Efficient Entity Disambiguation via Similarity Hashing

Masterarbeit im Fach Informatik
Master’s Thesis in Computer Science
von / by
Dat Ba Nguyen

angefertigt unter der Leitung von / supervised by
Dr. Martin Theobald

begutachtet von / reviewers
Dr. Martin Theobald
Prof. Dr. Gerhard Weikum

August 2012
Hilfsmittelerklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Non-plagiarism Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, den 31. August 2012,

(Dat Ba Nguyen)

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department, Saarland University.

Saarbrücken, den 31. August 2012,

(Dat Ba Nguyen)
Abstract

The task of Named Entity Disambiguation (NED), which maps mentions of ambiguous names in natural language onto a set of known entities, has been an important issue in many areas including machine translation and information extraction. Working with a huge amount of data (e.g. more than three million entities in Yago), some parts in an NED system which estimate the probability of a mention matching an entity, the similarity between a mention and an entity and the coherence among entity candidates for all mentions together might become bottlenecks. Thus, it is challenging for an interactive NED system to reach not only high accuracy but also efficiency.

This thesis presents an efficient way of disambiguating named entities by similarity hashing. Our framework is integrated with AIDA which is an on-line tool for entity detection and disambiguation developed at Max-Planck Institute for Informatics. We apply various state-of-the-art approaches, for example Locality Sensitive Hashing (LSH) and Spectral Hashing, to some forms of similarity search problem such as near-duplicate search for mention-entity matching, and especially related pair detection for entity-entity mapping which is not the default application of using hashing techniques due to the usually low similarities between entities.
Acknowledgements

First and foremost, I would like to thank my advisor Dr. Martin Theobald for his invaluable guidance. I feel deeply indebted to his strong advice, encouragements and instructions. He has not only assisted me in completing this thesis, but he has also helped me to broaden my attitude towards research, and to develop my personality.

A special note of thanks to Prof. Gerhard Weikum for giving me the opportunity to work with his group in the Max-Planck Institute for Informatics. It is a priceless experience to me.

Many thanks to Johannes and Stephan for helping me when I started working on AIDA.

I specially have Quan’s family, my Vietnamese friends in Saarbrucken, my AOE team, my badminton team to thank for their encouragement, delicious food and relaxation after exhausting working hours.

Thanks are sent to the colleague students in the lab room 4.14 and room 4.11 for the memorable time working with them.

Finally, I would like to thank IMPRS-CS for the assistance and financial support that I received from them. It gives me total freedom to work on my master thesis.
To my family
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>vi</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Problem statement</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Contributions</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Outline</td>
<td>3</td>
</tr>
<tr>
<td><strong>2 Related Work</strong></td>
<td>5</td>
</tr>
<tr>
<td>2.1 Named Entity Recognition</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Automatic Knowledge Base Construction</td>
<td>6</td>
</tr>
<tr>
<td>2.3 Named Entity Disambiguation</td>
<td>7</td>
</tr>
<tr>
<td><strong>3 Similarity Hashing</strong></td>
<td>11</td>
</tr>
<tr>
<td>3.1 Similarity Search</td>
<td>11</td>
</tr>
<tr>
<td>3.2 Distance Measures</td>
<td>12</td>
</tr>
<tr>
<td>3.2.1 Definition</td>
<td>12</td>
</tr>
<tr>
<td>3.2.2 Jaccard Distance</td>
<td>13</td>
</tr>
<tr>
<td>3.3 Min-wise Hashing</td>
<td>13</td>
</tr>
<tr>
<td>3.3.1 Resemblance Estimation</td>
<td>14</td>
</tr>
<tr>
<td>3.3.2 B-bits Min-wise Hashing Algorithm</td>
<td>15</td>
</tr>
<tr>
<td>3.4 Locality Sensitive Hashing (LSH)</td>
<td>16</td>
</tr>
<tr>
<td>3.5 Spectral Hashing</td>
<td>17</td>
</tr>
<tr>
<td><strong>4 AIDA - Entity Detection and Disambiguation Tool</strong></td>
<td>19</td>
</tr>
<tr>
<td>4.1 Stanford Named Entity Recognition Tagger</td>
<td>19</td>
</tr>
<tr>
<td>4.2 Features and Measures</td>
<td>20</td>
</tr>
<tr>
<td>4.2.1 Popularity Prior</td>
<td>21</td>
</tr>
<tr>
<td>4.2.2 Mention-Entity Similarity</td>
<td>21</td>
</tr>
<tr>
<td>4.2.3 Entity-Entity Coherence</td>
<td>22</td>
</tr>
<tr>
<td>4.3 Graph Model and Algorithm</td>
<td>22</td>
</tr>
<tr>
<td>4.3.1 Mention-Entity Graph</td>
<td>22</td>
</tr>
<tr>
<td>4.3.2 Graph Algorithm</td>
<td>23</td>
</tr>
</tbody>
</table>
4.4 Discussion ................................................................. 24

5 NEDSH: Named Entity Disambiguation System via Similarity Hashing ................................................. 25
  5.1 System Architecture .................................................. 25
    5.1.1 Mention-Name Matching ....................................... 26
    5.1.2 Mention-Entity Similarity Estimating ....................... 27
    5.1.3 Entity-Entity Coherence Estimating ......................... 28
  5.2 Implementation ...................................................... 29
    5.2.1 Java Remote Method Invocation .............................. 29
    5.2.2 Data Structures and Libraries ............................... 30
    5.2.3 Representation for Hashed Objects ......................... 32
    5.2.4 API Functions ................................................ 32
  5.3 Integration into AIDA ............................................... 33

6 Experiments .............................................................. 37
  6.1 Experiments on NEDSH ............................................... 38
    6.1.1 Experiments on Mention-Name Matching .................... 38
    6.1.2 Experiments on Mention-Entity Mapping .................... 40
  6.2 Experiments on Integration with AIDA .......................... 41

7 Conclusion and Future Work ......................................... 45
  7.1 Conclusion ........................................................ 45
  7.2 Future Work ....................................................... 45

List of Figures ........................................................... 46

List of Tables ............................................................ 49

Bibliography ............................................................. 51
Chapter 1

Introduction

1.1 Motivation

Natural language text on the Internet, including web pages, articles and blogs, contains mentions of entities that can be recognized and categorized into different categories such as persons, locations, organizations, etc. by Named Entity Recognition (NER) tools. However, a name is often ambiguous even when we know the category that it belongs to. Let’s take these following sentences as an example

“They performed Kashmir, written by Page and Plant. Page played unusual chords on his Gibson.”

“Page” can be found as a mention for a person name by NER tools, however how can we know that it should refer to guitarist Jimmy Page and not to Google founder Larry Page? Establishing mappings between mentions of potentially ambiguous names to the actual entities is the target of Named Entity Disambiguation (NED) task.

The particular architecture described in AIDA [1], which is an on-line tool for entity detection and disambiguation developed at Max-Planck Institute for Informatics, is to build the undirected weighted graph consisting of mention-name, name-entity and entity-entity edges. The weight of an edge is estimated to be the similarity between two vertices. Then the correct entities will be exported by finding the best sub-graph such that:

1. It contains all mention nodes and exactly one mention-name-entity path for each mention.

2. It has the best score. The score for a sub-graph is defined as the sum of all its edge weights.
However, there is a problem, that we must face throughout an NED system, called similarity search. It might be the process of finding out names similar to a mention, estimating the similarities between mentions and entities via their contexts or estimating the similarities of entity-entity pairs. With a large amount of data, they may become bottlenecks.

Additionally, similarity search in an NED system must be applied in a wide range of thresholds. For example, the similarity between a mention and a name should be very high. However, the threshold which is used to extract related entity pairs (the preprocessing step to speed up the process of estimating the coherence among entity candidates), on the contrary, is small according to the work on AIDA. Therefore, it is challenging to solve similarity search problem well or in other words to have a good NED system.

1.2 Problem statement

In this thesis, we consider the input of natural language text (web page, news article, etc.) with mentions (noun phrases that potentially denote named entities) and aim to map them onto their proper entries in a knowledge base such as Yago \[2, 3\]. With the purpose of building an efficient Named Entity Disambiguation system, we are confronted with following problems:

1. **Typos in natural language text:** Efficiently searching similar names to a given mention in a large set (for instance: more than 6 million names of entities in Yago). In some systems (e.g. AIDA), this step might be cut off, which means they work only on exactly matching of mention-name. However, if there is a typo, they might never find out the correct entity for a mention.

2. **Mention-entity similarity:** Efficiently estimating the similarity between a mention and an entity via their contexts. For any NED systems, mapping mentions to entities that have the similar contexts to the input context is necessary. In this thesis, we aim to find an efficient way to estimate these similarities in terms of speed and storage.

3. **Entity-entity similarity:** Efficiently estimating the coherence among entity candidates for all mentions. The default method, which is to calculate the similarities between all pairs of entities in a candidate set, is expensive \(O(n^2)\) where \(n\) is the number of entity candidates). Therefore, an efficient way of estimating the coherence among entity candidates may bring great benefits.
For the second and the third problems, we will not work on how to create the context of an entity, however we use the results from AIDA which is a state-of-the-art project on extracting features for concepts of NED applications.

1.3 Contributions

We make the following contributions:

1. We provide an efficient method for Named Entity Disambiguation that efficiently solves those problems mentioned in Section 1.2.

2. We provide a disambiguation service via a generic API which employs recent similarity hashing techniques (Locality Sensitive Hashing and Spectral Hashing) to match a given mention to similar names. The connection to the server where all names are indexed is implemented via remote services (RMI service). Therefore, this part can be easily integrated into any NED systems.

3. We also integrate our work into AIDA such as the preprocessing step of extracting related entity pairs that helps to speed up the process of estimating the coherence among entity candidates for all mentions. Particularly, this step works with small similarities which is not the default application of similarity search. Plus, this work contributes to “KORE: Keyphrase Overlap Relatedness for Entity Disambiguation” paper [4].

1.4 Outline

The rest of the thesis is organized as follows. Related work in Named Entity Disambiguation is shown in Chapter 2. In Chapter 3, we provide some basic concepts in similarity search. Chapter 4 presents a state-of-the-art NED system named AIDA and some ways to make it work better. In Chapter 5, we describe an efficient Entity Disambiguation system by using similarity hashing. Our experiments are shown in Chapter 6. Finally, Chapter 7 makes a conclusion and proposes some extending directions of our work.
Chapter 2

Related Work

A Named Entity Disambiguation system maps noun phrases (or mentions) that potentially denote named entities onto entities in a knowledge base. Mentions might be recognized and categorized by Named Entity Recognition (NER). Therefore, not only NED problem but related concepts including NER and knowledge base also are introduced in this chapter.

2.1 Named Entity Recognition

Automatically identifying and classifying named entities is an important task for many natural language processing tasks such as information extraction, information retrieval and machine translation. Specific Named Entity Recognition task first introduced at Message Understanding Conference 6 (MUC-6) attracted wide participation. Its goal is to identify mentions of entities in text, and their labelling with one of several entity types [5] as follows:

- **ENAMEX**: proper names and acronyms designating persons, locations, and organizations.
- **TIMEX**: absolute temporal terms.
- **NUMEX**: numeric expressions, monetary expressions, and percentages.

It is extended to 7 classes (time, location, organization, person, money, percent, date) at MUC-7 (1997). Since then, it has witnessed a significant improvement on NER systems for multiple languages [6–9].
There are two main types of NER system: manually built rule-based systems [9, 10] and statistically based systems [6, 7] (both require a dependency parser to work on a pre-processing step). On the one hand, a rule-based system consists of a set of manually created rules which utilizes morphological information (e.g. upper case, lower case characters), syntactic information (e.g. part of speech) or contextual information. It does not require an annotated training corpus, however the conflict over rules might be the problem. Therefore, this method fits with uncommon languages rather than English. On the other hand, a statistically based system is built by employing machine learning models such as Hidden Markov Model, Support Vector Machine and Conditional Random Field to train on an annotated corpus. With good English corpora (MUC-6, MUC-7, CoNLL 2003), statistically based systems achieve very high accuracy (e.g. 92% of F-mesure [6]).

However, note that an entity (for instance, George W. Bush, the former president of the U.S.) might be referred by multiple mention forms (such as “George Bush” or “Bush”). Plus, a mention can also refer to multiple entities. For instance, the mention “Bush” in “President Bush said that it’s the time to leave Iraq.” can refer to two U.S. presidents or the football player Reggie Bush. Thus, even though mention “Bush” is recognized as a mention for a person name, it seems that we need more specific information like the actual entity which it refers in the real life or in a knowledge base. This is out of reach of NER task.

### 2.2 Automatic Knowledge Base Construction

Automatic knowledge base construction in machine-readable representations, which is a basic field in AI, becomes more and more important with the dramatic growth of data on the Internet. Its target is to extract and structure knowledge from text corpora by using information extraction technologies including pattern matching, natural-language parsing and statistical learning [11, 12]. At first, almost successful knowledge bases were made manually. These can be listed as WordNet [13] or Cyc or OpenCyc [14]. However, they suffer from low coverage, high cost for quality assurance. Along with the success of Wikipedia and algorithmic advances in Information Retrieval, DBpedia [15] converts Wikipedia content into structured knowledge. DBpedia has harvested facts from Wikipedia infoboxes at large scale and also interlinks its entities to other resources. Known as another extension, Yago [2, 3] integrates the information of class memberships from Wikipedia category names with the taxonomic backbone of WordNet.

Once a knowledge base is built, all objects (e.g. places, people, organizations) are represented as entities. Particularly, to deal with synonymy and ambiguity, Yago provides
several relations, including “means”, “isCalled”, etc., that map names onto entities. For example, “means” relation describes:

\[
\text{Einstein means Albert\_Einstein}
\]

which means “Einstein”, which is a name, may refer to entity “Albert\_Einstein” - a person in real life.

The knowledge base also keeps information of authoritative sources for an entity, for example, the corresponding Wikipedia article. Therefore, with a data-mining step, the context of an entity (features or key-phrases) could be extracted. They might be the link anchors texts of an Wikipedia article including category names, citation titles and external references \[1\]. Those key-phrases help to find out the correct entities for mentions in a specific context.

### 2.3 Named Entity Disambiguation

Named Entity Recognition applications can recognize noun phrases that potentially denote named entities as mentions. However, a mention is usually not a canonical entity, uniquely denoted in a knowledge base. In other words, it is still ambiguous. In Natural Language Processing, Named Entity Disambiguation (NED) task works further with the target of mapping each mention onto a canonical entity. It is similar to the more general task called Word Sense Disambiguation \[16\] which maps content words onto a
predefined inventory of word sense. However, the ambiguity of entity names tends to be much higher.

Entity Matching problem [17, 18], deciding if two given mentions in the text (e.g. Michael Jackson and Michael J. Jackson) refer to the same entity, might be considered as a simple form of NED. Shen [19] presented an approach combining focusing on exploiting syntactic similarities among mentions and semantic integrity constraints. For example, a mention with age 2 cannot match a mention with salary 200K. This work achieves a good accuracy for some domains (e.g. for entities who are researchers, authors of papers in conferences).

Another similar task is Entity Resolution which is also known as entity reconciliation or record linkage [20, 21]. Entity Resolution aims to map a given mention (might include a textual context) onto a semantic class. A variant of this problem is to check if two mentions or records are duplicates [22]. The typical first step is to estimate the similarities between mentions in context and possible entity candidates. Plus, the new approach of applying machine learning to consider the joint disambiguation of multiple entities [23] comes closer to NED.

Back to NED problem, Bunescu [24] was the first to come up with the approach of using Wikipedia for explicit disambiguation by defining a similarity measure to compare the input context of a mention to the Wikipedia category of each entity candidate. This framework was extended by using richer features for the similarity comparison [25–27]. Particularly, instead of using the similarity function directly, [27] introduced a supervised learning step to estimate feature weights. [27] also added a notion of semantic relatedness between candidate entities in the unambiguous mentions in the context. Plus, another feature, that is the similarity of common noun phrases in the input context and Wikipedia article names, was considered by [26]. However, these approaches are limited to mapping each mention separately.

[28] has been the first work on using collective-learning model for joint mapping of all mentions. This method built a factor graph of the pair-wise coherence of entity candidates for two different mentions. Finding the maximum of posteriori estimator for the joint probability distribution of all mappings is considered as a hard optimization problem. Therefore, instead of computing the optimal objective, an approximation based on heuristics like relaxing an integer linear program was presented. [28] also gave a simple local hill-climbing algorithm that is comparable in speed and quality to LP relaxation. However, this work still faces high computational cost problem.

Recently, there is a bunch of projects on automatically building knowledge bases from natural-language text including KnowItAll [29], YAGO and its tool SOFIE [30, 31],
Chapter 2 Related Work

StatSnowball [32], ReadTheWeb [33], and the factor-graph work by [34]. Among them, only SOFIE can map names onto canonical entities. Nevertheless, the method, folding the NED into its MaxSat-based reasoning for fact extraction, is computationally expensive. Thus, it is not used for on-line disambiguation of entire natural language texts.

Hoffart [1] introduced a robust method for collective disambiguation which combines three measures:

- The probability of a mention matching an entity.
- The similarity between the context of a mention and the context of an entity.
- The coherence among entity candidates for all mentions in the input context together.

Figure 2.2 shows the architecture of collective Named Entity Disambiguation systems. First, mentions of entities in the text are extracted by a Named Entity Recognition application. After that, all names that are similar to a mention are listed. Based on the information of name-entity from the knowledge base, the entity candidate set is generated. As a result, the weighted graph of mention-name-entity is built with the weights estimated to be the similarity between mention-name, the probability of a name representing for an entity or the coherence among entity candidates for all mentions. Mentions are matched to entities by the best joint mapping sub-graph. Note that Hoffart [1] maps mentions onto entities directly, which speeds up their system (AIDA). However, it can not deal with typos in natural language.

It is certain that estimating the weights which include mention-name similarity, name-entity similarity and entity-entity coherence, is important for an NED system. This thesis does not focus on how to extract key-phrases (context) for an entity but on how to
efficiently estimate the similarities between mention-name, name-entity and entity-entity. In other words, we attempt to speed up NED systems with an equal or almost equal accuracy.
Chapter 3

Similarity Hashing

This chapter introduces state-of-the-art hashing methods including Locality Sensitive Hashing, Min-wise Hashing and Spectral Hashing. By using these methods, we can solve similarity search problem (for example: the process of matching a given mention to all similar names) to achieve an efficient Named Entity Disambiguation system. All of these methods are implemented in our system and can be selected by a user for disambiguating entities. Additionally, we also give the comparison between Locality Sensitive Hashing and Spectral Hashing for the mention-name matching part in the experiments on our system (Section 6.1.1).

3.1 Similarity Search

The nearest neighbour search problem arises in a large various database applications including image databases, document collections, time-series databases and genome databases [35–38]. It involves a collection of objects (images, videos, documents and even plain text, etc.) that are characterized by relevant features and represented as points in a space. Given a query in form of point in this space, we need to find objects having high similarities to the query.

Definition 3.1 (Nearest Neighbour Search (NNS)[39]). Given a set $P$ of objects represented as points in a space, pre-process $P$ so as to efficiently answer queries by finding the point in $P$ closest to a query point $q$.

This definition can be naturally extended to the case of $k$-Nearest Neighbour Search where we wish to return all $k$ points in the space that are closest to the query point. However, in some cases where we are required to return all points that are close enough to the query point, this problem is defined as another form:
Definition 3.2 (Similarity Search). Given a set $P$ of objects represented as points in a space, a query $q$ and a threshold $\tau$, pre-process $P$ so as to efficiently return all points $p$ such that $d(q, p) \leq \tau$, where $d(q, p)$ is the distance between point $p$ and the query $q$.

According to these definitions, there are three sub-problems which have to be solved as follows:

- Select features or construct a representation of an object.
- Define a distance measure. Several common measures are Jaccard, Hamming, Euclidean distance, etc.
- Apply an efficient algorithm for searching.

Features selection certainly depends on the context of using similarity search. We will discuss this step in more detail in Chapter 5 on the context of Named Entity Disambiguation problem.

3.2 Distance Measures

3.2.1 Definition

Suppose we have a set of points, called a space. A distance measure (called metric) on this space is a function $d(x, y)$ that takes two points in the space as arguments and produces a real number, and satisfies the following axioms [40]:

- $d(x, y) \geq 0 \forall x, y$ (non-negative).
- $d(x, y) = 0$ if and only if $x = y$ (indiscernible).
- $d(x, y) = d(y, x) \forall x, y$ (symmetry).
- $d(x, y) + d(y, z) \geq d(x, z) \forall x, y, z$ (triangle inequality).

where the first condition (non-negative condition) can be proved by the others:

$$2d(x, y) = d(x, y) + d(y, x) \geq d(x, x) = 0.$$
3.2.2 Jaccard Distance

Considering a special type of space where every point takes only (1 or 0) for each dimension, we can represent a point as a set:

\[ S \subseteq \Omega = \{0, 1, 2, ..., D - 1\}, \]

where \( D \) is the number of dimensions. In the rest of this chapter, we only consider similarity search problem over sets on this type of space. Now the Jaccard distance \( d \) of two sets \( S_1 \) and \( S_2 \) is defined by \( 1 - R \) where \( R \) is the Jaccard similarity (or the resemblance - a normalized similarity measure) between \( S_1 \) and \( S_2 \):

\[ R = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}. \]

We can easily prove that Jaccard distance satisfies all axioms mentioned in Section 3.2.1. In other words, it is a metric.

Note that:

\[ 1 - R = d \leq \tau \iff R \geq 1 - \tau. \]

In the context of this thesis, for similarity search problem, instead of using Jaccard distance, we use Jaccard similarity to find similar sets (points) for a query, which follows the above formula.

3.3 Min-wise Hashing

Typically, in text processing area (i.e. web (or document) duplicate detection), features (or shingles) are n-gram of words (or tokens). Therefore, the number of features should be large to avoid collision (e.g. \( 2^{40} \) or \( 2^{64} \) with 5-grams) \([41, 42]\). Min-wise hashing is first introduced as a good approach to duplicate Web page removal by reducing the number of dimensions for features’ space \([41-43]\). Since then, there have been considerable theoretical and methodological developments to make it become a standard technique for estimating set similarity (e.g. resemblance). In this section, we will give an introduction to this algorithm.

**Definition 3.3** (Min-wise Independent Permutations\([43]\)). A subset \( F \) of a symmetric group is min-wise independent if for any set \( X \subseteq [n] \) and any \( x \in X \), when permutation \( \pi \) is chosen at random in \( F \) we have:

\[ \Pr (\min \pi (X) = \pi (x)) = \frac{1}{|X|}. \]
3.3.1 Resemblance Estimation

We are still working on two sets $S_1$ and $S_2$ in space $\Omega$ mentioned in Section 3.3.1. Suppose a random permutation $\pi$ is performed on $\Omega$,

$$\pi : \Omega \rightarrow \Omega$$

An elementary probability argument shows that:

$$Pr\left(\min(\pi(S_1)) = \min(\pi(S_2)) \right) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = R.$$ 

There are several methods applied for computation of min-hashes. In our work, we use linear transformation:

$$\pi : h(x) = ax + b \mod P.$$ 

$$\min(\pi(S)) = \min(a S + b \mod P).$$

where $P$ is the dimensionality of output vectors that we wish. Normally, $P$ should be a big prime number to avoid collisions. Plus, $a$ and $b$ are random permutations such that $a \neq 0$.

For example, with

$$\pi : h(x) = 5x + 7 \mod 31$$

$$S = \{3, 10, 7, 4, 5\},$$

we have:

$$\min(\pi(S)) = \min\{22, 26, 11, 27, 1\} = 1.$$ 

After employing $k$ min-wise independent permutations (MIPs) $\pi_1, \pi_2, ..., \pi_k$, we might estimate $R$ without bias as follows:

$$\hat{R} = \frac{1}{k} \sum_{j=1}^{k} 1\{\min(\pi_j(S_1)) = \min(\pi_j(S_2))\}.$$
Algorithm 3.1 The B-bits Min-wise Hashing Algorithm [44].

**Input:** Sets $S_n \in \Omega = \{0, 1, 2, ..., D - 1\}, n = 1$ to $N$.

**Output:** Pairwise resemblance estimations in this set.

1. Generate $k$ random independent permutations $\pi_j : \Omega \rightarrow \Omega, j = 1$ to $k$.
2. For each set $S_n$ and each permutation $\pi_j$, store the lowest $b$-bits of $\min(\pi_j(S_n))$, denoted by $e_{n,i,j}$, $i = 1$ to $b$.
3. Estimate the resemblance between two sets. For example $S_1$ and $S_2$.
4. Compute:
   \[
   \hat{E}_b = \frac{1}{k} \sum_{j=1}^{k} \left\{ \prod_{i=1}^{b} 1\{e_{1,i,\pi_j} = e_{2,i,\pi_j}\} = 1 \right\}.
   \]
   \[
   r_1 = \frac{f_1}{D} \quad \text{and} \quad r_2 = \frac{f_2}{D} \quad \text{where} \quad f_1 = |S_1|, \quad f_2 = |S_2|.
   \]
   \[
   A_{1,b} = r_1 [1 - r_1]^{2^b - 1},
   \]
   \[
   A_{2,b} = r_2 [1 - r_2]^{2^b - 1},
   \]
   \[
   C_{1,b} = A_{1,b} \frac{r_2}{r_1 + r_2} + A_{2,b} \frac{r_1}{r_1 + r_2},
   \]
   \[
   C_{2,b} = A_{1,b} \frac{r_2}{r_1 + r_2} + A_{2,b} \frac{r_1}{r_1 + r_2},
   \]
5. Estimate $\hat{R} = \frac{\hat{E}_b - C_{1,b}}{1 - C_{2,b}}$.

3.3.2 B-bits Min-wise Hashing Algorithm

[44] developed a new b-bits min-wise hashing approach. For each set $S$ and each permutation $\pi$, they store the lowest $b$-bits ($b = 1$ or 2) of $\min(\pi(S))$ instead of storing the entire hashed value of $\min(\pi(S))$. When applying this approach, they can not use the normal resemblance estimation discussed in Section 3.3.1 due to collisions. Therefore, they prove some basic theoretical results and provide an unbiased estimator of the resemblance for any $b$ (Algorithm 3.1). According to their experiments, it gains substantial advantages in terms of storage space by using 1 or 2 bits to store a hashed value. For example, using $b = 1$ may reduce the storage space at least by a factor of 21.3 (or 10.7) in comparison to $b = 64$ (or $b = 32$).

In a nutshell, even though we can use min-wise hashing to compress the data set by reducing the number of dimensions, it may still be impossible to be applied efficiently for similarity search problem. In particular, it takes $O(n)$ time to search for an object and $O(n^2)$ time to list all similar pairs (deduplication) in a set of size $n$. Therefore, it is not good enough for similarity search problem over a large number of items.
3.4 Locality Sensitive Hashing (LSH)

Min-wise hashing helps reduce the number of dimensions, however it is still quadratic in the number of items (sets). Locality Sensitive Hashing (LSH), presented in [39, 45], is a better similarity search technique that works efficiently for large and high-dimensional data sets.

Definition 3.4 (Locality Sensitive Hashing [46]). A locality sensitive hashing scheme is a distribution on a family $F$ of hash functions operating on a collection of objects, such that for two objects $x, y$,

$$\Pr_{h \in F}[h(x) = h(y)] = \text{sim}(x, y).$$

Where: $\text{sim}(x, y)$ is a similarity function defined on the collection of objects.

In this thesis, we focus on Jaccard LSH using min-wise independent permutations. The main idea is concatenating min-hashed values from some random permutations into a longer signature. For example, by employing $k$ independent min-hash permutations: $\pi_1, \pi_2, \ldots, \pi_k$, we have the new hashed value for an object (a set) $S$:

$$\min(\pi_1(S)) \oplus \min(\pi_2(S)) \oplus \ldots \oplus \min(\pi_k(S)).$$

Then, each object is hashed several times. Similar objects are more likely to be hashed to the same bucket than dissimilar objects are. We consider any pairs that are hashed to the same bucket for any of the hashes to be a candidate pair. We then check only these candidate pairs for the exact similarities.

Two things which should be considered are the number of false positives and the number of false negatives. On the one hand, the longer a signature is, the fewer objects are hashed into a bucket. Therefore, we might have a better precision but, by contrast, we lose some objects. That means the number of false positives decreases and the number of false negatives increases. On the other hand, increasing number of hashes results in generating more candidate pairs. Thus, the number of false positives increases and the number of false negatives decreases.

Next, we give a detailed analysis of this approach. Assume that we are working on a concatenation of $k$ min-hash permutations for a signature. Each object is hashed $l$ times. If the Jaccard similarity of $S_1$ and $S_2$ is $R$, then for each permutation $\pi$:

$$P[\min(\pi(S_1)) = \min(\pi(S_2))] = R.$$
Chapter 3 Similarity Hashing

For each hash $h$:

$$P[h(\pi(S_1)) = h(\pi(S_2))] = R^k.$$ 

Therefore,

$$P[h(\pi(S_1)) \neq h(\pi(S_2))] = 1 - R^k.$$ 

As a result, after $l$ rounds, the probability of $(S_1, S_2)$ not being a candidate pair is

$$P\left[ \forall j \in \{1, 2, ..., l\} \ h_j(\pi(S_1)) \neq h_j(\pi(S_2)) \right] = (1 - R^k)^l.$$ 

The probability of $(S_1, S_2)$ being a candidate pair is $1 - (1 - R^k)^l$. Therefore, to achieve an efficient system using Locality Sensitive Hashing, parameters $(k, l)$ need to be chosen carefully.

In terms of speed for deduplication purposes, it is much faster than min-wise hashing. The running time to list all similar pairs in a set of size $n$ is $O(k \cdot l \cdot m \cdot n)$ where $m$ is the upper bound for the number of features of an object.

3.5 Spectral Hashing

Similar to other approaches in semantic hashing [47], spectral hashing [48] aims to map efficiently each object in the database onto a compact binary code so that similar items tend to be mapped onto similar codes. Spectral hashing requires that each bit (in the code) has 50% chance of being zero or one, and bits are independent of each other. Then among these codes, spectral hashing seeks to the code which minimizes the average Hamming distance between similar points. Let:

- $\{x_i\}_{i=1}^n$ be the dataset in $R^d$.
- $\{y_i\}_{i=1}^n$ be the list of codes ($k$ bits).
- $W_{n \times n}$ be the affinity matrix. Where $W(i, j) = exp(-\|x_i - x_j\|^2/\epsilon^2)$, $\epsilon$ defines the distance in $R^d$ which corresponds to similar objects.

Then the problem can be formulated as follows:
minimize the Hamming distance: \( \sum_{ij} \| y_i - y_j \|^2 \).

subject to: \( y_i \in -1, 1^k \).
\[
\sum_i y_i = 0.
\]
\[
\frac{1}{n} \sum_i y_i y_i^T = I.
\]

For each single bit, this problem is the problem of graph partitioning and can be shown to be NP hard. For \( k \) bits, it can be considered as finding \( k \) independent balanced partitions, each should have as low cut as possible. The solution of this problem is a subset of threshold eigenvectors in the graph Laplacian (\( L_p \)) \([49, 50]\). Additionally, Weiss \([48]\) designed an efficient way of calculating the code of an object in the dataset by utilizing results on convergence of graph Laplacian eigenvectors to the Laplace-Beltrami eigenfunctions of manifolds \([51, 52]\) as follows:

- Find the principal components of the training dataset \( \{ x_i \} \) using PCA.
- Calculate the \( k \) (number of bits which we desire) smallest single-dimension analytical eigenfunctions of \( L_p \) using a rectangular approximation along every PCA direction.
- Seek the analytical eigenfunctions at zero, to obtain binary codes.

This approach outperform the state-of-the-art in the dataset of 80 million images from the Internet \([53]\). However, it assumes that a multidimensional uniform distribution generated the data. Therefore, it might not work very well in the context of NED task.
Chapter 4

AIDA - Entity Detection and Disambiguation Tool

In this chapter, we will move to a specific Accurate On-line Disambiguation of Named Entities system named AIDA [1] which is developed at Max-Planck Institute for Informatics\(^1\). AIDA considers an input text (web page, news article, etc.) with mentions that are recognized by Stanford NER Tagger [6] and aims to map them to entities in Yago [2, 3].

4.1 Stanford Named Entity Recognition Tagger

Stanford NER [6], which is developed by The Stanford Natural Language Processing Group, is a state-of-the-art Named Entity Recognition tool. It demonstrates a constraint model that is effectively combined with an existing sequence model (CRF) in a factored architecture to successfully impose various sorts of long distance constraints. For example, in Figure 4.1, there is a constraint of enforcing label consistency for two tokens “Tanjug”. Particularly, Stanford NER incorporates these constraints into a CRF-based statistical model by using Gibbs sampling. By doing this, it achieves excellent results in recognizing and classifying names of things (mentions) in natural language text up to seven classes: time, location, organization, person, money, percent, date. Figure 4.2 shows the snapshot of Stanford NER GUI.

Currently, AIDA only considers the boundary of mentions, but not types of these mentions. For instance, even though “Barack Obama” (Figure 4.2) is recognized as a mention

\(^1\)http://www.mpi-inf.mpg.de/yago-naga/aida/
for a person, AIDA only uses the information that mention “Barack Obama” must be disambiguated. The type of this mention (person) is ignored.

4.2 Features and Measures

AIDA introduces a robust method for collective disambiguation which combines three measures:

- The prior probability of an entity being mentioned.
- The similarity between the context of a mention and the context of an entity.
- The coherence among entity candidates for all mentions in the input context together.

\(^2\)http://nlp.stanford.edu:8080/ner
4.2.1 Popularity Prior

AIDA supports multiple forms of popularity-based priors, however the most successful model is based on Wikipedia link anchors. That is counting how often a mention, that constitutes an anchor text, refers to a particular entity. A data-mining step is then applied to estimate probability distribution of a name over candidate entities. For example, in the following sentence:

“They performed Kashmir, written by Page and Plant.”

mention “Kashmir” refers to Kashmir (the region) in 90.91% of all occurrences and in 5.45% to Kashmir (song).

4.2.2 Mention-Entity Similarity

There is no doubt that the key to map mentions onto entities is the matching of contexts on both sides. On the mention side, AIDA uses all tokens in the text (except stop-words and the mention itself) as context. On the entity side, AIDA employs an off-line data-mining step to determine characteristic key-phrases for each entity and their statistical weights. For example, key-phrase candidates for an entity might be Wikipedia article’s link anchors texts, including category names, citation titles, and external references, etc. For the similarity of a mention \( m \) and a candidate entity \( e \), this score is aggregated over all key-phrases of \( e \) (\( KP(e) \)) and all their partial matches in the text. Note that using partial matches for key-phrases helps to deal with many different forms of an object in natural language.

\[
simscore(m, e) = \sum_{q \in KP(e)} \text{score}(q)
\]

where:

\[
\text{score}(q) = \frac{\text{number}\_of\_matching\_words}{\text{length}\_of\_cover(q)} \left( \frac{\sum_{w \in q} \text{weight}(w)}{\sum_{w \in \text{cover}(q)} \text{weight}(w)} \right)^2
\]

In the above formula:

- The cover of a key-phrase is the shortest window of words in the text that contains a maximal number of common words with it. For instance, the cover length of the text “winner of many prizes including the Grammy” and the key-phrase “Grammy award winner” is 7.
• The weight of a word with an entity \( \text{weight}(w) \) is estimated by MI(mutual information) or the collection-wide IDF weight.

It is clear that this computation is expensive. A simple improvement is to represent the input context by key-phrases. For example, if the number of matching words is greater than half of the maximum length between key-phrase \( k \) and a cover, we count that the input context contains key-phrase \( k \). Once we construct the input context as a set of key-phrase, we can employ similarity search approaches such as min-wise hashing to estimate the similarity between it and each entity’s context (Section 5.1.2).

4.2.3 Entity-Entity Coherence

As all entity candidates are registered in a knowledge base (like YAGO), AIDA uses a simple measure of the distance between two entities in terms of type and subclassOf edges. Plus, it also quantifies the coherence between two entities based on the number of incoming links that their Wikipedia articles share. For example, the formula for two entities \( e_1, e_2 \) is

\[
\text{mw}_\text{coh}(e_1, e_2) = 1 - \frac{\log(\max(|IN_{e_1}|, |IN_{e_2}|)) - \log(|IN_{e_1} \cap IN_{e_2}|)}{\log(|N|) - \log(\min(|IN_{e_1}|, |IN_{e_2}|))}
\]

where: \( N \) is the total number of entities in the knowledge base, and \( IN_e \) is the set of all entities linking to entity \( e \). If this formula returns a negative number, 0 will be used instead. In fact, many entity pairs have the coherence of 0 (or \( \approx 0 \)). Therefore, by employing similarity search approaches to remove these pairs, we might make AIDA faster.

4.3 Graph Model and Algorithm

4.3.1 Mention-Entity Graph

AIDA directly maps mentions extracted by the Stanford NER Tagger onto entity candidates on the knowledge base. From the popularity, similarity, and coherence measures discussed in Section 4.2, a weighted undirected graph with mentions and candidate entities as nodes is constructed as shown in Figure 4.3. In comparison with the architecture of a general collective NED system (Figure 2.2), there is no mention-name mapping. There are three kinds of edges in AIDA’s graph model:
• A mention-entity edge is weighted with a similarity measure or a combination of popularity and similarity measure

• An entity-entity edge is weighted based on Wikipedia-link overlap, or type distance, etc.

• An entity-keyphrase edge is weighted statistically by an off-line data-mining step as discussed in Section 5.1.2.

The mention-entity graph is dense on the entity side with hundreds or thousands of nodes, because there might be many candidate entities for common mentions (e.g. common first names, last names, etc.).

**Figure 4.3:** AIDA’s Architecture [1].

### 4.3.2 Graph Algorithm

Given a mention-entity graph, NED problem is defined as to compute a dense sub-graph that would ideally contain all mention nodes and exactly one mention-entity edge for each mention. Algorithm 4.1 is presented to solve this problem. Note that an entity is *taboo* if it is the last candidate for a mention it is connected to; and the weighted degree of a node in the graph is the total weight of its incident edges.

The output of the main loop would often be close to the desired result, but may still have more than one mention-entity edge for one or more mentions. In this case, AIDA considers an exhaustive enumeration and assessment of all possible solutions. Alternatively, AIDA performs a faster local-search algorithm where entity candidates are randomly selected.
Algorithm 4.1 Graph Distribution Algorithm[1].

**Input:** Weighted graph of mentions and entities.

**Output:** Sub-graph with one edge per mention.

1. pre-processing phase; //build the graph
2. **for** each entity **do**
3. calculate distances to all mentions;
4. **end for**
5. keep the closest (5 x mentions count) entities, drop the others;
6. //main loop
7. **while** graph has non-taboo entity **do**
8. determine non-taboo entity node with lowest weighted degree, remove it and all its incident edges;
9. **if** minimum weighted degree increased **then**
10. set solution to current graph;
11. **end if**
12. **end while**
13. post-processing phase; //process solution by local search or full enumeration for best configuration;

with probabilities proportional to their weighted degrees. After doing this step for a pre-specified number of iterations, the solution with the highest total edge-weight is chosen.

### 4.4 Discussion

The AIDA system provides a method of using popularity, similarity, and graph-based coherence for NED problem. Its experiments demonstrate a state-of-the-art performance in term of accuracy. However, there are some parts that could be improved to make this tool work better.

- AIDA directly maps mentions onto entities registered in the knowledge base, resulting in lack of ability to deal with natural language text. For instance, when there is a typo in mentions, and thus the right entity is not in the candidate set, AIDA can never find out the correct answer.

- As discussed in Section 4.3.1, the mention-entity graph is dense on the entity side with hundreds or thousands of nodes. Therefore, the computation to weight entity-entity edges is expensive - $O(n^2)$ (where $n$ is the number of entity candidates). The fact is that many entity-entity pairs are not (or hardly) related, employing a pre-processing step (LSH) to remove these pairs might gain great benefits in terms of running time. However, this is a challenging problem due to the low similarities between entity-entity pairs.
Chapter 5

NEDSH: Named Entity Disambiguation System via Similarity Hashing

Based on the AIDA’s architecture and problems described in Chapter 4, we figure out an efficient way to disambiguate named entities by similarity hashing. It can be implemented to work as an independent system (NEDSH) or partly integrated into an existing NED system (AIDA). This chapter is constructed as follows:

- In the first section, we introduce an efficient system of disambiguating named entities.
- The second section describes the implementation in detail.
- Finally, the third section shows how to partly integrates the new architecture into AIDA to make it work better.

5.1 System Architecture

Our work mainly focuses on speeding up the performance of an NED system; therefore, we will not work on how to label noun phrases as mentions in the input text, or extract key-phrases for an entity from the knowledge base (Yago). Instead, these results from AIDA are reused.

Based on the architecture of collective disambiguation systems described in Section 2.3, we modify AIDA’s architecture by adding one more step of finding the entity candidate
set for a mention. Particularly, instead of directly mapping a mention onto entities in the knowledge base, we first detect names that are similar to the mention. The entity candidates then will be listed based on these names. In the context of Yago, we use “means” relation. Note that, it is a \( n : m \) relation, which means a name can be mapped to multiple entities and some names might map to the same entity. By doing this, the system can deal with typos in natural language. For example,

“They performed *Kashmis*, written by Page and Plant.”

There is a typo (Kashmis) in this example. Therefore, AIDA will never be able to find out the correct entity (Kashmir song) in this case due to the missing of entity “Kashmir” in the candidate set. However, in NEDSH, the name “Kashmir” is listed as a similar name of the mention “Kashmis”. Thus, the entity “Kashmir song” is in the candidate set. It certainly takes additional time for mapping mentions to similar names; however, we need this step to make the system work well on natural language. Plus, based on our experiment shown in Chapter 6, this step is pretty fast.

NEDSH mainly follows the ideas of collective disambiguation from AIDA. However, we concentrate on three main points of building the undirected weighted mention-name-entity graph to make NEDSH efficiently disambiguate named entities in natural language text as follows:

- Efficiently matching a mention to similar names.
- Efficiently estimating the similarity between the input context and each entity candidate’s context.
- Efficiently estimating the coherence among entity candidates for all mentions.

### 5.1.1 Mention-Name Matching

There are a large number of names, for example: more than 6 million names in “means” relation in Yago. Plus, we need to deal with the short forms of person names (first names and last names) in natural language. For example, instead of “Larry Page”, only “Page” appears as a mention in the input text. Therefore, NEDSH also considers all first names and last names of person names, resulting in an increase in the number of names up to over 6.5 million. There is no doubt that finding out all similar names for a mention in such a huge set is not easy.

NEDSH employs a server that indexes all names by state-of-the-art hashing techniques including Locality Sensitive Hashing (LSH) and Spectral Hashing (SH). Users can
select the hashing technique which they desire. The communication to the server is implemented via Remote Method Invocation (RMI) as shown in Figure 5.1. The details of our implementation and API functions are shown in Section 5.2.

To hash names, NEDSH first constructs the representation for each name (Section 5.2.3). Based on our real experiments, representing a lower case name by a 2-gram set brings the best quality. Once a name is represented by a representation vector, it can be loaded into the hash table (LSH table or SH table) in the server. Although loading all names into the hash table is time-consuming, NEDSH does this just once. Therefore, this does not affect the main process of disambiguating entities. How to choose parameters of the hash table will be discussed in Chapter 6. When all names are loaded into the hash table, similar names for a mention can be requested via remote communication between client-server (RMI). Note that we use a post-filter at the end to remove all false positives from the hash table.

5.1.2 Mention-Entity Similarity Estimating

Estimating the similarity between the input text (mentions’ context) and each entity candidate’s context is an important step for any NED systems. The fact that the number of entity candidates is not very large (several hundreds), it is possible to go through all the entities in the candidate set to compare their contexts with the input context.
Figure 5.2 shows an efficient method of storing features of an entity in NEDSH that might help to speed up this process and decrease the amount of storage space required.

By using Min-hashing method (for example: employing $k$ random permutations), NEDSH can decide the fixed number of features ($k$) for an entity which need to be stored. How to choose this number to guarantee the quality of estimating process as well as take advantages of the decrease in the number of dimensions is discussed in Chapter 6.

NEDSH uses the method that AIDA use to construct the context vector for a mention from input text. Then, the $k$ random permutations are used to convert this vector to a new form of size $k$ (MIPs vector) which is used to compare with the MIPs vector of each entity candidate.

Note that we can make this process even better by adding one more pre-processing step of combining similar key-phrases (features) of entities. For instance, “the president of US” and “the US president” should be combined. This process will be discussed in detail in Section 5.3. In this section, we only focus on Min-wise hashing technique to reduce the number of dimensions in the feature space.

### 5.1.3 Entity-Entity Coherence Estimating

This section demonstrates an efficient method to estimate entity-entity coherence in NEDSH. As we discussed in Section 4.4, the mention-entity graph is dense on the entity side with hundreds or even thousands of nodes. A straightforward approach that requires
Chapter 5 NEDSH: Named Entity Disambiguation System via Similarity Hashing

O\(n^2\) computations might become a bottle-neck. Therefore, an efficient computation is vital for this process in particular and for the whole system in general. Figure 5.3 provides an overview of two stages in this process.

![Figure 5.3: Overview of Entity-Entity Coherence Estimating Technique.](image)

NEDSH first uses an LHS table to generate all similar entity pairs. Based on our experiment, the similarities between entities are small (i.e. 0.01 for Jaccard similarity). Thus, we use “related entity pairs” instead of similar entity pairs for this part. A related entity pair consists of two entities sharing at least one hash bucket. All these pairs are candidates to compute the exact relatedness. For other pairs, we assume the relatedness is sufficiently low to consider the entities as unrelated. By doing this, we can remove a large number of unrelated pairs, resulting in speeding up the computation. This part is integrated into AIDA, and thus will be discussed more in Section 5.3.

5.2 Implementation

5.2.1 Java Remote Method Invocation

Java Remote Method Invocation (Java RMI) helps to build distributed Java technology-based applications, in which methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts.

Figure 5.4 shows an overview of RMI architecture. RMI applications often comprise two separate programs, a server and a client. On the one hand, a server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects. On the other hand, a client program obtains a remote reference to one or more remote objects located in a server and then invokes methods on
them. RMI provides the mechanism by which the server and the client communicate and pass information.

One of the central and unique features of RMI is its ability to download the definition of an object’s class if the class is not defined in the receiver’s Java virtual machine. All of the types and behaviour of an object, previously available only in a single Java virtual machine, can be transmitted to another, possibly remote, Java virtual machine. RMI passes objects by their actual classes, so the behaviour of the objects is not changed when they are sent to another Java virtual machine. This capability enables new types and behaviours to be introduced into a remote Java virtual machine, thus dynamically extends the behaviour of an application.

5.2.2 Data Structures and Libraries

Regarding the code for LSH, we use open map structures provided by Colt library from European Organization for Nuclear Research. Colt has been used in numerous applications of scalable scientific and technical computing. In particular, it provides fundamental general-purpose data structures optimized for numerical data, such as re-sizeable arrays, dense and sparse matrices (multi-dimensional arrays), linear algebra, associative containers and buffer management. The map package offers flexible object oriented abstractions modelling automatically resizing maps. It is designed to be scalable in terms of performance and memory requirements.

---

1http://docs.oracle.com/javase/tutorial/rmi/overview.html
Next, we will introduce some main classes in our implementation as follows:

- **class Counter**: This class represents a name. It contains two main arrays of integer. The first one (`keys` array) is the representation vector which consists of global ids of n-grams for a name. It is sorted to speed up the process of calculating the exact similarity between two names in the post-filter. The second one (`vals` array) contains the weights (the frequencies) of each n-gram.

- **class MinHashTable**: This class is a hash table that hashes each name by the concatenation of $k$ permutations. The hash code function to concatenate $k$ hash values ($k$ permutations) into a longer signature is the sum of these values. It is a simple combination which might create collisions. However, it can take advantages of the fast computation. Plus, from our experiments, this function still brings good results in similarity search among strings.

- **class LSHTable**: This is the main class of our implementation for Locality Sensitive Hashing method. It consists of $l$ `MinHashTable`. Two names are similar if they are hashed into the same bucket in any of $l$ `MinHashTable`.

Regarding the implementation of Spectral Hashing, we modify open-source packages\textsuperscript{2,3} which follow the efficient way of calculating Spectral Hashing codes \textsuperscript{48} for objects (names). The step-by-step computation was discussed in Section 3.5.

We set up an RMI server to store the hash table (LSH table or Spectral Hashing table) that indexes all names in the knowledge base (e.g. more than 6.5 million names in Yago including all first names and last names for person names). This server waits for requests for similar names of a mention and returns a list of names to clients. This server might also store all entities and their representations (MIPs vectors). By doing this, clients may also request the similarities between the input context and the context of each entity in the candidate set.

While the first LSH table storing all names is built only once, the second one which is used to extract related entity pairs in the entity candidate set is built separately with each input text (document). Therefore, the amount of time to build an LSH table needs to be considered in this case. We store the hashed values of all entities which are generated by an off-line pre-processing step. Then these values are reused whenever we need to hash an entity. By doing this, we speed up this process.

\textsuperscript{2}https://bitbucket.org/rubyu/hashing/src/e18ba28f6a44/java
\textsuperscript{3}http://www.cs.huji.ac.il/~yweiss/SpectralHashing/
5.2.3 Representation for Hashed Objects

Before hashing objects (names, entities), we need to construct a representation of each object. For entities, the only thing we do is assigning each key-phrase a globally unique id. By doing this, we can represent each entity by a vector of ids. For names, because we need to deal with typos in natural language text, we decide to use n-gram at character level to represent a name.

Tokenizer: we first use the default delimiter set consisting of the space character, the tab character, the newline character, the carriage-return character, and the form-feed character to tokenize the name. For example:

Token set (Michael Jackson): \{Michael, Jackson\}.

N-gram: For each token, we extract the n-gram set (signature set) at character level. This is useful in cases of typos in natural language text. For example:

2-gram set (Michael Jackson): \{mi, ic, ch, ha, ae, el, ja, ac, ck, ks, so, on\}.

Finally, we assign each n-gram a globally unique id, and thus we can represent a name by a vector of ids. For example,

Representation vector (Michael Jackson): \{4, 7, 15, 27, 90, 34, 2, 19, 47, 53, 31, 78\}

where 4 is the id of “mi”, etc.

5.2.4 API Functions

Start the server:

```java
public static void startLSHServer(int k, int l, int n_gram).
public static void startSHServer(int Nbits, int n_gram).
```

The first function creates an LSH table with the parameters \(k, l\) and the second function creates a Spectral Hashing table with the parameter \(Nbits\) which is the number of bits used to hash a name. These two functions allow users to choose the hash table which they desire. Then all names are represented by vectors (the \(n\)-gram set), and loaded into the hash table. If \(n\)-gram is a negative number, a name is only represented by the set of tokens.
Match a mention to similar names:

\[
\text{public static List<String> getSimilarNames(String mention).}
\]

This function returns all similar names for a given mention.

Match a name to entities:

\[
\text{public static List<String> getEntities(String name).}
\]

This function returns all entities related to a given name.

Estimate the similarity between a mention and an entity:

\[
\text{public static double getSimilarity(String mention, int[] context, String entity).}
\]

This function returns the similarity between a given mention in a specific context and an given entity. We assume that the context vector of a mention is already constructed from the input text.

Shut-down the server:

\[
\text{public static void shutdownServer().}
\]

This function stops the RMI server.

### 5.3 Integration into AIDA

In three main parts that we focus on this thesis, the first part which matches a mention to similar names is an independent part. As a result, we can be easily integrated into any NED systems including AIDA. Plus, we did also implement an efficient method to estimate entity-entity coherence that is integrated into AIDA as well. We not only employ an LSH table to extract related entity pair candidates, but we also use another LSH table to combine similar key-phrases (e.g. “President of the United States” and “the United States President”) as a pre-processing step.

Figure 5.5 shows a proposal of the two-stage hashing scheme. To group highly similar key-phrases, we first represent a key-phrase by a set of tokens (as discuss in Section 5.2.3) and assign each token a globally unique id. We hash each key-phrase twice \(l = 2\), and we employ 2 random permutations \(k = 2\) for each time (the average length of a key-phrase in our knowledge base is 2.5 tokens). Each pair of MIPs values is combined to form a new hashed value by summing up their values. For example, assume that
Chapter 5 NEDSH: Named Entity Disambiguation System via Similarity Hashing

Figure 5.5: Overview of the Two-stage Hashing Technique [4].

\[
\text{MIPs(entity } e) = \{13, 5, 3, 8\}
\]

then we hash entity \( e \) twice with:

the first hashed value: \( 13 + 5 = 18 \),
the second hashed value: \( 3 + 8 = 11 \).

Each key-phrase is finally represented by these two hashed values. Note that we do not perform this stage-one hashing to reduce the dimensionality of the key-phrase space, but to capture the notion of partial overlapping key-phrases and to improve the second stage of grouping entities. Additionally, this step increases the similarity between two entities, which helps LSH work better with very low similarities among entities.

To extract related entity pairs, we try to use two kinds of LSH table as follows:

- \( LSH_G \) is a reasonably fast approximation which has nearly the same quality as AIDA, we hash each entity 200 times (\( l = 200 \)), and for each time we use a random permutation (\( k = 1 \)). \( LSH_G \) is geared towards high recall so that the actual computation is executed between all somewhat related entities, but filters out noise. \( LSH_G \) is not much faster than AIDA, but the quality is close to exact AIDA. Sometimes, the quality is even improved as noisy candidates are removed.

- \( LSH_F \) is a really fast approximation which degrades the approximation quality a bit, we hash each entity 1000 times (\( l = 1000 \)), and for each time we use 2 random permutations (\( k = 2 \)), again combining the two MIPs values by summing them up before hashing. \( LSH_F \) is geared towards higher precision with bands of size two, allowing \( LSH_F \) to prune even more entity pairs, speeding up the subsequent computation of the semantic relatedness due to fewer comparisons.
There is an option to switch between $LSH_G$ and $LSH_F$ depending on the main priority of the system which might be the quality or the speed. Note that this is the pre-process which only extracts related entity candidate pairs, and thus, we need to employ a post-filter to calculate the exact similarities for all these pairs. This helps to remove all false positives cause of $LSH_G$ and $LSH_F$. 
Chapter 6

Experiments

In order to judge the effectiveness of our proposed approach, we conducted experiments on three main points: mention-name matching, mention-entity mapping and entity-entity coherence estimating. Especially, the experiments on entity-entity coherence estimating were done on the integration with AIDA, which is a state-of-the-art on-line disambiguation tool for named entities.

We used three measures: precision, recall and F-measure which are orthogonal metrics used for evaluating the goodness of an information retrieval system.

**Definition 6.1** (Precision [54]). Let $D$ be a set of documents, $R \subseteq D$ be the set of relevant documents with respect to a query $q$, $A \subseteq D$ be the set of documents retrieved. The precision is the fraction of retrieved documents that are relevant:

$$\text{Precision} = \frac{|R \cap A|}{|A|}.$$

**Definition 6.2** (Recall [54]). Let $D$ be a set of documents, $R \subseteq D$ be the set of relevant documents with respect to a query $q$, $A \subseteq D$ be the set of documents retrieved. The recall is the fraction of the documents that are relevant to the query and that are successfully retrieved:

$$\text{Recall} = \frac{|R \cap A|}{|R|}.$$

While precision measures the fidelity of a result (how exactly the system delivers the answer), the recall measures the completeness of the result (how many correct answers the system covers). F-measure [54], which is the trade-off between precision and recall, is defined as follows:

$$F_\beta = \frac{(1 + \beta^2) \times \text{precision} \times \text{recall}}{\beta^2 \times \text{precision} + \text{recall}}.$$
where $\beta \leq 1$ means we weight precision more than recall, and by contrast, $\beta \geq 1$ means we weight recall more than precision. In this thesis, we use $F_1$ which weights recall and precision evenly.

6.1 Experiments on NEDSH

6.1.1 Experiments on Mention-Name Matching

Experimental set-up: Because it is impossible to test on the whole set of names (more than 6 million names in Yago), we extract a small set of 667 names. They all contain “paris” and have lengths less than or equal to 20, which results in high similarities among them. In order to cover typos in natural language text, we add a wrong form of each name (by deleting, inserting or changing a random character) to the set. Finally, we use this set (let call ND), which now contains 1333 names, to conduct our experiments.

Locality Sensitive Hashing: First, We calculated the Jaccard similarities for all pairs of entity in ND dataset. After that, we store all pairs which have the similarities $\geq 0.8$. Note that the threshold of 0.8 is chosen based on our experiments to make NEDSH work well for natural language. It should be high to not return too much name candidates, but it also should low enough to deal with typos. For example, considering the name “paris hilton” and a typo “paris hiltom”:

- 2-gram set of “paris hilton” = \{pa, ar, is, hi, il, lt, to, on\}.
- 2-gram set of “paris hiltom” = \{pa, ar, is, hi, il, lt, to, om\}.
- Jaccard similarity = $\frac{8}{10} = 0.8$.

If the threshold is less than 0.8, we will not have the name “paris hilton” in the candidate set. Once all similar pairs are generated, we can evaluate the performance of an LSH table with the parameters $(k, l)$ on ND dataset as follows:

- Load all names in ND dataset into an LSH table built with parameters $k$ and $l$.
- Request “similar names” for each name in ND dataset and compare the result to the similar pairs which we did store.

Figure 6.1 shows the performance of LSH on ND dataset in a range of $k$ and $l$. First, we fix $l$ at 48, and change $k$ from 2 to 30. Even though the highest F-measure value (0.86) is at $k = 14$, we choose $k = 12$ with the acceptable F-measure (0.80) but the very high
recall (0.90). The reason is that the recall should be high (i.e., $\geq 0.9$), otherwise we will lose a large number name candidates. This would badly affects the quality of our system. Plus, the precision does not need to be very high because we can use a post-filter to calculate the exact Jaccard similarities after receiving results from the LSH table, which increases the precision up to 1.0 (Figure 6.2). We then fix $k$ at 12, and change $l$ from 4 to 60. Again, we find that $l = 48$ brings the highest F-measure among points where the recall $\geq 0.9$.

Spectral Hashing: We calculated the Hamming distances for all pairs of entity in ND dataset. After that, we stored all pairs which have the distances $\leq 3$. Finally, we tested the performance of Spectral Hashing on ND dataset in a range of numbers of bits used to hash a name. Figure 6.3 shows the performance of spectral hashing. It seems that the precisions at points owning the recall $\geq 0.9$ are not high ($\approx 0.4$). It might cause the data in ND dataset not generated by a multidimensional uniform distribution (discussed in Section 3.5).

Running time: We tested the running time of the RMI server (Section 5.2.4) in the whole name set in Yago (more than 6 million names). The experiments were conducted on a computer with 16 AMD Dual Cores 3.0 GHz and 256 GB memory under Linux OS.
We requested the RMI server for similar names with 100 queries, which were randomly chosen. The running time of both Locality Sensitive Hashing and Spectral Hashing is shown in Table 6.1. Even on client side, it only takes several milliseconds (5 ms for LSH and 11 ms for Spectral Hashing) on average for a query. This speed is fast enough to guarantee that our API can be integrated into on-line Named Entity Disambiguation tools.

Table 6.1: Running Time for 100 Queries on Mention-name Matching.

<table>
<thead>
<tr>
<th></th>
<th>Server side</th>
<th>Client side</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locality Sensitive Hashing</td>
<td>116 ms</td>
<td>513 ms</td>
</tr>
<tr>
<td>Spectral Hashing</td>
<td>541 ms</td>
<td>1125 ms</td>
</tr>
</tbody>
</table>

6.1.2 Experiments on Mention-Entity Mapping

Table 6.2: Performance of Min-wise Hashing on Mention-Entity Mapping.

<table>
<thead>
<tr>
<th>Number of Permutations</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0.64</td>
<td>0.89</td>
</tr>
<tr>
<td>14</td>
<td>0.74</td>
<td>0.95</td>
</tr>
<tr>
<td>16</td>
<td>0.48</td>
<td>0.93</td>
</tr>
<tr>
<td>18</td>
<td>0.84</td>
<td>0.96</td>
</tr>
<tr>
<td>20</td>
<td>0.84</td>
<td>0.83</td>
</tr>
<tr>
<td>22</td>
<td>0.77</td>
<td>0.92</td>
</tr>
<tr>
<td>24</td>
<td>0.73</td>
<td>0.93</td>
</tr>
<tr>
<td>26</td>
<td>0.75</td>
<td>0.93</td>
</tr>
<tr>
<td>28</td>
<td>0.75</td>
<td>0.90</td>
</tr>
<tr>
<td>30</td>
<td>0.80</td>
<td>0.91</td>
</tr>
</tbody>
</table>

We conducted experiments on a small dataset of 6000 entities (let call ED) which were randomly chosen in Yago. The representation vector for each entity was constructed via key-phrases which are the link anchors texts of an Wikipedia article including category
names, citation titles and external references [1]. After that, we calculated the exact Jaccard similarities for each pair of entity, and store all pairs which have the similarities $\geq 0.8$. Once all similar pairs are generated, we can evaluate the performance of hashing approaches (min-wise hashing and b-bits min-wise hashing) on the ED set.

Table 6.2 shows the performance of min-wise hashing approach on a range of number of permutations ($12 - 30$). By using 30 permutations, we achieve good values of precision and recall ($0.80$ and $0.91$). Table 6.3 shows the performance of b-bits min-wise hashing approach ($b = 1$) on a range of number of permutations ($64 - 640$). It seems that the accuracy is better, and the storage capacity required is also smaller than those in normal min-wise hashing.

### 6.2 Experiments on Integration with AIDA

**Dataset:** we conducted experiments on three datasets:

- **CoNLL-YAGO:** The CoNLL-YAGO dataset is originally used in [1]. It is based on the CoNLL 2003 dataset, which consists of 1393 newswire articles with an average article length of 216 words. In each article, all mentions are annotated and mapped to the correct entity in YAGO2.

- **KORE50:** Hoffart [4] creates 50 difficult test sentences (14 words on average per sentence) from five domains (celebrities, music, business, sports, and politics) manually. The sentences were formulated according to a set of criteria such as: short context, high density of entity mentions, highly ambiguous mentions, etc.

- **WP:** This dataset [4] is a prepared slice of Wikipedia with similar characteristics as KORE50. It contains all articles in categories ending with “heavy metal musical
groups”. Each article is split into sentences, and only sentences containing at least 3 named entities as link-anchor texts are kept. After that, all occurrences of person names are replaced by the family name only (for example: use “Jackson” instead of “Michael Jackson”). Finally, WP consists of 2019 sentences with an average length of 52 words per sentence).

Even though there is no typo in those datasets, KORE50 and WP are close to natural language because of short contexts, short forms of person name, etc.

**Entity Relatedness Measures:** We introduce two relatedness measures which are used in AIDA [1, 4].

- Milne-Witten relatedness measure (MW): AIDA uses Milne and Witten’s measure of semantic relatedness as discussed in Section 4.2.3. This method has outperformed all other coherence-aware alternatives in the experiments of [1].

- Key-phrase overlap relatedness measure (KORE): KORE (an extension of AIDA) was first introduced in [4] as a novel notion of semantic relatedness between two entities represented as sets of weighted (multi-word) key-phrases, with consideration of partially overlapping phrases. This measure improves the quality of prior link-based models, and also eliminates the need for (usually Wikipedia-centric) explicit interlinkage between entities.

We also applied the new architectures (\(LSH_G\) and \(LSH_F\) discussed in Section 5.3) with KORE for our experiments.

**Table 6.4:** Named Entity Disambiguation Accuracy.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Evaluation</th>
<th>MW</th>
<th>KORE</th>
<th>LSH-G</th>
<th>LSH-F</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoNLL-YAGO</td>
<td>Micro Avg.</td>
<td>82.31</td>
<td>80.71</td>
<td>81.76</td>
<td>81.18</td>
</tr>
<tr>
<td></td>
<td>Macro Avg.</td>
<td>82.00</td>
<td>80.59</td>
<td>81.22</td>
<td>80.08</td>
</tr>
<tr>
<td></td>
<td>Link Avg.</td>
<td>81.34</td>
<td>80.21</td>
<td>81.80</td>
<td>80.80</td>
</tr>
<tr>
<td>WP</td>
<td>Micro Avg.</td>
<td>84.73</td>
<td>85.36</td>
<td>84.68</td>
<td>84.50</td>
</tr>
<tr>
<td></td>
<td>Macro Avg.</td>
<td>83.86</td>
<td>84.56</td>
<td>83.84</td>
<td>83.61</td>
</tr>
<tr>
<td></td>
<td>Link Avg.</td>
<td>82.45</td>
<td>80.12</td>
<td>80.64</td>
<td>80.36</td>
</tr>
<tr>
<td>KORE50</td>
<td>Micro Avg.</td>
<td>57.64</td>
<td>63.89</td>
<td>64.58</td>
<td>53.19</td>
</tr>
<tr>
<td></td>
<td>Macro Avg.</td>
<td>56.00</td>
<td>62.17</td>
<td>62.60</td>
<td>52.07</td>
</tr>
<tr>
<td></td>
<td>Link Avg.</td>
<td>63.21</td>
<td>70.75</td>
<td>71.70</td>
<td>58.58</td>
</tr>
</tbody>
</table>

Table 6.4 shows NED accuracy on the three datasets. The KORE-based NED performed about as well as the original AIDA method, which uses the MW measure based on the rich link structure of Wikipedia. MW performs better on the CoNLL-YAGO
dataset, KORE performs better on the KORE50 and WP datasets. Particularly, $LSH_G$ outperformed others in KORE50 dataset as noisy candidates are removed. The efficiency is demonstrated on Figure 6.4. $LSH_F$ is better than others, especially, far faster than the original method (MW) used in the first version of AIDA [1].
Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we presented an efficient method for Named Entity Disambiguation task via similarity hashing. This can prevent bottlenecks such as the process of mention-name matching, mention-entity mapping and entity-entity coherence estimating. We also provided a disambiguation service via a generic API which employs state-of-the-art similarity hashing techniques (i.e. Locality Sensitive Hashing) to match a given mention to similar names or to get the similarity between a mention in a specific context and an entity. This API can be easily integrated into any NED systems.

Additionally, we also integrated our work into AIDA to speed up the process of estimating the coherence among entity candidates for all mentions. Especially, to deal with the problem of small similarities among entities which is not the default application of similarity search, we introduced a proposal of the two-stage hashing scheme. The first stage of combining similar key-phrases increases the similarity between two entities, and thus, improves the quality of the second stage of grouping entities. Plus, this work contributed to “KORE: Keyphrase Overlap Relatedness for Entity Disambiguation” paper [4].

7.2 Future Work

We see some possible directions of future work as follows:

- **Full integration into AIDA:** We will fully integrate the efficient method we proposed into AIDA. For example, the mention-name mapping part which is
provided by the API (Section 5.2.4). By doing this, AIDA will be able to work with natural language text (e.g. to deal with typos).

- **Weighted LSH:** The fact that a key-phrase might be vital for some entities, but not very important for others, we have the idea of putting weights between an entity and a key-phrase into LSH. For example, we can estimate the weight between an entity \((e)\) and a permutation \((\pi)\) to be the weight between entity \(e\) and the key-phrase that corresponds to \(\min(\pi(e))\). The weight between a hashed value, which is the concatenation of \(k\) min-hashed values, and entity \(e\) can be estimated to be the average weight of each min-hashed value and entity \(e\). By doing this, each bucket contains not only entities but also the probability (the weight) that each entity belongs to it. In other words, we can judge the probability that two entities are similar even in a bucket. For example, in bucket \(b\), entity \(e_1\) has the weight of 0.1, and entity \(e_2\) has the weight of 0.00001. If the weights are ignored, the LSH table will return \(e_1\) and \(e_2\) are similar. However, we can see that the weight of \(e_2\) are extremely small, and thus this conclusion does not make sense. We finally design a formula that somehow estimates the probability of each entity candidate pair to become a similar pair by the combination of these weights (an entity candidate pair consists of two entities in the same bucket). In the context of low similarities among entities, the number of overlapping key-phrases between two entities is small. Therefore, this method might bring a great benefit because it observe the weight of each key-phrase under each entity.
List of Figures

2.1 Overview of the DBpedia Components [15]. ................. 7
2.2 General Architecture for Collective Disambiguation Systems. .............. 9

4.1 An Example of the Label Consistency Constraint [6]. ................. 20
4.3 AIDA’s Architecture [1]. ............................................. 23

5.1 Overview of Mention-Name Matching Method. ......................... 27
5.2 Overview of Mention-Entity Similarity Estimating Technique. ............. 28
5.3 Overview of Entity-Entity Coherence Estimating Technique. ............... 29
5.5 Overview of the Two-stage Hashing Technique [4]. ....................... 34

6.1 Performance of Locality Sensitive Hashing on Mention-Name Matching. 39
6.2 Performance of Locality Sensitive Hashing with Jaccard Post-filter on
Mention-Name Matching. ................................................. 39
6.3 Performance of Spectral Hashing on Mention-Name Matching. .......... 40
6.4 Running Time on CoNLL-YAGO. ................................ 43
List of Tables

6.1 Running Time for 100 Queries on Mention-name Matching. . . . . . . . . 40
6.2 Performance of Min-wise Hashing on Mention-Entity Mapping. . . . . . 40
6.3 Performance of B-bits Min-wise Hashing \( (b = 1) \) on Mention-Entity Mapping. 41
6.4 Named Entity Disambiguation Accuracy. . . . . . . . . . . . . . . . . . . . . . 42
Bibliography


Spring Symposium on Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering, pages 44–49, 2006.


[23] Parag Singla and Pedro Domingos. Entity resolution with markov logic. In Proceedings of the Sixth International Conference on Data Mining, ICDM ’06, pages


[31] Ndapandula Nakashole, Martin Theobald, and Gerhard Weikum. Scalable knowledge harvesting with high precision and high recall. In *Proceedings of the fourth ACM international conference on Web search and data mining*, WSDM ’11, pages 227–236,


