_2, df.sorted_qnodes_2,
                    df.cta_class, df.answer, df._clean_label))
                    df['answer2'] =
                    df['_dummy_3'].map(lambda x: self.choose_candidate_with_cta(x))
                    df['answer_Qnode'] = df['answer2'].map(lambda x: x[0])
                    df['final_confidence'] = df['answer2'].map(lambda x: x[1])
                    df['db_classes'] =
                    df['answer_Qnode'].map(lambda x: self.qnode_typeof_map.get(x))
                    df['lev_group'] = df['answer2'].map(lambda x: x[2])
                    return df

            #PROCESS CANDIDATES AND OBTAIN THE CLASS FOR THE CANDIDATES SELECTED
            class Wikifier(object):

                def __init__(self):
                    config = json.load(open('wikifier/config.json'))
                    self.asciiiiii = set(string.printable)
                    self.es_url = config['es_url']
                    self.es_index = config['es_index']
                    self.dbpedia_label_index = config['dbpedia_labels_index']
                    # average query score
                    self.aqs = None
                    self.es_doc = config['es_doc']
                    self.es_search_url = '{}/{}/{}/_search'.format(self.es_url,
                    self.es_index, self.es_doc)
                    self.dbpedia_label_search_url = '{}/{}/{}/_search'
                    .format(self.es_url, self.dbpedia_label_index, self.es_doc)
                    self.names_es_index = config['names_es_index']
                    self.names_es_doc = config['names_es_doc']
                    self.names_es_search_url = '{}/{}/{}/_search'
                    .format(self.es_url, self.names_es_index, self.names_es_doc)
                    self.wikidata_dbpedia_joined_index =
                    config['wikidata_dbpedia_joined_index']
                    self.wikidata_dbpedia_joined_doc =
                    config['wikidata_dbpedia_joined_doc']
                    self.wiki_dbpedia_joined_search_url = '{}/{}/{}/_search'
                    .format(self.es_url, self.wikidata_dbpedia_joined_index,
                    self.wikidata_dbpedia_joined_doc)
                    self.query_1 =
                    json.load(open('wikifier/queries/wiki_query_1.json'))
                    self.query_2 =
                    json.load(open('wikifier/queries/wiki_query_2.json'))
                    self.query_more_like_this =
                    json.load(open('wikifier/queries/wiki_query_more_like_this.json'))
                    self.query_dbpedia_labels =
                    json.load(open('wikifier/queries/wiki_query_dbpedia_labels.json'))
                    self.seen_labels = dict()
                    self.lev_similarity = AddLevenshteinSimilarity()
                @staticmethod
                def clean_labels(label):
                    if not isinstance(label, str):
                        return label
                    clean_label = ftfy.fix_encoding(label)
                    clean_label = ftfy.fix_text(clean_label)
                    clean_label = re.sub(r'[[].*[]]', ' ', clean_label)
                    clean_label = re.sub(r'[(].*[)]', ' ', clean_label)
                    clean_label = clean_label.replace('\\', ' ')
                    clean_label = clean_label.replace('/', ' ')
                    clean_label = clean_label.replace("\"", ' ')
                    clean_label = clean_label.replace("!", ' ')
                    clean_label = clean_label.replace("-->", '')
                    clean_label = clean_label.replace(">", ' ')
                    clean_label = clean_label.replace("<", ' ')
                    clean_label = clean_label.replace("-", ' ')
                    clean_label = clean_label.replace("^", ' ')
                    clean_label = clean_label.replace(":", ' ')
                    clean_label = clean_label.replace("+", ' ')
                    return clean_label

                @staticmethod
                def create_query(t_query, search_term):
                    t_query['query']['bool']['should'][0]['query_string']['query']
                    = search_term
                    t_query['query']['bool']['should'][1]['multi_match']['query']
                    = search_term
                    if len(search_term.split(' ')) == 1:
                        t_query['size'] = 100
                    else:
                        t_query['size'] = 20
                    return t_query

                def search_es_names(self, name):
                    query = {
                        "query": {
                            "function_score": {
                                "query": {
                                    "bool": {
                                        "must": [
                                            {
                                                "multi_match": {
                                                    "query": name,
                                                    "type": "phrase",
                                                    "fields": [
                                                        "abbr_name"
                                                    ],
                                                    "slop": 2
                                                }
                                            }
                                        ]}
                                }
                            }
                        },
                        "size": 50
                    }
                    response = requests.post(self.names_es_search_url, json=query)
                    if response.status_code == 200:
                        hits = response.json()['hits']['hits']
                        results = [
                            '{}:{}:{}'.format('3', x['_id'], x['_score'])
                            for x
                            in
                            hits]
                        return results if len(results) > 0 else None
                    else:
                        print(response.text)
                    return None
                def search_es_second(self, search_term):
                    search_term_tokens = search_term.split(' ')
                    query_type = "phrase"
                    slop = 0
                    if len(search_term_tokens) <= 3:
                        slop = 2
                    if len(search_term_tokens) > 3:
                        query_type = "phrase"
                        slop = 10
                    query = self.query_2
                    query['query']['function_score']['query']['bool']
                    ['must'][0]['multi_match']['query'] = search_term
                    query['query']['function_score']['query']['bool']
                    ['must'][0]['multi_match']['type'] = query_type
                    query['query']['function_score']['query']['bool']
                    ['must'][0]['multi_match']['slop'] = slop

                    # return the top matched QNode using ES
                    response = self.search_es(query, query_id='2')
                    if response is not None:
                        return response
                    elif len(search_term_tokens) > 3:
                        for i in range(0, -4, -1):
                            t_search_term = ' '.join(search_term_tokens[:i])
            query['query']['function_score']['query']['bool']
                            ['must'][0]['multi_match']['query'] = t_search_term
                            response = self.search_es(query, query_id='2')
                            if response is not None:
                                return response
                            else:
                                continue
                    return None

                def search_es_more_like_this(self, search_term, query_id='4'):
                    search_term_tokens = search_term.split(' ')
                    min_term_freq = 1
                    tokens_length = len(search_term_tokens)
                    max_query_terms = tokens_length
                    minimum_should_match = tokens_length - 1
                    query = self.query_more_like_this
                    query['query']['more_like_this']['like'] = search_term
                    query['query']['more_like_this']['min_term_freq'] =
                    min_term_freq
                    query['query']['more_like_this']['max_query_terms'] =
                    max_query_terms

                    for i in range(minimum_should_match, 0, -1):
                        query['query']['more_like_this']['minimum_should_match'] = i
                        response = self.search_es(query, query_id=query_id)
                        if response is not None:
                            return response
                        else:
                            continue
                    return None
```

```
def run_query(self, search_term):
    print(search_term)
    try:
        response = list()
        t_query = self.create_query(self.query_1, search_term)
        response_1 = self.search_es(t_query)
        if response_1 is None:
            t_query = self.create_query(self.query_1, search_term
            .lower())
            response_1 = self.search_es(t_query)
        if response_1 is None:
            search_ascii = ''.join(filter(lambda x:
            x in self.asciiiiii, search_term))
            t_query = self.create_query(self.query_1, search_ascii)
            response_1 = self.search_es(t_query)
        if response_1 is not None:
        response.extend(response_1)
        # Relaxed query
        response_2 = self.search_es_second(search_term)
        if response_2 is not None:
            response.extend(response_2)

        # Query on abbreviated names
        response_3 = self.search_es_names(search_term)
        if response_3 is not None:
            response.extend(response_3)
        response_4 = self.search_es_more_like_this(search_term)
        if response_4 is not None:
            response.extend(response_4)
        return '@'.join([r for r in response if r.strip() != ''])
    except Exception as e:
        traceback.print_exc()
        raise e

def search_es(self, query, query_id='1'):
    # return the top matched QNode using ES
    response = requests.post(self.es_search_url, json=query)

    if response.status_code == 200:
        hits = response.json()['hits']['hits']
        results = [
                '{}:{}:{}'.format(query_id, x['_id'], x['_score'])
                for x in hits]
            return results if len(results) > 0 else None
        return None

    def search_es_dbpedia(self, query, query_id='5'):
        # return the top matched QNode using ES
        response = requests.post(self.dbpedia_label_search_url,
        json=query)
        if response.status_code == 200:
            hits = response.json()['hits']['hits']
            results = ['{}:{}:{}'.format(query_id,
            self.format_dbpedia_labels_index_results(x), x['_score'])
            for x in hits]
            for hit in hits:
                id = hit['_id']
                _d = hit["_source"]
                _d['id'] = id
                self.dburi_to_labels_dict[id] = _d
            return results if len(results) > 0 else None
        return None

    @staticmethod
    def format_dbpedia_labels_index_results(doc):
        _source = doc['_source']
        _id = doc['_id']
        if 'qnode' in _source:
            return '{}${}'.format(_id, _source['qnode'])
        return _id

    @staticmethod
    def query_average_scores(df):
        scores = dict()
        lambdas = {
            '1': 1.0,
            '2': 1.0,
            '3': 1.0,
            '4': 1.0,
            '5': 1.0,
            '10': 1,
            '42': 2.0
        }
        qnodes = df['_candidates']
        for qnode_string in qnodes:
            try:
                if qnode_string is not None and
                isinstance(qnode_string, str) and
                qnode_string.strip() != '':
                    q_scores = qnode_string.split('@')
                    for q_score in q_scores:
                        if q_score != 'nan':
                            q_score = q_score.replace('Category:', '')
                            q_score = q_score.replace('Wikidata:', '')
                            q_score_split = q_score.split(':')
                            score = q_score_split[2]
                            id = str(q_score_split[0])
                            if id not in scores:
                                scores[id] = {
                                    'count': 0,
                                    'cumulative': 0.0,
                                    'lambda': lambdas[id]
                                }
                            scores[id]['count'] += 1
                            scores[id]['cumulative'] += float(score)
            except Exception as e:
                print(e)
                print(qnode_string)
                raise Exception(e)

        for k in scores:
            scores[k]['avg'] =
            float(scores[k]['cumulative'] / scores[k]['count'])
        return scores

    def get_candidates_qnodes_set(self, df):
        qnode_set = set()
        candidate_strings = df['_candidates'].values
        for candidate_string in candidate_strings:
            qnode_set.update
            (self.create_list_from_candidate_string(candidate_string))
        return qnode_set
```

```
@staticmethod
def create_list_from_candidate_string(candidate_string):
    qnode_set = set()
    if candidate_string is not None and
    isinstance(candidate_string, str):
        c_tuples = candidate_string.split('@')
        for c_tuple in c_tuples:
            if c_tuple is not None and isinstance(c_tuple, str)
            and c_tuple != 'nan':
                try:
                    vals = c_tuple.split(':')
                    if vals[0] != '5':
                        qnode_set.add(vals[1])
                    else:
                        _ = vals[1].split('$')
                        if len(_) > 1:
                            qnode_set.add(_[1])
                except:
                    print(c_tuple)
    return list(qnode_set)

def create_qnode_to_labels_dict(self, qnodes):
    qnode_to_labels_dict = dict()
    if not isinstance(qnodes, list):
        qnodes = [qnodes]
    while (len(qnodes) > 0):
        batch = qnodes[:100]
        qnodes = qnodes[100:]
        query = {
            "query": {
                "ids": {
                    "values": batch
                }
            },
            "size": len(batch)
        }

        response = requests.post(self.wiki_dbpedia_joined_search_url,
        json=query)
        if response.status_code == 200:
            es_docs = response.json()['hits']['hits']
            for es_doc in es_docs:
                qnode_to_labels_dict[es_doc['_id']] =
                es_doc['_source']
    return qnode_to_labels_dict

@staticmethod
def create_qnode_to_type_dict(qnode_to_labels_dict):
    qnode_to_type_map = dict()
    for qnode in qnode_to_labels_dict:
        dbpedia_instance_types =
        qnode_to_labels_dict[qnode]['db_instance_types']
        qnode_to_type_map[qnode] = list(set(dbpedia_instance_types))
    return qnode_to_type_map

@staticmethod
def create_qnode_to_dburi_map(qnode_to_labels_dict):
    qnode_to_dburi_map = {}
    for qnode in qnode_to_labels_dict:
        dbpedia_urls = qnode_to_labels_dict[qnode]['dbpedia_urls']
        if len(dbpedia_urls) > 0:
            for dbpedia_url in dbpedia_urls:
                if dbpedia_url.startswith
                ('http://dbpedia.org/resource'):
                qnode_to_dburi_map[qnode] = dbpedia_url
            if qnode not in qnode_to_dburi_map:
            qnode_to_dburi_map[qnode] = dbpedia_urls[0]
        else:
            qnode_to_dburi_map[qnode] = None
    return qnode_to_dburi_map

@staticmethod
def create_high_precision_tfidf_input(label_hp_candidate_tuples):
    _ = {}
    for label_hp_candidate_tuple in label_hp_candidate_tuples:
        label = label_hp_candidate_tuple[0]
        candidate = label_hp_candidate_tuple[1]
        _[label] = candidate
    return _

def get_dburi_for_qnode(self, qnode, qnode_dburi_map):
    if qnode is None:
        return None
    if qnode_dburi_map.get(qnode) is not None:
        return qnode_dburi_map[qnode]

    _qdict = self.create_qnode_to_labels_dict(qnode)
    return self.create_qnode_to_dburi_map(_qdict)[qnode]

@staticmethod
def create_lev_similarity_dict(label_lev_tuples,
candidate_selection_object):
    if not label_lev_tuples:
        return {}
    label_lev_similarity_dict = {}
    for label_lev_tuple in label_lev_tuples:
        if label_lev_tuple:
            label = label_lev_tuple[0]
            lv_qnodes_string = label_lev_tuple[1]
            _l_dict = candidate_selection_object
            .sort_lev_features(lv_qnodes_string, threshold=0.0)
            label_lev_similarity_dict[label] = _l_dict
    return label_lev_similarity_dict

@staticmethod
def create_answer_dict(df):
    answers = list(zip(df.label, df.cta_class,
    df.answer_Qnode, df.answer_dburi))
    _dict = {}
    for answer in answers:
        _dict[answer[0]] = (answer[1], answer[2], answer[3])
    return _dict

def wikify_column(self, i_df, column, case_sensitive=True):
    raw_labels = list()
    if isinstance(column, str):
        # access by column name
        raw_labels = list(i_df[column].unique())
    elif isinstance(column, int):
        raw_labels = list(i_df.iloc[:, column].unique())
    _new_i_list = []
    for label in raw_labels:
```

```
      _new_i_list.append({'label': label,
        '_clean_label': self.clean_labels(label)})
   df = pd.DataFrame(_new_i_list)

   # find the candidates
   df['_candidates'] = df['_clean_label'].map(lambda x:
   self.run_query(x))
   df['_candidates_list'] = df['_candidates'].map(lambda x:
   self.create_list_from_candidate_string(x))
   self.aqs = self.query_average_scores(df)
   all_qnodes = self.get_candidates_qnodes_set(df)
   qnode_to_labels_dict =
   self.create_qnode_to_labels_dict(list(all_qnodes))
   qnode_dburi_map =
   self.create_qnode_to_dburi_map(qnode_to_labels_dict)
   qnode_typeof_map =
   self.create_qnode_to_type_dict(qnode_to_labels_dict)
   cta = CTA(qnode_typeof_map)
   tfidf = TFIDF(qnode_to_labels_dict)
   df = self.lev_similarity.add_lev_feature(df,
   qnode_to_labels_dict, case_sensitive)
   cs = CandidateSelection(qnode_dburi_map,
   self.aqs, qnode_typeof_map)
   df = cs.select_high_precision_results(df)
   df_high_precision = df.loc[df['answer'].notnull()]
   label_lev_similarity_dict =
   self.create_lev_similarity_dict(
   list(zip(df._clean_label, df.lev_feature)), cs)
   label_hp_candidate_tuples =
   list(zip(df_high_precision._clean_label,
   df_high_precision.answer))
   high_precision_candidates =
   self.create_high_precision_tfidf_input(label_hp_candidate_tuples)
   label_candidates_tuples =
   list(zip(df._clean_label, df._candidates_list))
   tfidf_answer = tfidf.compute_tfidf(label_candidates_tuples,
   label_lev_similarity_dict,
        high_precision_candidates=high_precision_candidates)
   cta_class = cta.process(df_high_precision)
   df['cta_class'] = cta_class.split(' ')[-1]
   df['answer_Qnode'] = df['_clean_label'].map(lambda x:
        tfidf_answer.get(x))
        df['answer_dburi'] = df['answer_Qnode'].map(lambda x:
        self.get_dburi_for_qnode(x, qnode_dburi_map))
        answer_dict = self.create_answer_dict(df)
        i_df['{}_cta_class'.format(column)] = i_df[column]
        .map(lambda x: answer_dict[x][0])
        i_df['{}_answer_Qnode'
        .format(column)] = i_df[column].
        map(lambda x: answer_dict[x][1])
        i_df['{}_answer_dburi'.format(column)] = i_df[column].
        map(lambda x: answer_dict[x][2])
        return i_df

    def wikify(self, i_df, columns, format=None, case_sensitive=True):
        if not isinstance(columns, list):
            columns = [columns]
        for column in columns:
            i_df = self.wikify_column(i_df, column,
            case_sensitive=case_sensitive)
        if format and format.lower() == 'iswc':
            _o = list()
            for index, row in i_df.iterrows():
                for column in columns:
                    _o.append({'column': column, 'r': index,
                    'q': row['{}_answer_Qnode'.format(column)]})
            return pd.DataFrame(data=_o)
        if format and format.lower() == 'wikifier':
            _o = list()
            for index, row in i_df.iterrows():
                for column in columns:
                    _o.append({'f': '', 'c': '', 'l': row[column],
                    'q': row['{}_answer_Qnode'.format(column)]})
            return pd.DataFrame(data=_o)
        return i_df

#RUN CTA MODEL
class CTA(object):
    def __init__(self, dburi_typeof):
        self.dburi_typeof = dburi_typeof
        self.super_classes =
        pd.read_csv('wikifier/caches/SuperClasses.csv',
        header=None)[0].tolist()
        self.db_classes = json.load(open
        ('wikifier/caches/DBClasses.json'))
        self.db_classes_closure = json.load(open
        ('wikifier/caches/DBClassesClosure.json'))
        self.sparqldb = SPARQLWrapper("http://dbpedia.org/sparql")
    def is_instance_of(self, uri):
        self.sparqldb.setQuery(

            select distinct ?x where {{ <{}>
            <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?x .
            ?x <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
            <http://www.w3.org/2002/07/owl#Class> .}}

            .format(uri))
        self.sparqldb.setReturnFormat(JSON)
        results = self.sparqldb.query().convert()
        instances = set()
        for result in results["results"]["bindings"]:
            dbp = result['x']['value']
            instances.add(dbp)
        return instances

    def evaluate_class_closure(self, urilist, classuri):
        matches = 0
        classuriclosure = set()
        if classuri in self.db_classes_closure:
            classuriclosure = set(self.db_classes_closure[classuri])
        validuri = []
        for uri in urilist:
            if uri in self.dburi_typeof:
                instances = self.dburi_typeof[uri]
            else:
                instances = self.is_instance_of(uri)
                self.dburi_typeof[uri] = list(instances)

            for instance in instances:
                if instance in classuriclosure:
                    validuri.append(uri)
                    matches += 1
                    break
        score = matches / len(urilist)
        return [score, validuri]

    def find_class(self, urilist, classlist, currentans, ans_list,
    threshold):
        ans_list.append(currentans)
        if len(classlist) == 0:
            return

        max_score = -1
        max_validuri = []
        max_class = ''
        for superclass in classlist:
            [score, validuri] = self.evaluate_class_closure(urilist,
            superclass)
            if max_score < score:
                max_score = score
                max_validuri = validuri
                max_class = superclass

        if max_score >= threshold:
            subclasses = self.db_classes[max_class]
            self.find_class(max_validuri, subclasses,
            max_class, ans_list, threshold)
            return

    def process(self, df, threshold=0.508, class_level=0):
        urilist = df['answer'].tolist()
        if len(urilist) == 0:
            return ""
        ans_list = []
        self.find_class(urilist, self.super_classes, '',
        ans_list, threshold)
        ans_list = ans_list[1:]
        if len(ans_list) <= class_level:
            return ""
        return "␣".join(ans_list)
```

.1.4 Frequency Model

```
dataset = pd.read_csv("")
uris = dataset["answer_dburi"].tolist()
uris = pd.Series(uris)
uris = uris.dropna()
#RETRIEVE CLASS FROM EACH INSTANCE
def is_instance_of(uri):
        sparqldb = SPARQLWrapper("http://dbpedia.org/sparql")
        sparqldb.setQuery(

            select distinct ?x where {{ <{}>
            <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?x .
            ?x <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
             <http://www.w3.org/2002/07/owl#Class> .}} limit 10)

        .format(uri))
        sparqldb.setReturnFormat(JSON)
        results = sparqldb.query().convert()
        instances = set()
        for result in results["results"]["bindings"]:
            instances = result['x']['value']
        return instances
wiki_class = []
instance_class = []
for uri in uris:
    instance_class = is_instance_of(uri)
    wiki_class.append(instance_class)

#CALCULATE THE FREQUENCY FOR EACH CLASS
wiki_class = pd.Series(wiki_class, name='Class')
value_counts = wiki_class.value_counts(normalize=True,dropna= False)
frequency =
pd.DataFrame(value_counts.rename_axis('Class')
.reset_index(name='Frequency'))
```

.1.5 Weighted Average Calculation

```
#COMPUTATION OF DIFFERENT CONFIGURATIONS BETWEEN LEVENSHTEIN
#(OR JARO WINKLER) AND TFIDF
 denominator = float(_lev_score + tfidf_score)
                    if denominator <= 0.001:
                        _dict[qnode] = ((_lev_score * 0.4) +
                        (tfidf_score * 0.6)) * 100
                    else:
                         _dict[qnode] = ((_lev_score * 0.4) +
                         (tfidf_score * 0.6))/ denominator
```

.1.6 Final Model

```
#DEFINE BOTH METHODS
def process(self, urilist, threshold=0.508, class_level=0):


        if len(urilist) == 0:
            return ""
        ans_list = []
        self.find_class(urilist, self.super_classes, '',
        ans_list, threshold)
        ans_list = ans_list[1:]
        if len(ans_list) <= class_level:
            return ""
        return "␣".join(ans_list)

    def process_frequency_match(self, qnodes):
        if len(qnodes) == 0:
            return ""

        class_list = list()
        for qnode in qnodes:
            _list = [x for x in self.dburi_typeof.get(qnode, [])
            if x.startswith('http://dbpedia.org')]
            class_list.extend(_list)
        wiki_class = pd.Series(list(class_list), name='Class')
        value_counts = wiki_class.value_counts(normalize=True,
        dropna=False)
        frequency =
        pd.DataFrame(value_counts.rename_axis('Class').
        reset_index(name='Frequency'))
        print(frequency['Frequency'].tolist()[0])
        return frequency['Class'].tolist()[0]

#PROCESS INTEGRATED METHOD
def wikify_column(self, i_df, column, case_sensitive=True, debug=False):
        raw_labels = list()
        if isinstance(column, str):
            # access by column name
            raw_labels = list(i_df[column].unique())
        elif isinstance(column, int):
```

```
        raw_labels = list(i_df.iloc[:, column].unique())

    _new_i_list = []
    for label in raw_labels:
        _new_i_list.append({'label': label, '_clean_label':
        self.clean_labels(label)})
    df = pd.DataFrame(_new_i_list)
    # find the candidates
    df['_candidates'] = df['_clean_label'].map(lambda x:
    self.run_query(x))
    df['_candidates_list'] = df['_candidates'].map(lambda x:
    self.create_list_from_candidate_string(x)[0])
    df['_candidates_freq'] = df['_candidates'].map(lambda x:
    self.create_list_from_candidate_string(x)[1])
    self.aqs = self.query_average_scores(df)
    all_qnodes = self.get_candidates_qnodes_set(df)
    qnode_to_labels_dict =
    self.create_qnode_to_labels_dict(list(all_qnodes))
    qnode_dburi_map =
    self.create_qnode_to_dburi_map(qnode_to_labels_dict)
    qnode_typeof_map =
    self.create_qnode_to_type_dict(qnode_to_labels_dict)
    cta = CTA(qnode_typeof_map)
    tfidf = TFIDF(qnode_to_labels_dict)
    df = self.lev_similarity.add_lev_feature(df,
    qnode_to_labels_dict, case_sensitive)
    cs = CandidateSelection(qnode_dburi_map, self.aqs,
    qnode_typeof_map)
    df = cs.select_high_precision_results(df)
    df_high_precision = df.loc[df['answer'].notnull()]
    label_lev_similarity_dict =
    self.create_lev_similarity_dict(list(zip(df._clean_label,
    df.lev_feature)), cs)
    label_hp_candidate_tuples =
    list(zip(df_high_precision._clean_label,
    df_high_precision.answer))
    high_precision_candidates =
    self.create_high_precision_tfidf_input(label_hp_candidate_tuples)
    label_candidates_tuples =
    list(zip(df._clean_label, df._candidates_list))
    tfidf_answer =
    tfidf.compute_tfidf(label_candidates_tuples,
    label_lev_similarity_dict,
    high_precision_candidates=high_precision_candidate)
    hp_qnodes = df_high_precision['answer'].tolist()
    df['answer_Qnode'] = df['_clean_label'].map(lambda x:
    tfidf_answer.get(x))
    df['answer_dburi'] = df['answer_Qnode'].map(lambda x:
    self.get_dburi_for_qnode(x, qnode_dburi_map))
    #process Frequency Model
    cta_class = cta.process_frequency_match
    (df['answer_Qnode'].tolist())
    #if Frequency Model doesn't produce an output, try the CTA
    if cta_class == "" or cta_class == "{}":
        cta_class = cta.process(hp_qnodes)
    df['cta_class'] = cta_class.split(' ')[-1]
    answer_dict = self.create_answer_dict(df)
    i_df['{}_cta_class'.format(column)] = i_df[column].map(lambda x:
    answer_dict[x][0])
    i_df['{}_answer_Qnode'.format(column)] =
    i_df[column].map(lambda x: answer_dict[x][1])
    i_df['{}_answer_dburi'.format(column)] =
    i_df[column].map(lambda x: answer_dict[x][2])
    return i_df

def wikify(self, i_df, columns, format=None, case_sensitive=True):
    if not isinstance(columns, list):
        columns = [columns]

    for column in columns:
        i_df = self.wikify_column(i_df, column,
        case_sensitive=case_sensitive, debug=True)
    if format and format.lower() == 'iswc':
        _o = list()
        for index, row in i_df.iterrows():
            for column in columns:
                _o.append({'column': column, 'r': index,
                'q': row['{}_answer_Qnode'.format(column)]})
        return pd.DataFrame(data=_o)
    if format and format.lower() == 'wikifier':
        _o = list()
        for index, row in i_df.iterrows():
            for column in columns:
                _o.append({'f': '', 'c': '', 'l': row[column],
                'q': row['{}_answer_Qnode'.format(column)]})
        return pd.DataFrame(data=_o)
    return i_df
```

.1.7 OpenStreetMap Retrieval

```
#ACCESS TO OPENSTREETMAP WEBSITE
url = 'https://nominatim.openstreetmap.org/search'
lst_of_uniqs_class = dict()
lst_of_uniqs_type = dict()
empty dict to store unique labels
of types (key) and their count (value)
for col in df_new:
    clean_col = ftfy.fix_encoding(col)
    clean_col = ftfy.fix_text(col)
    clean_col = col.replace('\n', '␣')
    payload = {'q': clean_col,
        'format': 'json',
        'county' : 'Los␣Angeles␣County',
        'country' : 'United␣States␣of␣America',
        'extratags' : 1,
        'limit' : 10
    }
    resp = requests.get(url=url, params=payload)
    data = json.loads(resp.text)
    data = pd.DataFrame(data)
    if 'class' in data.columns.values:
        for itm in data['class'].tolist():
            if itm not in lst_of_uniqs_class: # as key
                lst_of_uniqs_class[itm] = 1
            else:
                lst_of_uniqs_class[itm] += 1
    if 'type' in data.columns.values:
        for itm in data['type'].tolist():
            if itm not in lst_of_uniqs_type: # as key
                lst_of_uniqs_type[itm] = 1
            else:
                lst_of_uniqs_type[itm] += 1
most_common_class  = max(lst_of_uniqs_class.items(),
key=operator.itemgetter(1))[0]
most_common_type = max(lst_of_uniqs_type.items())
```

.2 Datasets Description and Example

The datasets used for this research are available, thanks to the city of Los Angeles, at: `https://data.lacity.org/`. An example of dataset about Businesses is reported below.

ACCOUNT #	BUSINESS NAME	STREET ADDRESS
0000019227-0001-6	CINCO IRON WORKS INC	890 ONTARIO BLVD
0000020289-0001-8	MCWHIRTER STEEL INC	42211 7TH STREET E
0000028383-0001-7	MILLERS FAB/WELD CORP	6100 INDUSTRIAL AVENUE
0000034550-0001-6	CMC STEEL FABRICATORS INC	12451 ARROW ROUTE
0000036594-0001-3	ANVIL STEEL CORPORATION	137 W 168TH STREET
0000039799-0001-4	MASTERCRAFT IRON CO INC	7463 VARNA AVENUE
0000045151-0001-1	VCS METAL WORKS INC	14756 KESWICK STREET
0000048318-0001-7	SANIE MANUFACTURING CO INC	2600 S YALE STREET
0000050435-0001-7	ALEX SANCHEZ	1868 AUTUMN LANE
0000057905-0001-8	R/B REINFORCING STEEL CORP	13581 5TH STREET

Table 1: Dataset Examples about Businesses in Los Angeles (part 1)

ZIP CODE	LOCATION	LOCATION DATE
91761-1835	890 ONTARIO 91761-1835	07/17/2003
93535-5400	42211 7TH 93535-5400	04/17/2000
92504-1120	6100 INDUSTRIAL 92504-1120	06/19/2000
91739-9601	12451 ARROW 91739-9601	01/01/1975
90248-2728	137 168TH 90248-2728	02/01/1975
91605-4011	7463 VARNA 91605-4011	01/01/1995
91405-1205	1161 OGDEN 90046-5332	07/22/2003
92704-5228	2600 YALE 92704-5228	07/01/1992
92084-3344	1868 AUTUMN 92084-3344	11/13/2002
91710-5166	13581 5TH 91710-5166	11/18/1995

Table 2: Dataset Examples about Businesses in Los Angeles (part 2)

In this case, for example, the useful column to select for the experiment would be the **BUSINESS NAME** column. While, for OpenStreetMap retrieval, **STREET ADDRESS** column would be isolated.

.3 Integral Results

In this final section we report the results in detail of the final model output. As it is possible to see, the first column contains all the datasets used for the experiments (D=41). Then, there are two columns devoted to the answers of the final model, highlighting the cases where it has been necessary to access OpenStreetMap (OSM). The final column, instead, shows the target class; thus, the one expected from that dataset. The results have been assessed as follows:

- **green cell**: the class has been correctly identified,
- **red cell**: the class has been wrongly identified,
- **yellow cell**: the sub or super class has been correctly identified, but not the exact class,
- **{}**: the algorithm has not been able to detect a class for that dataset.

DATASET	CTA + FREQUENCY	OSM	CLASS
Affordable Housing	{}	place:house	Building
Animals	Mammal		Animal
Apparel	Organisation		Organisation
Arts Ed Profile	Organisation		Educational Institution
Auto Parts Lease	Organisation		Organisation
Care	Organisation		Hospital
City Lobbyists	Organisation		Company
City Projects and Agencies	Organisation		Company
Cultural Centers Theaters	Venue		Venue
Cultural Event	Museum		Event
Department of Recreation	{}	leisure:park	Park
Education Facilities	Organisation		Educational Institution
Elected Official Salary	Settlement		Profession
Events from LA Festival	Social Event		Social Event
Foothill commercial	Organisation		Organisation
Fused Maps	Library		Educational Institution
GA Bootcamp	Organisation		Organisation
Hospitals	Hospital		Hospital
Housing and City Services	{}	office:company	Organization
Immigration Workshop	{}	amenity:library	Educational Institution
Invoices and Purchase	Organisation		Organisation
LA Active Businesses	Organisation		Organisation
LA City Departments	Agent		PublicService
LA City Events	Settlement		SocialEvent
LA Florists	Work		Organisation
LAcity.org Website Traffic	Software		Software
Law Firms Named	Organisation		Organisation
Library Branches	{}	place:house	Educational Institution
Multipurpose Centers	{}	place:house	Venue
Museums	Museum		Museum
Music Released	Musical Work		Musical Work
Payroll by job classes	Person		Profession
Public Housing Sites	{}	natural: peak	Garden
Registered Foreclosure	Organisation		Organisation
Renewable Projects	Power Station		Power Station
Restaurants	Organisation		Restaurant
Skateparks	{}	leisure:pitch	ParkAttraction
Streets Name	PopulatedPlace		Street
Trees	Plant		Plant
Whats Happening in LA	Organisation		Event
Wilshire	Organisation		Organisation

Table 3: Final Model Results