A LARGE SCALE NAME MATCHING AND SEARCH FRAMEWORK

Stella Margonar

March 2013

Technical Report # DISI-13-026

Thanks: Fausto Giunchiglia (advisor), Juan Pane (co-advisor)

UNIVERSITÀ DEGLI STUDI DI TRENTO

Facoltà di Scienze Matematiche, Fisiche e Naturali

M.Sc. of Science in Computer Science

Final thesis

A Large Scale Name Matching and Search Framework

Tutor:
Fausto Giunchiglia
Università degli Studi di Trento

Graduant:
Stella Margonar

Co-Tutor:
Juan Pane
Università degli Studi di Trento

Academic year 2011/2012

Abstract

The ability of identifying whether two strings represent names referring to the same real world entity is essential for avoiding information integration problems, such as duplication of records. We study this problem in a scenario where the amount of data to analyze becomes large.

Our purpose is to develop a framework that address the name match and search problem, combining together different strategies, and is able to consider also the semantic of the string representing a name. Moreover we propose a dataset for evaluating name matching algorithm which consider semantic variation of names.

Contents

1 Introduction **13**

2 Problems Definition **17**

2.1 Name Matching 17

2.2 Name Search 19

2.3 Named Entity Recognition 22

2.4 Named Entity Disambiguation 23

3 Scenarios **25**

3.1 Entity Importing 25

3.2 NLP 26

3.3 Entity Identification 27

3.4 Offline Name search 27

3.5 Entity matching 28

3.6 Online name search 28

4 State of the Art **31**

4.1 Name Variations 31

4.1.1 Plain String Matching 31

4.1.2 Algorithms for Misspelling Variations 32

4.1.3 Format Variations 34

4.1.4 Phonetic Algorithms 35

4.1.5 Information Retrieval Techniques for Spelling Correction 37

4.2 Name Variants 39

4.2.1 Translations 39

4.2.2 Other Variants 40

4.3 Evaluations from papers 41

5 Algorithm Design **47**
5.1 Name Match Function . . . 47
5.1.1 Algorithms used . . . 47
5.1.2 Function High level design . . . 49
5.2 Standard Search . . . 56
5.3 Auto completion Search . . . 60
5.4 Named Entity Recognition function . . . 63
5.5 Named Entity Disambiguation function . . . 63

6 Ordering **65**
6.1 Similarity Measure . . . 65
6.2 User Ranking . . . 68
6.3 Statistics Usage . . . 69

7 Name Database **73**
7.1 Concept Definition . . . 73
7.2 EType Specific Structure . . . 74
7.2.1 EType Person . . . 75
7.2.2 EType Location . . . 76
7.2.3 EType Organization . . . 76
7.3 Database Architecture . . . 77

8 Evaluation **79**
8.1 Dataset . . . 79
8.1.1 Semantic Dataset . . . 80
8.1.2 Census Dataset . . . 82
8.2 Evaluation Modality . . . 84
8.2.1 Experiment Setup . . . 84
8.2.2 Parameters . . . 85
8.2.3 Values of Parameters . . . 86
8.3 Measure . . . 87
8.4 Results . . . 88
8.5 Remarks . . . 93

9 Conclusions **95**

Bibliography **99**

A Dataset Appendix **103**

B Freebase API **105**

List of Tables

4.1 Authors' name variation depending on bibliographic style, from [13] 35
4.2 Soundex conversion table 36
4.3 Example of edit distance integrated with misspelling statistics, from [22] . 38
4.4 Dataset for Evaluation 45

6.1 Example of tokens for two person names 66
6.2 Example of String Similarity computation on two person names 67
6.3 Other example of tokens for two person names 68
6.4 Other example of String Similarity computation on two person names . . . 68
6.5 Pairs stored for each function 69

8.1 Statistics on Datasets used for evaluation 80
8.2 Example of Semantic Dataset Entity of type Person 80
8.3 Example of two match and not match pairs from the Semantic dataset . . . 81
8.4 Example of Census Dataset Entry 83
8.5 Results on Census dataset from William W. Cohen et al. [11] 83
8.6 Example of alternative names and translation from Freebase for entity "/en/rome" 84
8.7 System Configuration for the experiment 86
8.8 True False Positive Negative definition 87
8.9 F-measure Results Summary 89
8.10 Result Summary 91
8.11 Results on Census dataset from William W. Cohen et al. [11] compare with our experiment 92

B.1 Example of Freebase query on identifier and alternative names for entity Rome 105
B.2 Example of Freebase query on identifier and alternative names for entity Rome 106

List of Figures

7.1 Database ER Model 78

8.1 Charts for precision recall on the datasets 88
8.2 Charts for $F_1 - measure$ on the datasets 89
8.3 Charts for time on the datasets 90
8.4 Time Comparison for positive and negative match pairs 93

Acronyms

NED Named Entity Disambiguation

NER Named Entity Recognition

OCR Optical Character Recognition

LCS Longest Common Substring

IBM International Business Machines

GUID Global Unique Identifier

NLP Natural Language Processing

BNF Backus Normal Form

Chapter 1

Introduction

The ability of identifying whether two strings represent names referring to the same real world entity is a complex task when the amount of data to analyze becomes large, and it is essential for avoiding information integration problems, such as duplication of records.

There are different fields of research that deal with problems related to names in computer science: the identification of names in a text (Named Entity Recognition (NER)), the disambiguation of which entity corresponds to a name (Named Entity Disambiguation (NED)), the search of entities based on their name, and the recognition of two strings as names corresponding to the same entity (Name Matching).

All these problems are made more complex by the variable nature of names. As presented by the taxonomy of Bignotti et al. in [13], names can have variant (meaning alternative names for the same entity) and variations (orthography variations of the string, like misspellings or format modification).

This thesis will analyze these four fields cited related to entity name, and will design a framework able to provide an implementation of algorithms suitable to address these problems for large scale scenarios. In the analysis we will identify the essential high level functions that will be implemented in the algorithm, trying to reduce, if possible, the problems to a minimum set of functions.

The framework will implement this set of functionality combining together different state of the art techniques for string matching and new approaches for addressing the particular nature of names.

The novel aspect that this thesis has compared with the current state of the art works, is the consideration in the analysis and design of the taxonomy presented by Bignotti et al. in [13]. At the moment, the main name matching techniques available in the state of the art are based on the orthography of the string representing a name. Therefore, they are able to identify two strings as referring to the same entity (name matching) only if these strings are in some way similar considering their syntactic aspect. In this work we will introduce in the comparsison of names, their semantic value as references of the same real world entity. Each entity can then have multiple names, which can be classified based on the taxonomy from Bignotti et al. in [13] as variations (already considered by state of the art) and variant, meaning names that are not similar considering their orthography, but have refer to the same real world entity. In summary we will not only consider similar name as name that orthographically are comparable, but also semantically related.

The thesis is structured in 7 chapters that focus on different aspects of the work, and cover the entire process followed in the development of the framework.

Chapter 2 describes and analyzes the four problems considered regarding names in computer science. For each problem it is given a formal definition and the high level signature of the function that will be designed for addressing that particular problem.

Chapter 3 defines the scenarios in which the framework will be applied. In particular we will take into account two main use cases: the *Offline Usage* which groups together operations executed not interactively by the user, and considered as background or mainenance tasks (like Natural Language Processing (NLP), Entity Identification, Name Search and Entity Matching), and the *Online Usage* which consider operation execute actively by the user for interacting with the system (like Name Search Online, see section 3.6).

Chapter 4 presents the current status of the research for the name related problems. The chapter is divided in two main sections that focus of different type of names: section 4.1 reports the main algorithms available for addressing name variations, while works about name variant are presented in 4.2.

Chapter 5 gives a more detailed specification of the functions described in chapter 2, in order to provide an implementation proposal for each problem. In particular the *name matching* function (defined in section 2.1) will be described in section 5.1, the *name search* function (see section 2.2) is presented in section 5.2 and 5.3, *named entity recognition* (see section 2.3) is illustrated in section 5.4, and the *named entity disambiguation* function (see section 2.4) is defined in section 5.5.

Chapter 6 discusses the problem of ordering, which consists in assigning a ranking value to the result obtained by the different functions described in chapter 5. The ranking value will be design as function of two parameters: the name similarity, as defined by name match function (see 2.1), and the user experience.

Chapter 7 presents the design and implementation choices made for the development of the system database. In addition here are formally defined the concepts of Entity, Entity Type (EType) and Entity Name, and the structure of each Name depending on the entity type.

In chapter 8 will be described the process used for evaluating the framework presented in the previous chapters, compared with other string matching algorithm already present in the state of the art. For this purpose, will be used two datasets, as presented in section 8.1. All the result collected from the experiment regarding those configurations are presented and discussed in section 8.4.

Chapter 2

Problems Definition

In this section will be analyzed four problems related to name matching for Entities, which are Name Matching (see section 2.1), Name Search (see section 2.2), Named Entity Recognition (NER) (see section 2.3) and Named Entity Disambiguation (NED) (see section 2.4).

For each of them will be provided the formal problem definition, and a high level description of the function needed to address the problem. The specific algorithm design will be provided in section 5.

2.1 Name Matching

Name Matching can be defined as "the process of determining whether two name strings are instances of the same name" (Patman and Thompson)[24]. As described by Bignotti, Pane and Giunchiglia [13] the class of nouns is dived into 3 subclasses: common nouns, proper nouns and pronouns. The main subject of this work will be proper nouns. A proper nouns is defined as a non-empty set of nouns which must contain at least a proper nouns. Examples of proper names are:

- "Raffaella Mondini": person name, composed by two proper nouns ("Raffaella" and "Mondini");
- "Garda Lake": location name, composed by a common noun ("Lake") and a proper noun ("Garda");
- "University of Trento": organization name, composed by a common noun ("university"), proper noun ("Trento") and a conjunction ("of").

The challenge for the name matching problem is to understand whether two strings correspond to the same name regardless name variations. A taxonomy of name variations has been developed and verified by Bignotti et al. [13]:

Full translation: translation from a source language to a target language of the entire name. Example: *Goofy*[en] vs. *Pippo*[it];

Part-of translation: for name composed by both common and proper nouns, the translation from one language to another will affect the former (also called trigger word) while keeping unaltered the latter. Example: *University of Trento*[en] vs. *Università di Trento*[it];

Misspellings: punctation, capitalization, spacing, omissions, additions, substitutions, phonetic variations, switching pairs of neighboring letters. Example: *J.K. Rowling* vs. *J K Rowling*, *Fausto* vs. *Fasuto*;

Format: different positioning and format of name elements. Example: *Fausto Giunchiglia* vs. *Giunchiglia Fausto*, *J.K. Rowling* vs. *Joanne Kathleen Rowling.* For more detailed examples see Table 4.1.

Name variations are caused by different factors. As reported by Christen in [10], when data are collected the way they are gathered affects different types of errors and variations in names: handwritten data, read with Optical Character Recognition (OCR) technology, can cause misspellings for letters with similar shape (m and n, u and v), manual typed words will contain misspellings of letters close on the keyboard, spoken collected words have higher probability to contain phonetic errors, while data stored in limited length fields can influence the user who will use shorter name than the original (e.g.: *Gianpaolo* vs *Giampi*).

All these name variations make difficult to decide whether two strings correspond to the same name through exact string comparison. Thus, as suggest by Christen in [10] "an approximate measure of how similar names are is desired. Generally a normalized similarity measure between 1.0 (identical names) and 0.0 (names totally different) is used." Name variations are the main difficult faced by traditional name matching, while in this work we also consider name variant (alias, pen names, nicknames [13]).

As input for the problem we have two strings, representing names, plus the entity type to which they belong. The possible entity types are:

- Person
- Location
- Organization
- Event

Accordingly to the given entity type, the strings could have been subject to different variations and variants. If we take the entity type "Person" as an example, we know that "there is no legal regulation about what constitutes a name" (Borgman and Siegfried) [7] and cultural context is one of the main factor that changes the name composition. A more detailed analysis of which are the possible format variations of names accordingly to cultural context will be given in section 7.2.1. Knowing the entity type considered, allows us to apply different algorithms and techniques ad-hoc for the name structure.

Formal Function Definition

Given two names $Name_1$ and $Name_2$ and the belonging entity type for the names (respectively $EType_1$ and $EType_2$), the expected output is a number $sim \in [0, 1]$ representing the probability that $Name_1$ is a variation or variant of $Name_2$.

$$nameMatching :< String, EType > \times < String, EType > \rightarrow sim \in [0, 1] \tag{2.1}$$

The function necessary to solve this problem is called *nameMatching* and its signature is:

$$\text{float } nameMatching(\text{String } Name_1,\ \text{String } Name_2,\ \text{EType } EType); \tag{2.2}$$

2.2 Name Search

The problem in analysis in this section consists in searching for an entity using its name as key for the research. The most common application example of this problem is a standard search in Wikipedia website: the user inserts the name of the topic it is interested in, and the system will retrieve the entity to which that name corresponds.

The challenges that raise from this problem can come both from the user who writes the query and from the source where the data are stored: Pfeifer et al. in [25] describe *vagueness of query* as the problem that misspellings can be contained in the user query, while as *uncertainly of knowledge* the possibility that those errors could appear in the database where the entities are stored.

Name Search and Name Matching (see section 2.1) are similar problems because the techniques that could be used for one could be adapted for solving also the other. The main difference between them is that the former, given a string, has to generate a list of candidates that correspond to variations and variants of the original string and match to the respective entities; while the latter has to compare two strings and decide whether they identify the same entity.

More formally name search problem takes as input a query inserted by the user (generally a string) and computes the solution with the support of a provided set of user defined entities. The expected output is a list of names representing the candidates entity names that match the user query.

Since both the entities and the query are user defined, we should expect that they could contain errors and be incomplete. A subproblem that should be considered is about which is the source of errors: do we assume that the database is error-free or not? And what about the query?

The second problem to take into account is about *closed vs open world assumption*. For simplicity and for the purposes of this thesis, our function will be design in a closed world. This means that the name search will regard only the entities given in the initial bootstrap: if the algorithm does not find any name that matches the query in the given entity set, then it will answer that the query does not correspond to any named entity. On the other hand, in an open world assumption we allow the user and the system to add new knowledge to the dictionary of entities. Then, in case the algorithm is not able to match the query with any available named entity, the algorithm will answer that he does not know yet any name that can match that query. In addition, another process will will offer to the user (or the system) the possibility to add such named entity to the knowledge base (KB).

The general vision of name search with respect to name variations and variant is a query expansion, which consists in the addition of new terms to the user search, in order to improve precision and overcome misspelling or similar variations.

There are two main application of a hypothetical function which performs name search accordingly to name variations and variant.

Autocompletion: in this case name search should be able to find all possible candidates that match the first input characters of the query. The list will be displayed to the user while he/she is typing in order to suggest sooner possible completions of the query.

The computation of this function has to be fast in order to anticipate the user in the typing of the query.

During the design of this function we can operate a choice for what concerns the location of errors in the data: accordingly to Pollock and Zamora in [27] fewer errors typically occur at the beginning of names. We could then assume that the inserted partial query does not contain errors, or better, that if the partial query is shorter than a fixed number of characters n then it is not likely to contain errors.

Standard Search: This function is meant to operate a name search starting from a complete query, returning as result a list of candidates entity names.

In the high level design of this algorithm we identify two main possible approaches:

- a query expansion of the search with all the variant and variations of the name obtained through a dictionary lookup;
- a canonization of the name, meaning that each name has a canonical form, shared by all the names for a particular entity: when a query is inserted is first standardized (obtaining its canonical form, or a list of probable canonical forms) and the corresponding entity is returned as result. Another idea for managing name canonical forms is to insert a list of canonical forms for the name at the entity creation time. This set of names can also be used as index for retrieving the corresponding entity name.

Formal Function Definition

As described in the previous section the functions that should be implemented for solving the name search problem are two.

The first accomplishes the autocomplete feature, which should take as input a short list of initial characters of a name, and generate a list of probable name candidates for that input.

Let Σ be the alphabet of the dataset, $c_1, c_2, c_3 \in \Sigma$ characters contained in Q (the query), E the set of entities available, and $name(e)$ the function that given an entity e returns its name.

$$\begin{gathered} NS\_autocomplete : Q =< c_1, c_2, c_3 > \rightarrow \\ C = \{n | \exists e \in E \text{ s.t. } n = name(e) \text{ and } n =< c_1, c_2, c_3, n' >\} \text{ with } n' \text{ string} \end{gathered} \tag{2.3}$$

The second function regards the standard name search which takes as input an entire string q and, based on the set of entities available, generates a list of candidate names for the query.

$$NS : Q \rightarrow C = \{n | \exists e \in E \text{ s.t. } n = name(e), n \text{ matches } Q\} \tag{2.4}$$

Of course the algorithm necessary to choose the proper candidates will use techniques related to the name matching function.

Both approaches have the same method signature:

$$\text{List<String, float>} \; searchPrefixName(\text{String } Q); \tag{2.5}$$

$$\text{List<String, float>} \; searchName(\text{String } Q); \tag{2.6}$$

2.3 Named Entity Recognition

The objective of Named Entity Recognition (NER) "is to identify and categorize all members of certain categories of *proper names* from a given corpus." (Borthwick, Sterling, Agichtein and Grishman)[8]. In other words, NER is the task of analyzing a source (for example a text), recognizing which elements (if we are considering a text then strings) correspond to entity names, and labeling them with the corresponding entity type.

This problem has various application context, for example in biology NER is the problem of "finding references to entities (mentions) such as genes, proteins, diseases, drugs, or organisms in natural language text, and labeling them with their location and type" (Leaman, Gonzalez et al.) [19].

An application which is related to the purpose of this work is described by Pouliquen et al. in [28] where the objective of the research is to develop a tool which is able to extract and relate entities (in particular personal names) from texts written in different languages.

Accordingly to what is reported in [2] multilingual approach is very important: only the 26,8% of Internet users are (only) English speaking people. This implies that web pages and resources contain names for entities in different languages (name variants), and with different formats (name variations).

The mean in which the source is provided affects the task, as well as the variety of sources which are given. At the moment names can be recognized in 4 main ways: [28]

- through a lookup procedure if a list of known names exists.
- by analyzing the local lexical context (e.g. 'President' Name Surname).
- because part of a sequence of candidate words is a known name component (e.g. 'John' Surname).
- because the sequence of surrounding parts-of-speech (or other) patterns indicates to a tagger that a certain word group is likely to be a name.

For example, if we consider some different source from a news article which do not give to the words contained a context (take for example a list of names) it could be difficult to use the second and forth technique.

The approach to NER problem proposed by Pouliquen et al. [28] can be used also for the multilingual part of name matching problem (see section 4.2.1), and will be taken into account during the development of the thesis for the algorithm part.

Name matching and NER are strongly related, because the latter has to rely on the former for what concerns name variations and variant in the sources, in order to identify the correct entity type that correspond to that name.

Formal Function Definition

Named Entity Recognition (NER) is the problem of deciding which is the probability p that a particular string s (part of a sentence) is a name for an entity E. Which formally consist in calculating $p = P(\exists E \text{ s.t. } match(name(E), s)$.

At the moment NER methodology consist in analyzing the text structure and accordingly to that, assigning a probability to each token (word) of that element to be a name or part of a name. This result is obtained basing the analysis on the grammatical structure of the sentence and the identification of trigger words (like "Mr", "Doctor"). Formally:

$$Structure\_analysis : T =< s_1, s_2, .., s_k >\rightarrow T' = \{(s_i, p_i)|s_i \in T \text{ and } p_i \in [0, 1]\} \quad (2.7)$$

where T is the original text defined as a sequence of strings, T' is the mapping of each string to the probability of being a name.

The probability for two names to match each other computed in name matching function (see eq. 2.2) can be used for contributing to the solution of this problem. Given the probability p (obtained from the text structure analysis) that a string s is a name, the solution of the problem is the probability p' calculated with the application of name search and matching functions:

$$\begin{aligned} NER : T' = \{(s_i, p_i)|s_i \in T \text{ and } p_i \in [0, 1]\} \rightarrow \\ T'' = \{(s_i, p'_i)|s_i \in T \text{ and } p'_i \in [0, 1]\} \end{aligned} \quad (2.8)$$

where p'_i is calculated accordingly to the provided entity set E.

The idea behind the use of these two function in the resolution of NER problem is that the probability of a string to be a name, is strictly related to the probability of that string to be matched with some name in the entity dataset E.

The final function that will address NER problem will take as input a string, and through name matching techniques, will return a list of couples composed by an EType and the probability that such string is a name for an entity of that EType.

$$\text{List<EType, float>} \; etypeNameRecognition(\text{String } name); \quad (2.9)$$

2.4 Named Entity Disambiguation

The problem of Named Entity Disambiguation (NED) is defined as "the task of linking an entity mention in a text to the correct real-world referent predefined in a knowledge base" (Ploch) [26], and it is related to name matching (see 2.1). In other words it is the problem of determining, given a string representing a name (also called surface form) to which entity it belongs.

This problem is also known by the name of *Entity Linking* [30] or also *Entity Deduplication* [29].

One of the challenges that this task raises is the many-to-many relation between names (in this case also called *surface forms*) and entities: "an entity can be referred to by multiple surface forms, and a surface form can correspond to multiple entities". For example take the name *Cristoforo Colombo*[it]: it is the Italian name for the famous explorer *Christopher Columbus*, but if we search for it on Wikipedia, we will discover that it is also the name for other entities (an airport, a station, multiple ships and movies). On the other way around, the person *Cristoforo Colombo*[it] can be described with different names: the translations in other languages (*Christophorus Columbus*[lat], *Cristóvão Colombo*[por], *Christopher Columbus*[en]), or the professional name *Admiral Columbus.*

Formal Function Definition

Given as input a name N and provided a set of named Entity E, Named Entity Disambiguation (NED) is the problem of finding an entity $e \in E$ such that N is a matching name for e and it represent that entity.

$$NED : n \in Names, E \text{ set of Entities} \rightarrow e \in E \tag{2.10}$$

where e is the entity corresponding to the name n.

Or we could also return a list of candidate entities enriched with the probability for each of them to be the correct one for the name n.

$$\begin{aligned} NED\_candidate : n \in Names, E \text{ set of Entities} \\ \rightarrow \{(e, p) | e \in E \text{ and } p \in [0, 1]\} \end{aligned} \tag{2.11}$$

With the proper assumptions (like for example the use of canonical forms described in section 2.2), the function necessary to solve NED problem can be reduced to the one used for solving the other problems

The function signature then will be:

$$\text{List<Entity, float>}\ entityCandidates(\text{String } n); \tag{2.12}$$

Chapter 3

Scenarios

The framework developed in this thesis is designed to work in different contexts of application.

In this section will be described the scenarios in which the thesis will be used. In particular we will take into account two main use cases.

Offline Usage: this use case groups together that kind of operations executed by the system for maintenance and updates. The main function of this scenario is *Entity Importing* (see section 3.1) which deals with the insertion of new entities in the database, and tries to reuse the knowledge already present in the system. This is the case for example of a user which wants to import its mail contacts to his / her mobile phone contacts list.

The main bricks that contributes to build its logic are *NLP* (see section 3.2), *Entity Identification* (see section 3.3), *Name Search* (see section 3.4) and *Entity Matching* (see section 3.5).

Online Usage: this use case instead includes the type of scenario that take place when the user is actively interacting the system. In particular in our case the scenarios considered are the entity *Name Search Online* (see section 3.6) and the creation of new entities. This latter case is used when the user does not find the entity he/she is looking for, and want to add information to the system. The procedure called in this situation is the same as the one described for Entity Identification (see section 3.3).

3.1 Entity Importing

This scenario describes the situation of importing an entity in a system where there are already present some initial data. The problem that we have to face in this case is to understand whether the entity we are importing is already present in the database or not, and, according to specific case, merge the two entities information or import the new object.

The input of this problem is a natural language description of an entity, from which can be extracted its information. This task is taken into account by the *NLP* algorithm (see section 3.2) which generates an object representing the considered entity, enriched with its attributes, which can include its name.

Once the entity is generated, the system needs to understand whether there are already some information in its database about the same object. If this is the case, the present entity needs to be enriched with the new information, otherwise a new entity is inserted in the database to represent the imported object. The *Entity Identification* function (see section 3.3) is the one in charge of identifying whether the entity is already present or not.

```
void entityImporting(String text) {
   Entity import = NLP(text);

   Entity e = entityIdentification(import);
   if (e != null)
      e.enrich(import)
   else
      createEntity(import)
}
```

Listing 3.1: Entity Importing

3.2 NLP

Natural Language Processing (NLP) explores how computers can be used to understand and manipulate natural language text or speech to do useful things. NLP starts analyzing texts at word level, trying to understand word by word what they are and their meaning. At this point NED and NER function come into the picture: when the NLP algorithm identifies a string as a possible name (because of its orthography, context or other techniques) the NER or NED function can be called for the identification of the meaning of that name. Accordingly to the information that we want to retrieve, we will call a different function: NER for retrieving the EType, or NED for the whole entity.

In our scenario the NLP function must generate an entity containing the attributes extracted from the text analyzed.

3.3 Entity Identification

This scenario describes the situation of understanding whether the entity we are importing is already present in the database, in order to reuse information already present in the system and then reduce heterogeneity.

Since it is too expensive to check whether the entity analyzed matches with all the other entities present in the database, the approach that will be used is search a set of candidates entities that will be tested for matching with the original entity.

The framework that will be developed in this thesis can support this situation participating in the research of the possible corresponding entity, based on its name. In this phase the system needs to find an entity which name matches with the current imported.

In listing 3.3 is summarized the pseudo code for entity identification algorithm. Given the input entity e the code calls the *Name Search (standard)* function (see section 2.2) which search in the database the entities that most likely will be a correct match for e based on their names. Then the algorithm can apply the *entityMatching* function to verify whether the entities really match. As we will see in section 3.5 this last function uses the *nameMatch* function (see section 2.1) as one of the steps to verify the match between the entities.

```
Entity entityIdentification(Entity e)
   // search for matching candidates, based on their name
   candidates = nameSearch(e.name);

   for (Entity c in candidates)
      // if they match, add information to the system entity
      if (entityMatching(e,c))
         return c;
   }
   return null;
```

Listing 3.2: Entity Identification

3.4 Offline Name search

The offline Name Search is used, as we saw in the previous sections, when we want to look for entities to match with an entity to import.

The input is a full name (unlike *Online Name Search* which accepts prefix of names), and accordingly to the context in which we are working, the system can perform the search function (to be more precise *nameSearch*, section 5.2) at different levels.

The possible contexts are:

Local Search: the search function is executed on the database of the local peer we are working on;

Central search: the database considered in this case is the one of the central service, which can include information retrieved also from external services and data sources;

Distributed Search: the context analyzed in this case is the one of a distributed system, which includes multiple peers, each of them can execute locally the search function. The extra value in this case is the possibility to search for a particular named entity in the different peers of the network.

3.5 Entity matching

One other possible application of the name matching framework is the process of matching entities. For the purposes of this thesis an entity is an abstraction of a real world object and this model is composed by different elements that are: an identifier, a name and a set of attributes.

Two entities match if they represent the same real world object, and the process to decide if it is the case needs to compare all the elements that compose them. For what concerns the comparison of the names, the $nameMatch$ 5.1 function can be used for this purpose.

```
boolean entityMatching(Entity e1, Entity e2) {
   if (nameMatch(e1.name, e2.name))
      // other function for attribute matching
      return attributeMatching(e1.attributes,e2.attributes);
   else
      return false;
}
```

Listing 3.3: Entity Identification

Since we use $nameMatch$ only as part of *entityMatching*, which is applied on names already retrieved from the database, we can assume that those entities and names, but more importantly the name tokens, are present in the system memory. This supposition implies that the function will be faster because it does not have to query the database or tokenize the name again.

3.6 Online name search

This second scenario takes place when a user wants to find an entity based on some criteria, which could be the entity name. This task could take place in different environments, which include desktop application and mobile systems. In both cases the user expects to find the correct answer (meaning the entity) as soon as possible, ideally after inserting few characters.

In this scenario the thesis work can contribute with its name search features. In particular the function that will be used is *Name Search (autocomplete)* which search for names starting from short prefixes.

Chapter 4

State of the Art

Accordingly to the taxonomy of variations defined by Bignotti et al. in [13], strings representing entity names can change in very different ways. A traditional exact string matching alone is not enough for solving the problem in analysis in this thesis.

As stated by Pfeifer et al. in [25] "Most non-linguistic similarity measures can be subdivided into three different categories:

- (plain) string similarity,
- similarity with respect to typing errors
- phonetic similarity. "

For each of the three classes, specific algorithms have been developed and evaluated, as we will describe in this section. In particular in section 4.1 will be described procedures design to manage name variations, while in section 4.2 the ones for name variants. In section 4.3 is presented a synthesis of the results obtained by other experiments about the described algorithms.

4.1 Name Variations

4.1.1 Plain String Matching

4.1.1.1 N-Grams (or Q-Grams)

N-Grams is a technique that compares two strings, finding similarities between the correspondent substrings of length n (called *grams*) and returning as similarity measure the number of identical grams. A n-gram is a continuos sequence of n items for a given sequence of text. We talk about digrams (grams of length 2), trigrams (grams of length 3) and so on.

In the application of this algorithm there are two factors to take into account:

- the gram size
- the addition of blank spaces at the beginning or the end of the word

The second parameter is used to give more importance to the initial (or ending) part of the considered word. Adding one ore more extra blank space to the beginning (or ending) of the word, generates more grams for the first (or last) character, increasing the comparison of this part of the string.

The similarity measure between two strings s_1 and s_2 obtained from this method is calculated based on the two sets of n-grams N_1 and N_2 of the words:

$$similarity\_coefficient = \frac{|N_1 \bigcap N_2|}{|N_1 \bigcup N_2|} \tag{4.1}$$

4.1.2 Algorithms for Misspelling Variations

4.1.2.1 Damerau Levenshtein metric

This metric consists in the comparison of two strings based on some typing errors. In particular the distance between two strings is determined by the number of insertion, substitution and deletion of characters necessary to transform one string into the other.

Example: S_1 = "Goettee", S_2 = "Goethe"

In this case Damerau-Levenshtein distance is equal to 2, because in order to transform S_1 into S_2 we should transform the second "t" of S_1 into a "h" and delete the ending letter "e". From this value we can derive the similarity measure which is the inverse of the distance.

In equation 4.2 is represented the dynamic programming algorithm for computing Damerau-Levenshtein metric.

$$lev_{a,b}(i,j) = \begin{cases} 0 & i = j = 0 \\ i & j = 0 \text{ and } i > 0 \\ j & i = 0 \text{ and } j > 0 \\ min \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+[a_i \neq b_j] \end{cases} & \end{cases} \tag{4.2}$$

The dynamic programming algorithm complexity is $O(n \cdot m)$ where n and m are the strings' lenght, but with there exists more efficient versions, like the one described by Berghel and Roach in [5], which can improve the complexity up to $O(n + d^2)$, where n is the lenght of the longer string and d the edit distance.

4.1.2.2 Jaro and Winkler

This metric is based on the number and order of common characters between two strings. It "accounts for insertions, deletions and transpositions. The algorithm calculates the number c of common characters (agreeing characters that are within half the length of the longer string) and the number of transpositions $T_{s',t'}$" (Christen) [10].

The similarity measure is calculated as:

$$Jaro(s,t) = \frac{1}{3}\left(\frac{|c|}{|s|} + \frac{|c|}{|t|} + \frac{|c| - T_{s',t'}}{|c|}\right) \tag{4.3}$$

A variant of this metric is the one proposed by *Winkler*, which takes also in account the length P of the longest common prefix of S and T: it improves Jaro metric "by applying ideas based on empirical studies which found that fewer errors typically occur at the beginning of names. Winkler algorithm therefore increases the Jaro similarity measure for agreeing initial characters" (Christen) [10].

$$P' = max(P, 4)$$
$$Jaro\_Winkler(s,t) = Jaro(s,t) + (P'/10) \cdot (1 - Jaro(s,t))$$

4.1.2.3 Skeleton and Omission key

This two very similar techniques were developed in order to cope with typing errors.

The main idea behind them is that "consonant carry more information than vowels" (Pfeifer) [25]. The two algorithms transform each string into a new one containing the same letter, but organized in a different order: they put the consonant at beginning of the word, followed by the remaining vowels.

Omission Key algorithm in addition, orders the constant according to an heuristic for the frequency with which a specific letter is omitted (RSTNLCHDPGMFBYWVZXQKJ where R is the most omitted).

If for example we consider the name "Android" it will be transformed with Skeleton algorithm in "Ndrdaoi" and by Omission Key algorithm in "Ddnraoi".

Once the algorithm has transformed the name into the code, as a similarity measure an alphabetically ordered list of the keys is constructed.

4.1.2.4 Skip-gram (Keskustalo, Pirkola, Visala, Leppänen and Järvelin [18])

Skip-grams are based on the idea of not only forming digrams (2-grams) of two adjacent characters, but also digrams that skip one or more characters. They are proved to improve the matching of cross lingual spelling variants.

4.1.3 Format Variations

4.1.3.1 Longest Common Substring (LCS)

The LCS problem consist in finding the longest subsequence of character in (generally) two string. The algorithm used to solve this problem is based on dynamic programming: given two strings $X(x_1, x_2, .., x_m)$ and $Y = (y_1, y_2, .., y_n)$ and define as X_i with $i < m$ (respectively Y_i) the sequence $x_1, x_2, .., x_i$ and is called prefix.

$$LCS(X_i, Y_j) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ (LCS(X_{i-1}, Y_{j-1}), x_i) & \text{if } x_i = y_i \\ longest(LCS(X_{i-1}, Y_j), LCS(X_i, Y_{j-1})) & \text{if } x_i \neq y_j \end{cases} \tag{4.4}$$

LCS algorithm computational complexity for two strings of length n and m respectively is $O(n \times m)$.

This algorithm is used in the string matching field in order to "find the longest common subsequence and then from this compare the difference". (Hall and Dowling [16]).

4.1.3.2 Jaccard [6]

Jaccard metric is design for names that contain similar words but that appear in different order. For example "Mario Rossi" and "Rossi Mario" represent the same person, but if we analyze them with another string matching algorithm which compute the number of transformation necessary (for example Levenshtein), the two strings will result to be very distant.

Jaccard algorithm converts each name in a subset of tokens, where each token is a single word), and compare the similarity on the multisets. Given two names s and t and respectively define as S and T the multiset of token for s ad t, Jaccard similarity measure is computed as:

$$Jaccard = \frac{|S \bigcap T|}{|S \bigcup T|} \tag{4.5}$$

4.1.3.3 Standard format variations

Especially when we consider author names, there are different format standardization that depends on the bibliographic style followed by the publisher.

Table 4.1 from [13] shows how bibliographic style (on the first column) can affect and transform authors name into string syntactically different and difficult to match through naïve algorithm.

Style	**Mary-Claire van Leunen**	**Oren Patashnik**	**Charles Louis de la Vallee Poussin**
ieeetr, phjcp, abbrv	M.-C. van Leunen	O. Patashnik	C. L. de la Vallee Poussin
unsrt, IEEE, plain	Mary-Claire van Leunen	Oren Patashnik	Charles Louis de la Vallee Poussin
ama	Leunen Mary-Claire	Patashnik Oren	Vallee Poussin Charles Louis
cj, nar, acm	van Leunen, M.-C.	Patashnik, O.	de la Vallee Poussin, C. L.

Table 4.1: Authors' name variation depending on bibliographic style, from [13]

If on one side the variety of format convention adopted by publisher generates many name format variations, on the other side each bibliographic stile for one name can be computed automatically following the respective rules.

4.1.3.4 Name Parsing

Another possible approach to the different composition of names is the one described by International Business Machines (IBM) in [17]. They propose an analytical name matching system which parses and recognizes the different parts that combined form a name. Based on the generated parsing tree , and a set of trigger words (in this cased called *name tokens*) the algorithm is then able to match names which are variations (both mono and multilingual) of the same original name.

The tool at the moment is able to parse and match only names which refer to entity belonging to EType *Person* or *Organization*.

4.1.4 Phonetic Algorithms

4.1.4.1 Soundex [25]

This algorithm, accordingly to some specific rules, transform each word into a code composed by the initial letter followed by a list of digits, which represent the pronunciation of the string. This allow to compare word misspelled because of lack of knowledge of the user who inserted the name, or cultural difference. Take for example the surname "Bayer", this could be typed by a German speaking person as "Baier". Accordingly to Soundex this two names, which have different string representation, share the same soundex code.

The main problem of soundex in name matching is the lack of support for multilingual words. The algorithm is developed for English names only and has to be tuned for other languages.

4.1.4.1.1 How to convert a name in a Soundex code:

1. Remove all vowels, the consonants ‘H’, ‘W’ and ‘Y’ and all duplicate consecutive characters. The first letter is always left unaltered.

2. Create the Soundex code by concatenating the first letter with the following three letters replaced by their numeric code according to Table 4.2

Soundex									
B	F	P	V					$\rightarrow$	1
C	G	J	K	Q	S	X	Z	$\rightarrow$	2
D	T							$\rightarrow$	3
L								$\rightarrow$	4
M	N							$\rightarrow$	5
R								$\rightarrow$	6

Table 4.2: Soundex conversion table

4.1.4.2 Phonix [25]

Phonix is a phonetic algorithm that operates string transformation into specific codes, but is far more complex than Soundex.

The main difference with the previous algorithm is that it gives more importance to the ending sound of the word because this part is considered in a specific rule, and it also operates a phonetic substitution.

As Soundex, this algorithm performs well only with English words.

4.1.4.3 Stabilization Algorithm

Stabilization Algorithm is the class of procedure that transform strings accordingly to specific of rules. Its aim is to overcome phonetic and orthographic error in names.

The procedure is similar to the one applied by Soundex and Phonix: substitute one or a sequence of letters with a specific one that represents the class of letters with a similar sound. Example from Holloway [15]:

PH	$\rightarrow$	F
M	$\rightarrow$	N
A,E,I,O,U	$\rightarrow$	A

The use of this class of algorithms should be limited because "typically, the more characters which are removed or stabilized by an algorithm, the more error and variation is overcome" [15]. One other consequence that the application of this technique causes is the variation of frequency of names, which means that different names (that do not represent the same entity, but are similar) are transformed into the same string. Example: SMITH, SMITHE, SMITHY, may all become SNT.

Specific stabilization algorithms can be used to deal with the problem of language specific character (like accents) or sounds ("ch" instead of "k"). Pouliquen et al. in [28] are describe a set of substitution rules developed in order to transform names from texts in different languages:

accented character	$\rightarrow$	non-accented equivalent
double consonant	$\rightarrow$	single consonant
ou	$\rightarrow$	u
wl(beginning of name)	$\rightarrow$	vl
ow, ew(end of name)	$\rightarrow$	ov, ev
ck	$\rightarrow$	k
ph	$\rightarrow$	f
$\check{z}$	$\rightarrow$	j
$\check{s}$	$\rightarrow$	sh

In this case their work focuses on east Europe languages, but we should also consider german letter which include vowels with umlaut (ö, ü) or ß.

Plus in their work they use also a specific transliteration for non-latin scripts as Greek, Russian and Arabic, in order to represent their particular character into the latin alphabet.

4.1.5 Information Retrieval Techniques for Spelling Correction

A further technique that can be adopted in the spelling correction algorithms is imported from the information retrieval field.

As defined by Norvig in [22] the task of spelling correction consists in "given a typed word, w, determine what word c was most likely intended." The types of error that transform the word c into the typed w could be several, and several are the different techniques for detecting them as we saw in the previous sections about name variations. From Norvig [22] we can import a formula that allows us to generalize the problem:

$$argmax_c P(c|w) = argmax_c P(w|c)P(c) \tag{4.6}$$

where $P(w|c)$ is the probability that the user would type w when c is intended, and $P(c)$ is the probability that c is the intended word.

These probability can be estimated based on *statistics*. Statistics can be calculated from a dataset (in this case a text) counting for each different word its number of occurrences. The result is a map of string-number which tells us which is the popularity of each word. This type of statistics can be used for choosing between two possible correction which is the most probable, according to some principles.

As Manning explains in [20] in spelling correction there are two main principles: when looking for alternatives to a word for correcting errors, the algorithm should choose between the *nearest* term, determined by a measure of similarity, and the *most common*, based on the number of occurrences of a term in the examined dataset.

For what concerns the nearness of two words, we already discussed in section 4.1.1, 4.1.2, 4.1.3 and 4.1.4 which is the state of the art for string matching algorithms. In addition Manning [20] and Norvig [22] propose two improvements to the edit distance (or Dameray-Levenshtein 4.1.2.1) algorithm, related to the information retrieval field:

- "set weights in this way depending on the likelihood of letters substituting for each other" [20], for example assign a higher weight to substitution of letters closed on the keyboard, and lower to the ones which are far from each other.

- use a *statistic of misspellings*, from which is possible to extract the probability of a specific error (for example the probability that the user types "sch" instead of "ch"), for computing the probability of each candidate corrected word in relation with the transformation done from the original word. For an example see table 4.3.

w	**c**	**w | c**	**P(w | c)**	**P(c)**	10^9 **P(w | c) P(c)**
thew	the	ew | e	.000007	.02	144.
thew	thew		.95	.00000009	90.
thew	thaw	e | a	.001	.0000007	0.7
thew	threw	h | hr	.000008	.000004	0.03
thew	thwe	ew | we	.000003	.00000004	0.0001

Table 4.3: Example of edit distance integrated with misspelling statistics, from [22]

Both the two improvements are related to the computation of statistics from some source, but in this case standard statistics that report the frequency of a word are not sufficient: we need to have statistics about the errors present in the words.

Creating a statistics of misspellings is more complicated than just counting occurrences of words. As described in [22], first of all we need a list of pair of words $<w,c>$ (with w the typed word and c the indented word), and from this we should extract the generic error that is done from w to c. For example, consider the couple $<dag, dog>$. In order to transform *dag* in *dog* the letter *a* should be substituted with a *o*. Then the error done here is a|o (read "typed a instead of o"). The new statistic obtained trough this analysis will contain couples of string-number, where in this case the string is not a word but the type of error done.

In some cases these kind of statistics are not enough detailed for detecting the right word c intended. Consider this example (from [22]): the word "they're" consider singularly is correct, but if we consider the sentence "in they're words" should be corrected into "their". A possible solution would be to create statistics of multiple words, which means, to consider the occurrences of sequences of words (two or more). This answer has a big problem as reported by both [22, 20], which is the size of the statistic created, and the computational complexity of the computation.

In order to improve the search of a word in the statistic dataset, Manning [20] proposes an approach which has empirical support and consists in integrating different algorithms that deals with string matching. First they use k-gram (see section 4.1.1.1) and Jaccard (see section 4.1.3.2) to extract the set of candidate word which are a possible correction for the query. Then they compute the edit distance (see section 4.1.2.1) for each term in this set with the query, and select the terms with the smaller distance. A similar approach is proposed by Lalwani et al. in [4] where the n-gram distance between two names is computed using the ascii sum of the grams generated from the names. In this case the comparison is faster, since it requires only a integer difference.

This technique will be considered in the algorithm design for the thesis.

4.2 Name Variants

4.2.1 Translations

One of the main focus of this work is name matching with respect to multilingual variations. Differently from monolingual variations (as misspellings, format) which slightly affect the string composition (character substituted, or switched) and can be managed by phonetic or string matching algorithm, multilingual variations can completely change the appearance and the pronunciation of the word, making the cited algorithms impossible to use in the general case.

The first thing to take in account is the management of the translation. The problem has been considered also by Pouliquen et al. [28], where a specific tool for named entity recognition in a multilingual environment is described. Their solution, besides the phonetic variation and canonization of names, consists in storing for each name all its known variant each time that a new one is discovered.

Similarly, there are available dataset and dictionaries where there is reported a correspondence of names with their known translations in the other languages. One example is the name translator tool offered by the website "Behind the name" [3].

Besides the problem of translation itself, for non-Latin scripts we should also consider the problem of a different alphabet: Unicode system can help us, but cannot do all the work.

The stabilization algorithm proposed by Pouliquen et al. [28] has been already used within their NER software for person names.

Another proposal for dealing with multinational names has been proposed by Oshika, Evans, Machi and Tom in [23], which approach is to decomposed the name search problem into two main components:

- "Language classification techniques to identify the source language for a given query name, and
- Name association techniques, once a source language for a name is known, to exploit language-specific rules to generate variants of a name due to spelling variation, bad transcriptions, nicknames, and other name conventions."

Other works have been developed about the first step of identifying the source language of a name. Another example is proposed by Dunning in [12] which describes a tool able to "learn with just few training to identify the language of a 20 character length string." This would be useful for dealing with multilingual variation of the names, but it's not suitable for two main reasons: names length is not always at least 20 characters (if we consider just first names, usually they are at most 10 characters), and the possibility of misspellings or phonetic variations, that this algorithm is not able to deal with. Anyway this techniques should be take into account for some particular cases, when name has been standardized from misspellings, and are sufficiently long.

4.2.2 Other Variants

For name variant we mean name alteration as nicknames, aliases, pen names ([13]) that can modify completely the appearance of the original name. The algorithms listed previously are then not able to match a name with its variants based only on string or phonetic similarity.

Instead, an approach similar to the one for multilingual should be chosen. As suggested by Pouliquen et al. [28] a dictionary structure, where names are stored together with all their name variant, is advised.

Another option to address the problem of retrieving and managing name variants is proposed by Bunescu and Pasca [9]. In this case they try to disambiguate named entity using the english version of Wikipedia web site as a tool for relating entities and their possible names.

They state that “there is a many-to-many correspondence between names and entities. This relation is captured in Wikipedia through redirect and disambiguation pages”.

For each alternative name that can be used to refer to an entity in Wikipedia, there exists a *redirect page* (which title is that specific alternative name) which redirect to the main article for the corresponding entity. For example, if we search for “Reginald Kenneth Dwight” we will be redirected to the page of “Elton John” (the singer), because the query corresponds to one of the known name variant (in this case his birth name) for the entity “Elton John”.

Disambiguation pages instead work the other way around: they represent the other side of the relation many-to-many between entities and names. There is a disambiguation page for each name that can denote more than one entity. For example if we search for “Newton” the website is not able just from that string to understand to which entity we refer. “Newton” can denote several people (for example *Isaac Newton*, the physicist), the force unit (named after the physicist), a town (*Newton*, South Australia), and many other entities. The system then will show to us a disambiguation page where all the entities contained in Wikipedia that can be referred by the name “Newton” are listed.

4.3 Evaluations from papers

The listed algorithms have been evaluated with testing dataset in order to understand which performs better in terms of precision and recall in the field of personal name matching. In table 4.4 is the complete list of the dataset used in the following evaluations.

The two evaluation cited here has been performed over personal name, which are not the only context of this thesis (we consider also location, event, organizations).

In the first evaluation from Pfeifer [25] they tried to evaluate each algorithm with some variations. From the result obtained, they then choose the best version of each algorithm, and tried to combine in order to obtained improved procedures.

In this paper Soundex resulted to be the worst in phonetic encoding algorithms, and this result is caused by the missing phonetic substitution that the other phonetic algorithms have.

Phonix was tested with 3 variants: short code (with no ending sound), long code (with no ending sound) and short code (with ending sound). The shorter code outperformed the longer version because of its too strict accuracy for the creation of code. Moreover the ending sound performed better than the version that does not consider it.

Between the algorithms for matching with respect to typing errors, Damerau-Levenshtein and Skeleton Key performed with similar result, and they are both better than Omission Key.

For what concern n-grams they noticed that "for increasing n, performance is decreasing. [..] This behavior is independent of the number of blank spaces used." Digrams performed also better than all the phonix algorithm versions.

Similarly [10] shows that pattern matching algorithms outperform phonetic encoding algorithms. In this work they tested also Jaro and Winkler technique which performed well for personal name matching.

In conclusions, for not-combined algorithms the best one accordingly to precision-recall measure for [25] is n-grams with n=2, while for [10] no specific algorithm performed always better than the others, but for the complete dataset used the three best are Jaro, Winkler, Damerau-Levenshtein, immediately followed by Phonix.

The only different result between these papers is about the combination of algorithms: while Pfeifer [25] finds the best results with the combination of Digrams, Skeleton and Phonix (with ending sound), [10] obtained in its result that the two combination of techniques tested (edit distance with soundex, match syllables encoded with phonex) performed worse than the other algorithms.

Dataset Name	**Used by**	**EType**	**Description**	**Location**	**Size**
AP	Pfeifer [25]	Person	AP Newswire (1989)	TREC (Text REtrieval Conference), not publicly available	
BIB	Pfeifer [25]	Person	Literature database (1993)	not publicly available	
CACM	Pfeifer [25]	Person	ACM abstracts	SMART, not publicly available	
FR	Pfeifer [25]	Person	Federal Register (1989)	TREC, not publicly available	
TELEFON	Pfeifer [25]	Person	Phone book database of the University of Dortmund (1993),	not publicly available	
WSJ	Pfeifer [25]	Person	Wall Street Journal (1988)	TREC, not publicly available	
ZIFF	Pfeifer [25]	Person	Information from Computer Select Disks (Ziff-Davis-Publishing)	TREC, not publicly available	
Complete	Pfeifer [25]	Person	Combination of all previous databases		14000 names
Midwives (given names)	Christen [10]	Person	Given names of midwives	NSW Department of Health [1]	15,233
Midwives (surnames)	Christen [10]	Person	Surnames of midwives	NSW	14,180
Midwives (full names)	Christen [10]	Person	Full names of midwives	NSW	36,614

[1]Centre for Epidemiology and Research, NSW Department of Health. New South Wales mothers and babies 2001, http://www.health.nsw.gov.au

Dataset Name	**Used by**	**EType**	**Description**	**Location**	**Size**
HeiNER database		All	Set of multilingual lexical Named Entity resource extracted from Wikipedia	HeiNER[2]	1.5 million entities
Eventseer		Event, Person, Organization	Academic Events Agenda	http://eventseer.net/	19,901 events, 1,033,569 people, 3,194 organizations
Devdir		Organization	Directory of Development Organizations	http://www.devdir.org	70.000 org
Universities Worldwide		Organization	database of universities	http://univ.cc	8991 universities, 204 countries
Union List of Artist Names		Person	artist names	ULAN [3]	638,900 names
GeoNames		Location	geographical database	GeoNames[4]	7 million entities
Us Census 2000		Person	surnames from 2000 US census	US census website [5]	151671 surnames
Us Census 1990		Person	surnames from 1990 US census	US census website [6]	88000 surnames
US Census 19th century		Person	surnames from transcription of 19th century US census	Us census website [7]	

[2]http://heiner.cl.uni-heidelberg.de/index.shtml
[3]http://www.getty.edu/research/tools/vocabularies/ulan/index.html
[4]http://www.geonames.org
[5]http://www.census.gov/genealogy/www/data/2000surnames/index.html
[6]http://www.census.gov/genealogy/www/data/1990surnames/index.html
[7]http://www.us-census.org/inventory/

Dataset Name	**Used by**	**EType**	**Description**	**Location**	**Size**
Enron Corpus	Minkov, Cohen, Ng [21]	Person	database of emails from Enron Corporation	Enron corupus website[8]	600000 emails, 158 employees
FOAF		Person	Friend of a Friend dataset	UMBC website[9]	201,612 RDF triples
HeiNER		All	HeiNER database, also contains translations and statistics	http://heiner.cl.uni-heidelberg.de	1.5 million entities

Table 4.4: Dataset for Evaluation

[8]http://sgi.nu/enron/corpora.php
[9]http://ebiquity.umbc.edu/resource/html/id/82/foafPub-dataset

Chapter 5

Algorithm Design

In this section will be given a more detailed specification of the functions described in 2, in order to provide an implementation proposal for each problem. In particular *name matching* function (defined in section 2.1) will be described in section 5.1, *name search* function (see section 2.2) is presented in section 5.2 and 5.3, *named entity recognition* (see section 2.3) is illustrated in section 5.4, and *named entity disambiguation* function (see section 2.4) is defined in section 5.5.

5.1 Name Match Function

In this section will be analyzed the design and implementation choices made for the development of name matching function. As defined in equation 2.2, the function should take as input two names, $Name_1$ and $Name_2$, with the eType, and return a number, between 0 and 1, which represents the distance (see section 6) between $Name_1$ and $Name_2$.

The function assumes that the two names belong to two entities which eTypes are compatible. For example the eTypes “Employee” and “Professor” are compatible, while “Town” and “Student” cannot match. The eType provided as input represent the most abstract eType to which both names belong, and, as defined in section 2.1, this can be *Person*, *Location*, *Organization* or *Event*. This eType will be use in the implementation of some functions in order to make the analysis of the names more specific and accurate, based on the structure and properties typical of the considered eType.

5.1.1 Algorithms used

The *nameMatch* function will use different matching algorithms (presented in section 4) combined and ordered properly for producing a distance measure in an efficient way. In order to minimize the time necessary to determine whether two names match, the algorithms execution order should be based on their probability to find more often a correct match, which means that the algorithm which detect the type of match that occurs most often will be applied first. The function will stop as soon as one strategy finds out that the names match with a distance lower than a predefined threshold

The main functions that will be used for name matching are specified in listing 5.1

```
// 1. check string exact equality
boolean stringEquality(String name1, String name2);

// 2. check string matching (misspellings)
float stringMatching(String name1, String name2, EType eType);

// 3. check dictionary for (name1,name2) or variations
float dictionary(String name1, String name2, EType eType);

// 4. match reordered tokens (considers misspellings)
float tokenReorderingMatching(String[] tokens1, String[] tokens2, EType eType
    );

// 5. match all tokens for maximizing the matching
float tokenRecombineMatching(String[] tokens1, String[] tokens2, EType eType)
    ;
```

Listing 5.1: Main Functions used for approaching Name Matching problem

The function can be combined in order to be executed by different threads simultaneously without interfering with each other. In listing 5.2 are specified the design for the three proposed threads that combine such functions.

The details about the design of these functions will be given in the following sections.

As we can see functions 1 (*stringEquality*) and 2 (*stringMatching*) are compose together because they both check string variations without querying the name database. Moreover *stringMatching* is applied only if the *stringEquality* function fails, in order to avoid meaningless operation and save time in the computation.

Thread number 2 instead check whether in the name database there is a correspondence for the two names, considering also their variation (like misspellings).

Thread number 3 splits the names in tokens and applies two functions on those elements. The first one *tokenReorderingMatching* reorder the tokens accordingly to their lexical graphical order and combines the single token similarity. The second one *tokenRecombineMatching* is applied only when the reordering function did not find an acceptable match, and match the tokens according to the name field their represent, or in case data are not available, recombine tokens for obtaining the highest similarity.

```
/* Thread t1() */ {
   if (stringEquality(name1, name2))            // 1.
```

```
        return 1;
    else
        return stringMatching(name1, name2, eType); // 2.
}

/* Thread t2() */ {
    return dictionary(name1, name2, eType);             // 3.
}

/* Thread t3() */ {
    tokens1 = tokenize(name1);
    tokens2 = tokenize(name2);

    sim1 = tokenReorderingMatching(tokens1, tokens2, eType); // 4.
    if (sim1 > T)
        return sim1;
    else
        return tokenRecombineMatching(tokens1, tokens2, eType); // 5.
}
```

Listing 5.2: Matching Threads

5.1.2 Function High level design

Name Match function (see listing 5.3) needs to properly combine the threads described in section 5.1.1. Since they can be run at the same time, the first step is to start executing all of them. Even if they starts at the same time, we are not sure that they complete their execution simultaneously. We need to define a policy for accepting the returned value in order to optimizing the average time complexity.

As specified in section 5.1.1, the threads need to be combined for retrieving the most recurrent matching as first. In this case we consider the case of two name, one the misspelling on the other, as the most frequent, followed by the recombination of tokens, and as third the possibility that the two name variant are queried.

Because of this the name matching function first waits for the result of thread 1, then thread 3 and if neither of them return an acceptable value, then it waits also for thread 2 and combines the obtained results in a proper way.

In the following paragraphs we will describe the design of the 5 functions of listing 5.1.

```
float nameMatch(String name1, String name2, EType eType) {
    t1.start();
    t2.start();
    t3.start();

    waitforresult(t1);
    if(t1.result > T)
       return t1.result;

    waitforresult(t3);
    if (t3.result > T)
       return t3.result;
    else
       waitforresult(t2);
       combine(t1.result, t2.result, t3.result);
}
```

Listing 5.3: Name Matching function structure

5.1.2.1 String Equality

The simplest strategy that we can apply is the *plain string comparison*, which consists in the verification that $Name_1$ and $Name_2$ are exactly the same string. This algorithm has in the worst case (two string which have the same characters, except the last one) $O(n)$ complexity, where n is the length of the shortest string between $Name_1$ and $Name_2$.

```
// 1.
boolean stringEquality(String name1, String name2) {
   return name1 equals name2;
}
```

Listing 5.4: String Equality Function

5.1.2.2 String Matching

This function focuses on the matching complete names, with respect to their misspellings variations.

The algorithm should check whether the two names match with respect to misspellings contained inside them. As reported in section 4.1.2 there are plenty of algorithms available for detecting whether two strings are misspelling of each other. The problem is to perform this computations in a quick and effective way, combining the different strategies.

```
// 2.
float stringMatching(String name1, String name2, EType eType) {
     // if many common char, perform misspelling check
     if ((diffLen(name1,name2) < L) AND (commonChar(name1,name2) > T)) {
         sim = misspellingSimilarity(name1,name2);
         if (sim > T)
             return sim;
      else return 0;
     }
```

```
    }
```

Listing 5.5: String Matching Function

Most of the algorithms for string matching have complexity of $O(n \cdot m)$ where n and m are the two strings length, and in some cases the matching between the two names cannot be found by these type of algorithms because the two strings are variant name for the same entity, but for what concerns the typographic appearance are very different. We should then decide whether it is the case to perform the misspellings verification or not, in a way less computationally complex than the matching itself.

The first observation to do is that two strings which lengths differs of a number k of characters, will always have an edit distance $\geq k$. Because of this, before performing the edit distance algorithm, the function checks the length difference for the names.

The length control is not the only strategy that we can apply. One other possible choice could be to count how many single characters the strings share, regardless the order in which they appear. In listing 5.6 is reported the pseudo code for one possible implementation of this function. It uses a Hash Table for storing the number of occurrences for each character used in the names, and then compare the differences. In this case the complexity would be even lower, since insert and search operation for a hash table (of sufficient size) are $O(1)$. The overall complexity is then $O(n + m + alphabet) = O(n + m)$

```
  float commonChar(String name1, String name2) {
    // arrays of alphabet size for storing the char occurrences
    int[] a; int[] b;

    for (c in name1)
       a[c]++;
    for (c in name2)
       b[c]++;

    diff = 0;
    for i in 0,alphabet_size
       diff += abs(a[i] - b[i]);

    return (name1.length + name2.length - diff) / 2;
  }
```

Listing 5.6: Common Char function with Hashing

If the length difference is acceptable and the measure of common chars provided by this function is higher than a predefined threshold, then the *misspellingsSimilarity* function is called.

This last function is in charge of verifying whether the names match with respect to misspellings, and combines different string matching strategy for computing the distance (for what concerns typing errors) between the names.

The main algorithm that will be used in this function is the Damerau-Levenshtein (or Edit Distance, see section 4.1.2.1), which provides a distance measure defined by the number of edit operation necessary to transform one string into the other. Another algorithm that is used is Jaro-Winkler (see section 4.1.2.2), which works similarly to Edit Distance, but gives more emphasis to the beginning of the word: this is important in the matching of names, because, as reported by Pollock and Zamora in [27], fewer errors typically occur at the beginning of names.

5.1.2.3 Dictionary Lookup

The first control that this function should do is to check whether the system already knows that $Name_1$ and $Name_2$ match. This can be done with a search within a database where are stored pairs of names, which are known matches, and the respective distance value. The function that is responsible of this task is called *dictionaryLookup*.

```
// 3.
float dictionary(String name1, String name2, EType eType) {
     if (dictionaryLookup(name1, name2, eType)) // check exact tuple (name1,
         name2,eType)
       return 1;
    else
    return nameVariantSimilarity(name1, name2, eType); // check variations on
        alternative names
}
```

Listing 5.7: Dictionary Function with variations

In case a pair $(Name_1, Name_2)$ for the input EType is found by this function, then the algorithm is sure that the two names match, and can immediately return the similarity value retrieved from the database.

Depending on the database design the stored matches could be of different type. Our proposal is to store couple of names which are alternative names for the same entity and name translations. In section 7 will be described in more details the database architecture.

In case *dictionaryLookup* does not find a match for the input names in the system database, $nameVariantSimilarity$ function is called. This function checks for variations on the alternative names of $Name_1$ reapplying the string similarity thread (thread nr 1, see section 5.1.2.2 and 5.1.2.1).

This choice is made in order to detect matching which request both a variant and a misspelling variation. For example consider the names:

$$Name_1 = \textit{New York City}\ \text{[en]} \quad Name_2 = \textit{La Grande Mele}\text{[it]}$$

if we want to go from $Name_1$ to $Name_2$ we need to pass through one other name:

$$\text{New York City} \overset{variant}{\rightarrow} \text{La Grande Mela} \overset{misspelling}{\rightarrow} \text{La Grande Mele}$$

This process needs two steps to reach the target name, and comparing just the misspellings between *New York City* and *La Grande Mele*, or their variant would not be enough to detect the match.

Because some of the variants are translation of the name, one further strategy that can be applied in the search of candidates is the normalization of names as suggested in section 4.1.4.3. The conversion of names according to a set of stabilization rules can help to deal the multilingual problem and find a translation match for the considered names.

The translation function could be more accurate in case it is provided also with the source language of the names. This would make the algorithm more precise and the dictionary lookup faster: the research would be reduced to the set of names belonging to a particular language.

```
float variantSimilarity(String name1, String name2, EType eType) {
    // retrieves variant(n1)
    variant(name1);

    // order list of variant accordingly to some criteria for analyzing
       first variants which are highly probable to match name2
    orderVariant();

    // for each variant of n1
    for n in variant(name1) {
        // check plain string sim
        if (n == name2)
            return 1;
        else {
        // reapply t1
        t1.start();
        return t1.result;
        }
    }
    return 0;
}
```

Listing 5.8: Variations on Dictionary

5.1.2.4 Divide in Tokens

Before defining the functions that analyze the name tokens we need to describe how tokens are generated.

In the easiest situation all the tokens are separated by a blank space, but this is not always the case. For example, because of misspellings, two words composing the name could be attached together, or in case of acronyms, separated by punctuation.

We need a function which, given a string representing the name, is able to divide it into words that represent the name tokens. This function, called $tokenizeNames$, should be able to separate the strings considering both the punctuation case and the missing space case.

For what concerns the punctuation case, the algorithm for facing this problem is not complicated and can be implemented with the tokenize functions provided by the programming language used (for example *StringTokenizer* in Java). The missing space case instead is more complicated to address. We need to understand if the string represents a single or multiple tokens, and, in this latter case, figure out where to put the missing space. Placing the blank space implies to know which combination of sub string composing that word is the one that correctly represents the tokens. For taking this decision the algorithm should compute a probability for each of these combinations, and this probability should be retrieved from statistics on token occurrences, which need to be dependent on the field that a token represent (we need statistics separated for name, surnames, towns, trigger words, etc). Since the problem of detecting two tokens not separated by punctuation is computationally expensive, it will not be addressed at this point of the algorithm.

According to the name design specific for each eType described in section 7.2 we will have that in many cases name tokens are already precomputed in the database.

5.1.2.5 Token Reordering

The token reordering function in defined in order to take into account format variation. The format of a name is highly dependent on the eType we are considering: there are specific rules for generating format variations for authors names, accordingly to the bibliographic style adopted (see section 4.1.3.3), specific order for the representation of an address depending on the country, and particular acronyms that are used in organization names.

The algorithm necessary to address properly all this format variation needs to know the eType considered and for each word that compose a name which part of the name it represents. This means that we need to split the name in tokens. Since the field recognition is an expensive task (if not provided by the name structure see section 7.2), and the format variation can be often detected without using it, in this function we will focus on matching tokens accordingly to their lexical graphic order, while the field analysis will be considered in the $tokenRecombineMatching$ (see paragraph 5.1.2.6).

In this phase of the algorithm we can anyway check some high level format variations, like word position swapping, use of abbreviations or different formats for compound names.

This function arranges tokens by alphabetic order, and generates pairs of tokens based on this order. In this way we can detect tokens which match with respect to misspellings or simple format variations.

Example: $Name_1$ = "Dott. Rossi Mario", $Name_2$ = "Dottor Maro Rossi". The pairs of tokens will be $(Dott., Dottor), (Maro, Mario), (Rossi, Rossi)$. We can easily see that the two names match if we rearrange tokens in alphabetical order.

This approach is not able to detect all the matching: for example tokens which are variant of each other do not always share the first letters, and then this ordering would not pair this kind of tokens together.

Once the pairs of tokens are generated, it comes the moment to check whether the two tokens match. This task is managed by the application of thread 1 (string similarity) and thread 2 (dictionary lookup) on the pairs of tokens. In this way we are able to verify name matching considering token misspellings and different position. The overall similarity measure for the two original names in equation 5.1 is defined in details in section 6.

$$\sum_{(t_1,t_2)\in pairs} sim_{string}(t_1,t_2) \cdot \frac{|t_2|}{|Name_2|} \tag{5.1}$$

```
// 4.
float tokenReordering(String[] tokens1, String[] tokens2, EType eType) {
   // combines tokens from name1 with ones from name2 in a proper way for
       reordering
   Map<String,String> tokenPairs = combine(tokens1, tokens2);

   for t1,t2 in tokenPairs{
      // apply T1 on (t1,t2)
      sim = combine(T1.result, T2.result);

      // combine similarity of tokens, normalized on their length
      simTotal += sim * (t2.size / name2.size);
   }
   return simTotal;
}
```

Listing 5.9: Token Reordering function

5.1.2.6 Token Combination

The *tokenRecombineMatch* is called only if the previous token analysis did not return a match for the input names. Because this is the last function applied, we can use in its definition techniques which are too complex for the previous functions. In particular there are two possible approaches for recombining the tokens:

Field Recognition order tokens based on the name field they represent. The system then will need to understand to which field each token corresponds. This computation is based on statistics and on the structure of each name, which are both strong eType dependent.

Maximize Match generate pairs of tokens that maximize the name similarity. This means that we have to compute all the possible permutation of tokens. This approach will be used only when no field recognition techniques are available.

5.2 Standard Search

In this section will be described the design and implementation choices made for the development of the name search functions.

As defined in section 2.2 in order to address this problem we need to develop two different functions: one for dealing with auto completion search and one for the standard search. In equations 2.6 and 2.5 are reported the two method signature for the functions that we are going to define.

In this section will be described the approach adopted for the standard search, while the auto completion case will be treated in section 5.3.

There are two possible approaches available for addressing the name search problem. The first is *query expansion*: given a name, return a list of strings which represent alternative names for the one given as input. The result will be used for enlarging the research for a named entity (for example in distributed search). The second one is *entity name search*: given a name, return a list of entities which have the input string as one of their alternative names. In this case the entity referral is returned, and from this the system can retrieve the other entity names, the eType, plus a description.

At the moment the proposed algorithm follows the former approach, but in its implementation it actually retrieves the entities related to the input name, and from them finds the alternative names.

As defined in equation 2.4 the function for performing the standard search takes as input a string representing a name, and should return a set of names (enriched with a ranking value) which correspond to the entities of which the input string is a possible alternative name.

The function should support the retrieval of entity names with respect to variations (misspellings and format modification) and variant (alternative names and translations).

The string in input represents a name, but we are not sure about this string to be a correct spelled name, contained in the database. We could encounter for example the case in which the user has typed “The Big Appel” instead of “The Big Apple”, and, without spelling correction or other methodologies, we can not retrieve the related name “New York”, because the input string is not present in the data source.

The function (reported in listing 5.10) uses the *entityNameSearch* function which retrieves the Global Unique Identifier (GUID) of entities which have name variations or variant of the input query. The *nameSearch* function then returns for each on these entities the corresponding list of names. As we will see in section 5.4 and 5.5 the same function will be use for the NER and NED algorithm which will instead retrieve the eType and the other information for each entity.

The *entityNameSearch* function uses 4 main functions, which address different type of variations. The details about these algorithms will be given in the following paragraphs.

```
 List<String, float> nameSearch(String input) {
     // retrieve entities which has "input" (or its variations) as name
     list = entityNameSearch(input);

   // return the list of names
     return retrieveNames(list);
 }
```

Listing 5.10: Name Search Standard

```
List<GUID,float> entityNameSearch(String input) {
     // 1.
   List<GUID,float> listEquals = searchEquals(input);

   String[] tokens = generateTokens(input); // tokens used also by other
      functions

   // 2.
   List<GUID,float> listReordering = searchReordering(tokens);

   // 3.
   List<GUID,float> listMisspellings = searchMisspellings(input);

   // 4.
   List<GUID,float> listToken = searchToken(tokens);

   return union(listEquals,listReordering, listMisspellings, listToken);
}
```

Listing 5.11: Entity Name Search

5.2.0.7 Plain Dictionary Search

This function looks in the database table for entities which have the input name as one of their possible name. For each of these entities, the algorithm retrieves their other possible names (or only their canonical one).

In the implementation proposed in listing 5.12 the function assumes a table structure which contains for each tuple its Id, the name, and the GUID for the entity referred. It first retrieves all the GUID of entities which has *input* as name, and then all the other names for entities identified by one of the GUIDs. The ranking in this case is high (1 if we consider rank $\in [0,1]$) because the retrieved names are the exact variant of the input string.

```
List<GUID,float> searchEquals(String input) {
     // supposing db table containing (ID, GUID, name)
     queryDB = "select e.GUID from Dictionary where e.name=" + input;

     // how to add RANKING: we could suppose that in this case ranking = 1 (
        max)
    return execute(queryDB);
}
```

Listing 5.12: Plain Search

5.2.0.8 Token Reordering

This function handle the case in which the tokens composing the query name are arranged in a different order from the ones for the name stored in the database.

The algorithm searches in the dictionary of the name composed by the tokens ordered in alphabetic order, and also ordered by field.

The former order suppose that there is a dedicated field in the name table containing each name with lexical graphic ordered tokens. For example if we are considering the name "Professor Fausto Giunchiglia" there will be a specific field *ordered_tokens* containing the string "Fausto Giunchiglia Professor".

The latter case instead arrange the tokens considering the name field they represent, which means whether they correspond (for the Person case) to the first name, surname, title and so on. The different tokens are recognized based on the specific eType structure for names (see section 7.2). Then the algorithm searches inside a specific table that relates each name token with the name field it represents.

```
List<GUID,float> searchReordering(String[] tokens) {
   // generate string ordering tokens alphabetically, supposes words ordered
      in this way in the db, or in a field of the name table
```

```
    List<GUID,float> list1 = searchEquals(orderAlphabetic(tokens));

    // reordered tokens based on name field (recognized via statistics or
       trigger words)
    List<GUID,float> list2 = searchEquals(orderByField(tokens));

    return union(list1, list2);
}
```

Listing 5.13: Token Reordering Search

5.2.0.9 Dictionary Search with Variations

Its task is to look for variations of the input name in the database. Those variations are based on misspellings in the implementation of the *searchMisspellings* function consists in generating a index, based on some of the name property, which will help the function to efficiency extract from the database a set of candidate names, which will be then more accurately compared with the input name through the *misspellingSimilarity* function (see section 5.1.2.2).

The key factor of this function that determine the efficiency of the algorithm is how this index is computed. One possible idea would be to use the *commonChar* function to determine whether the two names share an acceptable number of characters. To make it more efficient, and since it uses some preprocessing phase for each string, the preprocessed string could be stored in a dedicated column of the database. Another possible alternative would be to compare the *n-grams* (see section 4.1.1.1) of the names in an efficient way. In this case defining the size n of the grams and the way they are compared is fundamental. One possible choice for quickly compare the strings is to compute the ASCII sum of the n-grams, and use the difference as similarity measure, as proposed by Lalwani et al. in [4]. Alternatively we could combine these two proposed strategies, using an array which contains the occurrences of each letter in the word, and which can be stored efficiently in the database.

```
List<GUID,float> searchMisspellings(String input) {
    // select from the db those names which are likely to be variations of
       input
    querySelect = "select * from DB where name func(input)";
    candidates = execute(querySelect);

    // apply edit distance on those names, to pick only the correct ones
    for s in candidates
        if (misspellingSimilarity(s, input)> T)
            result.add(s.GUID);

    // in this case the rank could be the edit distance
    return result;
 }
```

Listing 5.14: Misspelling Search

5.2.0.10 Search for Tokens

The aim of this function is to search for names which are present partially in the database. The function needs then to manage the case of *sub-query* and *super-query*, which mean respectively query with less and more tokens than the database name. Consider for example the name "Fausto Giunchiglia". If we search just for "Fausto" (sub query) or "Professor Fausto Giunchiglia" we want in both cases to obtain the complete name ("Fausto Giunchiglia") as one of the possible results.

The proposed implementation of this function then search for each single token and rank the results based on the number of token from which each of them is retrieved. Considering the query "Fausto Giunchiglia" and the names: $Name_1$ = Fausto Rossi, $Name_2$ = Giunchiglia Fausto, $Name_3$ = Professor Giunchiglia: $Name_1$ will be the one with the highest ranking because it is return both by the token "Fausto" and "Giunchiglia", followed by the other names which are returned only from one token.

```
List<GUID,float> searchToken(String[] tokens) {
   HashMap<GUID,int> candidates;

   for t in tokens {
      tokenResult = searchSingleToken(t);
      for r in tokenResult
         candidates.put(r, candidates.get(r)+1);
   }

   return candidates.keys();
}
```

Listing 5.15: Token Combination Search

5.3 Auto completion Search

The subject of this section is the name search with auto completion. As defined in equation 2.5, the input for this function is a string, representing the partial query that the user typed, and it should return a ordered list of entity names, which are possible completion for the user query.

5.3.0.11 Prefix Index

For the design of this function we opted for the implementation of a index which relates each prefix with a set of GUID of the entities in the system having a name tokens which starts with the input prefix.

In particular the index is composed by three fields: the string representing the actual prefix , a string containing the GUID of entities more frequently searched when that input is inserted, and the list of the prefix of length increased of one, which are the possible extension of the input query.

In the following functions we assume to use the prefix information retrieved from the database as a class:

```
class Prefix {
   String prefix;
   List<GUID> topGUID;
   Etype etype;
}
```

Listing 5.16: Token Combination Search

The shortest prefix admitted is of size 2, because accordingly to principle used by Jaro-Winkler algorithm (see section 4.1.2.2) statistically the two first letters of a name are spelled correctly.

Moreover, since the number of possible name prefixes can be huge, and scan the whole index would slow down the search algorithm the prefixes can be divided in different tables, according to their length (one table for each prefix length, or at least for the shorter prefixes, which will be queried more frequently), and to their first letter.

5.3.0.12 Prefix Search

Assuming that we have at disposal a prefix index as described in the previous paragraph, we can easily design a prefix search function. The high level pseudo code of such a function is written in listing 5.17.

Once the user had input a string representing a prefix, the function normalize the prefix (replace accents and other special characters), and search within the database the corresponding prefix attributes. From these information it retrieves the most ranked entities for the input string, and can also anticipate the user retrieving the top ranked entities for the successors prefixes.

From the entities ids then it returns the list of corresponding (canonical) names.

In listing 5.17 the function accepts as input also the etype of the name. This parameter is present in case the function is used in some context where we already know which etype the user wants to retrieve, in order to make the search more specific. In the general case where the etype is not provided, the algorithm computes the list result as the union of the search for all the possible etypes.

```
Set<String, float> searchPrefixName(String input, EType etype) {

    // normalize input prefix for accents and other special characters
    String normalized = normalize(input);

    // search the list of GUID corresponding to the input prefix
    List<String,float> nameList = searchPrefix(input, etype);

    // return the list of retrieved names
    return nameList;
}
```

Listing 5.17: Autocompletion Search

```
List<String> searchPrefix(String input, EType etype) {
    int len = input.length;
    String tableName = "prefix" + len;

    String prefixQuery = "select fullname,frequency from " + tableName +"
        where prefix = " + input +" and etype = " + etype ;

    return execute(prefixQuery);
}
```

Listing 5.18: Search Prefix GUIDs

The autocompletion search retrieves names which are the most selected by users when typing a query. But in case the user input a name which is not between the most populare for its prefix, the system will not be able to return it in autocompletion search. In this case the user should rely on the standard search algorithm which takes as input complete name.

5.3.0.13 Observations

The index based on prefix allows us to implement a fast name search function, but the insertion phase will be more complex. In listing 5.19 is reported a possible way to manage the insertion of the prefixes. The main task for the insertion function is to add to the index new prefixes related to the inserted names, and link them together.

The other problem for this index concerns the management of the list of top ranked entities per prefix. This list must be updated accordingly to name search statistics about the result chosen by the user with respect of the input query. In the initial phase, when no statistics are available, the top list will be generated starting from the first inserted entities.

```
void insert(int GUID, List<String> names) {
   for n in names {
      tokens = tokenize(n);
      for t in tokens {
         prefixes = generatePrefixes(t);
         for p in prefixes {
            addPrefix(p,GUID);
}}}}
```

Listing 5.19: Insert Prefix Algorithm

5.4 Named Entity Recognition function

In this section will be discussed how to implement a function for addressing the Named Entity Recognition problem. In equation 2.9 is reported the method signature for the *etypeNameRecognition* function proposed. Its task is to, given a name retrieve a list of pairs $(EType, float)$ representing the list of possible ETypes of entity corresponding to that name, and the related probability of them to be correct.

Given the function developed for the name search function (see section 5.2), the *etypeNameRecognition* can be constructed composing them. In listing 5.20 is reported a possible implementation of the *etypeNameRecognition* using the two functions called by *nameSearch* (see listing 5.10). The only difference is in the information returned from the function given the list of GUID of entities related to the input name: *nameSearch* returns the list of alternative names, while *etypeNameRecognition* returns the eType of those entities.

```
List<EType,float> etypeNameRecognition(String name) {
   list = entityNameSearch(input);

   return retrieveEType(list);
}
```

Listing 5.20: Etype Name Recognition function

5.5 Named Entity Disambiguation function

The design of Named Entity Disambiguation function defined by equation 2.12 is described in this section. Similarly to *etypeNameRecognition* (see section 5.4) also *entityCandidates* can be implemented using the functions defined for name search. Listing 5.21 reports the relative implementation, where we can see that the only difference between this and the NER function is in the returned information: *etypeNameRecognition* returns the eType of the entity, while *entityCandidates* returns the entire Entity (which from the scope of this thesis is composed by a GUID, a set of names, an EType and a description).

```
List<Entity,float> entityCandidates(String name) {
   list = entityNameSearch(input);

   return retrieveEntity(list);
}
```

Listing 5.21: Named Disambiguation function

Chapter 6

Ordering

In this section we will discuss the problem of ordering, which consists in assigning a ranking value to the result obtained by the different functions (see section 5).

The ranking value is a positive real number $r \in [0, 1]$ assigned to each result (name, eType, entity) returned by the 4 functions (see section 2), which determines in which order they should be presented to the user. The value is determined in a way that the results most related to the input query have a ranking close to 1, while the less likely to match the name will have a ranking close to 0.

The way this ranking value is computed can be based on two different factors:

Name Similarity which considers whether two names match according to the algorithms for string similarity and the stored variants,

User Experience which based the similarity of names on the choices made from the user in the previous queries.

The overall measure should be a combination of these two approaches, and it is described in more details in section 6.3. We decide to trust also on the users choices since their experience is, based on our intuition, more reliable for names than the algorithm for matching, because people can decide whether the names match based on their previous knowledge.

In section 6.1 is presented the name similarity function, while the two measures based on user statistics are described in section 6.2. The last section, 6.3, discuss the possible usage of the statistics collected from the log.

6.1 Similarity Measure

The similarity measure is a real number $r \in [0, 1]$ assigned to a target name with respect to a source name where 1 represents perfect match between the names, which means that for certainty they represent the same entity, and 0 means that the names correspond to completely different entities. In case of name matching we consider as source the first inserted, for name search the input query.

The computation of this value is based on the variations and variant of the names, which consider the strings appearance and the entries present in the system dictionary (like alternative names and translations). In particular for the misspellings variations we use the Damerau Levenshtein metric (see section 4.1.2.1) which considers the number of edit (substitution, insertion, deletion) operation necessary to transform a string into another. The overall distance between the two names is normalized over the size of the strings so we can obtain a value in [0,1].

Depending on the name and the results of the different strategy applied, the similarity can be computed on the entire name, or on each of the tokens that compose the names.

In equations 6.2 and 6.1 is represented the function that combines together the different name similarity strategies used in the *nameMatch* function (see section 5.1).

$$sim(Name_1, Name_2) = \begin{cases} \frac{sim_{string}(Name_1,Name_2)}{|Name_2|} & \text{if } sim_{string}(Name_1, Name_2) > T \\ \\ \sum_{(t_1,t_2)\in pairs} sim_{string}(t_1, t_2) \cdot \frac{|t_2|}{|Name_2|} & \text{otherwise} \end{cases} \tag{6.1}$$

where $pairs = \{(t_1, t_2)|t_1 \in Name_1, t_2 \in Name_2, \text{based on heuristics}\}$ generated based on heuristics on the name field each token belongs to. For example if we have two person names:

	$Name_1$	$Name_2$
Full Name	Fausto Giunchiglia	Fabio Conchiglia
Given Name	Fausto	Fabio
Surname	Giuchiglia	Conchiglia

Table 6.1: Example of tokens for two person names

the resulting set of pairs will be: $pairs = \{(\text{Fausto, Fabio}), (\text{Giunchiglia, Conchiglia})\}$.

$$sim_{string}(string_1, string_2) = \begin{cases} |string_2| & \text{if } string_1 = string_2 \\ \frac{(|string_2| - editDist(string_1, string_2))}{|string_2|} & \text{if } |(|string_1| - |string_2|)| < L \\ & \text{and } editDist(string_1, string_2) > T \\ sim_{string}(variant(string_1), string_2) & \text{if } sim_{string}(alt(string_1), string_2) > T \\ 0 & \text{otherwise} \end{cases} \tag{6.2}$$

where $editDist(s,t)$ = nr of edit operation necessary to transform s into t. Edit operation = {add, delete, substitution} and all same weight

The similarity function is designed such that the resulting value is always a real number included in [0,1], and that is not symmetric: $sim(a,b) \neq sim(b,a)$. This is because we always consider one of the name as the source name and the other as the target name (see section 3).

Consider the previous example in table 6.1. The first function called is sim(Fausto Giunchiglia, Fabio Conchiglia), which first tries to compare the full name with the other function sim_{string}(Fausto Giunchiglia, Fabio Conchiglia).

The two strings then enter into a list of comparison. Since they are not identical, the algorithm jump to the edit distance case. Edit distance algorithm is applied since the length different is lower than the threshold (suppose L=3), and its value is 6, unfortunately higher than the threshold (suppose T=4). The last step tried by this function is to look for alternative names of $string_1$ (in our case "Fausto Giunchiglia"), like nicknames and translations, that match $string_2$ ("Fabio Conchiglia"), but the algorithm cannot find any of them. Then we obtain that sim_{string}(Fausto Giunchiglia, Fabio Conchiglia) = 0.

The sim function obtains the returned value, and since 0 < T (for every T), he goes to the second case, which is token analysis. The set of pairs (as showed in the previous example) for the input names is $pairs = \{(\text{Fausto, Fabio}), (\text{Giunchiglia, Conchiglia})\}$. On each pair is applied the sim_{string} function:

Pair(t_1, t_2)	sim_{string}	$sim_{string}(t_1,t_2) \cdot \frac{\|t_2\|}{\|Name_2\|}$
(Fausto, Fabio)	$^2/_5$	$^2/_5 \cdot (^5/_{15}) =^2 /_{15} =$
(Giunchiglia, Conchiglia)	$^7/_{10}$	$^7/_{10} \cdot (^{10}/_{15}) =^7 /_{15}$

Table 6.2: Example of String Similarity computation on two person names

The overall similarity of ($Name_1$, $Name_2$) then is $^9/_{15} = 0,6$.

Consider another example:

	$Name_1$	$Name_2$
Full Name	Papa Giovanni Paolo II	Pope John Paul II
Given Name	Giovanni	John
Midname	Paolo	Paul
Title	Papa	Pope
Qualifier	II	II

Table 6.3: Other example of tokens for two person names

$$\text{pairs} = \text{(Giovanni, John), (Paolo, Paul), (Papa, Pope), (II,II)}$$

If we suppose that we do not have any information that these two names correspond to the same entity, we will obtain that sim_{string}(“Papa Giovanni Paolo II”, “Pope John Paul II”) = 0.

Pair(t_1, t_2)	sim_{string}	$sim_{string}(t_1, t_2) \cdot \frac{\lvert t_2\rvert}{\lvert Name_2\rvert}$
(Giovanni, John)	1 (translation)	$1 \cdot ({}^4/_{14}) = {}^4/_{14}$
(Paolo, Paul)	1	$1 \cdot ({}^4/_{14}) = {}^4/_{14}$
(Papa, Pope)	1	$1 \cdot ({}^4/_{14}) = {}^4/_{14}$
(II, II)	1	$1 \cdot ({}^2/_{14}) = {}^2/_{14}$

Table 6.4: Other example of String Similarity computation on two person names

$$\text{sim(“Papa Giovanni Paolo II”, “Pope John Paul II”)} = {}^{14}/_{14} = 1$$

6.2 User Ranking

For the implementation of measures based on community experience, the system needs to keep track of the choice made by the users. We should implement a system that uses a special log for storing the queries and the results chosen by the user, and updates regularly (once a day or week) a dedicated database table which stores the overall name ranking.

Since we are interested in what the user inserted as query, and which is the result he/she was expecting to get, we do not need to store the complete log information about its query, which would contain also more technical information. We only need to store pairs of the type $(inserted\_query, selected\_result)$. In table 6.5 are listed the type of tuples stored for each function.

Function	Log		
Name Matching	($name_1$,	$name_2$, similarity,	user)
Name Search (standard)	(query,	GUID selected,	user)
Name Search (autocomplete)	(prefix,	GUID selected,	user)
Named Entity Recognition	(query,	GUID selected,	user)
Named Entity Disambiguation	(query,	GUID selected,	user)

Table 6.5: Pairs stored for each function

The table for storing this ranking should be composed by:

```
Rank (query , result , hit)
```

Listing 6.1: Named Disambiguation function

where the combination of *query* and *result* is the primary key of the table, and *hit* represents the times a user chose *result* after inserting *query*.

Local statistics represents the data collected from a single user, stored in his private database and are not visible to other users. Global statistics are generated as the union of the local statistics of all the users in the systems. Each time a log is stored in the local database, it is also sent to the central service which updates its database, without keeping track of where those data come from, for preserving user privacy.

The strategy applied for managing these ranking information should be different according to the level of statistics we are considering. Local statistics should be stored entirely, even when the value of *hit* is very low, as long as the size of those data can be stored in the local database. In this case a different strategy applied for the global data should be preferred. Since the data collected from all the system users will be huge, in order to keep the table reasonably small, we can store the pairs only when the hit is higher than a threshold; the problem in this case is that entries frequently used in a short amount of time will be stored, while entries used regularly for long time, but not often for a short period (the maintenance time) will not be stored.

6.3 Statistics Usage

Once we find an efficient way to store statistics, we have at out disposal a set of data that we can use for improving the name search and matching in different ways.

The first application of these information is the computation of the **ranking**. As introduced in section 6.2, we can order the results returned by each function based on the frequency that the user select them. When the user inputs a query q for one of the functions defined in section 2, the algorithm retrieves a list of results $\{n_1, n_2, ..., n_k\}$ that must be ordered based on some parameter. In this case the parameter considered is the user usage statistics. In the database we have (as previously described) a table containing tuples $\{(q, n_1, hit_1), (q, n_2, hit_2), ...\}$ (see table definition in listing 6.1). The user will then obtain as result from the function the ordered list of names $< (n_i, hit_i) >$ for $i \in [1, k]$ where $hit_i \geq hit_{i+1}$.

Using this approach we build an ordering based on the user usage, and in particular on his/her usual specific misspellings and format variations. When the user inputs a query q, the algorithm should search as first in the local ranking, and only when there is no pair (q, x) in the local table, then the service should look into the central one for the suggested result.

Moreover when the algorithm uses statistics from the community, it could even detect name matching that are not discovered by the similarity measure (see section 6.1).

In listing 6.3 and equation 6.3 is showed how the local ranking (and similarly the global one) is computed, and how the three different measures are combined together to obtain the overall ranking.

```
float retrieveLocalRanking(String query, String GUIDselected) {
   int localHit = "select hit from rank where query="+query+" and GUID="+
      GUIDselected;
   int countLocal = "select count distinct(GUID) from rank where query="+query;

   float LR = (localHit / countLocal) // percentage of hit with respect to that
      specific query
   return LR;
}
```

Listing 6.2: Local Ranking computation

Similarity Combination 1:

$$\begin{gathered} ranking = \alpha \cdot localRanking + \beta \cdot globalRanking + \gamma \cdot Similarity \\ \text{where } \alpha + \beta + \gamma = 1 \end{gathered} \tag{6.3}$$

Similarity combination 2:

```
if (localRanking > 0)
   return localRanking;
if (globalRanking > 0)
   return globalRanking;
else
   return similarity;
```

Listing 6.3: similarity combination

The second application of ranking is the management of the top rank names related to the prefix search. For speeding up auto completion algorithm, we are developing a tree structure for prefixes which keeps information about the corresponding names in the leaves, but maintain some information about the most frequent names for each single prefix. An easy way to manage those list of GUID would be to use the statistics retrieved from the user.

The third application of user usage data is **misspellings correction**. As explained in section 4.1.5, from these data we can extract information about what people meant to write when inserting the query.

We can operate two different level of misspelling correction: one which considers the entire name, and another which takes into account the single error. The former approach, given a query q selects the result n such that in the database the tuple (q, n, h) where $h = max(hit)$ for tuples with $query = q$ (see table definition in listing 6.1). It selects the most chosen name which is intended by the user when writing the specified query. The latter approach instead analyze the entire dataset, and extract new statistics on the single letter misspellings (see table 4.3). Once we obtains those statistics we can use them for correcting the input from the user, or improve the edit distance algorithm (used for misspelling variations) giving weight to the different operations.

Chapter 7

Name Database

In this section will be described the design and implementation choices made for the development of the system database.

7.1 Concept Definition

Entity Definition An Entity is an object identified by a GUID and has name list plus two more attributes which consists in the eType and the entity description, reported in an external link.

$\langle \textit{Entity} \rangle$ ::= '`(`' $\langle \textit{GUID} \rangle$ '`,`' $\langle \textit{full\_name\_list} \rangle$ '`,`' $\langle \textit{EType} \rangle$ '`,`' $\langle \textit{url} \rangle$ '`)`'

$\langle \textit{EType} \rangle$ ::= '`Person`'
| '`Location`'
| '`Organization`'
| '`Event`'

$\langle \textit{full\_name\_list} \rangle$ ::= $\langle \textit{full\_name} \rangle$ '`,`' $\langle \textit{full\_name\_list} \rangle$
| $\langle \textit{full\_name} \rangle$

Full Name A Full Name is the one of the possible name of an entity. It has a local ID (different from the GUID), and is composed by tokens of different type, which can be trigger words or the atomic names.

$\langle \textit{full\_name} \rangle$ ::= $\langle \textit{token\_list} \rangle$

$\langle \textit{token\_list} \rangle$ ::= $\langle \textit{atomic\_name} \rangle$ $\langle \textit{token\_list} \rangle$
| $\langle \textit{trigger\_word} \rangle$ $\langle \textit{token\_list} \rangle$ $\langle \textit{atomic\_name} \rangle$ $\langle \textit{token\_list} \rangle$
| nil

$\langle \textit{trigger\_word} \rangle ::= \langle \textit{title} \rangle$
$\mid \langle \textit{qualifiers} \rangle$
$\mid \langle \textit{others} \rangle$

This grammar generates names where trigger words and atomic names can appear in every order. Following these rules we can match names like:

Fausto Giunchiglia →	<atomic_name> <atomic_name>
Professor Giunchiglia Fausto →	<title> <atomic_name> <atomic_name>
Giunchiglia Fausto Prof →	<atomic_name> <atomic_name> <title>
Garda Lake →	<other_tr_word> <atomic_name>

A *Token* is a single word which composes a name. Each token represents a field in the entity name (like "surname", "first name"). One special type of token are *Trigger Words*: special words, which can be part of a name without being a proper name, and can help to understand which type of entity we are considering and give more information about the name structure. Examples of trigger words are "Doctor" or "Lake".

One problem that raises from this grammar definition is the distinction between a token representing a name or a trigger word. Even if we suppose that we have a complete list of trigger word, we can not be sure that the considered string is a trigger word. There can be the case in which a token which is a name, is identical to a trigger word. In this cases the position of the token can help the algorithm, but we can not completely rely on this criteria.

The proposed Backus Normal Form (BNF) grammar defines the complete name structure for an entity, and then it also describes the structure for the allowed queries. For name matching (see equation 2.2) the query is composed by two names, which therefore should be compatible with the grammar. Name Search (equation 2.6), NER (section 2.9) and NED (section 2.12) take as input a string representing a name, which need to be accepted by the syntax.

The only query which is not captured by this definition is the one for search with auto completion (see section 2.5): in this case the function expects the user to input the prefix of a name, which we can suppose is every possible set of initial letter accepted by the BNF grammar.

7.2 EType Specific Structure

Each entity type has a different structure and different types of trigger words that compose its name. In [14] (paper enrico eType) they try to define a particular BNF for each eType.

In the following sections we will present the structure derived from [14](paper Enrico eType)of the names for the etypes considered in our system, which are: *Person*, *Location*, *Organization* and *Event*.

7.2.1 EType Person

The EType *Person* collects all the entity that represent a person. In [14] Bignotti et al. define that the structure of a name for this eType can be simplified in 5 fields, each of them can contain multiple tokens. Consider the example of "Professor Fausto Giunchiglia III".

Given Name	*Fausto*	
Mid Name		
Surname	*Giunchiglia*	
Titles	*Professor*	represents all the titles relative to the social status or profession
Qualifiers	*III*	differs from titles because it contains those qualifiers that usually appears at the end of the name, and give extra information about the person.

While *Given Name*, *Mid Name* and *Surname* sets contains word which are names, *Titles* and *Qualifiers* include the trigger words that compose the name. In particular in [14] the different titles are classified in four sub categories:

- **Kinship** that represent family or social relations, ex: Mr, Mrs, Miss
- **Religious and Monarchy** used for religious or aristocratic person, ex: Pope, King, Duke, Prince
- **Military** rank of military officers, ex: General, Marshal, Lieutenant
- **Professional** titles that are specific to the person job, ex: Professor, Doctor, Engineer

This classification is important because accordingly to the different category changes the way the trigger word will be imported in the system, and more importantly, how the name are compared

We define that when a name contains a title of type *Religious and Monarchy* then the comparison of name will consider the entire string, and the translation of the single name token is allowed; instead if the names does not contain any title of this type, then all the tokens belonging to the set *Titles* are not included in the comparison, and translation of single name token is not allowed.

Consider the example *Name* = "Professor Fausto Rossi". In this case *Title* = "Professor", which is of type *Professional.* Then the string that will be analyzed by the matching and search functions will be "Fausto Rossi", and the translation of the tokens is not allowed (we cannot translate *Rossi*[it] to *Reds*[en]).

Now consider the example *Name* = "Papa Giovanni Paolo II". *Title* = "Papa", which is a *Religious* title. The string considered then will be the entire name "Papa Giovanni Paolo II", and it could be translated (we can translate *Giovanni*[it] to *John*[en]).

This distinction is done because it is in ordinary usage to translate names of historical or religious people, and their name come almost always with the respective title.

7.2.2 EType Location

Similarly to the eType person, in [14] is presented also a structure for names of Locations. In this case consider the example of the name "Piazza San Pietro".

Name	*Pietro*	
Toponyms	*Piazza*	gives information about the type of location the entity represents
Qualifiers	*San*	other qualifiers which are not toponyms

In this case we always consider the full name in the comparison operations since if we would delete the toponyms and qualifiers the matching with other names would be incorrect. For example take $Name_1$ = *Piazza San Pietro* and $Name_2$=*Basilica di San Pietro* we would consider only "Pietro" for both names, deriving that they match, which is not the case in the real world.

Moreover when analyzing locations' names we should allow the translation and format variations of toponyms and qualifiers, and in some cases also for the name. Considering the previous example, we should be able to match *Piazza San Pietro*[it] with *St. Peter's Square*[en].

7.2.3 EType Organization

Inside Organization EType are identified four sub categories:

- Organization (no-profit and other organization)
- Corporation
- Educational Organization
- Sport Team

Each of them has a particular name structure but for what concerns the purposes of this work we can generalize that the structure of names of the EType Organization is:

Name	*Apple, Trento, Vodafone, Milan*	organization name, without designators
Designator	*Inc, s.p.a, University, Corp., AC*	

In the general case the entire name should be considered in the comparison, but we can do some observations about the different type of designators. When we consider Educational Organization we should allow that the designator can be translated. Consider the example of *University of Trento*[en] and *Università di Trento*[it].

Instead for corporations the designator is used mainly for legal purposes, while in the other cases it is not included. And if we consider sports team the number of designator is huge, difficult to translate or classify, and similarly to corporation, rarely used. We should then consider designator for not educational organizations as stop words, which will not be considered in the comparison.

Moreover when we talk about organization (meaning the general EType) they are often addressed with their name acronym: *FIAT* instead of *Fabbrica Italiana Automobili Torino*, *IBM* instead of *International Business Machines.* We should then always compute the name acronym before the comparison. One possible choice would be when a new organization entity is created in the system, its acronym should be stored as an alternative name for that entity.

7.3 Database Architecture

In this section will be described the design choices made for the implementation of the database represented in the ER model in figure 7.1.

In the model we can see that there is a kind of symmetry between the *Individual Name* and the *Trigger Word* part: the two tables both have a many-to-many relation with *Full Name*, both have a translation (or variation) table, can have a specific type (*NameElement* and *TriggerWordType*) and statistics.

There is a difference between the statistics information collected for the two elements. For *Individual Name* the statistic counts the frequency of a specific name as a particular *Name Element* (for example how many times the name "Franco" is used as Given Name, or how many times is used as Family Name). Instead for trigger word the statistics relate together the *Trigger Word* with the *EType.* In this way we can have information about which is the most likely etype of the name in case it contains a paritcular trigger word.

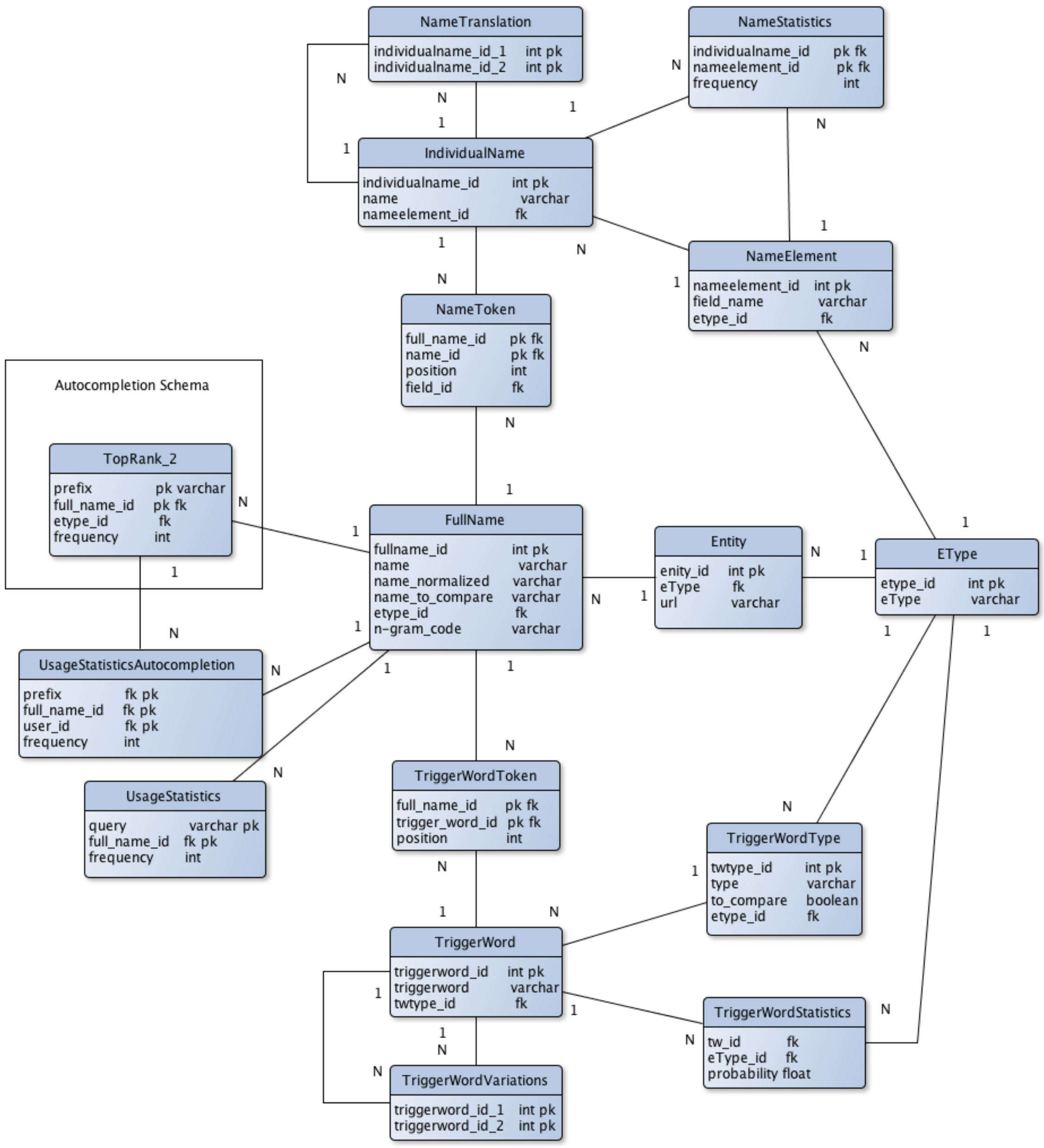

Figure 7.1: Database ER Model

Chapter 8

Evaluation

This chapter describes the evaluation of the framework presented in this thesis, referring to the name matching algorithm presented in section 5, compared with other string matching algorithm already present in the state of the art (see chapter 4).

Section 8.1 presents the datasets used in the evaluation with their main characteristics, and, if available, the results obtained in other works on these datasets.

The datasets will be tested with different configuration of the framework (see section 5) and of other algorithms from the state of the art, more precisely *Levensthein-Damearau* (see section 4.1.2.1) and *Jaro-Winkler* (see section 4.1.2.2), in order to explore the results for different values of different configuration properties. The proposed configuration are presented in section 8.2.

All the result collected from the experiment regarding those configurations are presented and discussed in section 8.4.

8.1 Dataset

The dataset that will be used for evaluating the algorithm are two:

1. The "Semantic Dataset" from Bignotti et al. [13]. The dataset contains people entities each of them with variant names and variations, presented more in details in section 8.1.1, and it is available for download at `http://disi.unitn.it/~knowdive/namedataset`
2. The "Census dataset" from Bilenko et al. [6]. This dataset is available at `https://github.com/TeamCohen/secondstring/blob/master/data/censusText.txt`, and contains people names, each of them with a syntactic variation, see section 8.1.2.

Table 8.1 summarize the properties of the two datasets, which will be described in more details in the next sections.

	Semantic Dataset	Census Dataset
Number of Entities	20	502
Average Name per Entity	10	2
Average Variants per Entity	5.25	1
Average Variations per Entity	5.25	1
Average Tokens per name	2.25	3
Average name length	13.6	14.68
Total Match Tasks	1500	1500
Positive Match Tasks	750	750
Negative Match Tasks	750	750

Table 8.1: Statistics on Datasets used for evaluation

8.1.1 Semantic Dataset

The Semantic Dataset is constructed starting from the work of Bignotti et al. in [13], and it contains a list of entities, each of them with a name, a list of alternative names, and a list of variations of those names. The entities in the dataset belong all to the entity type (eType as defined in 2) Person. Table 8.2 presents an example of an entity of type Person and its names available in the dataset. Listing 8.1 reports the xml code used for representing the entity in table 8.2. The corresponding schema is reported in appendix A.

Original Name	Variant Names	Variations
Mark Twain	Samuel Clemens	mark twain
	Samuel Langhorne	Mask Twain
	Clemens	Twin Mark
	Samuel L. Clemens	Mark Twainn
		Mark Twsin
		Mark Tawin
		Clemens Samuel

Table 8.2: Example of Semantic Dataset Entity of type Person

```
<entry>
    <etype name="Person" id="50"/>
    <variant name="Mark Twain" />
    <variant name="Samuel Clemens" />
    <variant name="Samuel Langhorne Clemens" />
    <variant name="Samuel L. Clemens" />
```

```
        <variation name="mark twain" />
        <variation name="Mask  Twain" />
        <variation name="Twin Mark" />
        <variation name="Mark Twainn" />
        <variation name="Mark Twsin" />
        <variation name="Mark Tawin" />
    </entry>
```

Listing 8.1: Xml for Semantic Dataset Entity of type Person

As defined in 2, *variant names* are alternative names used for the same entity, like alias, pen names, nicknames, and translations in other languages, while *name variations* are syntactic variations of the original name, like misspellings and format variations.

In addition to the entities, the dataset provides also a list of match pairs composed by two names and the expected result. The dataset includes 1500 match pairs, of which:

- 750 pairs that are expected to match (from now on called *match pair*), generated combining the original name with its variants, and all its variants with the name variations. In case the list of pairs is bigger than 750 elements, the 750 final pairs are randomly selected from the entire list;
- 750 pairs that are expected to not match (called *not match pair*). These are generated combining randomly names, variant and variations with the names of other entities. Similarly to the match pairs described in the previous point, in case the list is too large, the final set is choose randomly from the entire list.

Table 8.3 reports and example of two pairs in the dataset, where the first one is a "match pair" and the second in a "not match pair", and and listing 8.2 reports the xml code used for representing the match tasks (see xml schema in appendix A).

	Match Pair	**Not Match Pair**
Name 1	Samuel Clemens	Samuel Clemens
Name 2	Mark Twainn	Papa Giovanni Paolo II
Expected result	match	NOT match

Table 8.3: Example of two match and not match pairs from the Semantic dataset

```
    <matchEntries>
        <etype name="Person" id="50"/>
        <correct>true</correct>
        <name1 name="Samuel Clemens" type="variant"/>
        <name2 name="Mask  Twain" type="variation"/>
    </matchEntries>
```

```
<matchEntries>
    <etype name="Person" id="50"/>
    <correct>false</correct>
    <name1 name="Samuel Clemens" type="variant"/>
    <name2 name="Papa Giovanni Paolo II" type="variant"/>
</matchEntries>
```

Listing 8.2: Xml for two match and not match pairs from the Semantic dataset

The main property of this dataset, which distinguish it from others name matching dataset is the presence of variant names for entities, and not just syntactic variations.

8.1.2 Census Dataset

The Census Dataset is used by William W. Cohen et al. [11] for evaluating different string matching techniques.

The dataset presents two different list of entries A and B, both of them include an identifier, name and address of American people. The difference between the two lists is that list A contains correct spelled entries, while list B contains the same entries but misspelled. The misspelling can be present in all the elements of the entries, except from the identifier, which is used to identify the relation between entries in A and B.

Starting from this two lists, a set of 1500 match pairs has been build similarly to the *Semantic Dataset* (see section 8.1.1). The matching pair are composed only of the name, without taking into account the address, considering that this work address the name matching problem and not general string matching.

Each name in this dataset is composed of: a surname, (optional) a name, (optional) the middle name initial, and the address. In our experiment, since it is about name matching, we will consider only the three tokens that represent the name, and not the address.

Table 8.4 presents an example of an entry in the Census dataset: the first part is the name from list A (correct spelled), while the second one is the corresponding entry (with the same identifier) from list B (variation name).

Listing 8.3 and A.1 reports the respective xml code for the dataset and two example match pairs, in the same format of *Semantic dataset* 8.1.1.

```
<entry>
    <etype name="Person" id="50"/>
    <variant name="DOMINIC A RUDASILL" />
    <variation name="DOMINI A RUDAGILL" />
</entry>
```

Listing 8.3: Xml for Census Dataset Entry

	List A	**List B**
Identifier	ID44501272383618100002	ID44501272383618100002
Family Name	RUDASILL	RUDAGILL
Given Name	DOMINIC	DOMINI
Middle Name	A	A
Address	122 WARE	122 WARE

Table 8.4: Example of Census Dataset Entry

```xml
<matchEntries>
    <etype name="Person" id="50"/>
    <correct>true</correct>
    <name1 name="DOMINIC A RUDASILL" type="variant"/>
    <name2 name="DOMINI A RUDAGILL" type="variation"/>
</matchEntries>
<matchEntries>
    <etype name="Person" id="50"/>
    <correct>false</correct>
    <name1 name="DOMINIC A RUDASILL" type="variant"/>
    <name2 name="STG MARTINEZ" type="variation"/>
</matchEntries>
```

Listing 8.4: Xml for two match and not match pairs from the Census dataset

The Census dataset contains only variations of names, and does not consider variant (as alternative names and translation), as the Semantic Dataset.

The Census dataset was used also in another work to compare the performance of different string matching algorithms. In the work of William W. Cohen et al. [11] they used this dataset for evaluating several string matching algorithm, among them *Levenstheine-Damearau* and *Jaro-Winkler*, as we use here. In their evaluation they considered the entire entry in the dataset, and not just the token representing the name. Table 8.5 reports the results obtained in their work.

String matcher	**Max** $F_1 - measure$	**Average Precision**
Jaro	0.728	0.789
Levenstein	0.865	0.925

Table 8.5: Results on Census dataset from William W. Cohen et al. [11]

8.2 Evaluation Modality

In this section will be described the modalities used for the experiment execution. Here we evaluate the Name Matching algorithm presented in section5.1, because it is the main part of the framework, and the other two algorithms (Standard Search, section 5.2 and auto completion search, section 5.3) use functionality implemented by the Name Match algorithm.

The evaluation of the two search function could be implemented easily starting from the two datasets presented in 8.1, and is left as future work.

8.2.1 Experiment Setup

As presented in section 8.1, the dataset are used to create a list 1500 of pairs for each of them, which include both correct and not correct match.

The name matching algorithm presented in 5 needs two external component in order to work: a name tokenizer (section 8.2.1.2) and the dictionary (section 8.2.1.1).

8.2.1.1 Dictionary

Because of the presence of alternative names in the Semantic Dataset, the system needs to be initialized with a dictionary which contains relations about variant names of entities. This dictionary is retrieved from the Freebase database through their APIs [1]. The type of names extracted are mainly translation and some variant. Table B.2 in appendix B reports an example of two calls to Freebase APIs for retrieving the identifier and the alternative names of the entity *Rome* representing the Italian City. Table 8.6 presents the resulting list of variant names obtained by the system from the API invocation.

Entity	**Variant Names**
/en/rome	Roma
	The eternal City
	Rome,Italy
	A Cidade Eterna
	Rome
	Rom

Table 8.6: Example of alternative names and translation from Freebase for entity "/en/rome"

This operation is not done for the Census dataset since the entities contained in that list are not present on Freebase database.

8.2.1.2 Name Tokenization

The algorithm designed in section 5 contains a specific function for analyzing the tokens that compose a name. In order to compare these tokens the algorithm needs a tool that is able to split the string representing the name into tokens and assign them a semantic. The semantic is necessary to understand per each token which part of the name it represent (see section 7.2).

The tokens will be compared in two by two for a more accurate name similarity. The pairs of token can be generated using different strategies. The strategies described in section 5.1.2.5 are two:

- the semantic pairing strategy, which considers the name element that each token represents in the name
- the syntactic pairing strategy, which considers the first letter of each token and their alphabetic order.

The use of this two different strategies will affect the accuracy and the execution time of the algorithm. The syntactic one should be faster, but also less accurate. We need to understand which is the accuracy and time difference between the two strategies and decide which is the most suitable one.

For this evaluation we will use a tokenizer based on heuristics for splitting the name string into tokens, and we will not give semantic to the part of names, considering all of them at the same level. Then the pair strategy that will be use will be the syntactic one, since we do not have semantic information about the tokens.

8.2.2 Parameters

In the experiment are tested different configurations of the algorithm, which differ one another by some parameters. In this section will be described in details the properties tested, how we expect that they will change the result, and which are the configuration on which the experiment will be run.

8.2.2.1 Threshold

Threshold is the value which defines whether the similarity between two name is high enough to define a match between the two entries. Threshold can have values from 0 to 1 (included).

8.2.2.2 String Comparator

The String Comparator is the algorithm used for computing the string matching similarity between two names and between their tokens. The system supports two implementation of the comparator: one is LevenstheinDamerau algorithm 4.1.2.1 and the other is JaroWinkler 4.1.2.2. We expect that the application of a different comparator will generate different similarity values and then will need different threshold for finding the best result on the dataset.

8.2.2.3 Matcher

The Matcher is the algorithm used for executing the name matching. In our experiment we will use two different matchers: one is the "Baseline" matcher which does not consider the semantic of the names and consists only of string matching algorithm (the state of the art algorithms from *Levensthein-Damearau* see section 4.1.2.1, and *Jaro-Winkler* see section 4.1.2.2), while the second is the "Semantic" matcher, and consists in the framework presented in this work (see chapter 5).

8.2.3 Values of Parameters

Giving different values to the parameters presented in section 8.2.2 we prepared four different configuration of the framework that will be used in the evaluation.

Table 8.7 summarize the characteristics of each configuration used in the experiment.

Name	**Matcher**	**Comparator**	**Threshold**
$Baseline_1$	Baseline	Levensthein-Damerau	[0.0, 0.1, ... , 1.0]
$Baseline_2$	Baseline	Jaro-Winkler	[0.0, 0.1, ... , 1.0]
$Semantic_1$	Semantic	Levensthein-Damerau	[0.0, 0.1, ... , 1.0]
$Semantic_2$	Semantic	Jaro-Winkler	[0.0, 0.1, ... , 1.0]

Table 8.7: System Configuration for the experiment

The first two configuration, $Baseline_1$ and $Baseline_2$, consists in the state of the art algorithms, respectively Levensthein-Damerau and Jaro-Winkler. The second two, $Semantic_1$ and $Semantic_2$ instead use the algorithm presented in chapter 5 for name matching, are differs one from the other from the String Comparator (third column) used inside them.

Summarizing, we will use 4 configuration, 2 baseline and 2 semantic; we will have 2 configuration that use Jaro-Winkler, one semantic and one baseline, and the other two that use Levensthein-Damearau, one semantic and one baseline. With these configuration we will be able to understand how the modification of each parameter can affect the result obtained in the experiment.

As shown in the last column, all the configuration will be tested on 11 values of threshold, from 0.0 to 1.0, in order to identify.

8.3 Measure

In this section will be defined the measure used for evaluating each configuration. The algorithms will be evaluated using precision/recall, $F_1 - measure$ and execution time.

Precision, Recall and $F_1 - measure$ are based on the true/false positive/negative results obtained from the dataset:

True Positive (tp) is a correct match pair, which was recognized as match by the system,

True Negative (tn) is a not correct match pair, recognized as non matching by the system,

False Positive (fp) is a not correct match pair, for which the system determined that it was matching,

False negative (fn) is a correct match pair, for which the system said it was not matching

Table 8.8 summarizes the definition of these concepts.

	Match Pair	**Not Match Pair**
System match	True Positive	False Positive
System not match	False Negative	True Negative

Table 8.8: True False Positive Negative definition

From the values of true false positive and negatives we can compute the corresponding precision recall and $F_1 - measure$ for each experiment.

$$precision = \frac{tp}{tp + fp} \tag{8.1}$$

$$recall = \frac{tp}{tp + fn} \tag{8.2}$$

$$F_1 - measure = \frac{precision \cdot recall}{precision + recall} \tag{8.3}$$

8.4 Results

In this section are reported the results of the experiment executed on the different algorithms. Figure 8.1 shows the values for precision and recall for the four different configurations (see table 8.7).

These two graphs show the difference between the two datasets. The census dataset (on the left) has the most similar values of precision and recall analogous for all the configurations. In the Semantic dataset (on the right) instead the results are different: the two syntactic matchers, $Baseline_1$ and $Baseline_2$, perform worse than the two semantic matchers, $Semantic_1$ and $Semantic_2$.

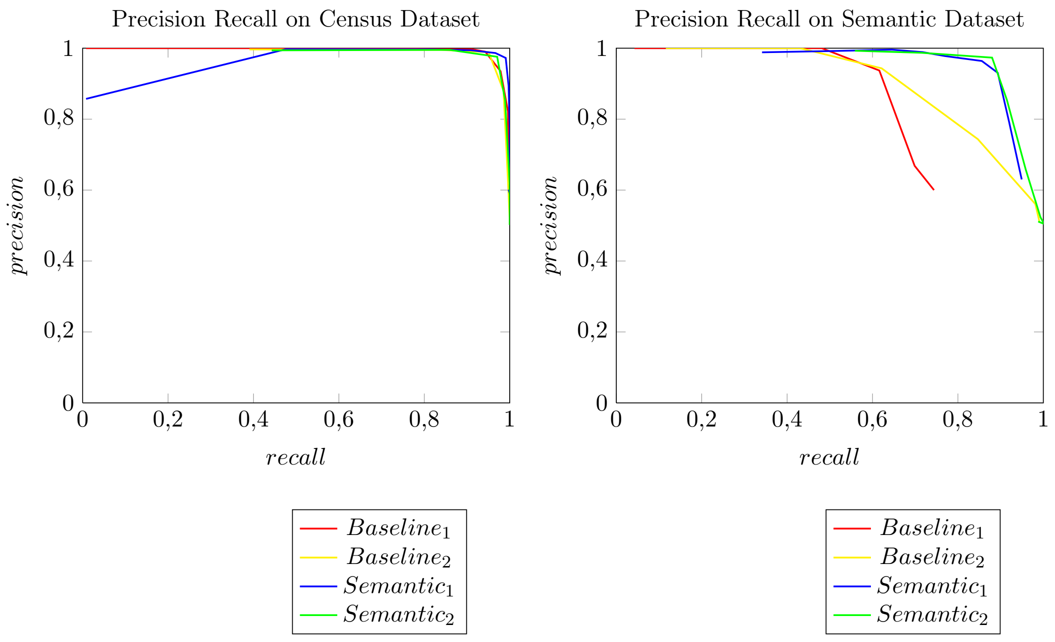

Figure 8.1: Charts for precision recall on the datasets

Figure 8.2 shows the results for the $F_1 - measure$ computed for each configuration on different values of threshold (with step difference = 0.1).

Similarly to the precision recall graph (see figure 8.1), we can observe in figure 8.2 that $Semantic_1$ and $Semantic_2$ outperform the other matcher on Semantic dataset, with difference of $F_1 - measure$ around 0.2 points. In the census graph we can observe that the results are similar for pair of configuration that use the same string matching algorithm: $Baseline_1$ with $Semantic_1$ and $Baseline_2$ with $Semantic_2$. For the first pair we can observe that $Semantic_1$ performs slightly better than the basic string matching algorithm. In the case of Jaro-Winkler based configuration ($Baseline_2$ and $Semantic_2$), the difference is smaller.

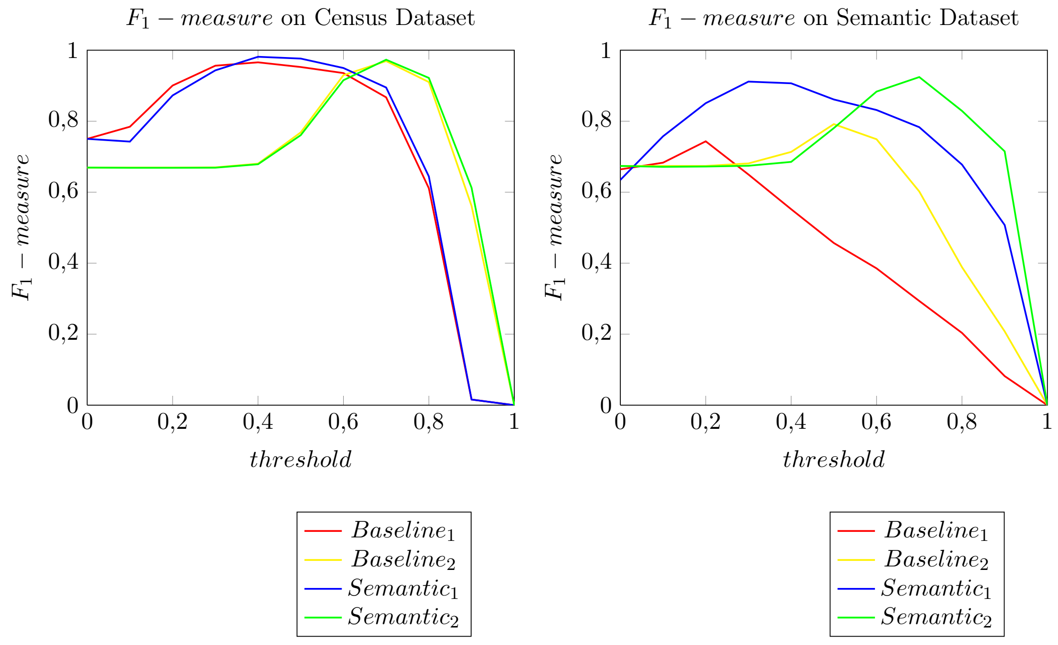

Figure 8.2: Charts for $F_1 - measure$ on the datasets

From these two graphs we can extract the threshold for each configuration that ensure the highest value of $F_1 - measure$, as shown by table 8.9.

Matcher	**Census**		**Semantic**	
	Threshold	$F_1 - measure$	**Threshold**	$F_1 - measure$
$Baseline_1$	0.4	0.9658	0.2	0.7433
$Baseline_2$	0.4	0.9694	0.5	0.7917
$Semantic_1$	0.4	0.9815	0.4	0.9067
$Semantic_2$	0.7	0.9732	0.7	0.9243

Table 8.9: F-measure Results Summary

The threshold represent the minimal similarity value that two names must have to be accepted as matching. We can notice that the two String Comparator (see section 8.2.2.2) have different values of threshold for their maximum $F_1 - measure$ in the Census dataset: for Jaro-Winkler the threshold is 0.7, while for Levensthein-Damearay is 0.4. The best values of threshold on the Semantic dataset instead are different depending on the type of matcher that is used: the two baseline matchers have lower optimal threshold than the corresponding semantic matchers.

In the algorithm implementation at the moment this value of threshold is also used to decide if it is necessary to adopt different strategies for detecting a match: first the system uses the basic string matcher, and if the result is lower than the threshold, computes the dictionary match, and if still the value is not high enough tries the token analysis.

This implementation choice implies two things:

- since the dictionary and token techniques are applied only when the string matching fails (is lower than the threshold), the results for the *Semantic* configuration are always higher than the two *Baseline* matchers;
- the threshold value affects also the execution time of the algorithm, as shown in figure 8.3.

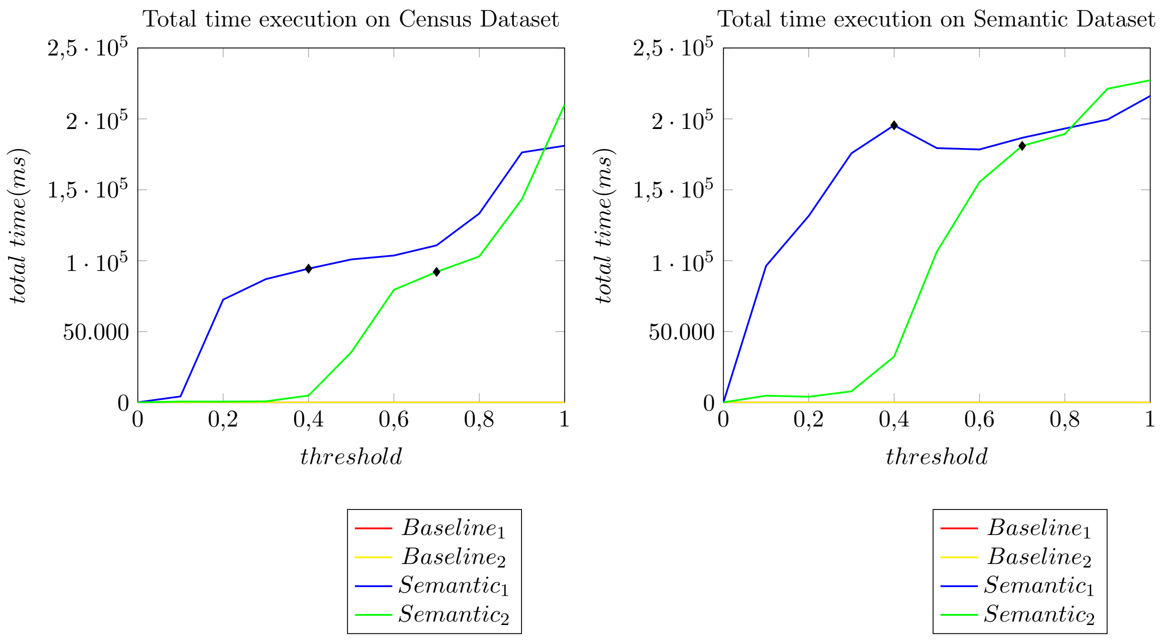

Figure 8.3: Charts for time on the datasets

In the two graphs representing the relation between total time execution and threshold (figure 8.3), are represented the results for all the four configurations. The time refers to the average time (on 10 runs) taken by the algorithm to execute 1500 matching tasks contained in the dataset. The diamonds on the lines represent the points with the optimal value for threshold obtained from the $F_1 - measure$ graph.

As we can observe, only for $Semantic_1$ and $Semantic_2$ there is a variability of the time execution in relation with the threshold. The other two configuration $Baseline_1$ and

Baseline$_2$ have constant time execution, too low compared to the time of the Semantic ones, and then they are not visible from the graph (they are close to the x-axis).

From the two graph we can also observe that there is a remarkable difference between the semantic and baseline matchers. This is caused by the invocation of function that use access to the database (the dictionary look up) and the analysis of the name tokens.

If the threshold increases, than increases also the probability that the string matching is not able to detect a match, and needs to invoke the dictionary look up and the tokenization, which are able to detect the semantic match, but also require more time.

Matcher	**Census**			**Semantic**		
	Threshold	$F_1 - measure$	**Time**	**Threshold**	$F_1 - measure$	**Time**
Baseline$_1$	0.4	0.9658	0.0068	0.2	0.7433	0.0142
Baseline$_2$	0.7	0.9694	0.0046	0.5	0.7917	0.0037
Semantic$_1$	0.4	0.9815	62.95	0.4	0.9067	130.33
Semantic$_2$	0.7	0.9732	61.43	0.7	0.9243	120.65

Table 8.10: Result Summary

In table 8.10 are reported for the configuration the best results obtained in the experiment for each dataset. Column *Time* refers to the average time take by the algorithm to execute a single match task in milliseconds.

From these values we can observe that there is a difference of the 0.1 between the two $F_1 - measure$ in favour of the configuration with *Levensthein Damearau* as string matcher, but an average of 10 milliseconds more per each match task.

Another observation that follows from the time results, is that the number of variant names (alternative names and translation) stored in the database affects the time necessary for a match task. In fact in Census dataset there are no variant stored for the names, since all the entries in the dataset are common people, while in the Semantic dataset almost all the names represent famous entities, which have around 10 variant each in the database.

The number of variant affects the time per match because in the dictionary lookup all the variant of the source name are taken in account to find a possible match between a variation of one of them and the target name. This is the example of the match between "Pope John Paul II" and "Papa Giovanni Paolo II", where the target name is a translation of single tokens of the source name.

From the results on Census datasets of the syntactic matcher *Levensthein-Damearau* and *Jaro-Winkler* we can notice that the maximum $F_1 - measure$ obtained is higher than the results obtained by William W. Cohen et al. in [11]. Table 8.11 shows a comparison between the values for $F_1 - measure$ obtained by the two works.

The difference in the results is caused by the input string considered in the matching tasks. In the work of William W. Cohen et al. [11] they considered the entire entry (name and address) as the string to be matched, while in our experiments, we considered only the tokens representing the person name.

String matcher	$F_1 - measure$ **from [11]**	$F_1 - measure$ **(our experiment)**
Jaro-Winkler	0.728	0.969
Levensthein-Damearau	0.865	0.965

Table 8.11: Results on Census dataset from William W. Cohen et al. [11] compare with our experiment

From the experiment results presented in section 8.4 we can assert that some of the hypothesis made during the system design are confirmed.

In 3 over 4 cases the configurations that use Jaro-Winkler as String comparator ($Baseline_2$ and $Semantic_2$) obtained higher $F_1 - measure$ than the respective configuration that use instead Levensthein-Damearau ($Baseline_1$ and $Semantic_1$). The only case in which Levensthein-Damearau string comparator has higher $F_1 - measure$ that Jaro-Winkler is on Census Dataset for the two Semantic matchers: $Semantic_1$ (which has Levensthein as comparator) obtains 0.008 points more than $Semantic_2$ for $F_1 - measure$. Our explanation about this lower result from Jaro-Winkler is because in Census Dataset the names variations contains misspellings in the first letters of the name, while in the Semantic dataset, most of the misspellings are in the central or ending part of the tokens. Jaro-Winkler indeed is designed for giving more weight to the first letters, because its hypothesis is that for names the misspellings occurs after the second letter of the string (as described by Christen in [10]).

In both datasets the Semantic matchers have higher $F_1 - measure$ than the corresponding Baseline. In particular in Census dataset the difference is minimal, while in the semantic dataset there is a difference of around 0.3 points for the Jaro based matchers and 0.35 points for Levensthein based ones.

The difference of $F_1 - measure$ results in the Census dataset is due to the fact that $Semantic_1$ and $Semantic_2$ algorithms, in case the string matching function fails, try also to analyze the similarity on single pairs of tokens, and then detect more true positive.

For the Semantic dataset the difference of $F_1 - measure$ is caused by the nature of the dataset itself as explained in 8.1. Each entity in the dataset have a list of alternative names which are not detectable through string similarity. These match need a semantic matcher, which uses some knowledge (the database) to match the names.

For what concerns result in time the Syntactic matcher outperform the Semantic ones.

As explained in results section 8.4, this is due to the use of database and token analysis.

The average time does not capture the fact that there is a noticeable difference in time between the task of matching a positive pair and a negative pair. In graph 8.4 are compared the match time for only positive match and negative match with different threshold. The diamonds on the lines highlight the threshold that guarantees the best F measure. The configuration used for the test is $Semantic_2$.

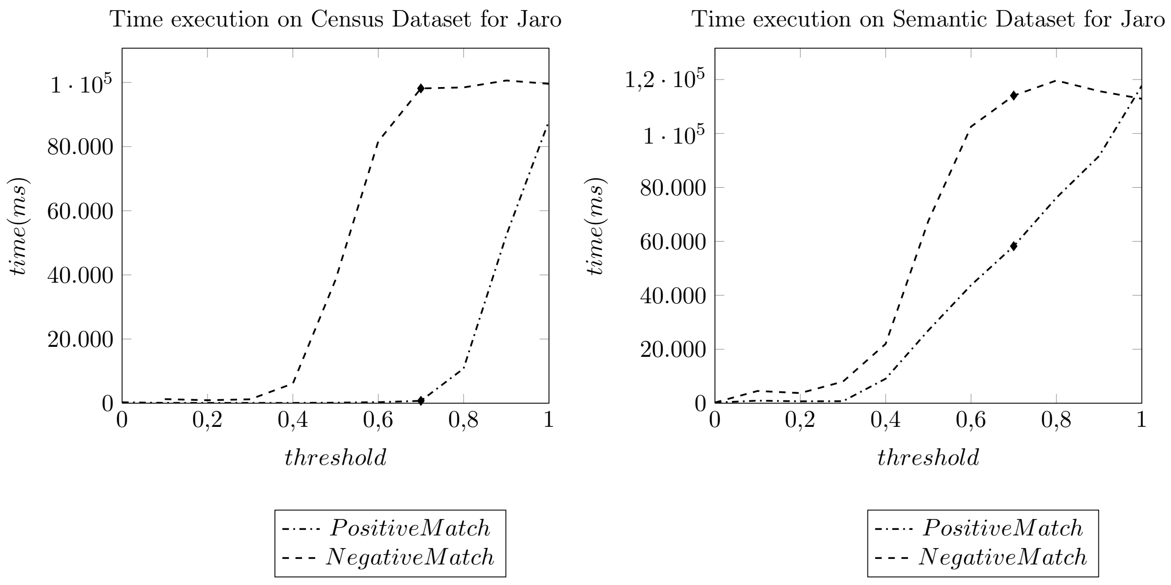

Figure 8.4: Time Comparison for positive and negative match pairs

The graphs highlight the fact that a not matching pair requires more time to be detected than a true matching pair, because a true match is recognized as soon as one of the function (string, dictionary and token in the order) used returns a true match, while a false match always requires to invoke all three strategies to be detected.

8.5 Remarks

Considering the best results for each configuration reported in table 8.10 we can observe that for the Census dataset $Semantic_1$ obtains a higher $F_1 - measure$ of 0.0083 points with a slightly longer execution time, 1.5 milliseconds more per match task than $Semantic_2$. On Semantic dataset instead $Semantic_2$ achieve a higher $F_1 - measure$ (0.02 points more than $Semantic_1$) and its execution takes in average 10 millisecond less per each match task.

Since the $F_1 - measure$ difference is small compare to the difference in time for the two best results in the different configuation, we can conclude that the matcher that obtain the best results is $Semantic_1$, which characteristics are presented in table 8.7.

Chapter 9

Conclusions

Analyzing the 4 problems regarding name in computer sciences (Name Matching, Name Search, Named Entity Recognition and Disambiguation), we have been able to reduce to three the number of functions essential to implement in the framework for addressing all these problems. The selected function, as presented in section 5, are:

- float $nameMatching$(String $Name_1$, String $Name_2$, EType $EType$);
- List<String,float> $nameSearch$(String $name[$, Etype $EType])$
- List<String,float> $autocompletionSearch$(String $name[$, Etype $EType])$

where the arguments inside squared brackets are optional.

All these three functions have been designed in details, and $nameMatch$ in particular have been implemented and evaluated. For the design and implementation, a specific database has been developed for storing properly the names accordingly to the structure defined for each entity type (EType) defined in section 7.2.

As result of the analysis, the thesis presents a framework for addressing the name related problems, which purpose is to allow the user to plug in different algorithms (for example different string comparator as showed in the evaluation section 8), and other approaches for comparing the different results.

The framework have been evaluated for what concerns the name match function on two datasets (Census dataset and Semantic dataset, see section 8.1), with different configurations that use different comparators and strategies (see section 8). Considering the best results for each configuration reported in table 8.10 we can observe that for the Census dataset $Semantic_1$ obtains a higher measure of 0.008 points with a slightly longer execution time, 1.5 milliseconds more per match task than $Semantic_2$. On Semantic dataset instead $Semantic_2$ achieve a higher F measure (0.02 points more than $Semantic_1$) and its execution takes in average 10 millisecond less per each match task. Since the

$F_1 - measure$ difference is almost negligible compare to the difference in time for the two best results in the different configuration, we can conclude that the matcher that obtain the best results is $Semantic_1$, which characteristics are presented in table 8.7.

Summary of Contributions

- Reduction of name related problems to 3 high level functions
- Design and partial implementation of these functions
- Creation of a dataset for name matching evaluation which address also name variants, available at `http://disi.unitn.it/~knowdive/namedataset`
- Implementation of a framework for combining different name matching and search techniques
- Improvement of the $F_1 - measure$ results on both the considered datasets compared to the current state of the art

Future Works

The work performed in this project provides basis for future research in several areas. At least three such areas can be identified, and include:

- Evaluation of the Name Search functions, similarly to the evaluation done on Name Match. The Semantic dataset could be use as well in these experiments in order to evaluate the semantic value of names.
- Improvement of the time efficiency for the name match function, since at the moment our implementation outperform the state of the art algorithm for what concerns the $F_1 - measure$, but requires more time to complete.
- Development of a Name Tokenizer, a tool able to identify the different tokens that compose a name, and assign to each of them the corresponding semantic, meaning the part of the name that the string represent (first name, for example, in case of a person entity).

Ringraziamenti

Il primo ringraziamento va al Professor Giunchiglia, per avermi offerto la possibilità di lavorare all'interno di questo progetto, e per la disponibiltà in questi mesi per i vari incontri riguardo il mio lavoro.

Un ringraziamento speciale va a Juan Pane e Alethia Hume, che mi hanno aiutato e supportato in questi mesi in tutte le fasi della tesi, e senza i quali non sarei riuscita a completare questo lavoro. Grazie ad Enrico Bignotti per il suo lavoro, senza il quale questa tesi non avrebbe altrimenti senso di esistere, e per l'aiuto fornitomi in questi mesi. Voglio ringraziare in genere il gruppo di ricerca KnowDive per il supporto fornito.

Poichè questa tesi non è solo il frutto del lavoro di 6 mesi, ma il risultato di 5 anni di sforzi, non posso dimenticare le persone che in tutto questo periodo mi sono state vicine. Primi fra tutti i miei genitori e la mia famiglia che mi hanno supportato finanziariamente, ma soprattutto psicologicamente. A voi andava sempre la prima chiamata o email dopo aver superato un esame.

Grazie ai miei amici qui presenti (anche se so che tra molto poco me ne pentirò!) per condividere con me questo momento, e aver dimostrato che si può essere ottimi studenti, e al tempo stesso sapersi lasciare andare e divertirsi. Grazie ai miei, ormai ex, compagni di università Alberto e Ciccio: nonostante la distanza di questi ultimi due anni, siamo riusciti comunque a vederci quando possibile, e ricordare tutti i vari cHP, merjitsu e sVArIOn. A tutti gli olandesi nel cuore disseminati in giro per l'Italia (Carlotta, Daria, Davide, Davide, Fabio e Carmine) va un ringraziamento specialissimo per aver contribuito a rendere speciale quella bellissima esperienza, che difficilmente dimenticheremo.

Ultimo ma tutt'altro che meno importante, un grande enorme ringraziamento va a Stefano, con cui insieme abbiamo cominciato questo percorso, e insieme lo stiamo terminando. Impiegherei meno a elencare gli esami che abbiamo preparato per conto nostro, che quelli fatti insieme, con tutte le discussioni e bestemmie del caso. Senza il tuo incoraggiamento non sarei mai partita per Amsterdam, e non saprei cosa mi sarei persa. Grazie per il supporto che mi hai sempre dato nei momenti di stress, ma anche per i magnifici risultati che abbiamo raggiunto insieme.

Bibliography

[1] *Freebase read services.*

[2] *http://www.internetworldstats.com/stats7.htm*, 2010.

[3] *http://www.behindthename.com/translate.php*, 2012.

[4] M. ade, N. Bagmar, and S. Parikh, *Efficient algorithm for auto correction using n-gram indexing.*

[5] H. Berghel and D. Roach, *An extension of ukkonen's enhanced dynamic programming asm algorithm*, ACM Transactions on Information Systems (TOIS) **14** (1996), no. 1, 94–106.

[6] M. Bilenko, R. Mooney, W. Cohen, P. Ravikumar, and S. Fienberg, *Adaptive name matching in information integration*, Intelligent Systems, IEEE **18** (2003), no. 5, 16–23.

[7] C.L. Borgman and S.L. Siegfried, *Getty's synoname^{TM} and its cousins: A survey of applications of personal name-matching algorithms*, Journal of the American Society for Information Science **43** (1999), no. 7, 459–476.

[8] A. Borthwick, J. Sterling, E. Agichtein, and R. Grishman, *Exploiting diverse knowledge sources via maximum entropy in named entity recognition*, Proc. of the Sixth Workshop on Very Large Corpora, 1998.

[9] R. Bunescu and M. Pasca, *Using encyclopedic knowledge for named entity disambiguation*, Proceedings of EACL, vol. 6, 2006, pp. 9–16.

[10] P. Christen, *A comparison of personal name matching: Techniques and practical issues*, Data Mining Workshops, 2006. ICDM Workshops 2006. Sixth IEEE International Conference on, IEEE, 2006, pp. 290–294.

[11] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg, *A Comparison of String Metrics for Matching Names and Records*, (2003).

[12] T. Dunning, *Statistical identification of language*, Computing Research Laboratory, New Mexico State University, 1994.

[13] Fausto Giunchiglia Enrico Bignotti, Juan Pane, *Semantic Name Matching*, Master's thesis, University of Trento, 2012.

[14] Vincenzo Maltese Enrico Bignotti, *Etype name structure*, (2013).

[15] Holloway Geoff, *The Math, Myth and Magic of Name Search and Matching*, 2004.

[16] P.A.V. Hall and G.R. Dowling, *Approximate string matching*, ACM Computing Surveys (CSUR) **12** (1980), no. 4, 381–402.

[17] IBM, *Overview of Names and Name Matching*, 2008, http://publib.boulder.ibm.com/infocenter/gnrgna/v4r1m0/index.jsp?topic=/com.ibm .gnr.gna.ic.doc/topics/gnr_gnm_con_namematchingapproaches.html.

[18] Heikki Keskustalo, Ari Pirkola, Kari Visala, Erkka Leppänen, and Kalervo Järvelin, *Nonadjacent Digrams Improve Matching of Cross-Lingual Spelling Variants*, 2003.

[19] R. Leaman, G. Gonzalez, et al., *Banner: an executable survey of advances in biomedical named entity recognition*, Pacific Symposium on Biocomputing, vol. 13, 2008, pp. 652–663.

[20] C.D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*, vol. 1, Cambridge University Press Cambridge, 2008.

[21] E. Minkov, W.W. Cohen, and A.Y. Ng, *Contextual search and name disambiguation in email using graphs*, Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval, ACM, 2006, pp. 27–34.

[22] P. Norvig, *Natural language corpus data*, Beautiful Data (2009), 219–242.

[23] B.T. Oshika, B. Evans, F. Machi, and J. Tom, *Computational techniques for improved name search*, Proceedings of the second conference on Applied natural language processing, Association for Computational Linguistics, 1988, pp. 203–210.

[24] F. Patman and P. Thompson, *Names: A new frontier in text mining*, Intelligence and Security Informatics (2003), 960–960.

[25] U. Pfeifer, T. Poersch, and N. Fuhr, *Retrieval effectiveness of proper name search methods*, Information Processing & Management **32** (1996), no. 6, 667–679.

[26] D. Ploch, *Exploring entity relations for named entity disambiguation*, ACL HLT 2011 (2011), 18.

[27] J.J. Pollock and A. Zamora, *Automatic spelling correction in scientific and scholarly text*, Communications of the ACM **27** (1984), no. 4, 358–368.

[28] B. Pouliquen, R. Steinberger, C. Ignat, I. Temnikova, A. Widiger, W. Zaghouani, and J. Zizka, *Multilingual person name recognition and transliteration*, arXiv preprint cs/0609051 (2006).

[29] S. Sarawagi and A. Bhamidipaty, *Interactive deduplication using active learning*, Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2002, pp. 269–278.

[30] W. Zhang, J. Su, C.L. Tan, and W.T. Wang, *Entity linking leveraging: automatically generated annotation*, Proceedings of the 23rd International Conference on Computational Linguistics, Association for Computational Linguistics, 2010, pp. 1290–1298.

Appendix A

Dataset Appendix

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="etype" type="eType"/>
  <xs:element name="matchEntry" type="matchEntry"/>
  <xs:element name="nameEntry" type="name"/>

  <xs:complexType name="dataset">
    <xs:sequence>
      <xs:element name="match" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="matchEntries" type="matchEntry" nillable="true"
                minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="matchEntry">
    <xs:sequence>
      <xs:element ref="etype" minOccurs="0"/>
      <xs:element name="correct" type="xs:boolean"/>
      <xs:element name="name1" type="name" minOccurs="0"/>
      <xs:element name="name2" type="name" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="eType">
    <xs:sequence/>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="id" type="xs:int" use="required"/>
  </xs:complexType>

  <xs:complexType name="name">
    <xs:sequence/>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="type" type="xs:string"/>
  </xs:complexType>
</xs:schema>
```

Listing A.1: Xml for two match and not match pairs from the Census dataset

Appendix B

Freebase API

Request for entities representing Rome:
```
https://www.googleapis.com/freebase/v1/search?query=rome&indent=true
```
Response
```
{
  "status": "200 OK",
  "result": [
    {
      "mid": "/m/06c62",
      "id": "/en/rome",
      "name": "Rome",
      "notable": {
        "name": "City/Town/Village",
        "id": "/location/citytown"
      },
      "lang": "en",
      "score": 171.902924
    }, ... ]}
```

Table B.1: Example of Freebase query on identifier and alternative names for entity Rome

Request for variant names for Rome:

```
https://www.googleapis.com/freebase/v1/rdf/en/rome
```

Response

```
@prefix key: <http://rdf.freebase.com/key/>.
@prefix ns: <http://rdf.freebase.com/ns/>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
....
    ns:common.topic.alias    "Roma"@en;
    ns:common.topic.alias    "The eternal City"@en;
    ns:common.topic.alias    "Rome, Italy"@en;
    ns:common.topic.alias    "A Cidade Eterna"@pt;
    rdfs:label    "Rome"@fr;
    rdfs:label    "Rom"@de;
    rdfs:label    "Roma"@it;
    rdfs:label    "Roma"@es;
    rdfs:label    "Rom"@da;
...
```

Table B.2: Example of Freebase query on identifier and alternative names for entity Rome