The University of Manchester

Lexical Disambiguation

MSc Project 2014-2015 - Progress Report

Christophe François

Supervisor: Dr Allan Ramsay

May 2015

Abstract

Word Sense Disambiguation is an open problem of Natural Language processing which consists in identifying the sense intended by an ambiguous word used in a sentence. A large number of approaches have been studied to solve the problem, notably the application of machine learning techniques to WSD. This project aims to apply a graph-based technique that consists in performing the Personalized PageRank algorithm (PPR) on WordNet definitions, and to improve it by combining several knowledge resources.

Contents

List of Figures

List of Abbreviations

LKB	Lexical Knowledge Base
NLP	Natural Language Processing
PPR	Personalized PageRank algorithm
TF-IDF	Term Frequency - Inverse Document Frequency
WSD	Word Sense Disambiguation

Word count: 7038

1 Introduction

1.1 Word Sense Disambiguation

Word Sense Disambiguation (WSD) is the task of assigning to a word its intended sense. It's a major problem in Natural Language Processing (NLP) and has been of concern since the first works in the area [9].

Natural language is inherently ambiguous, in contrast with formal programming languages that can be understood by computers. An ambiguity arises when a sentence can have different meanings. It can be syntactical, when it's due to the structure of the sentence. "I looked at the man with the telescope", for example, is unclear: did I use a telescope to look at the man, or did I see a man who had a telescope ?

Here, we're interested in lexical ambiguity, when a word can have several senses depending on the context of its use. For example, the verb "run" can mean "jog", "campaign", "execute"... The problem is often easily solved by a human, as we simply interpret the word in a sentence, almost unconsciously: the phrase "I'm running a program on a computer" doesn't leave much room for interpretation. It's not as easy however for a computer.

WSD is easy to us, but it is an AI-complete problem, which means it is as hard to solve as the hardest problems of Artificial Intelligence [17]: notably common sense, knowledge representation and reasoning. It is, however, a necessary part of most NLP tasks. In fact, it was historically conceived as a fundamental step in machine translation, the automatic translation of a text between two natural languages. Different senses of a word indeed often must be translated differently: for example, "run" can, depending on the context, correspond in French to "courir", "faire campagne", etc. Other applications include text processing, information extraction.

1.2 Aim and objectives of the project

A lot of approaches to WSD have been researched in the past decades. We can divide them in two categories: machine learning techniques applied to WSD, and knowledge-driven approaches which rely on the use of a Lexical Knowledge Base (LKB) like WordNet [8]. State-of-the-art knowledge-based methods consist in processing the underlying graph structure of the LKB [3]. In fact, a recent method based on the Personalized PageRank algorithm (PPR) has shown very good results [2].

The aim of the project is to develop a program performing automatic lexical disambiguation using the Personalized PageRank algorithm. It will be achieved with the following steps:

- extract the graph structure of WordNet definitions

· extract definitions of several dictionaries and add them to the WordNet graph

· apply the Personalized PageRank algorithm to the graph

· evaluate the software

1.3 Structure of the report

Next section presents several approaches to Word Sense Disambiguation, and more precisely the approach used for this project. I will then describe tools used in the development of the project, followed by its methodology. Finally, I will present the progress I've made in the project so far, and what the next steps are.

2 Background Research

2.1 Approaches to the problem of Word Sense Disambiguation

In the past decades, there have been a large number of attempts to solve the problem.

2.1.1 Early works in WSD

The first discussions on WSD were made in the context of Machine Translation. The issue arose with early attempts which performed a simple word by word translation with mixed results: the use of context is necessary as natural languages are inherently ambiguous. The memorandum of Weaver, in 1949 [21], discusses problems in mechanic translation, and in particular the need for disambiguation.

The use of the context of the word is immediately proposed as a solution to the problem, and the issue becomes to determine how much context would be necessary to perform the disambiguation of a word: which is the minimal number of words needed to be certain of the sense of a word? This number depends on several things: is the word a verb, a noun, a pronoun? How many senses does it have? Is it in a technical text or is it from a more general domain?

Consequently, researchers focused on the automatic translation of texts of specific domains, developing micro-glossaries which limited the senses of a word to those relevant to a domain. For example, in a mathematical text, the word "square" could mean the geometric figure or the mathematical operation, but not the urban area.

Weaver also highlights the statistical character of the problem. Several publications propose ideas regarding this approach: use of the most frequent sense of a word, estimation of the degree of polysemy in a text.

Those early publications present problems and ideas that are still relevant today; they were however unable to apply them due to lack of resources [9]. The advancements in the field coincide therefore often with the development of large-scale resources like knowledge bases and usable corporas.

2.1.2 The Most-Frequent-Sense baseline

A simple technique of disambiguation, and yet difficult to outperform by more elaborated algorithms, consists in attributing to each ambiguous word the sense that is most often intended by it.

It's easily computed on available sense-annotated corpora (like SemCor [15]), but can be refined as it is highly domain-dependent.

This baseline is hard to outperform and represents a useful measure of the performance of a WSD algorithm [17].

2.1.3 Data-driven or Machine Learning methods

With the emergence of larger corpora and more generally resources for natural language processing, searchers have been able to apply machine learning algorithms to the problem of WSD. In fact, about every machine learning algorithm have been adapted for WSD.

The most important issue is to define and compute features that can be used for supervised or unsupervised techniques. I will now describe some of these features before presenting several machine learning techniques commonly applied to WSD.

The features try to represent most of the information given by the context of the ambiguous word w. Such features include: [7]:

- words around w (in the local context): $w_{-2}, w_{-1}, w_{+1}, w_{+2}$...
- collocations of words around w: $(w_{-2}, w_{-1}), (w_{-1}, w_{+1}), (w_{+1}, w_{+2})$...
- triplets of words around w: $(w_{-3}, w_{-2}, w_{-1}), (w_{-2}, w_{-1}, w_{+1})$...
- part-of-speech information of the local context: $p_{-2}, p_{-1}, p_{+1}, p_{+2}$...
- broad context information (about terms that are not directly next to w)

Of course, the set of features chosen will depend on the algorithm applied to the problem; exemplar-based methods for example, described in section 2.1.3.b, tend to perform better on local context than with broad context (which contains a large number of features compared to the few training points).

In the following sections, several algorithms of machine learning applied to WSD are presented.

2.1.3.a Statistical approach, the Naive Bayes classifier

The Naive Bayes algorithm is a probabilistic classifier that applies Bayes' theorem to perform classification, with the assumption that features are independent from one another. Bayes' theorem is stated below, where A and B are independent events:

$$P(A|B)P(B) = P(B|A)P(A)$$

$P(A)$ is the probability of A, $P(A|B)$ is the probability of A when B is true. In machine learning, we can define for each point the events as its class and the value of its features; assuming independent features, we can decompose the conditional probability:

$$P(C|x_1, x_2, ..., x_n)$$

where C is the class of **x** and the x_i are its features, into:

$$P(C|x_1, x_2, ..., x_n) \propto P(C) \prod_{i=1}^{n} P(x_i|C) \qquad \textbf{(1)}$$

The result of the classification is the class that maximizes the value given in (1).

When applied to WSD, the class of a word is the sense associated to it. To disambiguate a word, we find all occurrences of this word in the corpus, observe their sense and their features to compute the different probabilities in (1); the algorithm returns the sense with highest probability given the values of the features of the ambiguous word.

2.1.3.b Exemplar-based method, the nearest neighbour classifier

The k-nearest neighbours (k-nn) algorithm is a classifier that consists in finding, for a given test example, the k examples of the training set that are closest to it. The algorithm then performs a majority vote between the neighbours' classes to return the class of the test example. Figure 1 shows an example of k-nn classification.

In the neighbourhood of the test example (green) in Figure 1, with k=3, there are two points of class red and one point of class blue, so the algorithm classifies the test as a point of class red.

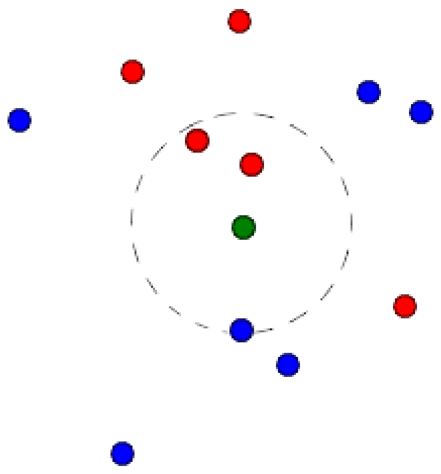


Figure 1: Example of k-nn classification, with k=3

This algorithm can be applied to WSD. Several techniques have been tried, using different sets of features (words before and after, part-of-speech information) and different distances to find the nearest neighbours.

For example, the commonly-used Hamming Distance is the number of features that are different between two examples. If we use the word before and the word after "like" as the feature set, then the Hamming distance between "I like trains" and "just like a" is 2, and the Hamming distance between "I like trains" and "I like puppies" is 1. More elaborated distances can be used for better results [7].

2.1.3.c Support Vector Machines

A Support Vector Machine (SVM) is an algorithm which, in a given dataset, tries to calculate the hyperplane that separates, with the largest margin, data points into two classes. Figure 2 shows an example of SVM applied to a simple dataset. The algorithm finds the optimal hyperplane with the maximum margin between classes red and blue.

By adapting SVM to a multi-class problem (for example with binarisation of all features), we can apply it to WSD. This technique produces among the best results, compared to other supervised techniques [12].

Although they tend to get better performance, supervised methods are limited by what is called the knowledge acquisition bottleneck (the lack of annotated corpora): these approaches require large

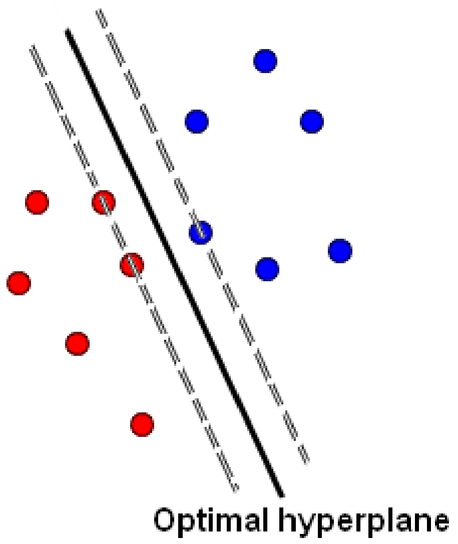


Figure 2: Example of SVM on a simple dataset

corpora for training, which are very expensive to create. Moreover, machine learning techniques are really dependent on the specificity of the data. Corpora for technical domains do exist (notably in biology and medicine [19]), but the manual labelling of large corpora for every domain is simply unfeasible.

Several approaches to relieve the knowledge acquisition bottleneck are being developed since several years [17]: automatic creation of training data, collection of corpora on the web, automatic enrichment of knowledge resources... Another way to tackle the issue is to perform unsupervised methods of disambiguation.

2.1.3.d Unsupervised methods

Unsupervised techniques are not subject to the knowledge acquisition bottleneck as they are applied to unannotated corpora.

The idea is that words with similar senses have similar contexts, so an unsupervised algorithm tries to compute clusters corresponding to different senses. It's an open problem known as Word Sense Induction [16], the objective being to identify automatically different senses of a word instead of using a manual sense inventory.

We can differentiate context clustering and word clustering. The first computes context vectors, and groups words with similar vectors (either with a measure of distance between vectors or by considering the frequency of co-occurrence of words). The second method consists in clustering semantically similar words (synonyms notably). In both cases, the output of the algorithm is an ensemble of clusters corresponding (hopefully) to different senses.

These methods tend to perform less well than other approaches, but the comparison is difficult since the clustered senses have to be matched with existing sense inventories. However, their independence to manual resources gives them interesting potential [17].

2.1.4 Knowledge-based approaches

In parallel to data-driven approaches, techniques relying on the use of Lexical Knowledge Bases (LKB) were developed.

LKBs are resources that provide semantic information on words. They can be simple machine-readable dictionaries, which provide senses and their definitions; thesauri, which explicit relationships between words (such as synonyms, antonyms, meronyms, hyponyms); or notably lexicons (such as WordNet), that combine information of a dictionary and a thesaurus but often provide other information such as a hierarchy of senses.

The advantage over supervised methods is that knowledge-based approaches don't necessitate expensive manually labelled corpora, and therefore can be used on a larger scale.

2.1.4.a Overlapping definitions, Lesk algorithm

This simple algorithm [13] consists of counting the number of words that are in both the context and each definition of the ambiguous word. The chosen sense is the one with the most overlapping words.

This technique gives mixed results, as it is highly dependent on the formulation of the definition: the exact same words need to be found to be counted. The results of the algorithm with thus vary a lot with different dictionaries for example.

2.1.4.b Similarity measure

This sort of technique isn't applicable to a machine-readable dictionary like Lesk algorithm because it exploits other information given by the LKB.

The similarity for example is a function that makes use of the semantic relationships between senses to compute a score between 0 and 1. For example, it can be calculated by finding the closest shared ancestor of two words by hypernymy relationship (A is a hypernym of B if A is a type of B, for example animal is a hypernym of cat).

From here, we can disambiguate a word by finding the sense that maximizes its similarity with the words found in the context.

2.1.4.c Graph-based approaches

These methods exploit the underlying graph structure of LKBs, extracting it to use an algorithm developed for graphs.

At this date, state-of-the-art methods consist in the application of the PageRank algorithm to the graph structure of a LKB [2]. This project aims to apply such an algorithm, which will be described in the following sections.

2.2 WordNet, a Lexical Knowledge Base

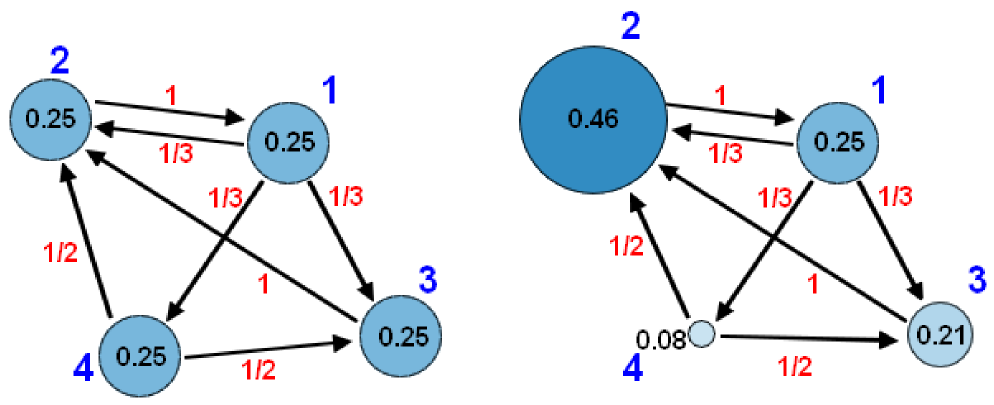
WordNet [8] is an enumerative lexicon, which is both a dictionary of senses with their definitions and a thesaurus with a hierarchy of said senses. It includes several lexical and semantic relationships between senses, such as hypernomy, antonymy...

It was developed for the English language at the University of Princeton, where it is still maintained, and it is at date the most widely used knowledge-base in WSD. The latest version, WordNet 3.1, provides 117,659 senses to 155,287 words. Those include nouns, verbs, adjectives and adverbs.

The hierarchy of WordNet is composed of synsets, which are sets of words corresponding to a given sense. Each synset is accompanied by its definition and sometimes examples of usage. For example, the synset dog_n^1 contains the words (or lemmas) $\{dog, domestic\_dog, Canis\_familiaris\}$, and corresponds to the dog as a species. The word dog, however, has several senses and belongs accordingly to several synsets: $\{dog_n^1, frump_n^1, dog_n^3, ..., chase_v^1\}$.

For each synset, WordNet also provides a certain number of relationships to other synsets:

- hypernyms and hyponyms (if A is a kind of B, then B is a hypernym of A and A is a hyponym of B); for example, dog_n^1 is a hyponym to $canine_n^2$ but a hypernym to $corgi_n^1$.
- holonyms and meronyms (if A is part of a B, then B is a holonym of A and A is a meronym of B); for example, dog_n^1 is a meronym of $pack_n^6$ as a member of a group of dogs, and a meronym of $canis_n^1$ as a member of the Canis genus.



(a) Initial system (b) After 1 iteration

Figure 3: PageRank algorithm applied to a simple graph

Since Wordnet is a widely used tool, several projects exist to improve it. WordNets for several languages have been developed, creating opportunities for WSD in other languages but also interesting tools for machine translation.

Although it's very used, there are some limitations to the WordNet lexicon. It is often criticized for the fine-grainedness of its definitions, which is often far above what one could consider necessary in the task of WSD [9].

2.3 A graph-based method using PPR

2.3.1 PageRank and Personalized PageRank

PageRank [18] is an algorithm created in 1998 to order web pages by their relative importance. It's especially known for being the first algorithm used by Google Search, which was in fact initially an application of PageRank.

The idea is that if a page is important, then a lot of pages will link to it; so the algorithm counts the number and importance of links that lead to the page. The goal is to emulate the behaviour of a random surfer. Starting from a random web page, what is the probability that, after clicking a number of times, he will arrive to a certain page? That probability is the PageRank of the page.

2.3.1.a Description of the algorithm on a simple graph

Figure 3.a shows a simple graph composed of 4 nodes and 7 edges.

The value of a node (so a page) represents the probability that the surfer has clicked on a link leading to this page. In the PageRank algorithm, this probability is uniform in the initial state. The edges represent links going from a page to another; the surfer then clicks on a random link of the page with a uniform probability for each link.

On the graph, we first apply a uniform value of $\frac{1}{4}$ to all 4 nodes and apply weights to each edge. The value of an edge is the inverse of the number of edges coming from the same node; for example, 3 edges originate from the first node and are therefore applied a weight of $\frac{1}{3}$.

During the first iteration, the surfer clicks on a link to another page. The probability that the surfer is on a given page can be calculated using the probabilities of the previous iteration and the probability that he clicked on the link leading to the page.

For example, on the graph of Figure 3, we can calculate the probability that the surfer is in each node after 1 iteration. There is one edge leading to node 1, it originates from node 2:

$$P_1^{(1)} = P_2^{(0)} \times w_{2\to 1} = 0.25 \times 1 = 0.25$$

In the same way, for node 2:

$$P_2^{(1)} = P_1^{(0)} \times w_{1\to 2} + P_3^{(0)} \times w_{3\to 2} + P_4^{(0)} \times w_{4\to 2} = 0.46$$

The values of each node after the first iteration are shown in Figure 3.b. We can generalise the formula for a node a, if we consider that the weight of a non-existing edge is 0, to:

$$P_a^{(i+1)} = \sum_{j\in\{nodes\}} P_j^{(i)} \times w_{j\to a}$$

We repeat this iteration until the equilibrium (or after a certain number of iterations for cases of non convergence). The final state is shown in Figure 4 for the example graph. The value of each node is the PageRank.

2.3.1.b Matrix notation

We define the transition matrix in the algorithm as the weighted adjacency matrix of the graph. The adjacency matrix A is a matricial representation of a graph, of size the number of nodes, defined by: $A_{i,j} = 1$ if there is an edge from node j to node i. For example, for the graph in Figure 3, the adjacency matrix is A and the transition matrix is M:

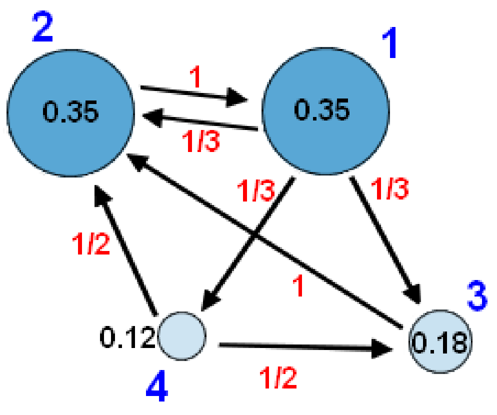


Figure 4: Final state of the previous graph

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \qquad M = \begin{pmatrix} 0 & 1 & 0 & 0 \\ \frac{1}{3} & 0 & 1 & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & 0 \end{pmatrix}$$

The initial state of the system is stored in a vector **v** of size the number of nodes, and $v_i = P_i^{(0)}$.

$$\mathbf{v} = \begin{pmatrix} 0.25 & 0.25 & 0.25 & 0.25 \end{pmatrix}$$

The next state of the system is then obtained by computing the dot product of M by **v**, $M\mathbf{v}$. In the previous example,

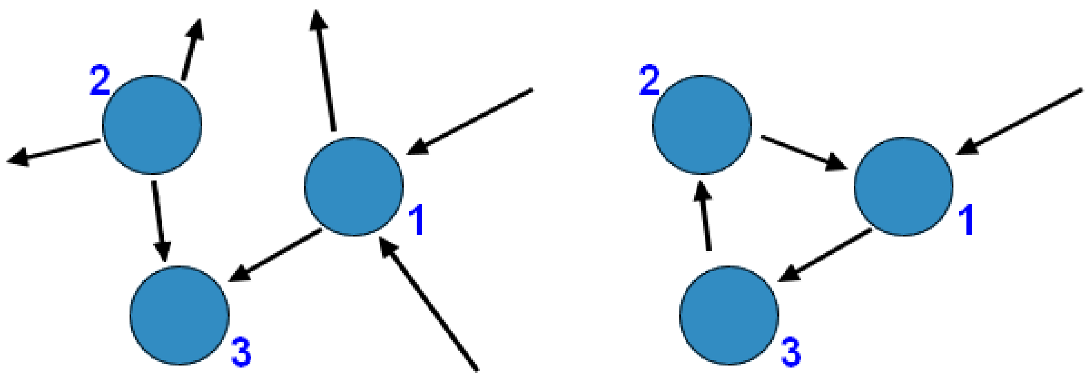
$$M\mathbf{v} = \begin{pmatrix} 0.25 & 0.46 & 0.21 & 0.08 \end{pmatrix}$$

At each iteration, we multiply the result of the previous one by M, so at iteration i, we have calculated $M^i\mathbf{v}$. When the equilibrium is attained at iteration n, then we have computed $M^n\mathbf{v}$, which is the rank vector of the system.

$$M^n\mathbf{v} = \begin{pmatrix} 0.35 & 0.35 & 0.18 & 0.12 \end{pmatrix}$$

2.3.1.c Damping factor

The simple iteration seen in the previous section can give bad results if, notably, there is a node or a loop in the graph with no outgoing edge. In the case of a single node, as in Figure 5.a, there is a constant loss of total rank in the graph: at every iteration, the rank of the 3rd is not redistributed, and the rest of the graph keeps giving it rank. Therefore, the ranking vector of the graph converges to the



(a) Rank sink caused by a single node (b) Rank sink caused by a loop

Figure 5: Examples of rank sink in PageRank

null vector.

The other case, shown in Figure 5.b, is that of a loop in the graph with no outgoing edge. As there is no edge leaving the loop, there is no loss of total rank in the loop. In fact, the incoming edge makes the total rank of the loop increase for each iteration. In this case, the total rank of the nodes in the loop converges to 1 whereas the rank of the rest of the graph converges to 0.

Such a closed system is called a rank sink. To overcome the issue, the algorithm is improved with a damping factor α.

The iteration becomes:

$$\mathbf{v}^{(i+1)} = \alpha M \mathbf{v}^{(i)} + (1-\alpha)\mathbf{v}^{(0)}$$

The goal is, again, to reproduce the behaviour of a random surfer; if the surfer is caught in a loop of links, he will not stay in it but will start again from a random website. The damping factor corresponds to the probability that, at any point in his navigation, he decides not to click on a link but to go to a random other website. The original paper on PageRank proposes to use $\alpha = 0.85$ which is therefore the default value of most applications [18].

2.3.1.d Personalized PageRank

The PageRank model is designed to emulate the behaviour of a random surfer, starting its navigation from a random website. However, a surfer often doesn't consult websites randomly, but with a certain goal.

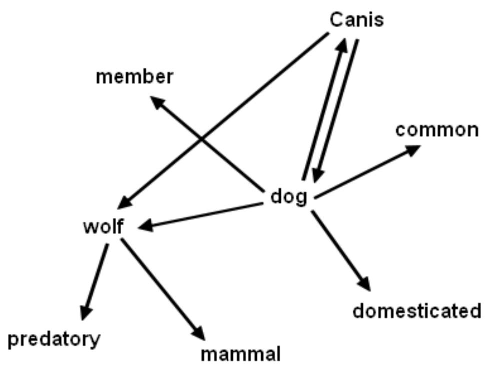


Figure 6: Simplified subgraph of senses linked by their definitions

The PPR aims to model this behaviour by modifying the initial state in the PageRank algorithm. The probabilities for each node are not uniform anymore but are weighted, personalized to obtain a different but more appropriate ranking system. The vector of the initial weights is called the Personalization vector.

2.3.2 Personalized PageRank in WSD

PPR can be applied to the graph structure obtained in graph-based approaches to WSD. I will describe the method that I intend to implement for this project, which is PPR applied to a sense dictionary [5].

One underlying graph structure of a sense inventory is the graph in which nodes correspond to senses. There is an edge from a node A to a node B if the sense B appears in the definition of A. In the case of a non-disambiguated dictionary, we don't know which sense is intended by a word in a definition, so we add all senses of the word to the graph. However, it won't give as good results as the disambiguated graph [5]. Figure 6 shows a simplified subgraph (constructed around dog_n^1, as in the canine species) with WordNet definitions.

We don't apply classic PageRank to the whole graph as it would return a ranking of senses present in the dictionary, and that is not interesting. There are two methods to use the algorithm for WSD.

The first is to find a subgraph that is relevant to the context of the disambiguation and to perform classic PageRank on it. Several methods have been used to compute that subgraph, for example by finding the shortest paths between words of the context [4].

The second is to perform PPR on the whole graph, putting weight on terms of the context to find relevant highest ranking senses; it gives better results as the first technique [5]. This is the approach I plan to use for my disambiguating software.

2.4 Matching definitions of different dictionaries

To improve performance of PPR applied to a dictionary (notably WordNet), I plan to combine several sense inventories and extract a graph that contains information about all sources. However, different sense inventories have a different structure for their senses, and a different classification. Therefore, it won't be possible to simply take definitions of all sources and put them in the graph, I need to find correspondences between the senses of each dictionary.

A definition is often composed of one, maybe two sentences. For a given word, we have two sets of senses with definitions, and the objective is to find matches between the sets. If we consider the sets of definitions as a corpus and each definition as a document, then the problem becomes to find similar documents in a corpus, for which several methods exist.

One of them is the measure of Term Frequency - Inverse Document Frequency (TF-IDF) Vector similarity, which has been shown to give among the best results in measure of sentence similarity [1].

The TF-IDF is a measure of the importance of a word in a document contained in a corpus. It's the product of two measures, the Term Frequency and the Inverse Document Frequency.

Term Frequency is the number of occurrences of the word in the document; its importance is obviously directly proportional to this measure. It can also be pondered in regards to the length of the document, to avoid giving higher scores to terms in longer documents.

Inverse Document Frequency is the fraction of documents of the corpus in which the term appears, which is then logarithmically scaled. This is a measure of the specificity of a term: if it appears in few documents, then it's probably more important, more relevant than a word appearing in every document of the corpus.

We can represent the document as a vector. Its features are the terms that appear in the corpus, and its values are computed using the TF-IDF measure.

After calculation of vector representations of each sentence, we can find similar sentences by a measure of cosine similarity, which is a measure of similarity between vectors. It's the quotient of the dot product of the vectors and the product of those vectors' magnitudes. For vectors a and b,

$$similarity = \frac{a.b}{||a||.||b||}$$

3 Research method

3.1 Development tools

3.1.1 Python Natural Language Toolkit

Python Natural Language Toolkit (NLTK) [6] is a Python library that allows to perform several tasks of natural language processing (NLP).

It contains a large number of modules useful in NLP: several can perform a NLP task such as Part-of-Speech tagging, Sentiment analysis, etc. Other modules are interfaces that allow easy access to several corpora and knowledge bases, and mainly WordNet, which is the reason why this project is developed in Python.

The rest of the section focuses on the WordNet interface provided by NLTK. I will present some of the functions that allow us to run through the WordNet hierarchy.

```
>>> from nltk.corpus import wordnet as wn

# get all synsets containing the input word
>>> wn.synsets('dog')
  [Synset('dog.n.01'), Synset('frump.n.01'), ..., Synset('chase.v.01')]

# get the definition of a synset
>>> wn.synset('dog.n.01').definition()
  'a member of the genus Canis (probably descended from ...'

# get all lemmas contained in a synset
>>> wn.synset('dog.n.01').lemma_names()
  ['dog', 'domestic_dog', 'Canis_familiaris']

# some semantic relationships are defined for synsets
>>> wn.synset('dog.n.01').hypernyms()
  [Synset('canine.n.02'), Synset('domestic_animal.n.01')]
>>> wn.synset('dog.n.01').hyponyms()
  [Synset('basenji.n.01'), ..., Synset('dalmatian.n.02')]

# others are defined for lemmas
>>> wn.lemma('happy.a.01.happy').antonyms()
  [Lemma('unhappy.a.01.unhappy')]
>>> wn.lemma('happy.a.01.happy').derivationally_related_forms()
  [Lemma('happiness.n.02.happiness'), Lemma('happiness.n.01.happiness')]
```

3.1.2 Sparse matrices with SciPy

Matrices involved in the PPR algorithm applied to word sense disambiguation are very large (their size is the number of senses in the dictionary), but contain, comparatively, only a little number of non-zero values. For example in the case of WordNet, the matrix is of size $117,659 \times 117,659$. Considering definitions as sentences of around 10 words, each belonging to one or more synsets, the number of non-zero values is of magnitude 10^6, which is very little when the number of cases in the matrix is of magnitude $(10^5)^2 = 10^{10}$.

Fortunately, the SciPy library (Scientific Computing Tools for Python [10]) provides support for sparse matrices, that is, matrices with few non-zero values. I will first present formats for sparse arrays and then some useful functions provided by the library.

- the Linked lists (LIL) format is used to construct the matrix; each row is a list, and the matrix is a list of rows. Entry (i, j) is put in the row of index i and contains the column index j and the value.
- the Compressed Sparse Row (CSR) format is composed of an array of non-zero values, an array of column indices of those values and a list of the indices corresponding to the first value (zero or not) of a row.
- the Compressed Sparse Column (CSC) format is the same as CSR but with an array of row indices and a list of columns indices. Both CSC and CSR are efficient formats for matrix operations.

Following are some useful functions provided by SciPy:

```
>>> from nltk.corpus import wordnet as wn

# creation of a sparse matrix
>>> A = sparse.lil_matrix([[2,0,0],[1,0,1],[0,1,0]])
>>> A[2,2] = 2
>>> A
  <3x3 sparse matrix of type '<class 'numpy.int32'>'
      with 6 stored elements in LInked List format>

# visualization of the matrix
>>> A.toarray()
  array([[2, 0, 0],
         [1, 0, 1],
         [0, 1, 2]], dtype=int32)

# conversion into other formats
>>> A1 = A.tocsr()
>>> A2 = A.tocsc()

# matrix operations
>>> A1 = A1.transpose()
>>> A1.dot(A2).toarray()
  array([[5, 0, 1],
         [0, 1, 2],
         [1, 2, 5]], dtype=int32)
```

3.1.3 Access to web pages

In order to retrieve data from web online dictionaries, we need to extract information in a HTML document. This is done using two Python modules: urllib, more particularly urllib.request, and Beautiful Soup [20].

The first package, provided by default with Python, accesses web pages using their URL and returns the HTML source code of the page.

This code is then parsed using Beautiful Soup, which makes a "soup" out of the HTML document, that is, a structure that allows easy information retrieval. Following are examples of functions to run through a HTML document.

```
>>> from urllib.request import urlopen
>>> from bs4 import BeautifulSoup

# access to a HTML page, gets a string of HTML code
>>> html = urlopen("http://www.manchester.ac.uk/").read()

# makes the soup
>>> soup = BeautifulSoup(html)

# prints 'alt' attributes from 'img' tags
>>> for img in soup.find_all('img'):
        print(img['alt'])
  Menu
  The University of Manchester
  Menu
  Search the University of Manchester site
  ...
```

3.2 Steps of development

3.2.1 Research and familiarisation with the tools

The first months have been spent on literature survey and learning of the tools involved in the development of the software.

The objective was to understand what the issue is, the motivations and difficulties of WSD. Which approach to choose? What are its advantages, its limitations? How to try and mitigate the latter?

The other important part was to familiarize with tools necessary to the development. I never programed in Python before, and I had to learn to use the NLTK and other Python toolboxes. Fortunately, the NLTK provides a book [6] which contains several tutorials for the modules that compose it but also for basic programming in Python; so in parallel to the background research, I completed the exercises proposed in the NLTK book. I also familiarized with other modules such as SciPy and the Beautiful Soup library which would be useful for certain parts of the software.

3.2.2 Personalized PageRank on WordNet definitions

The first part of my project is to apply PPR to a first version of the graph containing only WordNet definitions. It involves the conversion of WordNet into a sparse matrix using NLTK, and then the implementation of the PPR algorithm on Python.

3.2.2.a Conversion of WordNet

The first objective is to create the graph described in Section 2.3.2, on which PPR will be performed. The second objective is to make it into a computable structure, since it will contain 117,659 nodes and millions of edges. A way of representing a graph is to use a matrix A; A is of size N, the number of nodes, and for two nodes i, j, $A_{ij} = 1$ if there is an edge going from j to i, so if the synset j appears in the definition of i. We are going to use the matrix implementation of PPR, so the problem becomes to directly compute the transition matrix of the initial graph.

It's not sufficient to make it computable, however, since we have to compute a matrix of size $117,659 \times 117,659$. However, it will be mostly empty, so we can use sparse arrays to represent it, which is possible with SciPy as seen is Section 3.1.2.

Another important issue is to make the conversion computationally efficient, not only in term of memory but also in term of time, considering a lot of calls to the WordNet corpus will be necessary. Once the sparse matrix is computed, we can save it in order to save time in later uses.

3.2.2.b Implementation of PPR

There are three steps here: create a function that converts a sentence into a personalization vector, apply the PPR algorithm and convert the rank vector into a usable structure.

The computation of the personalization vector is very similar to the conversion of WordNet, so I only need to adapt the implementation of the latter.

The implementation of PPR is the most important step, as it will be used for the whole project, which consists in the improvement of the transition matrix on which PPR is applied. The implementation necessitates several operations on sparse arrays, which can be tricky.

Finally, the conversion of the rank vector uses similar functions as the conversion of WordNet and the input sentence.

3.2.3 Matching of definitions from other sources

The first step is to decide which sources will be used, then to actually retrieve definitions from them. The next steps will be to match them to WordNet definitions and finally to add them to the graph.

At first I will use the definitions of the Macmillan online dictionary. Depending on the advancement of the project, I may try to retrieve information from Wikipedia, as the first sentence of an article is often similar to a general definition of the title term.

In both cases, definition retrieval is performed by accessing the web pages of the dictionary, which is feasible in Python as explained in Section 3.1.3. An important part of the problem is to store the information in a structure that makes the matching easier.

I will then implement the techniques seen in Section 2.4 to match definitions of a word with one another. An important issue is that senses of different sources do not correspond exactly; they may not be as finely-grained, or simply use different distinctions in the senses. The basis of the system are WordNet definitions, so senses from other sources that don't have an equivalent in WordNet will be ignored. On the other hand, senses in WordNet that don't have equivalents in other sources might be disadvantaged compared to other senses in the PPR algorithm (with longer definitions and therefore more edges in the graph). They might have to be removed too, in order to not decrease accuracy. We can imagine that such senses might be very rarely intended and that it wouldn't be too harmful for performance. This issue will necessitate tests to find the best solution.

Finally, this new set of definitions will be converted into a transition matrix, which will necessitate the adaptation of the first conversion function to the new structure.

3.2.4 Personalized PageRank on all definitions

As stated in 3.2.2.b, the previous implementation of PPR is still adapted to the modified transition matrix. Some tests may however be necessary with the parameters of the algorithm (i.e. precision, damping factor and number of iterations).

3.2.5 Improvement of the algorithm

Depending on the time left after the various implementations, I have discussed several ways of improving the algorithm with my supervisor.

3.2.5.a Remove irrelevant words from definitions

Most definitions are complete sentences and therefore present terms that are not related in sense to the word they define. Articles and pronouns should already be removed as they are not included in WordNet, but in fact they can still cause problems, for example "he" can be defined as the symbol for "helium" or a letter of the Hebrew alphabet. Even though, some nouns and mainly verbs are irrelevant in several definitions, and could be removed for better accuracy, or at least weighted down.

3.2.5.b Bootstrapping of the algorithm

In order to improve performance of the algorithm, I will use my program (PPR with all definitions) to try and disambiguate the definitions of the various dictionaries used for the transition matrix.

The disambiguation won't be perfect, and we discussed the possibility of leaving ambiguous terms in the definition depending on the ranks obtained from the PPR: if the rank of the highest ranking term is too close to the others, the disambiguation is not considered reliable enough and the term is left ambiguous.

This first step will remove a certain number of unwanted edges in the transition matrix, improving the accuracy of the algorithm; it can then be used to retry to disambiguate definitions, removing more edges. We can iterate this process several times, with the hope of improving accuracy each time.

3.2.6 Planning

Figure 7 presents the planning given in the initial report, highlighting tasks that have already been performed in green and tasks that I am currently working on in red. The planning has been updated since the initial report, with current progress taken into account and after refinement of the project subject.

3.3 Evaluation of the software

For a long time, evaluation and comparison of WSD algorithms was difficult, as most tests were performed by their developers who had to choose testing data, sense inventories with which to perform evaluation. That's the reason for the organisation of the Senseval competition (now SemEval) in 1998

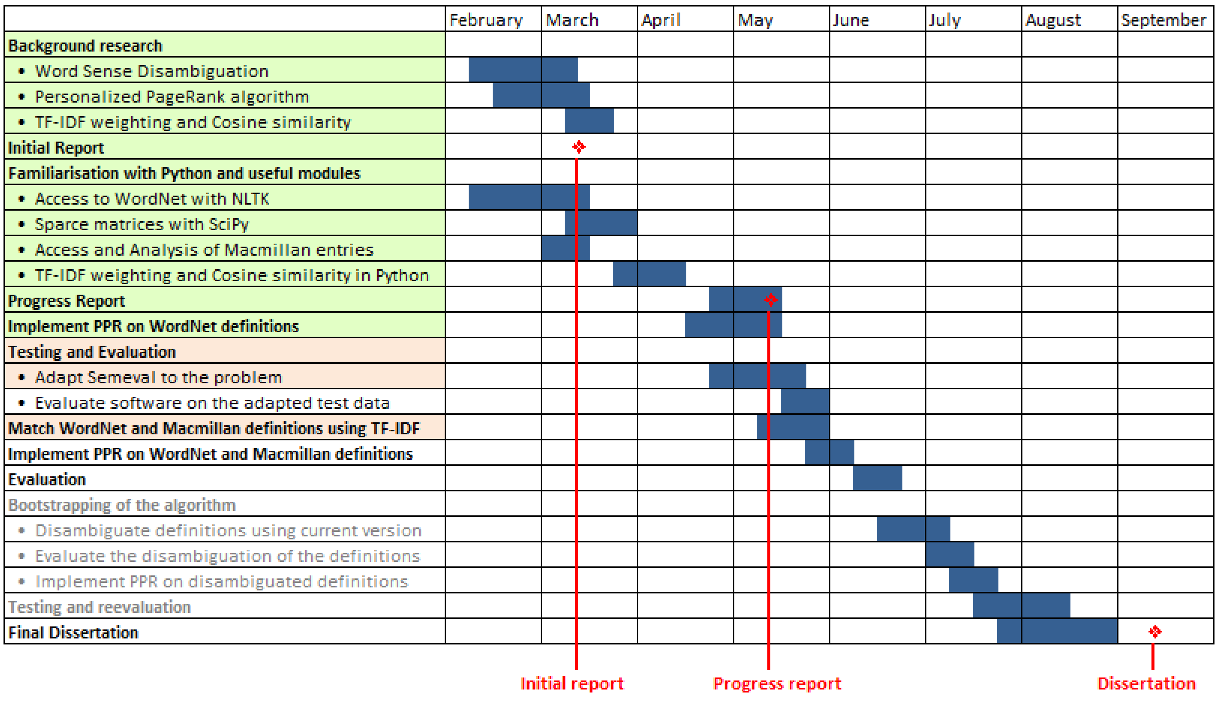


Figure 7: Gantt chart of the project

[11]. Organised every 3 years from 1998 to 2010 and yearly from 2012, it's a series of evaluations for several tasks of semantic analysis, and notably WSD (particularly in the first editions).

The SemEval competition is the reference for observation of state-of-the-art methods in WSD, and the tasks released for each edition are very useful corpora that can be used for evaluation.

I will use those corpora to evaluate the performance of my software. In particular, the Senseval-2 corpus, which uses WordNet definitions, is readily available with NLTK. However, the latest version of the toolbox uses the 3.0 version of WordNet, while Senseval-2 uses the 1.7 version (as it was organised in 2001), with different senses. The solution is either to find a mapping that links senses from both versions of WordNet, or to find a way to access WordNet 1.7 with NLTK and use it as a basis for the algorithm.

The evaluation is to be performed on every step of development: with WordNet definitions, after addition of other dictionaries and after improvement of the algorithm.

4 Project progress

The planning in section 3.2.6 shows which tasks I've already completed for the project. In this section I will present with more details what I have implemented for the disambiguation software.

4.1 WordNet conversion into a graph

I have implemented a Python module that contains all functions relative to the conversion of WordNet into a graph. They use several Python libraries: NLTK, SciPy and Pickle (a library provided by Python to save data structures onto the hard drive). The following functions are implemented:

- $makeGraph()$: the main function of the package, which performs the conversion. It takes no argument and returns the transition matrix (as defined in section 2.3.2) and a dictionary that maps synsets and their indices in the matrix.
- $saveGraph(graph, indices, g_file, i_file)$: a function that saves the graph onto the hard drive in pickle format. It takes the graph, its indices map, and the path to the files in which we want to save them as arguments. It returns no value.
- $loadGraph(g_file, i_file)$: a function that retrieves the transition matrix and its indices dictionary from the files given in arguments. It returns the graph and the indices that have been found in the files.
- $inverseIndices(indices)$: a function that inverses the keys and their values in a dictionary. It's used here to get a dictionary that attaches the synset to each index; it takes a dictionary as argument (here the indices map) and returns the reverse dictionary. The reverse map of indices will be used to read the result of the PPR algorithm

```
# importation of my module
>>> import wordnet2graph as wn2g

# conversion of WordNet into a graph
>>> graph, indices = wn2g.makeGraph()

>>> graph
  <117659x117659 sparse matrix of type '<class 'numpy.float64'>'
      with 6324961 stored elements in Compressed Sparse Column format>

>>> indices
  {'mare_liberum.n.01': 85975, 'choreograph.v.02': 19104, ...}
```

4.2 Personalized PageRank in Python

I then implemented the PPR algorithm for sparse matrices. I used the algorithm described in [14] (only the actual PPR, not the pre-processing of the transition matrix); the result is a Python module containing the following functions, accordingly to Section 3.2.2.b:

- $makePersVector(sentence, indices, size)$: a function that turns a string $sentence$ into a personalization vector, which is a sparse matrix in this implementation. The $size$ argument is the size of the vector, which is the number of sense nodes in the graph (here, for WordNet, $size = 117,659$). It returns the personalization vector for the input sentence.
- $personalizedPageRank(graph, persVector, damping, max\_iter, precision)$: the actual implementation of PPR. $damping$ is the damping factor, $precision$ is the value that determines if the algorithm can terminate at after the current iteration, max_iter is the maximal number of iterations if the $precision$ is never attained (which is not likely). It returns the ranking vector which is the output of PPR, and a sparse matrix here. Default values for $damping$, max_iter and $precision$ are respectively 0.85, 50 and 10^{-12}
- the module also contains some functions to help visualize the results. They all take the ranking vector and the indices dictionary as arguments. For the moment I implemented $wordPageRank$, which also takes a word in input and returns a dictionary that maps all senses of the word to their respective rank in the output of PPR. I also implemented $getDefinition$, which takes a word in input and returns the synset of this word with highest rank and its definition (mostly for human visualization). I may implement other similar functions depended on what will be needed in further development (for example for evaluation using the Senseval corpus).

The following code shows results obtained with the program.

```
# creation of the personalization vector
>>> persVector = ppr.makePersVector('dogs bark', indices, 117659)
>>> persVector
  <1x117659 sparse matrix of type '<class 'numpy.float64'>'
      with 17 stored elements in Compressed Sparse Row format>

# application of PPR with default parameters
>>> rankVector = ppr.personalizedPageRank(graph, persVector)
>>> rankVector
  <117659x1 sparse matrix of type '<class 'numpy.float64'>'
      with 117290 stored elements in Compressed Sparse Row format>

# visualization of the results
>>> ppr.getDefinition('dog', rankVector, indices)
  ('dog.n.01', 'a member of the genus Canis (probably descended ...')
```

Although I have not yet performed a real evaluation with SemEval, I have made some tests and some interesting observations. Whether the disambiguation is correct or not, ranks of senses of a given word are often very close. However, when some words from the definition of the sense are found in the context of the ambiguous word, the algorithm performs far better, with a large gap between the correct sense and the others; the addition of other sources to the graph should therefore allow a better performance

The following examples illustrate the limitations of the current algorithm. The expected result was the synset dog_n^1, that is, dog in the sense of animal species.

- 'dogs' in 'dogs bark' is correctly disambiguated
- 'dogs bark at cats' result in a wrong disambiguation, and in fact gives a sense of 'dog' that seems totally unrelated to the sentence: cad_n^1, defined as 'someone who is morally reprehensible'.
- however, 'dogs bark cats' is correctly disambiguated

Since WordNet doesn't support prepositions, the second and third sentences should give the same results. However, when we look for synsets for 'at', WordNet returns 2 synsets: $astatine_n^1$, the chemical element of symbal At, and at_n^2, a currency from Laos. The word 'at', which shouldn't be considered here, adds two definitions to the personalization vector! It seems therefore very interesting to find a way to remove irrelevant words from the context and definitions.

4.3 Access to Macmillan dictionary definitions

I have implemented a function $getDefinition(word)$ that, for the input word, accesses the corresponding page of the Macmillan online dictionary and analyses the HTML source code to extract the definitions of the word. The definitions are returned as a list of strings, one for each definition. An example of use is shown below.

```
>>> macmillan.getDefinition('dog')
[' an animal kept as a pet, for guarding buildings, or for hunting.
   A young dog is called a puppy', ' someone who is not attractive,
   especially a woman', ' someone who gives information about people
   to the police or to another authority', ' something that is of
   bad quality or very unsuccessful']
```

What I haven't implemented yet is an automatic iteration on WordNet lemmas to extract all definitions of Macmillan. Some tweaks will need to be made for WordNet lemmas composed of several words (like hot_dog), which are separated by a dash in Macmillan's URLs, for words that can belong to different parts of speech, and of course for words that are not present in the Macmillan dictionary. However it shouldn't take long once I find an interesting structure to store the definitions and match them to WordNet.

4.4 Next steps

Once the definitions retrieval is done, the next step in the development will be to match those definitions with WordNet's. This involves the implementation of sentences matching techniques such as seen in Section 2.4.

In parallel, I will adapt the software to Senseval-2 in order to evaluate it. Files from WordNet 3.0 and WordNet 1.7 share the same syntax and file structure so it might be possible to use the files of the 1.7 version to try and evaluate the program. If it doesn't work, I will need to map the senses from WordNet 3.0 to WordNet 1.7 for words used in Senseval-2; however, if I have to do it manually, it will only be possible for the Lexical Sample task (where only a given set of words are sense-tagged), which is less interesting than the All-words task (where all words are sense-tagged).

I will finally be able to apply PPR to the graph of all definitions, and evaluate it, before trying to improve the software with the approaches seen in Section 3.2.5.

References

[1] Palakorn Achananuparp, Xiaohua Hu, and Xiajiong Shen. The evaluation of Sentence Similarity measures. In *Data Warehousing and Knowledge Discovery*, pages 305–316. Springer, 2008.

[2] Eneko Agirre, Oier Lopez de Lacalle, and Aitor Soroa. Random walks for knowledge-based Word Sense Disambiguation. *Computational Linguistics*, 40(1):57–84, 2014.

[3] Eneko Agirre, Oier Lopez De Lacalle, Aitor Soroa, and Informatika Fakultatea. Knowledge-based WSD and specific domains: Performing better than generic supervised WSD. In *IJCAI*, pages 1501–1506. Citeseer, 2009.

[4] Eneko Agirre and Aitor Soroa. Using the multilingual central repository for graph-based Word Sense Disambiguation. In *LREC*, 2008.

[5] Eneko Agirre and Aitor Soroa. Personalizing PageRank for Word Sense Disambiguation. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*, pages 33–41. Association for Computational Linguistics, 2009.

[6] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python*. " O'Reilly Media, Inc.", 2009.

[7] Gerard Escudero, Lluís Màrquez, and German Rigau. Naive Bayes and exemplar-based approaches to Word Sense Disambiguation revisited. *arXiv preprint cs/0007011*, 2000.

[8] Christiane Fellbaum. *WordNet*. Wiley Online Library, 1998.

[9] Nancy Ide and Jean Véronis. Introduction to the special issue on Word Sense Disambiguation: the state of the art. *Computational linguistics*, 24(1):2–40, 1998.

[10] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[11] Adam Kilgarriff. Senseval: An exercise in evaluating Word Sense Disambiguation programs. In *Proc. of the first international conference on language resources and evaluation*, pages 581–588. Citeseer, 1998.

[12] Yoong Keok Lee, Hwee Tou Ng, and Tee Kiah Chia. Supervised Word Sense Disambiguation with support vector machines and multiple knowledge sources. In *Senseval-3: third international workshop on the evaluation of systems for the semantic analysis of text*, pages 137–140, 2004.

[13] Michael Lesk. Automatic Sense Disambiguation using machine readable dictionaries: how to tell a pine cone from an ice cream cone. In *Proceedings of the 5th annual international conference on Systems documentation*, pages 24–26. ACM, 1986.

[14] Takanori Maehara, Takuya Akiba, Yoichi Iwata, and Ken-ichi Kawarabayashi. Computing Personalized PageRank quickly by exploiting graph structures. *Proceedings of the VLDB Endowment*, 7(12), 2014.

[15] George A Miller, Claudia Leacock, Randee Tengi, and Ross T Bunker. A semantic concordance. In *Proceedings of the workshop on Human Language Technology*, pages 303–308. Association for Computational Linguistics, 1993.

[16] Mohammad Nasiruddin. A state of the art of Word Sense Induction: A way towards Word Sense Disambiguation for under-resourced languages. In *Proceedings of RECITAL 2013*, pages 192–205. ATALA, 2013.

[17] Roberto Navigli. Word Sense Disambiguation: A survey. *ACM Computing Surveys (CSUR)*, 41(2):10, 2009.

[18] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. 1999.

[19] Diana Raileanu, Paul Buitelaar, Spela Vintar, and Jörg Bay. Evaluation corpora for Sense Disambiguation in the medical domain. In *LREC*, 2002.

[20] Leonard Richardson. Beautiful Soup documentation, 2007.

[21] Warren Weaver. Translation. *Machine translation of languages*, 14:15–23, 1955.