MASTER'S THESIS | LUND UNIVERSITY 2013

Nedforia: A Tool for Automatic Named Entity Disambiguation

Marcus Klang

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2013-40

Nedforia

A Tool for automatic Named Entity Disambiguation

Marcus Klang
gda07mkl@student.lu.se

November 11, 2013

LUNDS UNIVERSITET
Lunds Tekniska Högskola

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Peter Exner, peter.exner@cs.lth.se

Examiner Pierre Nugues, pierre.nugues@cs.lth.se

Abstract

In 2011, IBM showed for the first time that a computer could surpass human question answering capabilities. This feat was accomplished by designing a system that could play the American game show Jeopardy! against human contestants. In this huge undertaking, one of the problems IBM had to solve was named entity disambiguation. Named entity disambiguation is about the automatic identification of a real-world reference to a given string. IBM's work was limited to English and very little research is dedicated to named entity disambiguation for Swedish.

In this thesis, I report the design and implementation of Nedforia, a named entity disambiguation tool for Swedish. Nedforia provides a full pipeline beginning with a dump of Wikipedia and ending with a disambiguator. Nedforia's core contribution consists of modules to parse wiki markup reliably, extract relevant pieces of information from Wikipedia, and manage large amounts of information. To evaluate the final disambiguation tool, I collected a test set created from 10 news articles where I hand-annotated and disambiguated the named entities. On this set, Nedforia could reach an F_1-score of 66.49% using a Swedish Wikipedia dump from 2013-02-25. In addition to Swedish, Neforia could easily be extended to support and manage multiple languages.

Keywords: Named Entity, Disambiguation, Wikipedia, Nedforia, Swedish

Acknowledgements

I would like to thank Peter Exner and Pierre Nugues for their excellent guidance and their ability to introduce clarity to a complex subject. They provided invaluable feedback and remarks.

Finally, I would also like to thank my parents Carina Klang and Tommy Klang. Their unwavering support made this thesis become a reality.

Contents

1 Introduction **9**
1.1 Motivation . . . 9
1.2 Problem . . . 9
1.2.1 Input and output . . . 10
1.2.2 Formulation . . . 10
1.3 Scope . . . 10
1.4 Contributions . . . 11
1.5 Outline . . . 11

2 Background **13**
2.1 NLP Concepts . . . 13
2.1.1 Language . . . 13
2.1.2 Entities . . . 14
2.1.3 Semantics . . . 15
2.1.4 Cosine similarity . . . 15
2.2 Wikipedia . . . 16
2.2.1 MediaWiki . . . 16
2.2.2 Content . . . 17
2.3 Entity catalog . . . 18
2.3.1 Resource Description Framework (RDF) . . . 18
2.3.2 DBpedia . . . 18
2.3.3 YAGO . . . 18
2.4 Precision and Recall . . . 19
2.4.1 The F-measure . . . 20
2.4.2 Usage . . . 20
2.5 Previous work . . . 20
2.5.1 Context . . . 21
2.5.2 Categories and Tags . . . 23
2.5.3 Relevance . . . 23
2.5.4 Concepts . . . 24

2.5.5 Mention-Entity Graph 26
2.5.6 Connections 27

3 Implementation **29**
3.1 Framework 29
3.1.1 Extensibility 29
3.1.1.1 Creational patterns 30
3.1.2 Multilingual 32
3.1.3 Scalability 33
3.1.4 Rich querying 33
3.2 Entity detection 34
3.3 Disambiguation 35
3.3.1 Method A: Title edit distance 35
3.3.2 Method B: Popularity 35

4 Architecture **37**
4.1 Overview 37
4.1.1 Document model 38
4.2 Import 39
4.2.1 Wiki markup parser and AST tree generation 40
4.3 Indexing and storage 42
4.3.1 Requirements 42
4.3.2 Indexing 43
4.3.3 Storage 43
4.4 Resources 43
4.5 Annotation 44
4.6 Front end 44
4.6.1 Command line 44
4.6.2 Web interface 45

5 Evaluation **47**
5.1 Gold standard 47
5.2 Implementation 48

6 Results **49**
6.1 Statistics 49
6.2 Detection 50
6.3 Disambiguation 51

7 Discussion **53**
7.1 Evaluation 53
7.2 Wikipedia 54
7.3 Data mining 54
7.3.1 Page classification 54
7.3.2 Parser 55
7.4 Resources for the Swedish language 55
7.5 Detection 56

7.6 Disambiguation 56

8 Conclusions **59**
8.1 Future work 59
8.2 Summary 59

Terminology **61**

Bibliography **63**

Chapter 1
Introduction

1.1 Motivation

In 2011, IBM showed for the first time that a computer could surpass human question answering capabilities. They reached this milestone when IBM's system Watson won over all its human contestants in the Jeopardy! quiz show [1]. In this huge undertaking, one of the problems IBM had to solve was to disambiguate named entities. Given a string in a text, named entity disambiguation is the automatic identification of the real-world reference to this string: person, organization, or country. However, their work was limited to English. This thesis is about named entity disambiguation in Swedish.

1.2 Problem

The problem of named entity disambiguation is best defined through an example like the string "Göran Persson." which is ambiguous.

Limiting us to well-known people, the Swedish version of Wikipedia contains four articles with a title including Göran Persson. These articles correspond to four different persons where two of them are politicians: One has been a prime minister and one is a local politician in Simrishamn, Skåne. And the name *Göran Persson* in these two excerpts:

> Göran Persson, en lokal politiker i Simrishamn gjorde ett uttalande
>
> "Göran Persson, a local politician in Simrishamn made a statement"

and

> Statsministern Göran Persson var ordförande för Socialdemokraterna
>
> "Prime Minister Göran Persson was chairman for the Social Democrats"

refers to two different entities. The second "Göran Persson" is the prime minister, while the first one is the local politician in Simrishamn.

1.2.1 Input and output

Concretely, this all means that given a text as input results in an output of a list with named entities found and their targets.

Detection

It starts with detection which is about finding named entities (result shown in bold):

Göran Persson, en lokal politiker i **Simrishamn** gjorde ett uttalande

"**Göran Persson**, a local politician in **Simrishamn** made a statement"

Disambiguation

The next step is to find candidates and disambiguate them. This produces a list of each found named entity connected with a single target:

Named Entity	Properties	Wiki page
Göran Persson	The party leader	Göran Persson
Göran Persson	The politician in Simrishamn	Göran Persson (född 1960)
Göran Persson	The musician	Göran Persson (musiker)
Göran Persson	Swedish officer in the 1500s	Jöran Persson
Simrishamn	A location in Skåne	Simrishamn

Table 1.1: Disambiguation candidates

Named Entity	Properties	Wiki page
Göran Persson	The politician in Simrishamn	Göran Persson (född 1960)
Simrishamn	A location in Skåne	Simrishamn

Table 1.2: Disambiguation result

1.2.2 Formulation

The purpose of this thesis is to answer the following questions:

- How to disambiguate named entities for Swedish based on the Swedish version of Wikipedia as the knowledge base?
- How do different methods compare to gather entity candidates?
- What are the particular issues when dealing with Wikipedia as a knowledge source.

1.3 Scope

Wikipedia is a large knowledge source and it does require some effort in order to make it searchable. The limits for this thesis is to incrementally test simple disambiguation methods and increase complexity and iteratively improve the solution until it can be used in applications.

1.4 Contributions

The contributions of this thesis is a new named entity disambiguation tool, Nedforia, that uses Swedish sources. As knowledge sources, I used Wikipedia, although Nedforia could accept other kind of texts as input.

Nedforia was designed for the purpose of named entity disambiguation but it consists of modules that can easily be adapted to serve other purposes, such as Wikipedia document modeling, Wiki markup parsing, prototyping environment for Natural Language Processing (NLP), and more.

Nedforia provides a full pipeline to parse Wikipedia and creates the necessary data structures needed to do disambiguation.

Another contribution is how well simple methods fare when doing disambiguation and initial results for named entity disambiguation in Swedish based on Wikipedia.

1.5 Outline

The thesis begins with Chapter 2 that introduces the reader to the topic of this thesis. It also provides a description of previous work. The following chapter 3 outlines the concepts and algorithms I implemented. Chapter 4 describes the actual implementation with more detail. Chapter 5 describes the evaluation methodology I used and Chapter 6 provides the results I obtained. The thesis ends with a discussion of the trade-offs I had to make, future improvements, and some issues to finally arrive at the conclusion of the thesis in Chapter 8.

Chapter 2
Background

This chapter describes of the concepts that is used in named entity disambiguation and some of the existing methods that are or have been used to do automatic named entity disambiguation.

2.1 NLP Concepts

NLP is a part of computer science that deals with how natural language as spoken by human beings can be processed by a program in order to e.g. extract knowledge, information, or other data for scientific purposes.

2.1.1 Language

The raw material consists of documents or texts, which at first appears as just a collection of letters, spaces, and symbols of varying shapes and sizes. One level down, texts are divided into paragraphs which are separated by spaces in the layout. The paragraphs are then separated into sequences of letters grouped together. The start and ending of these sequences is marked by punctuation e.g. periods, exclamation and question marks. Moving one level further down, these groups form the meaningful structure that we convert to sound when read aloud. The purpose is to allow communication between people.

This is a figurative description that applies to a language such as Swedish and English but not all forms of written language. I will now be more specific and extend the description to a usable presentation for language processing.

Strings are sequences of characters, this is the basic construct to handle the symbols in a text.

Tokens are the constituents of a sentence, a token could be a word, or punctuation. It is the smallest meaningful part of a written language.

Sentences is a set of tokens in a specific order that is dependent on the language. There is always some form of structure to the sequence of tokens which forms the meaning that is understandable by humans.

Part of Speech (POS) is a grammatical concept, where you attach words to lexical categories e.g. nouns or verbs.

Segmentation is the process of dividing text into the fundamental lexical constructs such as sentences and words. Segmentation in NLP looks primarily at punctuation marks and spaces but false positives that abbreviations can generate must be handled with care to get good results.

Morphology is about identifying and handling how words varies in form. This could happen when a word is in e.g singular or plural form. Some examples: car, cars, run, running, drive, driving, index, indices, apple, apples, worked, working.

Lemma form is the form of a word that is not affected by morphology, which is the base form, the neutral form. In Swedish e.g "är" is the surface form and "vara" is the lemma form.

Surface form is the form of a word found in a text and is related to morphology. In this thesis surface forms are normalized when used in a dictionary by using the lemma form of them.

Corpus is a collection of texts written by humans. Wikipedia in the context of this thesis is the corpus used.

Document is synonymous to a specific text in this thesis but it can be uniquely identified by some string and has properties such as a title.

Gold standard is something that provides the correct answers. An example of a gold standard is a manually annotated corpus with part of speech tags. Without a gold standard it is not possible to know if a particular result is correct or incorrect which makes evaluation difficult.

Stop words is words that occur frequently and they are often removed prior to conducting a search. This is done to avoid finding a large amount of unrelated matches. Any word could in theory be selected as a stop word but the actual choice is arguably best chosen based on real statistics such as word frequency. Stop words could be e.g and, or, I, you, he, she, and many more.

2.1.2 Entities

An entity is a thing in this thesis. It could be a person, organization, car, table, concept, and much more. It is often ambiguous as when writing just cat, dog or table. There are entities that have the requirement of being uniquely identifiable and they are called *named entities*, and some examples are Schrödinger's cat, Pierre Nugues, Peter Exner, Marcus Klang and Lunds Tekniska Högskola. Consequently, a proper noun or proper name is a named entity. They could be ambiguous in an input text just as a normal entity. However, with the significant difference that a named entity must be resolvable to a set of candidates that are unique.

In addition, a word that references or implies a specific entity is not a named entity. An example of this would be: given a text, a dog is named “Gus” and as the text progresses an instance of “the dog” is found. This instance would be resolved, by using context, to “Gus” which is unique but it is not uniquely identifiable by itself which is why it is not a named entity.

Named entities could also have an attached type such as person, organization and country, but this is not a requirement. Finally, all named entities are entities, but entities are not named entities.

2.1.3 Semantics

Semantics is about what something means. A computer program only sees strings of characters arranged in specific ways. By applying a function that does segmentation we get sentences and words but they are still only strings to a computer program. The problem of modeling dependency grammar attempts to solve a part of this problem. It is solved by using a so called dependency parser that forms connections between semantically connected words, usually modeled after noun and verb phrases. Given two examples:

> “Göran Persson is a politician in Stockholm”

and

> “Göran Persson saw a man running to a meeting”

the first one contains three noun phrases: “Göran Persson”, “a politician” and “Stockholm” and the last one contains a verb phrase “saw a man running to a meeting”. Other ways of emulating a sense of understanding is by creating a predicate-argument structure that can be used for search. A predicate-argument structure in a entity catalog[1] such as YAGO2 and DBpedia consists of triplets: subject, predicate, argument. Given an example, “Albert Einstein was born on 14 march 1878”, the subject would be “Albert Einstein”, the predicate “born”, argument “14 march 1878”. This structure is well suited to create efficient queries that could be used to find relevant knowledge.

2.1.4 Cosine similarity

To compute similarity between two texts there is a concept based on the idea of that related texts “point” in the same direction. This idea is converted into the concept of cosine similarity. In order to compute cosine similarity you need to convert the information into a vector space. To give an analogy to a geometric system cosine similarity is about computing the angle between two vectors, this can be achieved by using the dot product:

$$\cos\alpha = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \cdot \|\mathbf{b}\|} = \frac{a_1 \cdot b_1 + a_2 \cdot b_2 + \ldots + a_n \cdot b_n}{\sqrt{\sum\limits_{i=1}^{n} a_i^2} \cdot \sqrt{\sum\limits_{i=1}^{n} b_i^2}} \tag{2.1}$$

This gives $\cos\alpha$ between vector $\mathbf{a}$ and $\mathbf{b}$. $\cos\alpha$ has the range $[-1, 1]$. In the context of NLP, -1 means completely opposite, 0 means not related and 1 means related. The content

[1] Described in depth in section 2.3 on page 18

of a vector are weights which define how common a certain term is. The index must match between the two vectors so if the term does not exist in text A but in text B a corresponding 0 weight must be placed in A and B. That is the vector of A and B equals:

$$\mathbf{v} = \{w(x) | x \in A \cup B\} \tag{2.2}$$

$$w(x) = \begin{cases} f(x) & \text{if the input text has term x} \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

where $f(x)$ is a function that provides a weight given a token. An example is provided below of the two texts A and B, containing 2 terms each in order to keep the example small.

$$\begin{aligned}
A &= \{\text{Karlstad, Sverige}\} \\
B &= \{\text{USA, Karlstad}\} \\
A \cup B &= \{\text{USA, Karlstad,Sverige}\} \\
\mathbf{a} &= \{w(\text{USA}), w(\text{Karlstad}), w(\text{Sverige})\} = \{0.3, 0.04, 0\} \\
\mathbf{b} &= \{w(\text{USA}), w(\text{Karlstad}), w(\text{Sverige})\} = \{0.0, 0.04, 0.4\} \\
||\mathbf{a}|| \cdot ||\mathbf{b}|| &= \sqrt{0.3^2 + 0.04^2 + 0^2} \cdot \sqrt{0^2 + 0.04^2 + 0.4^2} \approx 0.12167 \\
\mathbf{a} \cdot \mathbf{b} &= 0.3 \cdot 0 + 0.04 \cdot 0.04 + 0 \cdot 0.4 = 0.0016 \\
\cos\alpha &= \frac{\mathbf{a} \cdot \mathbf{b}}{||\mathbf{a}|| \cdot ||\mathbf{b}||} \approx \frac{0.0016}{0.12167} \approx 0.0132
\end{aligned}$$

As can be seen above I gave the term Sverige and USA a rather high weight of 0.4 and 0.3 which contributed to a very low similarity between text A and B. These weights were arbitrary but you could use a probabilistic approach that models how common a term is. An example could be counting how common a term is in the corpus (term-frequency) and counting how many documents that contains the term (document-frequency).

2.2 Wikipedia

Wikipedia is an online encyclopedia that is managed and written by users from all around the world. It exists in many languages, where Swedish is one of them. Today Wikipedia is very diverse in topics and the Swedish edition is growing fast when reading [2]. The figures presented in [2] were the only ones I could find and because they were compiled by Wikipedia editors it should be noted that the figures might not be trustworthy.

2.2.1 MediaWiki

Wikipedia uses MediaWiki to drive its web based front end. MediaWiki [3] is an application written in the server side programming language PHP [4] that is suited for use in web development. MediaWiki is not only used for Wikipedia but many other wiki-like sites as well. Anyone can download MediaWiki and set up their own Wikipedia like page. MediaWiki provides the data model to store wiki pages and is an active project where the latest release was at the time of writing version 1.21.2, released 2013-09-03.

MediaWiki and in turn Wikipedia deals with pages; these pages or wiki pages are written in a special markup language called wiki markup[2]. Wiki markup is central to extracting information from Wikipedia. Wiki markup is a text based markup language that is more text-like than a markup language such as HTML. The language has constructs such as headers, links, tables, bullet lists, indentations, templates and much more.

2.2.2 Content

Like any larger library of information, structure is needed when data grows. This is handled through specific types of pages that exists within Wikipedia. I will now present the basic Wikipedia page types and some other important details:

Pages is the fundamental construct. A page has a title and it has some wiki markup, everything that will be described below is a specialization of a page and inherits its properties.

Articles are normal full length texts about a subject or concept e.g. Göran Persson, global warming or natural language processing.

Stubs are articles but are deemed to not be mature enough to be called an article. They are usually short and has not yet reached a satisfactory level of quality and scope determined by other Wikipedia editors.

Disambiguation pages are pages that tries to disambiguate ambiguous topics by describing the key differences between them. They also provide hints that could guide a reader to the correct page. These pages are of great interest to named entity disambiguation because they provides gold candidates for disambiguation and highlights ambiguous topics.

Templates are MediaWiki constructs that are used to reduce duplication of wiki markup code. One of the most commonly used template is arguably the infoboxes that provides property information e.g. birth dates, spouses, children, who is CEO for an organizations, and more. There are many different templates and they can include other templates in its definition.

Redirects are pages which redirect to other wiki pages. Because their title is different than the target they are good for extracting different surface forms for what is meant by an article or stub. Commonly acronyms are redirects and common misspellings are also redirects.

Categories are basic pages but when viewed in Wikipedia they have auto-generated content which shows all the articles that belongs to the selected category. Categories are allowed to be part of another category which creates a semantic hierarchy of related subjects.

The term entity in the context of Wikipedia is concretely an article or a stub.

[2] Read [5] for some examples of the language

2.3 Entity catalog

An entity catalog is a database of things automatically extracted from a knowledge source. I will describe two entity catalogs: YAGO and DBpedia. These two catalogs were automatically created from Wikipedia [6], and in the case of YAGO [7] also WordNet . They contain for instance facts extracted from infoboxes in Wikipedia. They define an ontology which deals with not just entities but also how they relate to each other e.g. a scientist is a person.

2.3.1 Resource Description Framework (RDF)

RDF is a W3C standard. It is defined by W3C as "RDF is a standard model for data interchange on the Web" [8]. It is in the context of this thesis used to store relations between different entities, and it defines a triplet structure. This triplet enables the attachment of metadata to a Uniform Resource Identifier (URI) (the subject) where the type is defined by a predicate and its value by an object. It is very generic and can be used to connect many different forms of data. The predicate-argument structure in section 2.1.3 can be stored using a RDF format. One RDF format uses a XML standard defined by W3C. However, this format is not used in practice when surveying what available formats exists for YAGO and DBpedia. This is most likely due to performance and size considerations.

2.3.2 DBpedia

The goal of the DBpedia project is to provide structured information extracted from Wikipedia and make it available on the Web. DBpedia uses infoboxes, page links, page text, page categories and more when extracting information.

DBpedia also interlinks to existing datasets such as YAGO and many others which results in a greatly expanded web of knowledge[6]. Some relevant datasets to this thesis are persons and the links between languages which includes Swedish. The latest version of DBpedia (which at the end of the thesis was version 3.9) classified 3.22 million things into what the authors claim to be a consistent ontology which includes 832 000 persons [9]. DBpedia's datasets are also freely available for download on their website in a multiple RDF formats [10]. One issue with DBpedia is that it does not yet have hand generated mappings for the Swedish version of Wikipedia [11]. This means that primarily language independent constructs are extracted and linked e.g titles, raw infobox information, links between pages, and links between different Wikipedia language editions. This means that Swedish is not of the same level of quality as English, German or any other languages that have manual bindings.

2.3.3 YAGO

YAGO stands for "Yet Another Great Ontology" [7]. YAGO was created from primarily WordNet and Wikipedia by utilizing heuristic algorithms, i.e. approximated algorithms that was optimized for the target domain where a universal solution is too hard or complex to create. YAGO contains facts such as the date of birth of a person, who that person married to and much more. YAGO has well defined structures and has been manually evaluated partially to verify that it is of high quality. The authors found it to have an accuracy of 95% [7]. This figure was derived from manual evaluation of randomly selected facts. However, the evalua-

tion does not verify that the fact is actually correct but rather that it matches the fact presented in Wikipedia. i.e. the evaluation does not deal with the problem of false information. YAGO was later extended in a second version "YAGO2" which extends YAGO with temporal and spatial information [12]. YAGO2 have dedicated resources to enable linking of Wikipedia article titles to YAGO entities in multiple languages[13] which makes it useful for this thesis.

2.4 Precision and Recall

In order to measure how good a specific information retrieval solution is the concepts of precision and recall is often used. Recall answers how much of the result which is relevant that has been found by the system. Precision answers how much of what is retrieved actually is correct or relevant.

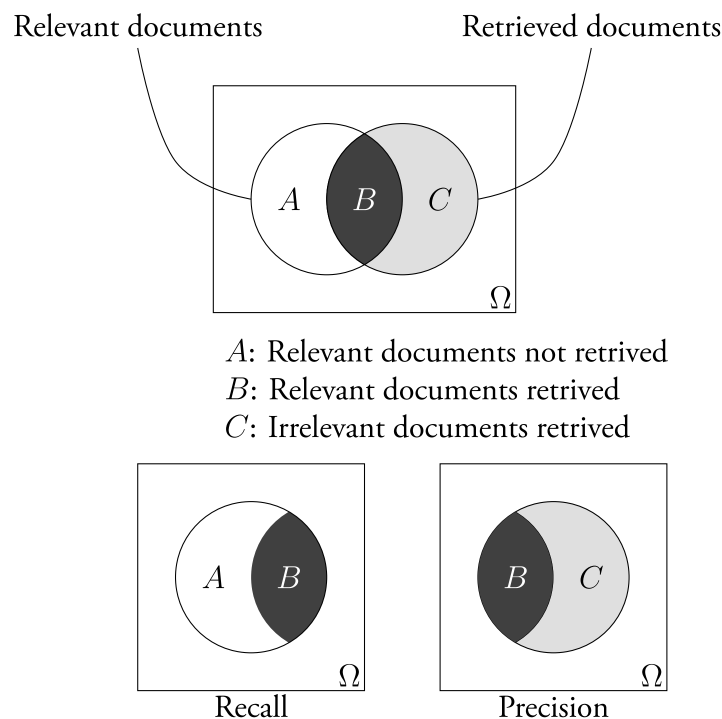

Figure 2.1: Precision and recall

Figure 2.1 gives a picture of how to interpret the two measures. Ω is all documents in the system. From this presentation they are concretely computed the following way:

$$\text{recall} = \frac{|B|}{|A| + |B|} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|} \tag{2.4}$$

$$\text{precision} = \frac{|B|}{|B| + |C|} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|} \tag{2.5}$$

These two metrics measures two different parts of an information retrieval system and to get a single overall score they are combined into the so called F-measure.

2.4.1 The F-measure

The F-measure has the generic definition of :

$$F_\beta = \left(\beta^2 + 1\right) \cdot \frac{\text{precision} \cdot \text{recall}}{\beta^2 \cdot \text{precision} + \text{recall}} \tag{2.6}$$

[14]. The most commonly used variant in practice of the F-measure is when $\beta = 1$, because this turns the F-measure into a harmonic mean between recall and precision which has the form:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{2.7}$$

The choice of a harmonic mean is arguably due to fact that when precision and recall are unbalanced the lower value tends to have a higher weight, resulting in a lower score. Take this example:

$$\begin{aligned} \text{precision} &= 0.9 \\ \text{recall} &= 0.1 \\ \text{harmonic mean} = F_1 &= 0.18 \\ \text{arithmetic mean} = \frac{\text{precision+recall}}{2} &= 0.5 \end{aligned}$$

here it is clear that the harmonic mean better represents the true score.

2.4.2 Usage

In figure 2.2 on the facing page an example consisting of three scenarios are presented. The goal is to catch all dead fish and the circle symbolizes what has been caught. In Scenario A, the typical case is presented where a mix of alive and dead fish has been caught. Scenario B is an edge case where precision is perfect but recall is poor and as can be seen by the F_1 score, a poor balance results in the lower value taking precedence. It is often easy to get perfect recall or precision but hard to get good performance of both.

2.5 Previous work

In this section, I will try to capture the ideas and techniques used to carry out Named Entity Disambiguation by others. The articles referenced in section uses primarily Wikipedia as its knowledge source. Every section that follows matches one article and provided is my interpretation. The sections and articles are:

Context is based on the article “Using Encyclopedic Knowledge for Named Entity Disambiguation” written by Bunescu and Paşca [15]

Categories and Tags is based on the article “Large-Scale Named Entity Disambiguation Based on Wikipedia Data” written by Cucerzan and Silviu [16]

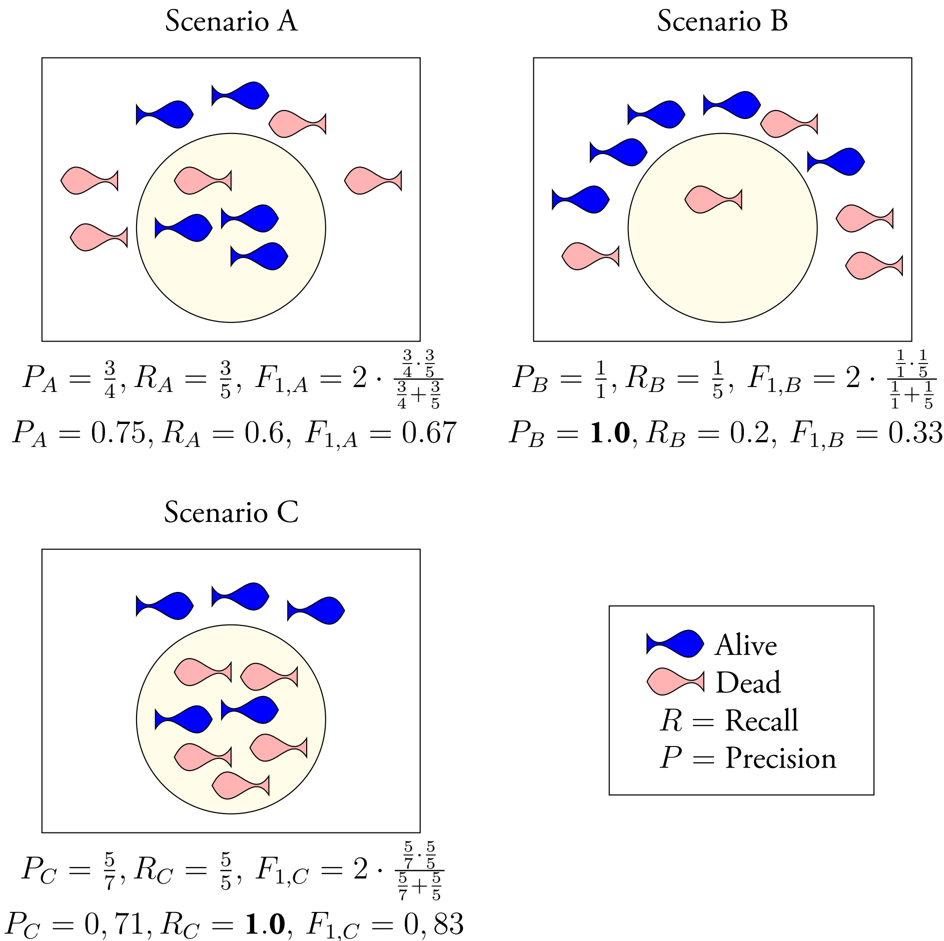

Figure 2.2: Precision and Recall example

Relevance is based on the article “Learning to link with wikipedia” written by Milne and Witten [17]

Concepts is based on the article “Named entity disambiguation by leveraging wikipedia semantic knowledge” written by Han and Zhao [18]

Mention-Entity Graph is based on the article “Robust disambiguation of named entities in text” written by Hoffart et. al [19]

2.5.1 Context

The earliest work of Named Entity Disambiguation (NED) conducted on Wikipedia that I could find was Bunescu and Paşca [15]. They extracted named entities from Wikipedia using a heuristic method that was based on the page titles. In order for a wiki page to be selected as a named entity any of the following three conditions had to be satisfied:

1. If the title of an wiki page contains multiple tokens, and every token are words that do not belong to any of the following types: determiners, conjunctions, prepositions, relative pronouns, or negations. Example: “House of Sweden”.

2. If the title of an entity is a single token, and the two first characters are uppercase letters. Example: “USA”

3. If all occurrences of the title in the main text fulfills the same restrictions as in condition 1 in at least 75% of the cases. The title occurrences must be in locations other than the start of sentences. Example: “Maryland”

From the conditions above a dictionary of named entities was created. The actual entries in the dictionary included all variants of what an article could be called which included titles of redirects, anchor texts and more. This dictionary was then used to get a list of candidates which was the input to the disambiguation method.

The first disambiguation solution used a scoring function based solely on cosine similarity to compute the rank of a potential candidate. The input to this scoring function was a window of 55 tokens from the input text centered around a candidate and compared against the full text of its potential entity target. The token weights used for cosine similarity was:

$$d_w = f(w) \ln \frac{N}{df(w)} \tag{2.8}$$

In this equation, w is a word in the document d, $f(w)$ is word frequency i.e. the number of times the word has been used, $df(w)$ is the word document frequency i.e. the number of documents that contain the word, N is the number of total number of documents i.e. Wikipedia articles. When computing e.g term frequencies stop-words, common words that are too frequent, or too rare were excluded. The authors made an evaluation of this method and they found that it failed to rank the proper candidate even when all the necessary context words were there.

To improve upon this solution the authors leveraged machine learning by using an technique called Support Vector Machine (SVM). The goal of the SVM was to find a vector of weights. This weight vector was used in the ranking which concretely meant computing a dot product between the weights and a candidate feature vector, expressed mathematically as:

$$\hat{e} = \arg\max_{e_k} \mathbf{w}\, \Phi(q, e_k) \tag{2.9}$$

Here $\hat{e}$ is the selected candidate, $\mathbf{w}$ is the weight vector and $\Phi(q, e_k)$ is the feature vector of a query q which in this context meant a token window[3] around a link in an article and the potential entity candidate e_k. The features were cosine similarity and a binary vector of features that was derived from categories and the vocabulary of Wikipedia. My interpretation is that they wanted features that could potentially rank a candidate higher when there was a strong correlation to the content of categories. The training data was derived from the articles themselves where the links that were unambiguous were used as a gold standard.

The size of input data and limited computational power made it hard for the authors to fully evaluate their method so they had to settle for a partial evaluation where they used 110, 540 and 2847 categories and reported accuracies between 55.4% (cosine similarity) and 84.8% (machine learning method) depending on the development/test data employed. I could not find the exact method used to compute the accuracy figure which means I do not know if the F-measure was used. [15]

[3] a number of tokens around a link, e.g “In Skåne [Göran Persson] is a politician” a token window of 4 around the link “Göran Persson” could mean the tokens: {Skåne, Göran, Persson, is}.

2.5.2 Categories and Tags

The work of Cucerzan and Silviu [16] is similar to Bunescu and Paşca [15]. Both articles define an extraction method to get named entities from Wikipedia as well as a disambiguation method that used redirects, article titles and disambiguation pages to get different surface forms and candidates for an entity. Cucerzan and Silviu also included real statistics from queries performed on a web search engine something Bunescu and Paşca did not do.

The disambiguation method used primarily three parts of information: entity surface forms, category tags and contexts.

Entity surface forms are all the different variations of what an entity could be called in Wikipedia (extracted from redirects, links, disambiguation pages).

Category tags was derived from list pages (list of ... or table of ... pages) and category labels. These provided a way of relating entities semantically.

Context was based on the page content of an entity and the page content of its in-links, i.e. those entities referring to the entity at hand. They initially considered to use all information of every entity but they limited themselves to only the ones mentioned in the first paragraph of the entity page due to data size considerations. This resulted in 38 million entity, context pairs for the English edition of Wikipedia.

The disambiguation method consists of two parts: the first one is to compute similarity between the input text and candidate contexts, the second is to compute category similarity between all candidates. By combining these two a rank is produced. The similarity is concretely computed with dot product of vectors where each element is a word or token however the elements are not normalized.

This system scored an accuracy of 91.4% when measured against top two stories in ten different news sites of varying type. The test set contained 756 surface forms of which 127 was not possible to recall. [16]

2.5.3 Relevance

Milne and Witten [17] differs from earlier work in that they added statistics of what surface forms could refer to (defined as commonness by the authors) and used a measure of relatedness between two entities. The equation that defines the relatedness measure is defined below:

$$relatedness(a, b) = \frac{\log(\max(|A|, |B|)) - \log(|A \cap B|)}{\log(|W|) - \log(\min(|A|,|B|))}, \tag{2.10}$$

where A and B are in-links to the entity a resp. b, and W is set of all links. This relatedness function measures essentially how compatible two articles are by comparing the articles that link to them. This means that if the same articles link to them then they are very related.

For the disambiguation component Milne and Witten used machine learning to train a classifier that can choose when to use relatedness and when to use commonness instead. The classifier was trained on three features: relatedness, commonness and a context quality which is a sum of weights derived from the context so that there is a measure of how good the context is. This disambiguator got an F_1-score of 96.9% (using Bagged C4.5 algorithm as described by the author) when performed on 11 000 links in 100 randomly selected articles. Milne and

Witten also described a link detector trained on Wikipedia. I determined this link detector to not to be of relevance for this thesis with the argument that it is not a named entity detector which would be of interest. [17]

2.5.4 Concepts

The previous works are all heavily based on the Bag of Words (BOW) model which means that they only used a bag of words instead of building a semantically related network of entities, i.e. how entities relate to each other and make use of that to disambiguate named entities. Han and Zhao [18] extracts surface forms, builds a dictionary from the text, just as earlier work and also adds commonness in the same way as Milne and Witten [17] described in the previous section. This creates a surface form dictionary that links a surface form to an candidate article. Important to note is that Han and Zhao excludes the Wikipedia articles that matches any of the following conditions:

- The article belongs to categories related to chronology, i.e. "Years", "Decades" and "Centuries".
- The first letter of the article title is not a capital one.
- The article title is a single stop word.

Han and Zhao has has a special meaning for the term concept. According to my interpretation a Wikipedia article becomes a concept when some page has linked to it and fulfills the conditions above, which would imply that a concept could be used interchangeably with an article when this requirement is fulfilled.

There are two parts of Han and Zhao's work, the first part defines how to find concepts from Wikipedia and an input text. The second part compares concepts found in the input text against concepts derived from the real Wikipedia articles. [18]

Part 1: The concept vectors This part begins with the goal of mapping surface forms to a selection of concepts which is then used in the disambiguation step to compare input text context against an candidate and rank accordingly. The entries of the surface form dictionary contains one or more possible concept candidates. Han and Zhao maps the surface form to a particular concept by selecting the candidate that has the highest value provided by this scoring function:

$$Score(s,c) = \frac{\sum\limits_{t \in \mathrm{T}} sr(t,c)}{|T|} \cdot Commoness_{s,c} \tag{2.11}$$

where s is the surface form in the dictionary, c is the concept, sr is the same as equation 2.10 on the previous page where the input data matches the original definition, and T is the context concepts. The context concepts are the targets of unambiguous links found in all concept candidates. $Commonness_{s,c}$ was computed by normalizing the commonness of a surface form, i.e. the probability a particular surface form refers to a special concept, which concretely means a count divided by the sum of all counts. The final product is a dictionary where each entry is a surface form linked to a concept. In the end a named entity candidate from a text is represented as a vector of concepts and a weight:

$$o = \{(c_1, w(c_1, o)), (c_2, w(c_2, o)), \ldots, (c_m, w(c_m, o))\} \tag{2.12}$$

where c_i represents a concept and the weight function w is equal to:

$$w(c, o) = |o|^{-1} \left(\sum_{c_i \in o, c_i \neq c} sr(c, c_i) \right) \tag{2.13}$$

This weight function returns a scalar value on how relevant a particular concept is compared to the rest. To reduce noise and improve performance, concepts with low weights were removed. It should also be noted that Han and Zhao did not use all surface forms. Han and Zhao computed a probability that a surface form could be a concept and removed those with low probabilities.[18]

Part 2: Disambiguation Han and Zhao introduces a new method to compute similarity between input and disambiguation candidates. The method has three steps and the first step is about computing alignment, I interpret it as a way to compare a concept to its most likely counterpart in another concept vector. Alignment is computed by the equation:

$$Align(c, o_k) = \operatorname*{argmax}_{c_i \in o_k} sr(c, c_i) \tag{2.14}$$

where o_k is an named entity observation. The align method maps a concept to its most likely counterpart in another concept vector. This alignment function is used when computing how semantically related two concept vectors are. The function to compute semantic relatedness is defined as:

$$SR(o_k \to o_l) = \frac{\sum_{c \in o_k} w(c, o_k) \times w(Align(c, o_l), o_l) \times sr(c, Align(c, o_l))}{\sum_{c \in o_k} w(c, o_k) \times w(Align(c, o_l), o_l)} \tag{2.15}$$

where o_l is another concept vector derived from a named entity candidate text. This equation describes a weighted average of semantic relatedness between all aligned concepts in o_l compared against concepts in o_k. The value is bounded within $[0, 1]$.The final step is the similarity function that makes use of everything before:

$$SIM(o_k, o_l) = \frac{1}{2} \times (SR(o_k \to o_l) + SR(o_l \to o_k)) \tag{2.16}$$

which is basically a mean of semantic relatedness when going in both directions: $o_k \to o_l$ and $o_l \to o_k$.

All this goes into the final step that is the actual disambiguation that uses input from everything prior to this point. Han and Zhao used a method called Hierarchical Agglomerative Clustering (HAC) to group named entities together with their most likely entities based on the similarity in equation 2.16. They tested the solution on a standardized dataset called WePS1 where they got their best F_1 score of 88%. [18]

2.5.5 Mention-Entity Graph

Hoffart et. al [19] was the first in this thesis to use a linear combination of different measures, the previous did not combine measures when computing the final result. This article uses the term *mention* to denote a named entity candidate found in a text. The measures selected were:

Popularity prior corresponds to the number of in-links to an Wikipedia entity.

Context similarity is about comparing the context of the input by computing a similarity between all tokens in the input against a key phrase defined for entities that were extracted from YAGO. A key phrase is a phrase that is derived from link texts, category names, citation titles and other references. The idea is that a key phrase shall contain keywords that is characteristic for an entity. When comparing contexts they used a concept called phrase cover when computing a similarity score. A phrase cover is the shortest window of words that contains a maximal number of words found in another phrase, which in this case was a key phrase. They also used a method to compute a syntax based similarity based on what happens when a subject in a text is switched out in a sentence for another. An example provided by the authors was "Page played unusual chords" and by switching "Page" into "Guitarist" it would yield a high score because it is referring to Jimmy Page, the musician. But when switching "Page" in the same example into "Entrepreneur" as in Larry Page (the Google founder), the similarity yields a lower score. The context similarity is computed between a mention in an input text and key phrases.

Coherence provides a way of comparing different entity candidates in a text in order to measure how compatible they are. The method used is the same as Milne and Witten [17], equation 2.16 on the preceding page.

These methods were combined into a so called "Overall Objective Function". This function is a linear combination of the measures output and a weight. These weights must sum up to 1.

The authors created a so called mention - entity graph where the outer nodes are mentions found in a text and then mapped to one or more candidate entities. The entities were also connected to other entities if they had relevance, e.g. when linking to each other. All edges in this graph were weighted using primarily the previous three measures. Then end goal was to simplify the graph so that there is only one link from an mention to an entity. This is where the innovation comes in, they used an algorithm that does this approximately because doing it exact was NP-hard[4] and not feasible for larger inputs. They also used something they called a robustness check, this check tries to mitigate the weak spots such as when the input text is short. The check verifies that popularity does not dominate the outcome in the popularity case and in the coherence case whether it makes sense which might not be the case when the text is heterogeneous.

When evaluating they used a method called MAP (mean average precision) to do its evaluation which is related but not the same as the F-measure. The best result for their solution was 89.05% which was reached when using popularity with robustness check, context similarity using key phrases and coherence without robustness check. [19]

[4] Problems for which efficient algorithms has not been found. Efficient algorithms typically have polynomial or better complexity.

2.5.6 Connections

I used the previous works to derive ideas. Basically adapting earlier work to work for Swedish was not the goal but instead getting a sense of what ideas exists and use some of them. In this thesis I used the idea of extracting anchor texts, redirection titles to find candidates. This idea was present in one form or another in all works but I first found it in Bunescu and Paşca [15]. Popularity was an idea that I adopted from the work of Cucerzan and Silviu [16]. The rest of the previous works provided insight into more complex methods that could be used to compute similarities of different types.

Chapter 3
Implementation

This chapter provides the high level goals of the tool Nedforia, the ideas that it is based on and the considerations that went into the design of Nedforia.

3.1 Framework

These were the goals I had when designing Nedforia:

- Extensible, easy to extend with new code and integrate with other components.
- Multilingual at the core, meaning that anything language dependent should be separated and managed. The goal is that new languages could easily be added by just implementing the language specific parts.
- Scalability on a single machine, be able to mange large amounts of data as fast and smooth as possible without over complicating the solution.
- Rich querying, be able to easily extend query operations on the data and see the results. This could be e.g. run computations on data directly without any preprocessing, query and find relevant data.

To accomplish these goals I iterated the development and improved the solution at each iteration as needed. In total three distinct version of Nedforia was made. The first one was completely scrapped due to being overly concrete and lacking scalability. The second version was good enough for some tasks but extension was hard. The second version was therefore heavily refactored into the third and final version.

3.1.1 Extensibility

The most time consuming part to get right was extensibility. There was a problem of balancing abstraction and concreteness, and be mindful about not over complicating things. Code

reuse was essential, not only because it makes the code unmaintainable otherwise but because differences in the implementation could make evaluation inconsistent. Working at the conceptual level helped to define abstractions. I also needed to move fast forward and be able to switch out parts quickly without breaking old code. To be able to switch out parts easily also meant that extensions could make use of the abstractions that had to be created and thereby reuse existing code.

3.1.1.1 Creational patterns

A standing problem with abstractions is that eventually you have to get an concrete instance from somewhere. This could be solved by using the factory pattern which is exemplified using UML in figure 3.1. In this figure the `SwedishLanguage` class depends on an abstraction `Language` and the concrete instance is provided by `LanguageFactory` given a string of the language name. This instance is then used by another implementation `Disambiguator` which makes use of the language abstraction to do its work.

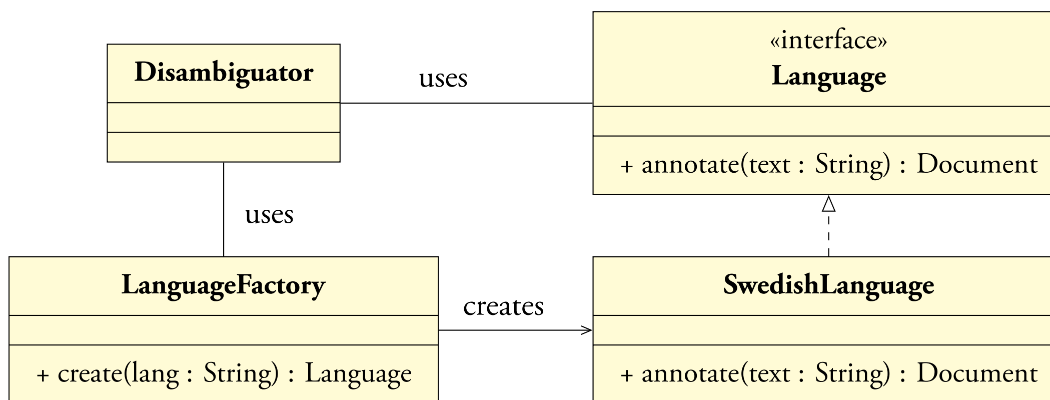

Figure 3.1: A sample of the factory pattern

The issue with this pattern is that the factory itself also has to be managed and created somewhere, which requires passing arguments to a constructor which could be e.g. a configuration class. This makes code that uses factories cluttered. This could have been solved by using a singleton but still the code rigidly depends on that particular factory and the singleton has to be initiated somehow before. The code that `LanguageFactory` has also directly depends on that concrete class and switching its implementation would be hard. There is however one very desirable property of the factory pattern and that is the simplicity of the code produced because it is very concrete and generally easy to follow.

Dependency injection

Considering the advantages and disadvantages of the factory pattern I ended up switching a part of simplicity with flexibility. The solution is still easy to use. Nevertheless, it is much harder to follow what happens in the code which could be attributed to the thing that magically provides instances, the actual injector implementation.

Dependency Injection is the generic name for this pattern. It solves the problem of hard coded dependencies such as factory classes in the code by making the code only depend on a dependency injector.

The dependency injector is constructed once and reused multiple times. The injector could be configured to provide different concrete implementations which allows a high level way of switching the implementation of an abstraction. The role of the injector is to provide instances of requested abstractions as well as concrete classes. The injector also injects the dependencies needed to construct these instances. There exists production ready libraries such as Guice [20] that implements dependency injection but I built my own solution because I needed thread safe construction of a concept I call *data resources* which are things such as indices and storage implementation. These resources did not quite fit into how Guice implemented this pattern.

The easiest way to describe how it all comes together is to present sample code:

Dependency injector configuration

```
public abstract class DependencyInjector {
    /**
     * Bind a type so that it can be resolved
     * @param from a abstract type like interface or abstract class
     * @param to a concrete type like a class
     */
    protected final void bind(Class<T> from, Class<T> to);

    /**
     * Inject an instance of type T
     * @param type the type to inject
     * @return an instance of the type T
     */
    public final T inject(Class<T> type);

    /**
     * Inject an instance of type T for language lang
     * @param type the type to inject
     * @param lang the language to use
     * @return an instance of the type T
     */
    public final T inject(String lang, Class<T> type);

    /**
     * Configure the injector so that types can be resolved
     */
    protected abstract void configure();
}
```

Injector example

```
/**
 * An sample injector
 */
public final class NedforiaInjector extends DependencyInjector
{
	private boolean useBerkleyDb;

	public NedforiaInjector(boolean useBerkleyDb) {
		this.useBerkleyDb = useBerkleyDb;
		configure();
	}

	protected void configure() {
		if(useBerkleyDb)
			bind(WikiStorage.class, BdbWikiStorage.class);
		else
			bind(WikiStorage.class, StdWikiStorage.class);
	}
}
```

Sample code for usage

```
//Example code using the injector
NedforiaInjector nedforia = new NedforiaInjector(false);
WikiPageStorage storage = nedforia.inject("sv", WikiPageStorage.class);
```

"Dependency injector configuration" listing defines the base class for an injector. In the "Injection example" listing an an injector extends the dependency injector base class and configures it. Finally, "Sample code for usage" displays how it all is used in practice.

In the Dependency Injector: `bind` is the method used in the `configure` method to map one type to another. All this configuration is typically done only once at the start of the application. The major difference in my implementation compared to how e.g. Guice implemented this pattern is that there are two variants of the `inject` method, one that is language independent and one that is dependent on the actual language to use. I used the language dependent version in the usage code because I wanted to access the Swedish Wikipedia data storage. The major drawback of my implementation is that an exception could be thrown at runtime if the configuration becomes inconsistent with the data stored in the workspace.

3.1.2 Multilingual

It was clear from the start that one thing that many existing solution lacked was to have multilingual support at the core. It seems like most existing work originated from English and then either tried to apply it to other languages or are currently in the process of converting their solutions to fit other languages as well. The solution was simple enough: instead of

hard-coding language dependence I separated everything language dependent into separate packages and made a top level contract that operates on a standardized data model.

The standardized model in question was based on the model [21] defined by ConLL (the Conference on Natural Language Learning) and was built by Peter Exner which I extended with some helper methods.

3.1.3 Scalability

Swedish Wikipedia is not as big as the English edition of Wikipedia but the dump I selected (2013-02-25) [22] still contains 1.6 million pages and uncompressed it consumes 13.02 GiB of storage space in its raw XML form.

The longer term goal, beyond this thesis, was to support English Wikipedia as well. Even the size of Swedish Wikipedia made scalability a concern for the implementation. To solve this I found that resorting to simple data structures was the best way. It was the solid and intuitive ideas that was the fastest and easiest ones to use. I used solutions based on primarily hash-tables which in average has a complexity of $O(1)$ if implemented correctly and tree based data-structures which could have $O(\log n)$ complexity for lookups. The tree based data-structure was particularly useful because it scales well and provides a sorted list of keys with connected data.

3.1.4 Rich querying

I found that dealing with a large dataset required a good front-end that allows quick navigation and provide the information I need during testing and evaluation of a solution. To set up an environment manually could be time consuming and this is something a proper front-end helps with. Using just the debugger is not desirable when there is much data because it does not allow you to easily jump around in the data and inspect how the code worked in different situations. When determining how the front-end should be built I quickly abandoned the idea of building a Java user-interface because it would be too time consuming and would not easily provide the flexibility and power I needed. The criteria I sat up for a graphical front-end solution was:

- It should be able to handle a reasonably large amount of data (5 - 10 MiB raw data output)
- Allows quick and easy construction of rich user interfaces
- Have good documentation in that the APIs or libraries I would use should be well documented or have a strong community that has solutions to common problems.

The best solution for this job was in my mind a modern web browser which meant the use of web technology. There has gone a large amount of effort into building a web browser and web technology is today so capable that building desktop grade applications using only a web browser is a reality.

The ability to access the application from virtually anywhere in the world by just using a modern web browser was also very appealing and I had prior experience of web development which meant that this was the best choice for me.

The use of web technology also meant that the server side had to support multiple requests at the same time which was solved by following a convention that all retrieval methods used must be thread safe. From earlier experience I know that if web development is done incorrectly it can be a slow and painful process. To solve this it meant balancing the amount of server side code and client code. I made the choice to run as much code as possible in the browser and let the server act as data service providing processed data. This meant finding good software for the server side was essential which is described further in section 4.6.2 on page 45.

3.2 Entity detection

Detection is about finding possible named entities in a text. All the following methods splits up the text into tokens and tries to match a sequence of tokens to candidates by using a dictionary created by using one of the methods below. The first method was simple to implement and based on the assumption that all Wikipedia articles are named entities, the second method is an extension of the first with a larger dictionary size and a small difference in how the named entities are matched. The last one uses the Named Entity Recognizer (NER) found in the library Stagger [23] created by Robert Östling [24].

Method 1: Noun clustering matched a sequence of words, an N-gram (which I limited to 10 tokens in sequence), to a list of entity candidates using a dictionary of normalized page titles. For Swedish this meant using the longest sequence of nouns found in the dictionary. The algorithm began with finding all sequences of nouns, all combinations and then methodically trying them all out.

Method 2: Surface form matching matched a sequence of tokens to known anchor texts, page titles and redirect page titles. This method only uses sequences of tokens where the first token is a noun (in case of Swedish). The matching is done against a dictionary of anchor texts mapped to a list of candidate entities. It is the longest exact match that is chosen as a source of candidates. It is similar to noun clustering but not equal because it looks at more data and can select words that are not nouns.

Method 3: Stagger NER uses the library Stagger's built in NER to find entities in a text and uses only the tokens it says to denote a named entity.

In table 3.1 there is an example of the surface form dictionary used in method 2 and 3.

Normalized surface form	Entity
berga	Helsingborg
helsingborg	Helsingborg
helsingborgs stadsförsamling	Helsingborg

Table 3.1: Samples in the dictionary of surface forms

Entity	Popularity
Helsingborg	1 842
Lunds Kommun	220
Skåne	3 748
Sverige	42 216

Table 3.2: Some samples for a popularity dictionary

3.3 Disambiguation

All methods here use the candidate list provided by some detection method defined in the previous section.

3.3.1 Method A: Title edit distance

I needed a method that did not require any special indices to be created which is why I came up with this method. This method only uses a normalized title of an entity page. Basically you use a dictionary of normalized titles which was created at import and a mapping to the corresponding entity.

The method computes the edit distance, the minimum number of changes needed to make it equal to its counterpart which is b in this case. The method used is defined as:

$$\text{sim}_{a,b}(i,j) = \begin{cases} \max(i,j) & if\ \min(i,j) = 0 \\ \min \begin{cases} \text{sim}_{a,b}(i-1,j)+1 \\ \text{sim}_{a,b}(i,j-1)+1 \\ \text{sim}_{a,b}(i-1,j-1)+\text{matches}(i,j) \end{cases} & otherwise \end{cases} \tag{3.1}$$

where a and b are strings and $i = |a|$, $j = |b|$ and $\text{matches}(i,j) = 1$ when the characters a_i, b_j do not match, in all other cases $\text{matches}(i,j) = 0$.

The disambiguation algorithm

1. Compute the similarity between a candidate entity title and a normalized form of the input tokens.

2. Select the candidate that has the lowest edit distance, if more than one has an edit distance of 0 which is the lowest possible value, then select the first random hit.

3.3.2 Method B: Popularity

This method needs a dictionary of popularity. This dictionary returns given a candidate a popularity value. This value is equal to the number of unique in-links to a Wikipedia article or stub. I selected this method because it has provided a reasonable result for other languages before when reading Hoffart et. al [19] under results and competitors summary. An example of the popularity dictionary is shown in table 3.2.

The disambiguation algorithm

1. Take the candidates from the entity detection and retrieve its popularity
2. Select the candidate whose popularity is highest

Chapter 4
Architecture

In this chapter I will provide a description of how the finalized system was actually implemented.

4.1 Overview

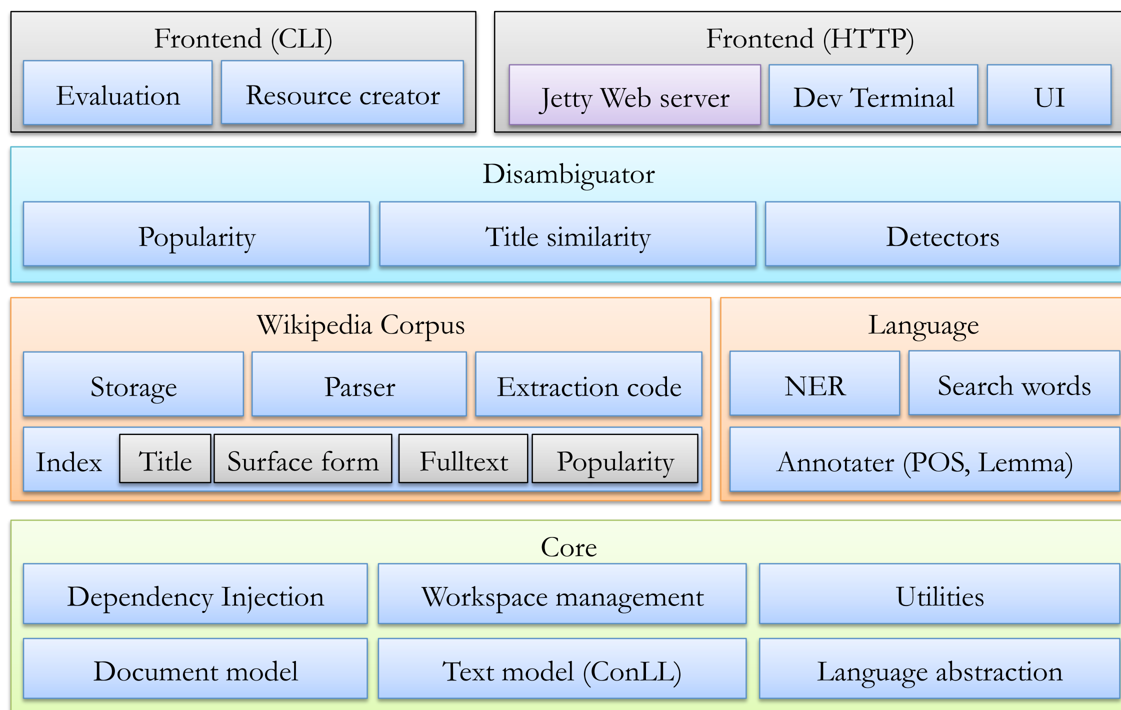

Figure 4.1: Architectural overview

Figure 4.1 shows an architectural overview of Nedforia. Every large block represents a module in Nedforia, there are essentially 5 parts:

Frontend is the interface to the application and it provides two entry points: one via a Command Line Interface (CLI) and one web interface over the HTTP protocol. The CLI provides access to offline methods such as index construction and initial import, these are tasks that could be time consuming and needs to be executed offline. The web interface provides search, a development terminal that allows users to query content within the system and the actual interface to the disambiguator.

Disambiguator is the part which actually implements disambiguation, it is a small part because it relies on code mostly in Wikipedia corpus and language.

Wikipedia Corpus is the largest part and it contains all parser code, storage implementation and index implementations for the Wikipedia import.

Language contains the language specific bits such as a NER provided by the Stagger component (Robert Östling [24]).

Core provides everything else, and it covers the document model that will be described later as well as a modified ConLL model from Peter Exner. The language abstraction basically provides the interfaces that needs to be implemented and they are all implemented in the language.It also implements the dependency injector solution, however the configuration is implemented in the frontends and wikipedia corpus.

4.1.1 Document model

In order to store a Wikipedia page I needed to define a structure of how it should be stored. The basic requirements was that it should support multiple languages, be able to assign properties, attaching data of types that can be extended as development progressed.

The first document model contained an id, title, language, text, annotated text, page type and wiki markup. This model was very hardwired to the Wikipedia model which made it difficult to generalize when Wikipedia is not a requirement. This lead to revisions that added support for multiple filtered texts and annotated documents. I used the library Protocol Buffers (protobuf) created by Google[25] to serialize[1] and deserialize all data. The choice of protobuf was rigidity, performance, and that messages can be extended with more fields in the future. I could have used Java serialization but the fact that protobuf messages are shorter and has better interoperability with other programming languages made protobuf a better choice.

The parts of the final document model

Id is a unique identifier of a document, a 64 bit integer value.

Title of the document.

Uri is an unique identifier, which was compatible with Wikipedia URLs.

Language is a ISO-631 identifier for the language, 2 letters ("sv" for Swedish, "en" for English) or 2 letter language, and 2 letter region ("en-US" for American English).

[1] encode data structures into a storage format

Properties is a dictionary of keys and values. Here, wiki page type was stored and its original id as a string.

Entries could be described as a miniature file-system where each entry is a file that can be serialized or deserialized by some handler. Every entry consists of three parts: path, type and the raw data. The paths are similar to ordinary filepaths used in a UNIX compatible environment e.g "/wiki/ast" which is the Abstract Syntax Tree (AST). The use of filepaths made it simpler to group information.

Text is the filtered content of the document, here multiple different parts can be stored by an identifier. By convention "main" is the core text, "bulletlist[3 digits]" are filtered versions of bulletlists and "template[3 digits]" are filtered versions of templates. The texts are stored as entries in the actual implementation.

Annotated documents or ConLL document which is the annotated versions of the filtered texts. These annotated documents are stored in a format where you can assign multiple properties to a token and arrange the tokens in sentences.

The entries can be extended by implementing a new handler. I used the same construction myself to extend a generic document when writing an handler that could serialize and deserialize the AST tree created by the parser. The only requirement for a handler is that the data types that the handler assigns must be unique otherwise a clash of formats could occur. Unsupported entries are passed through and supported even if they can't be deserialized. I made this decision because it simplifies the construction of an external reader, it only requires a crude understanding of the model to actually be able to read and manipulate it without destroying existing data. The only thing required when implementing a reader externally is to have the protobuf definition of the storage format.

4.2 Import

Figure 4.2 on the following page provides an overview of the import pipeline. It all starts with a Wikipedia dump and the one I selected was the one created on 2013-02-25. The format is compressed XML that uncompressed was 13.02 GiB. The dump contains raw wiki markup, a time stamp of the last edit, and a namespace id that is defined in a header. These namespace ids have a language dependent string attached to it. These namespaces are language dependent.

To extract information from the Wikipedia dump I read the compressed file directly and feed the uncompressed data as a stream to a Streaming XML parser a Streaming XML parser (StAX parser), The parser used was Woodstox [26] which appeared to be fast when reading comparative tests of performance between different parsers [27] and it was easy to use in my opinion. Loading the entire XML document i.e. the use of a DOM parser would be impractical due the memory required. The Woodstox parser is a XML pull parser which means that code has to tell the parser when to move forward instead of receiving handle incoming events during parsing as in e.g. a SAX parser.

This output is then feed into a type classifier that assigns a type to every page e.g if it is a category, template or article. Because the header is language dependent a matching configuration had to be provided in order to for a proper classification to take place. Redirect pages and disambiguation pages are not determined by looking at the namespace id instead they are

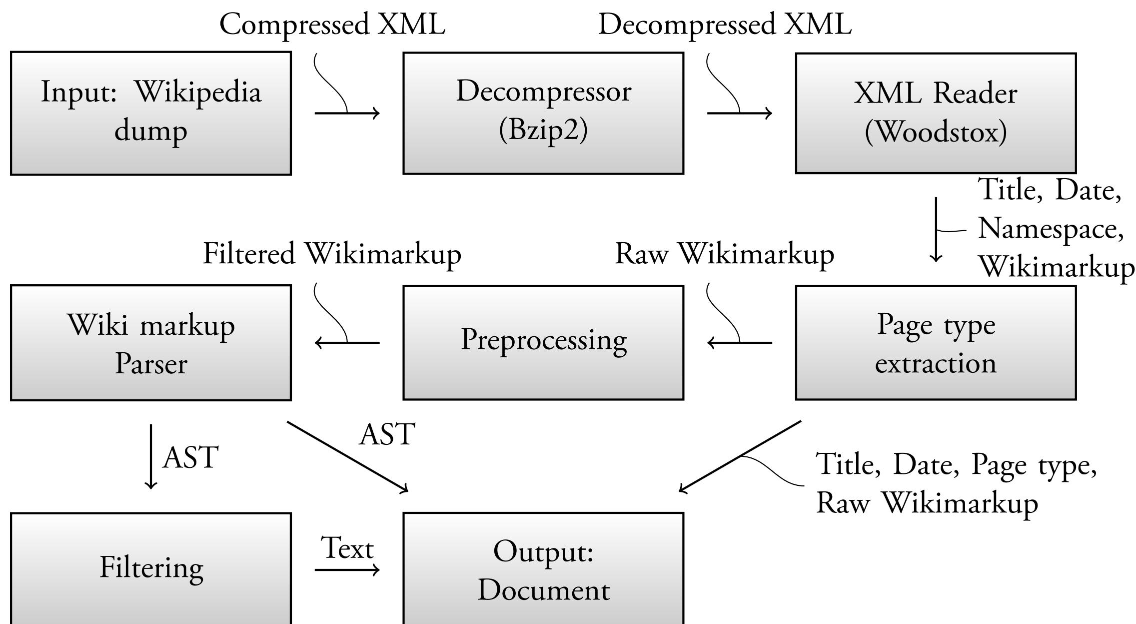

Figure 4.2: Import pipeline

based on a string search of the wiki markup. Redirect pages are matched by looking if the page starts with "#REDIRECT" or "#OMDIRIGERING" which is specific to the Swedish version. Disambiguation pages are matched by looking for a specific template anchor that exists in the pages, it is most commonly called "förgrening" or "gren" in the Swedish edition which were the two matched. Stubs are matched by looking for used templates where the template name ends with "stub". Pages with a namespace id of 0 were assigned the Article type.

All other type ids are called unknown if the title of the page does not begin with either "Kategori:" (category) or "Mall:" (template) which again is specific to the Swedish version, in the English edition they are called "Category:" and "Template:".

Once a page has been read and classified it is put through a parser to generate filtered text that is used to extract links from and to display.

4.2.1 Wiki markup parser and AST tree generation

I searched for wiki markup parsers but I failed to find one that could be integrated into the import. I did find one that worked in Java: Sweble [28]. Sweble fulfilled a subjective measure of quality but the performance was not good so I ended up with building my own, that way I got better performance and got full control.

The parser consists of two parts, a preprocessing stage and a deep parsing stage. All HTML and math formulas are removed in the preprocessing step. Also, all escaped characters or entity characters such as "&" (which corresponds to: "&") and quote characters like " ' ' " are converted into their proper Unicode representation before parsing. The preprocessing step also makes sure that all links and templates are ended properly by automatically inserting end symbols where needed. The algorithm uses the number of end symbols to determine the nesting depth e.g. "[[[link 1]] [[link 2]]" becomes "[[link 1]] [[link2]]" and not "[[[link1] [[link2]]". This choice made most sense when evaluating real errors made by users. This

algorithm makes sure that anything that follows an error is captured correctly. You could say that the preprocessing step auto corrects fatal errors that would crash the main parser by heuristically choosing a fix.

The deep parser consists of two parts: a tokenizer[2] and grammar logic. The tokenizer is built using token lists and lexer rules[3] defined in the rule language used by JavaCC which is hand tuned to accept any input and tokenize it. The full tokenizer is then produced by JavaCC. JavaCC does produce a full parser but it is not used, only its tokenizer and lexer are used. The grammar rules was instead programmed by hand. The reason behind this decision was because I could not fully capture the structure and make it fault tolerant at the same time. To use the grammar rules meant implementing full support or nothing and even that is very hard. My hand programmed version can selectively ignore bits it does not understand without failing. Creating a full compliant parser meant that it would most likely not be tolerant to unexpected code written by Wikipedia users and accept incorrect wiki markup code in graceful way.

The parser is top down recursive which means that it takes the full markup and iteratively captures more and more fine-grained details until the smallest detail has been found. I built it using smaller parser functions that can handle different contexts. This could be e.g. when you have found a link where the character "|" means to give an option, but in global context it is just a character that should be included or just ignored. The fact that some tokens have different meanings in varied contexts also contributed to the choice of building the parser by hand. All rules were derived from best case scenarios by sampling Wikipedia articles, particularly the cases that failed to parse instead of building a parser based on the user manual of how the code should ideally be written. The AST is built directly by the parser functions which has limited support for more complex scenarios. However, by conducting a second pass of the AST I suspect that it would be possible to get the structure right.

This is the list of elements that the parser captures:

- Headers, with level (could be used to reconstruct the relation between paragraphs)
- Paragraphs of text and links merged together.
- Lists, which include both unordered and ordered lists
- Tables, rudimentary support, future work could extend this into a proper column and cell structure
- Templates, with argument keys and values
- Links with target and options that are used e.g in miniature images
- Horizontal rules, i.e section dividers

The parser ignores indentations but does use it as a breaking rule when splitting up paragraphs. From the tree that was created a main filtered version was created that excluded bullet lists, tables and template content. Filtered versions of template content and bullet lists was also

[2] a tokenizer is the part that splits incoming strings into logical parts. e.g "Sweden is a country" into {Sweden, is, a, country} .

[3] a lexer is the part that finds tokens by using a rule instead of constants e.g. sequence of numbers

created to capture links because this significantly increased the number of found links by almost 3 times.

In the end a document is produced with an attached AST tree, filtered versions of the text and all links found in the wiki markup.

4.3 Indexing and storage

One of the largest issues I had was to find an indexing system that was simple and fast enough and still capable of handling large amounts of data. I tested a variety of storage engines but most failed for one reason or another. The conclusions presented here should be taken with a grain of salt because I did not strive to write the most optimized code for every solution. My goal was to construct a solution that was clean and simple and still get good performance characteristics.

4.3.1 Requirements

Some requirements I had when selecting a storage solution was:

- Simple to use API, Java compatible
- Allow high concurrency, many threads (8, 16 or more) that was primarily reading
- Embeddable, not depending on a full server that must be maintained and run separately
- Handle datasets that has millions of rows or keys. Basically if the system can handle the size of 100 million to 1 billion without problems then scaling is not a problem. This requirement is difficult to pin down so it all comes down to testing a solution and see if it is good enough.

Using any Relational Database Management Systems (RDBMS) was quickly ruled out because they were not practical due to the common problem of slow interfaces. I needed something that could process a large amount of simple (100 000 or more a second) questions from multiple threads. A RDBMS typically is connection orientated and execute statements making thread usage cumbersome. SQLite is the fastest straightforward embedded RDBMS I know of but the implementation I initially used: SQLite4java [29] which is single threaded and even with prepared statements[4] I could only get about 10 000 queries from Java answered a second, but with a fully Java based solution like the database engine BerkleyDB [30] I ended up with performance in the range of 250 000 - 300 000 questions answered a second on the same data . These results are heavily dependent on the input data making general conclusions hard, suffice to say: BerkleyDB worked good enough for my purposes. If a faster solutions is needed I would have to resort to memory based solutions which is heavily limited to the available RAM of the computer it runs on.

I tested other solutions such as Neo4j [31] and OrientDB [32]. Neo4j showed promise when using it with transactions as it was designed but when doing batch inserts which provided the maximum performance it failed. The frustrating bit is that I got problems with both solutions only when scaling up by using a larger data set. This made it hard to write down a

[4] SQL based queries that has been compiled to minimize overhead

bug report because it worked in smaller test cases I wrote. It could be that I misunderstood some aspect of how the solutions should have been used.

4.3.2 Indexing

I selected Lucene [33] as my fulltext indexing engine. Lucene is very well documented, modular, has great community support and can handle large amounts of data. The only drawback I found was that Lucene primarily uses hashtables meaning that a sorted list could not be extracted without doing the sorting in memory which would defeat the purpose. A sorted list is useful in some cases such as quickly ruling out that a sequence of tokens exists in a dictionary. The rule of thumb was that Lucene could provide quick answers for specific questions. When I needed a sorted list I then resorted to BerkelyDB. BerkleyDB uses a variant of a binary tree which is optimized for hard drive storage.

4.3.3 Storage

Most of the time the raw data was processed in a sequential fashion by the algorithms, otherwise data was accessed randomly which depending on the type of hard drive could be very slow. The absolute fastest way to do sequential read is to naturally store the information in sequential order on the disk. I created a custom solution in the end because I wanted something simple, that supported multiple threads writing or reading (not both). I also did not have to worry about supporting very memory-constrained systems which meant I could use 128 MB for an intermediary buffer before writing it to the file-system. I used a dedicated thread to do all writing. When reading linearly I prefetched data i.e. read ahead before needing it.

The storage format is a sequential list of records where each record has a header with a size defined by a 32 bit integer and followed by stream of compressed data with that length. I created a indexing file parallel to the storage file that is a list of longs (64 bits) so that you instantly can find where a record is stored. I used a library called Snappy [34] to compress the records and the records were a serialized version of the document model. I used snappy because it provides a good tradeoff between processor usage, size reduction, compression and decompression performance. In practice when using Snappy the overall size was reduced to 47,1% of the uncompressed size.

4.4 Resources

In Nedforia I created a concept that I call resources. There are two types of resources: data and computational. A data resource is basically some form of index, data storage or something else that you can read from or write data to. A computational resource could be a resource that generates statistics, does evaluations or more. An index that knows how to create itself is both a data and a computational resource.

The basic requirement for a resource is that it has to be able to tell if it exists. For data resources it must define how to open and close it, the open differs if it is a readable resource or just a resource that writes. The computational resource must specify resources it depends on and method that does creation and deletion. The fact that computation resources defines dependencies mean that Nedforia can check if the required resources exists and if they do not and are computational, they are automatically created. In Nedforia e.g. the popularity

dictionary that contains the mapping from entity to a popularity count is a computational data resource because it knows how to create itself and allows reading of data from the index.

4.5 Annotation

The filtered text that originated from the AST produced by the parser is annotated by using two components: Stagger and Maltparser. Stagger is a part of speech tagger created by Robert Östling. Stagger provides automatic segmentation of sentences and it is trained on the Stockholm-Umeå Corpus (SUC) version 3.0 with additions such as the Swedish morphological lexicon SALDO [24]. Stagger scored an 96.4% accuracy when trained and tested against SUC 2.0 and 96.57% against SUC 3.0. [24].

Maltparser is a dependency parser that creates a projective tree and gives a structure of how words relate to each other and was also trained on SUC. Maltparser got a F-measure score of 84% on Swedish in 2007 [35]. I used a model from 2012 downloaded from Maltparsers main page[36] which I did not find any updated scores for.

Maltparser [36] can be used to find noun and verb phrases as described in section 2.1.3 on page 15. The product of these two components is a document with sentences and tokens with attached properties such as surface form, lemma form, part of speech tags, dependency relations and morphological features such e.g. if the word is in singular form.

4.6 Front end

The front end has two interfaces: one that is accessible from a web browser and one that is accessible by invoking Nedforia from the command-line.

4.6.1 Command line

This is were most heavy work is invoked from. There are 2 main commands: import, create. Import does what it sounds like, it does a full import from a Wikipedia dump. This is done by specifying where it is, and where the workspace shall be located.

Create is the most used function after import, it allows you to specify a goal and the dependencies for that goal is automatically found and computed from code if possible. A typical goal could be annotation which performs annotation of all wiki pages. Annotation needs links to be resolved and this will be done before annotation begins. The create command fits a rudimentary definition of a build system.

To find resources that could be created from command-line I used reflection with the help of the library “reflections” [37] version 0.9.9-RC1 and it simplified the implementation of new commands. The reflection library was integrated with my build system which was Maven [38] and it has a plug-in that scans the project and creates a file containing all types that exists in the project. This mitigates some of the drawbacks that exists in the Java reflection implementation. For the command-line I used the library Commons CLI [39] to format and parse the command line, a simple and powerful library that was enough for my needs.

4.6.2 Web interface

This is were most of the power from using Nedforia comes. The web interface was built using Jetty [40] version 9 which is a web server that was embedded as a module inside Nedforia in conjunction with the Spring Framework [41][5] version 3.2.2.

From the Spring Framework I used a module that provides the MVC[6] pattern which simplified the construction of the server side code and JSP[7] as the view engine. The implementation of MVC in Spring Framework uses reflection to find controllers and know when a particular request handler should be called. To add support for a new URL it is as simple as adding a new java method and annotating it with the annotation type "RequestMapping".

All this combined results in quick and easy development of web applications on the java platform. The only downside is that there was a lot of dependencies. It took some painstaking effort to make it all work initially. Finding the right dependencies to use and assembling the right set of libraries that work together was not easy. Searching the Internet for solutions resulted in that this was eventually sorted out.

User Interface

There are three main parts: search functionality, document browsing and a web based terminal.

The search functionality enables the user to quickly find an indexed document and it was implemented using Lucene as the indexing engine. You can search via title, fulltext and type of document (wiki page type) in the user interface. The ranking was done by Lucene.

Document browsing displays the data computed and data extracted from a Wikipedia page. The information is in-links, out-links, entities, what entries exists in the document, the original raw wiki markup text, the AST tree and the filtered text with highlighted named entities as determined by Staggers NER.

Web based terminal is most likely the most important part of the user interface when doing prototyping and development. It is a web based command-line interface that allows the user to enter commands that runs code on the server side and returns the result to the browser. The output could be a table, HTML, text or formatted text. This allows infinite possibilities to get back powerful presentations of results from the server side code. The server side code uses java reflection to find commands and adding a new command to the terminal is as simple as adding a new method in a class. There is also a state object that is managed by the browser, this allows to execute commands that operates on specific input data. The terminal uses the JavaScript "jQuery terminal" on the client that was modified to accept HTML output as a result. There was also JavaScript code written to do asynchronous communication with the server that used JSON to transmit its content.

[5] It is a collection of libraries that simplifies web development on the Java platform.

[6] Model-View-Controller (MVC) is a architectural design pattern that is commonly used when designing user interfaces. It provides a good separation of concerns.

[7] JSP is a markup language that combines server side code (that could output dynamic data) with HTML code

The design is based on a template from Twitter called Bootstrap [42], it provided a clean design and was easy to work with.

Chapter 5

Evaluation

In this section, a presentation of how the results were computed will be provided along with the evaluation methods used.

5.1 Gold standard

All evaluations are based on a gold standard which defines what the correct answers should be. For Swedish I could not find any gold standards on named entity disambiguation using Wikipedia as a source of knowledge. The gold standard was therefore created by me using manual annotation.

I took in total 10 articles from Aftonbladet, Dagens Industri, Dagens nyheter, Helsingborgs Dagblad, IDG, NyTeknik, Svenska Dagbladet. In order to get a representative result I took articles of varying topics, the topics were ordinary news, sport news, tech news, economic news and an article about culture.

Annotation procedure The articles found were hand annotated in the following way:

1. I took the raw text and made Nedforia annotate it but not disambiguate. This produced a ConLL file that contained all sentences and tokens, it did however contain Staggers NER result.

2. Using an editor that I built and began to mark all Named Entities I could find by hand. The tool only showed the tokens and sentences, you could hover and see the specifics of each token but I ignored those details. Some examples of named entities I looked for was persons, organizations, countries and so on. I did not follow the guidelines of Wikipedia that the first concept or named entity found in a text should be linked but not the ones that followed, I marked them all. All choices here arguably impacts the result, there will be a deeper discussion of this in section 7.1.

3. I then attempted to resolve each named entity I had marked to a Wikipedia article. I used title search, fulltext search and Google to make sure that it did exist in Wikipedia and compared it with my dump.

4. Finally when I verified that the named entity existed I added the entity URL in a format my system understands. This is equal to the Wikipedia URL and if there was no hit the hand-annotated named entity was removed.

5.2 Implementation

The system tries to map sequences of tokens in an input text to an article. A link is a sequence of tokens and only when this sequence fully matches a gold link will it be considered as a possible match.

The possible match is checked against the gold standard to see if it points in the correct direction otherwise it will be determined as incorrect. To use the theoretical framework of precision and recall: a relevant document is a link that points to the correct article and matches the exact same tokens as the gold one and a retrieved document is a link the system mapped.

Concretely, I then computed precision and recall as follows:

$$\text{precision} = \frac{|\{\text{found gold link and it was correct}\}|}{|\{\text{found links}\}|} \tag{5.1}$$

$$\text{recall} = \frac{|\{\text{found gold link and it was correct}\}|}{|\{\text{gold links}\}|} \tag{5.2}$$

Chapter 6

Results

6.1 Statistics

It is hard to describe the contents of my dataset because it is large and I do not have the time to read through it all, it is simply too big. Because of this reason I will here present some statistics extracted from the dataset I used (2012-02-25 dump) and give some remarks about the results.

Type	Count	Relative freq.
Pages in dataset	1 690 600	100.0%
Articles and stubs	815 056	48.2%
Redirects	632 686	37.4%
Other (Template, Categories, Administrative pages, and more)	225 321	13.3%
Disambiguation pages	16 396	0.97%
Articles and stubs without links	1 141	0.067%

Table 6.1: Page statistics

In table 6.1 you can see the distribution of page types, one thing that stands out is that the number of redirects is high compared to the number of articles. This makes a case for that redirects are a good source for alternative names. There are also few articles and stubs with absolutely no links which means that most articles will have a popularity value.

All links was extracted from the AST and not the filtered text. The links used was however extracted from annotated text in articles and stubs. The results are presented in table 6.2 where it can be seen that roughly half of all the found links are used. The rest lies in primarily templates which I ignored.

Type	Count	Relative freq.
Links found	20 392 129	100.0%
Links resolved	18 612 117	91.3%
Links in articles and stubs	19 738 403	96.8% (100%)
Links resolved in articles and stubs	18 079 921	88.6% (91.6%)
Links used in articles and stubs	10 593 781	52.0% (53.7%)

Note: Numbers in parenthesis is relative to articles and stubs

Table 6.2: Link statistics

6.2 Detection

The first part of disambiguation is to find suitable candidates to disambiguate which is the task of the detector. Nedforia has two primary indices for doing this: the title index and the title + anchor + disambiguation index. Both are surface form indices in that they look at normalized tokens, but they do work a little differently. The title index uses a normalized form that is provided by Lucene and the larger index uses lower case lemma forms provided by in this case Stagger.

Type	Count
Search terms in title index	1 441 117
Search terms in title + anchor + disambiguation index	1 835 117

Table 6.3: Index statistics

A search term is a single sequence of tokens that has been used in e.g. a title or anchor text. I did not expect that the number of search terms to be only be 27.3% more when compared to the smaller title index in table 6.3. The reason for this is that the number of anchor texts was more than 7 times as great as the number of article, stub, disambiguation and redirect page titles combined. I hypothesized that the anchor texts contained significantly more variants than the page titles but this proved to be false when reading the actual results. The results arguably proves that most anchor texts corresponds to the page title.

Detection algorithm	Recall	Precision	F1-score
Title	83.58%	19.40%	31.49%
Title + Anchor texts + Disambiguation page titles	**86.57%**	19.21%	31.44%
Language NER (Stagger in this case)	72.14%	**81.46%**	**76.52%**

Table 6.4: Detection method results

For the detectors the final results are in table 6.4 and it shows how well the detectors find named entities when compared to my hand annotated test set. It is easy to see that you lose a lot of precision to get higher recall when using texts that is used to refer to an entity. Only when using a real NER does precision go up. It is interesting to see that the precision is always at the expense of recall, however this is a small test set and hard to general conclusions from.

6.3 Disambiguation

I computed results for all 6 combinations shown in tables 6.6 and 6.7, the summary of the results are presented in table 6.5. From the results, using a real NER and popularity gives the best results at the expense of recall. It is also clear that popularity is a good metric as can be seen since it consistently produces a better result compared to title similarity.

When using one of the surface form indices and popularity recall is good but precision is low. The reason for this is that it is over linking, it treats even common words as references to article pages which is an indication that many articles in Wikipedia does not describe a named entity.

F1-scores	Title similarity	Popularity
Title	11.81%	20.81%
Title + Anchor texts + Disambiguation page titles	11.74%	26.02%
Language NER (Stagger)	**28.50%**	**66.49%**

Table 6.5: Disambiguation f1-scores summary

Detector	Recall	Precision	F1-score
Title	31.34%	7.27%	11.81%
Title + Anchor + Disambiguation titles	**32.34%**	7.12%	11.74%
Language NER (Stagger)	26.87%	**30.34%**	**28.50%**

Table 6.6: Title similarity

Detector	Recall	Precision	F1-score
Title	55.22%	12.82%	20.81%
Title + Anchor + Disambiguation titles	**71.64%**	15.89%	26.02%
Language NER (Stagger)	62.68%	**70.78%**	**66.49%**

Table 6.7: Popularity

Chapter 7
Discussion

7.1 Evaluation

The major evaluation error sources are:

- Limited test set of only 201 named entities from 10 news articles
- The selection articles
- The way I determined that a named entity exists and could be disambiguated.

These three error sources could be mitigated by increasing the size of the test set or asking more persons to create a hand annotated test set. However, I could not find any publicly available named entity test set for Swedish on the internet. This made hand annotated by myself the only viable option for an evaluation.

When selecting the articles, I tried to the best of my ability to create a balanced test set that tested the good and the bad of Nedforia. Nevertheless, from an analytic standpoint I could subconsciously have affected the selection despite my best intentions. You could argue that a solution to this problem would be to make the selection automatic e.g. take the top article from multiple sources. But in order for this method to be successful I would argue that the number of articles you have to gather must be large. The size increases the chances for the test set to be balanced which due to time constraints was not possible.

Another important source of error is that I used title search and Google to determine if a named entity could be resolved to an entity within Wikipedia. This method has its drawbacks for the reason that it is dependent on normalized string matches in either the text or the title of an article. Despite this drawback I would argue that the method is good enough. In my opinion, no method could give a 100% definitive answer unless someone read it all and manually verified it.

7.2 Wikipedia

The Swedish edition contained as can be seen by table 6.1 on page 49 a large amount of redirects. These redirects in combination with anchor texts provided a good source to connect words in a text to a potential candidate. The recall was the highest when detecting named entities by using anchor texts, redirect titles and article titles. It could have been better if I dropped the requirement that it had to start with a noun, an example of this is "Vita huset" which in English is "White House", "Vita" is not a noun however the entire sequence is a named entity. The Stagger NER detected "Vita huset" correctly but its database did not contain many newer articles which is why the recall dropped. In addition, as can be seen by the results recall was always at the cost of precision.

A large untapped resource for disambiguation was the category system. It provides a way to semantically group relevant articles together. However the categories are not always consistent. Take the idea of turning all name pages such as "Mats" into a disambiguation page. One way of finding these is by looking at categories because "Mats" belongs to a category "Svenska mansnamn" which has the top category of "Förnamn". But in this top category articles such as "Tilltalsnamn", "Förnamn" are also found which are semantically related but they should not be classified as a disambiguation page. In this case there are only few exceptions but it does require some manual effort in order to achieve high quality.

7.3 Data mining

The part that affects the performance of the system the most would be how data is gathered and processed. If the information could not be extracted, the system would not be able to find it later.

7.3.1 Page classification

I used a heuristic to classify a page as described in section 4.2 on page 39. As I stated before pages that deal with names such as "Mats" are not classified as disambiguation pages. They contains many links to specific targets, and has a small hint of differences between the persons which makes it useful for disambiguation. I speculate that the reason is that "Mats" is a generic name which has a history with a meaning which results in that it do not fit the strict definition of a disambiguation page. However, for the problem of Named Entity Disambiguation it would be preferable to classify it as a disambiguation page. These kinds of gray cases where it is not a disambiguation page but it would have been desirable to be classified as such requires some amount of manual effort which is language specific.

Category pages, template pages and other administrative pages could reliably be classified by only looking at the title, this had to do with the fact that they must belong to a namespace which is separated by a colon e.g. "Kategori:Svenska mansnamn". Disambiguation pages was classified by looking for specific templates but there are more variants than those I searched for which could impacted the end result.

7.3.2 Parser

The parser that I built is not in any way fully compliant with the wiki markup, it basically ignores any bit it does not understand and excluded math formulas and HTML code early which could contain important information. There was a parser that claimed high precision and that was Sweble [28] which I ruled out due to being slow. The quality of the parser and the resulting AST does contribute to the end result because I harvest all filtered information from the AST. I recognized that templates are not good candidates for text but they do contain interesting links, one example is an episode summary of a TV series where the use of a template is common. The infoboxes also often link to other persons and capturing these was important. I started with just links in the main text but later extending it to use bullet lists and templates which resulted in that about 3 times as many links were found.

Including filtered copies of templates and bullet lists into the main text would also mean the introduction of unwanted noise. The main text should ideally be separated paragraphs containing only sentences. If you include filtered templates and bullet lists it would distort this paragraph separation and also introduce elements which is not natural text. This was solved by separating each bullet list and template into its own filtered part, in that way I could easily differentiate between the noise and good parts.

There was also a need to balance the amount of information extracted and the amount of information to ignore. Nedforia uses bullet lists, template content, and paragraphs at the top level, to harvest links from. Tables were initially completely ignored but contained important information such as bullet lists so I built partial support to parse tables into Nedforia. It should be relatively easy to create a good AST structure for the tables by doing a second processing pass of the tree.

7.4 Resources for the Swedish language

As mentioned earlier, I could not find any test data for named entity disambiguation for the Swedish language. A standardized comparison between different works arguably requires the access of an impartially and publicly available created gold standard for named entity disambiguation. The prior works I went through used different datasets to evaluate their methods. The fact that they did not use a coherent test set meant that I had a hard time drawing any far reaching conclusions from them, other than a subjective feeling that this method would work well and the results seem to confirm that.

However, for the Swedish language there does exists large hand annotated corpora and one of them is: Stockholm Umeå Corpus (SUC). The corpora contains part of speech tags, syntactic, and morphological features for a variety of texts and it is balanced, meaning it contains a selection from a variety of sources in order to cover most aspects of the language. This corpus was essential in order to create high quality part of speech taggers like Stagger. Nevertheless, there is one dataset that is largely missing for Swedish and that is an equivalent of WordNet that is easily accessible and is of high quality. A equivalent resource like WordNet could provide generalizations of words and provide semantic connection between words and much more.

7.5 Detection

Three methods were tested and only the real NER provided reasonable precision. The reason that the other two methods got poor precision was that they found too much. The root cause is that Wikipedia contains many articles that are not a named entity such as articles about almost every punctuation mark there is, numbers, years, etc.

The first method that from the beginning only used Lucene generated unrelated results, which had to do with its normalization functions that strips of suffixes and then removes very common words. When searching for "Danmark" the two top hits are in order "Hans av Danmark", and "Danmark". The words "Hans" (name) and "av" (eng: of) will be removed due to that they are very common i.e. stop words. This is due to "Hans" is a name in this context and not *his* as it means in another context.

One way to improve this would be to use heuristics to determine if an article is a named entity or not based on its title and content. Some methods to do this were used in previous works as can be read in section 2.5 on page 20. However, while these heuristics might work well for English, Swedish is a little different. An example of this is the named entity "Sveriges regering" with the English equivalent article "Government of Sweden". This named entity consists of multiple tokens but only the first token "Sveriges" has a capital letter and not the latter which would make the heuristics fail to detect this as a named entity. Nevertheless, many names would work such as "Göran Persson".

One approach to mitigate these drawbacks could be to combine heuristics and an entity catalog such as YAGO. Here, YAGO would be the gold standard dictating which articles are named entities and the cases when YAGO do not contain them, the heuristics would decide. In any case, having access to a good NER is crucial for precision. In addition, in order to get a good recall the entity catalog should also be somewhat up to date and capture many different topics, something Wikipedia primarily provides.

Another way that might improve detection is to implement coreference, which means to connect words in the text that refers to the same entity. This is helpful because you can then gather multiple presentations of the same entity such as abbreviations and other ways that you could refer to an entity. This would only work if articles from the start were assigned the Named Entity status.

7.6 Disambiguation

Title similarity was bad in most cases as it tries to match strings approximately and give scores after it. It meant that an close match always produced the best results. I used the same cost for all characters and this could be improved by using statistics of letters and adjusting the costs to match the statistical weights. However, I do not believe that this method should ever be used because it generates too many false hits. The biggest issue is that when you have two candidates with equal scores the ranking fails. It then requires some other method to differentiate like comparing the text via a method such as cosine similarity.

Popularity performed as expected, however it had a large drawback in that it will never find unpopular articles. Popularity is good for when you have something that almost always only refers to one single thing. But if there is a more unclear division of popularity

or if the popularity is low uncertainty would be high. Should the system ever hope to match human ability to disambiguate, then more data has to be put into consideration when ranking. Popularity could be improved by using commonness statistics. One example when popularity fails due to not taking into account how often a particular surface form refers to an candidate is the word "stad" which means city. The correct article is "Stad" (city) but "Frankrikes kommuner" which in English the Municipalities of France will be the one selected. This was very strange at first, but I found that in a few anchor texts that refers to "Frankrikes kommuner" it had been called just "stad". Commonness would most likely have solved this issue.

Context similarity is another way to improve previous methods described. By looking at the actual text, anchor texts, use key phrases, categories and compare it against the input text a greatly improved disambiguator could be created. The main issue with context similarity is that the quality is dependent on how the context is modeled and there exists many ways to model a context as can be seen by previous work. Basically what context similarity gives is a measure of how compatible a candidate is compare to an input text, it would be helpful when popularity is uncertain or when popularity needs to be suppressed.

Coherence could be used as a way to ensure overall compatibility of named entity matches but it is based on the assumption that the input text is homogenous. I would also describe it as a way to model a global context and make sure that the named entities found fits that context. One method would be to compare the semantic relatedness between all candidates and reevaluate candidates that has poor semantic relatedness. This method however quickly becomes unfeasible if the number of candidates is large, so approximations must be used to solve the combinatorial explosion that could otherwise occur.

Chapter 8
Conclusions

8.1 Future work

Improving the disambiguation and testing out heuristics for named entity detection are big candidates for future work. The disambiguation could be improved by using context similarity measures primarily by either implementing cosine similarity or some other method that work better. Another improvement would be to use an entity catalog such as YAGO to dictate detection and use heuristics to extend the YAGO catalog. Given some manual effort that is specific to Wikipedia I suspect that you could greatly improve extraction of some types of information.

Another thing which is important in order to scale up Nedforia is to implement a storage solution that works in a clusters such as when you use Hadoop [43]. Implementation of more languages such as English would be highly desirable in order to compare methods against the English language.

8.2 Summary

Much time and effort went into the technical and practical issues of solving the core problem of this thesis. This meant that more advanced methods could not be tested because there was little time to do that. I proved that it is possible to create a named entity disambiguator for the Swedish language using simple methods. The size and scope of Wikipedia presented some unique challenges and I found out that simple and solid solutions worked best. The final F_1-score of 66.49% is nothing spectacular but not surprising considering the method used. Subjectively the methods that find potential candidates from a given string works surprisingly well and can be confirmed by looking at the recall figures. There is a clear path for improvement and a foundation that hopefully will accelerate future work.

Terminology

API
: Application Programming Interface, A defined interface that specifies how software modules communicate with eachother. 42

AST
: Abstract Syntax Tree is a high level abstraction of how a bit of source code is writen. It can be used to extract expressions in the case of programming languages and in the case of wiki markup: links, paragraphs and more. 39, 41, 42, 44, 45, 49, 55

BOW
: Bag of Words is a class of methods that operates on a collection of tokens in some way. Something that distingues BOW methods is that they do not need to understand what the token mean or what is semantically related to it. 24

DOM
: Document Object Model is an API standard defined by W3C to access contents of e.g. XML, it typically has the entire document loaded in memory. 39

HTML
: HyperText Markup Langauge is the language used to define content in web pages. 17, 40, 45, 55

JSON
: JavaScript Object Notation is compact text based format which is used to serialize and deserialize javascript objects. 45

JSP
: Java Server Pages is a mix of HTML code and server side code for web development. 45

NED
: Named Entity Disambiguation is about the automatic identification of a real-world reference to a given string. 21

NER
: Named Entity Recognizer is a detector for named entities, it detects things like names, organizations, country names and so on. 34, 38, 45, 47, 50, 51, 54, 56

NLP
: Natural Language Processing is the disciplin of processing natural language that humans use to communicate with eachother. 11, 13–15

RDF
: Resource Description Framework is a way to attach metadata to web page or other sources. Examples of metadata could be when a page is created. In the case of an entity catalog such as YAGO it could be when a person has been born. 18

URI
: Uniform Resource Identifier is a way to identify a resource of some kind. An example of an URI could be a URL, URL extends with information about how you can retrieve a resource. 18

URL
: Uniform Resource Locator is commonly known as a web address. 38, 45, 48

XML
: eXtensible Markup Language is a hierarchical text based storage format that is portable and easy to read. 18, 39

YAGO
: Yet Another Great Ontology is a knowledge database that semantically connects entities such as persons and defines relations between them. 18, 19, 56, 59

Bibliography

[1] IBM. IBM Watson. 2011. Last accessed: 2013-11-10. URL: `http://www-03.ibm.com/press/us/en/presskit/27297.wss`.

[2] Wikipedia. Statestik. `http://sv.wikipedia.org/wiki/Wikipedia:Statistik`, 2013. Last accessed: 2013-11-10.

[3] MediaWiki. Welcome to MediaWiki.org, 2013. Last accessed: 2013-10-13. URL: `http://www.mediawiki.org/wiki/MediaWiki`.

[4] The PHP Group. PHP: Hypertext processor, 2013. Last accessed: 2013-10-13. URL: `http://www.php.net/`.

[5] Wikipedia. Wiki Markup, 2013. Last accessed: 2013-11-10. URL: `http://en.wikipedia.org/wiki/Help:Wiki_markup`.

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

[7] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706. ACM, 2007.

[8] W3C. Resource Description Framework (RDF). Last accessed: 2013-11-10. URL: `http://www.w3.org/RDF/`.

[9] DBpedia. About, 2013. Last accessed: 2013-11-10. URL: `http://wiki.dbpedia.org/About`.

[10] DBpedia. DBpedia 3.9 downloads, 2013. Last accessed: 2013-11-10. URL: `http://wiki.dbpedia.org/Downloads39`.

[11] DBPedia. DBpedia Data Set Statistics, 2013. Last accessed: 2013-11-10. URL: `http://wiki.dbpedia.org/Datasets/DatasetStatistics`.

[12] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. Yago2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 194(0):28 – 61, 2013. URL: http://www.sciencedirect.com/science/article/pii/S0004370212000719, doi:http://dx.doi.org/10.1016/j.artint.2012.06.001.

[13] Max-Planck-Institut Informatik. YAGO2s: A High-Quality Knowledge Base, 2013. Last accessed: 2013-11-10. URL: http://www.mpi-inf.mpg.de/yago-naga/yago/index.html.

[14] Nancy Chinchor. Muc-4 evaluation metrics. In *Proceedings of the 4th conference on Message understanding*, MUC4 '92, pages 22–29, Stroudsburg, PA, USA, 1992. Association for Computational Linguistics. URL: http://dx.doi.org/10.3115/1072064.1072067, doi:10.3115/1072064.1072067.

[15] Razvan Bunescu. Using encyclopedic knowledge for named entity disambiguation. In *In EACL*, pages 9 – 16, 2006.

[16] Silviu Cucerzan. Large-scale named entity disambiguation based on Wikipedia data. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 708 – 716, Prague, Czech Republic, June 2007. Association for Computational Linguistics. URL: http://www.aclweb.org/anthology/D/D07/D07-1074.

[17] David Milne and Ian H. Witten. Learning to link with wikipedia. In *Proceedings of the 17th ACM conference on Information and knowledge management*, CIKM '08, pages 509–518, New York, NY, USA, 2008. ACM. URL: http://doi.acm.org/10.1145/1458082.1458150, doi:10.1145/1458082.1458150.

[18] Xianpei Han and Jun Zhao. Named entity disambiguation by leveraging wikipedia semantic knowledge. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 215 – 224, New York, NY, USA, 2009. ACM. URL: http://doi.acm.org.ludwig.lub.lu.se/10.1145/1645953.1645983, doi:10.1145/1645953.1645983.

[19] Johannes Hoffart, Mohamed Amir Yosef, Ilaria Bordino, Hagen Fürstenau, Manfred Pinkal, Marc Spaniol, Bilyana Taneva, Stefan Thater, and Gerhard Weikum. Robust disambiguation of named entities in text. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '11, pages 782 – 792, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics. URL: http://dl.acm.org/citation.cfm?id=2145432.2145521.

[20] Google. Guice (version 3.0), 2011. Last accessed: 2013-11-10. URL: https://code.google.com/p/google-guice/.

[21] Conll-x shared task: Multi-lingual dependency parsing. Last accessed: 2013-11-09. URL: http://ilk.uvt.nl/conll/.

[22] Wikipedia. Wikipedia Dump (2013-02-25), 2013. Last accessed: 2013-11-10. URL: http://dumps.wikimedia.org/svwiki/20130225/svwiki-20130225-pages-articles.xml.bz2.

[23] Robert Östling. Stagger – The Stockholm Tagger, 2013. Last accessed: 2013-10-13. URL: http://www.ling.su.se/english/nlp/tools/stagger/stagger-the-stockholm-tagger-1.98986.

[24] Robert Östling. Stagger: A modern pos tagger for swedish. 2012.

[25] Protocol Buffers - Google's data interchange format (Version 2.5.0). Last accessed: 2013-11-09. URL: https://code.google.com/p/protobuf/.

[26] Codehaus Foundation. Woodstox - high-performance XML processor (version 4.2.0), 2013. Last accessed: 2013-11-10. URL: http://woodstox.codehaus.org.

[27] Eduardo Rodrigues. A comprehensive XML processing benchmark. 2008. Last accessed: 2013-11-10. URL: http://java2go.blogspot.se/2008/04/comprehensive-xml-processing-benchmark.html.

[28] Open Source Research Group at Friedrich-Alexander-Universität Erlangen-Nürnberg. Sweble wikitext components, 2011. Last accessed: 2013-11-10. URL: http://sweble.org/projects/swc/.

[29] ALM Works Ltd. sqlite4java (Version 0.282), 2012. Last accessed: 2013-11-10. URL: https://code.google.com/p/sqlite4java/.

[30] Oracle. Oracle berkeley db java edition (version 5.0.73), 2013. Last accessed: 2013-11-10. URL: http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html.

[31] Neo Technology, Inc. Neo4j (Version 1.9.4, Beta 2.0.0-M3), 2013. Last accessed: 2013-11-10. URL: http://www.neo4j.org/.

[32] Orient Technologies LTD. OrientDB (Version 1.3), 2013. Last accessed: 2013-11-10. URL: http://www.orientdb.org/.

[33] The Apache Software Foundation. Lucene (Version 3.6.2, 4.2), 2013. Last accessed: 2013-11-10. URL: http://lucene.apache.org/.

[34] Google. Snappy (Version 1.1.0-M3), 2013. Last accessed: 2013-11-10. URL: https://code.google.com/p/snappy/.

[35] Joakim Nivre, Johan Hall, Jens Nilsson, Atanas Chanev, Gülsen Eryigit, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2):95–135, 2007.

[36] Johan Hall, Jens Nilsson, and Joakim Nivre. Maltparser. Last accessed: 2013-11-10. URL: http://www.maltparser.org/.

[37] Reflections (Version 0.9.9-RC1), 2013. Last accessed: 2013-11-10. URL: https://code.google.com/p/reflections/.

[38] The Apache Software Foundation. Apache Maven (Version 3.0.5), 2013. Last accessed: 2013-11-10. URL: http://maven.apache.org/.

[39] The Apache Software Foundation. Apache Commons CLI, 2010. Last accessed: 2013-11-10. URL: http://commons.apache.org/proper/commons-cli/.

[40] The Eclipse Foundation. Jetty (Version 9.0.2), 2013. Last accessed: 2013-11-10. URL: http://www.eclipse.org/jetty/.

[41] GoPivotal, Inc. Spring Framework (Version 3.2.2), 2013. Last accessed: 2013-11-10. URL: http://projects.spring.io/spring-framework/.

[42] Mark Otto and Jacob Thornton. Bootstrap (Version 2.3.2), 2013. Last accessed: 2013-11-10. URL: http://getbootstrap.com/2.3.2/.

[43] The Apache Software Foundation. Apache Hadoop, 2013. Last accessed: 2013-11-10. URL: http://hadoop.apache.org/.