UNIUNEA EUROPEANĂ
GUVERNUL ROMÂNIEI MINISTERUL MUNCII, FAMILIEI ŞI PROTECŢIEI SOCIALE AMPOSDRU
Fondul Social European POSDRU 2007-2013
Instrumente Structurale 2007-2013
OIPOSDRU
UNIVERSITATEA "POLITEHNICA" din BUCURESTI

FONDUL SOCIAL EUROPEAN
Investeşte în oameni!
Programul Operaţional Sectorial pentru Dezvoltarea Resurselor Umane 2007 – 2013
Proiect POSDRU/6/1.5/S/19 – *Pregatirea competitiva a doctoranzilor in domenii prioritare ale societatii bazate pe cunoastere*

POLITEHNICA UNIVERSITY OF BUCHAREST, ROMANIA
Faculty of Automatic Control and Computers
Department of Computer Science and Engineering
UNIVERISTE PAUL SABATIER, TOULOUSE, FRANCE
Institut de Recherche en Informatique de Toulouse
Systemes d'Information Generalisés

Senate Decision No. 212 / 30 September 2011

DOCTORAL THESIS

Extragere de informaţii din surse de date nestructurate şi semi-structurate

Information extraction from unstructured and semi-structured sources

Author: Eng. Stefan Daniel DUMITRESCU

DOCTORAL COMMITTEE

President	Dumitru POPESCU	of	Politehnica University of Bucharest
Coordinator-1	Stefan TRAUSAN-MATU	of	Politehnica University of Bucharest
Coordinator-2	Florence SEDES	of	Université Paul Sabatier
Referent	Luca Dan SERBANATI	of	Politehnica University of Bucharest
Referent	Ion SMEUREANU	of	Academy of Economic Studies

Bucharest, November 2011

Contents

List of figures ... iii

List of tables ... iv

I. Introduction ... 1

II. Information Extraction related tools, methods and techniques ... 6

II.1. Text pre-processing ... 6

II.1.1. Tokenization and sentence splitting ... 6

II.1.2. Stop words ... 7

II.1.3. Stemming ... 8

II.1.4. Part-of-Speech Tagging ... 9

II.2. Machine learning approach and tools ... 10

II.2.1. Support Vector Machines ... 11

II.2.2. Conditional Random Fields – Linear-chain CRF ... 12

II.3. Parsers ... 14

II.4. Coreference resolution ... 16

II.5. General purpose corpora ... 18

III. Knowledge acquisition and representation ... 20

III.1. Ontologies as information repositories ... 20

III.1.1. RDF – Resource Description Framework ... 21

III.1.2. WordNet ... 23

III.1.3. YAGO ... 27

III.2. Information Extraction ... 28

III.2.1. Open IE – TextRunner ... 30

III.2.2. Canonic fact extraction – SOFIE ... 33

IV. Entity recognition and word sense disambiguation for Information Extraction ... 36

IV.1. Word Sense Disambiguation ... 36

IV.1.1. Supervised Disambiguation ... 38

IV.1.2. Unsupervised Disambiguation ... 43

IV.1.3. Knowledge-Based Disambiguation ... 46

IV.1.4. WSD Bounds ... 51

IV.1.5. Evaluation metrics ... 52

IV.2. Named Entity Recognition ... 53

IV.2.1. Classification of NER Approaches 54
IV.2.2. Named Entity detection and recognition techniques..........57
IV.2.3. Evaluation Metrics 59
IV.3. General Named Entity Recognition..........60
V. A General Entity Recognition (GER) System..........63
V.1. System overview..........65
V.1.1. Architecture..........66
V.2. Formalization..........67
V.3. Proposed custom graph algorithm – Linker Algorithm..........70
V.3.1. Description 71
V.3.2. Complexity analysis 82
V.3.3. Experiments..........86
V.4. Integrating the Linker Algorithm into the General Entity Recognition System...94
V.4.1. Module A - NLP Module 95
V.4.2. Module B - String Entity Processor Module 96
V.4.3. Module C - Canonic Entity Processor Module..........101
V.5. System evaluation 109
V.5.1. Evaluation methodology..........109
V.5.2. Evaluation set and standard creation 110
V.5.3. Testing the system 114
V.6. Conclusions..........116
VI. A knowledge-based approach for document classification..........118
VI.1. Introduction 118
VI.2. Domain Literature Review 118
VI.3. System Implementation 119
VI.3.1. Topic list creation..........120
VI.3.2. Processor Module..........122
VI.3.3. Analysis Module 126
VI.3.4. Evaluator Module..........132
VI.4. Evaluation..........133
VI.5. Conclusion 135
VII. Conclusions 137
VIII. References..........144
Appendix 154
Example run of the proposed GER System 154

List of figures

Figure 1. SVM, boundary 11
Figure 2. Two equivalent notations of dependency trees 15
Figure 3. ENJU deep parser visual example (constituency tree) for the sentence ‘I see what I eat’ 16
Figure 4. Example of synset partial hypernym graph 25
Figure 5. TextRunner architecture 31
Figure 6. A Hidden Markov Model example 55
Figure 7. Logical architecture of the proposed system 67
Figure 8. Example of the graph decision problem 72
Figure 9. Abstraction of the 3-partite graph in figure 8 72
Figure 10. Pseudocode for CDFS() function 74
Figure 11. Pseudocode for addSolution() function 76
Figure 12. Pseudocode for isSubSetOrEqual() function 76
Figure 13. Pseudocode for canMerge() function 76
Figure 14. Pseudocode for the computeScores() function 77
Figure 15. Example Result Set Array 78
Figure 16. Pseudocode for mergeNonOverlappingRSA() function 79
Figure 17. Pseudocode for createSolutionTree() function 80
Figure 18. Worst case scenario tree construction for step 4 merging function 83
Figure 19. Result Set size variation on complete graph 87
Figure 20. Algorithm step 2 time variation for complete graph 88
Figure 21. Algorithm step 3 time variation for complete graph 88
Figure 22. Time measurement when varying the number of edges 89
Figure 23. Result Set Array size when varying the number of edges 89
Figure 24. Dependency tree example 100
Figure 25. Connections found between String Entities 100
Figure 26. Influence Matrix example 101
Figure 27. Pseudocode for obtainInputGraph() function 106
Figure 28. Pseudocode for getEdgeWeight() function 106
Figure 29. Operational graph 107
Figure 30. Input graph derived from the operational graph 108
Figure 31. Document classification system architecture 119

List of tables

Table 1. Different semantic similarity measures 48
Table 2. Example of tree creation for merging non-overlapped Result Sets 80
Table 3. Algorithm input parameters average grouped by document category 91
Table 4. Algorithm average run-time grouped by document category 93
Table 5. Example of classes YAGO returns for the query about “engine” 97
Table 6. Example table showing the score percentage assigned to each topic based on its supporter count 130
Table 7. Comparison between the proposed system and a standard SVM state-of-the-art method 133
Table 8. A short comparison between overall system performance grouped by topic, before and after topic tweaking, for 5 out of the 50 total topics 134

I. Introduction

We live in an age of information. Everywhere around us information is embedded in the devices we wear, the tools we use, the media we watch or hear, in the workplace and at home. New information is added every second to the interconnected web of devices that is the Internet.

This exponential data explosion brings with it the availability of knowledge to end-users, the ability to interconnect, share, create and develop ideas and business together. However, this explosion also brings what is known as Information Overload, where humans are bombarded with too much information that actually makes them less productive. The volume of scientific knowledge has outpaced our ability to manage it. The continuous data addition without structure and tools to extract what is relevant leads to data intractability problems.

There are a number of solutions proposed, some proven, some in development and some only in concept that try to solve this issue. The biggest problem is information classification and retrieval. A user is spending too much time trying to ask a search engine the correct question that will minimize the list of results returned and then to manually crawl the different web pages until he or she finds the desired information.

An obvious solution is better content classification in directories and libraries. However, standard classification can only go so far, as the user is still required to manually sift through web pages (even if through a smaller number) to find the information. The response to a user query should be an actual *answer* not a list of decreasingly relevant documents. The solution to this problem is to *involve the compute*r in the search effort, which means that the computer needs *to understand* what it is searching and how it relates to the sea of information available to it. Semantic technologies provide a way for computers to understand the data they process.

However, for computers to be able to work on such datasets, they must first be extracted from the current web. Information Extraction (IE) is the task of extracting knowledge from text, usually in the form of facts (two entities that stand in a relation). This representation of data is relational in nature, creating complex graphs of relations between entities. Using this type of knowledge representation, computers can then answer user queries with actual facts instead of web pages.

The Web is a vast source of information. At the time of writing, a rough estimate of the size of the web reveals that there are over 15 billion indexed pages[1], with thousands more being indexed every day. Information is added in digital form on an ever-increasing rate, as more

[1] http://www.worldwidewebsize.com/

and more people connect to the Web and as the Web itself creates a framework of tools and means to add more diverse, heterogeneous data faster.

At first sight, this information appears to be available to anyone and everyone, and, in many cases, it is, thanks to search engines. Users input their queries in the form of keywords they consider relevant and are presented with a list of websites that should contain the queried information. Most queries however are 2 to 3 words in length, allowing search engines to present from the millions of possible sources a sorted list of ten supposing decreasingly relevant sites per page. This is (arguably) sufficient for common information queries like "*Ford Focus review*" or "*seo optimization*" ("*seo*" is short for *s*earch *e*ngine *o*ptimization), but vastly inefficient for more specific queries like "*windshield wipers size for Hyundai Accent 2003*".

There is no person that has searched for something on the Internet at some point and has not failed to find it, even with the power of an indexed web of billions of pages to choose from. Professional needs, local issues, comparative queries, simple or more complex questions (formulated as such) and so on, can be poorly expressed by a set of keywords. Even if there might be a way to express such queries in terms of just keywords, the semantic links between the words are lost as the query is seen as a bag of words, without order. This will result in poor results, with many false positive hits, displaying websites that contain as most of the keywords as possible, but do not address the query itself. For example, questions like "*which are the current presidents of EU?*", "*Japan's prime minister before Naoto Kan*", "*highest score of a B-division team in 1980 season in Poland*" or even more complex inquiries stated by simple queries like "*cities in which both Scorpions and Stones played*" or "*medicine that can be taken without interfering with medicine X in flu cases*" will not be solved by current standard search engines.

For such queries to be considered, there first has to be an understanding of the meaning of such queries (1st problem), and then the result of such queries should be an actual answer rather than a list of websites that contain such information (2nd problem). These problems raise the bar to another level of difficulty. The search is now a search for knowledge instead of a search for data [1].

Semantic Web

The Semantic Web idea is relatively new, being mentioned for the first time in 1998 by Tim Berners-Lee[2]. The Semantic Web translates roughly as a web of Meaning, a web where computers can understand the meaning of information. The focus shifts from links between web pages to links between entities, or better said, relationships between entities and entity properties. The current web is centered on the presentation of information while the Semantic Web centers on knowledge and its representation model.

[2] "Semantic Web Road map", http://www.w3.org/DesignIssues/Semantic.html

There is no single standard format to model the Semantic Web. The entities, relationships and properties can be represented in any of the available maturing or newly developed formats. The basic model is RDF – Resource Description Framework[3] with its associated RDF Schema (RDFS) and its notations like XML format, N3, Turtle, etc (presented in section III.1.1). More advanced models include OWL – Web Ontology Language, currently at its second version OWL2[4].

Even though the Semantic Web promises a revolution, this revolution will come at a slow pace. One of the major problems is that all the models and tools above do require knowledge to develop and use. Even the basic RDF standard was created by people with academic background, and this means that there is a learning curve to be climbed in order to use the Semantic Web tools at their true potential. This is why, as opposed to the explosive growth of the standard web where anybody can publish anything without requiring any special knowledge, the Semantic Web will grow slowly.

A main research direction focuses now on how to create the necessary standards that are versatile enough and do not require advanced training to use, and, possibly even more important, to create the tools that will make the Semantic Web as easy to access and develop as possible.

While this is a very difficult task, steps have been taken in the right direction. For example, Microsoft's EntityCube[5] gathers facts about named entities (people, publications, organizations, places, etc.); WolframAlpha[6] (computational knowledge engine) links together domain databases and is able to understand and process queries like "*next solar eclipse*" an present a tabular format with results and analysis, it can analyze an electronic circuit from the query "*RLC circuit 10ohm, 12H, 400uF*" and many others; Google's Squared[7] attempts to provide a table with entities as rows and columns as attributes in response to queries.

Freebase[8] and DBpedia[9] are two large, free sources of information. Freebase data may be viewed and edited by anyone and DBpedia dataset can be freely downloaded. Freebase provides a user friendly interface so that people can define types and relations, and they can add and search data. DBpedia also provides online access but using a SPARQL[10] (an RDF query language) endpoint. Both also provide data in RDF format.

In early 2010 the DBpedia data set describes 3.4 million entities with over 1 billion facts while Freebase contains about 12 million topics. One notable community effort is the

[3] RDF: http://www.w3.org/TR/rdf-schema/, W3C recommendation on 10 February 2004
[4] Approved W3C Recommendation on 27 Oct 2009, http://www.w3.org/TR/owl2-overview/
[5] http://entitycube.research.microsoft.com/
[6] http://www.wolframalpha.com
[7] http://www.google.com/squared/
[8] http://www.freebase.com/
[9] http://dbpedia.org/
[10] http://www.w3.org/TR/rdf-sparql-query/, W3C recommendation on 15 January 2008

mapping between these two ontologies – at present 2.4 million RDF links have been added to DBpedia pointing at the corresponding entities in Freebase. There is a noticeable, though slow, momentum gathering towards these new technologies.

All of the above systems rely on some form of knowledge database and internal knowledge representation format. Due to the size of the task at hand, almost all of the above systems (excepting community efforts) use some type of tool to extract information from a source and then convert it to its representation format.

In the present work we investigate such tools required to create large knowledge repositories – we look at the **Information Extraction** (**IE**) field. Information Extraction is a type of Information Retrieval that focuses on extracting structured information from unstructured (free, natural language text) and semi-structured sources (xml, html documents, etc). The extracted information needs to be in a structured format so that it can be machine-readable by computers. Structured format has many forms, but the most basic type is the "fact" or "triple" containing a subject, an object, and a predicate/relation that links the subject to the object. For example, the natural language statement "Ann is Mary's daughter" can be expressed as (`Mary`, *`hasDaughter`*, `Ann`). This simple example illustrates the need to identify words and detect the existing relations between them. As such, the field of Information Extraction is split into several tasks, like entity recognition, relationship extraction, coreference resolution, etc.

In this thesis we focus especially on the task of entity recognition, which is to identify words as candidate entities and recognize them in the context of a reference dictionary (more specifically in the form of an ontology). We consider named entities as well as common nouns. For named entities the task is to determine initially their type (whether they are persons, locations, organizations, etc – the Named Entity Recognition task), or the more difficult attempt to uniquely identify them directly in the reference source (detect more specific instances like city, country, region, etc, not just simple location). For common words the task at hand is to identify their senses, considering that words are polysemous (the task of Word Sense Disambiguation). This identification step (both for named and common entities) is essential for every Information Extraction system, as it usually provides a first stepping stone on which to perform more advanced text processing. While seemingly simple at a first glance, entity identification (with its two sub-tasks: NER and WSD) is very difficult, as shown by the many attempts over the past two decades summed up in specialized conferences and workshops like the Message Understanding Conferences, Sens/SemEval Workshops[11], CoNNL[12], CLEF[13], EACL[14] and other important events.

[11] http://www.senseval.org/
[12] Conference on Computational Natural Language Learning, http://ifarm.nl/signll/conll/
[13] Cross-Language Evaluation Forum, http://clef-campaign.org/
[14] European Association for Computational Linguistics, http://www.eacl.org

As a summary of the contents of the thesis, we will start from the basics, investigating the algorithms, models, tools, the sub-tasks required for any Information Extraction system. We then look at existing state-of-the-art IE systems like TextRunner[15] and SOFIE[16]. To be able to create knowledge bases in which to store the information harvested by such systems a representation method is needed. Thus, the thesis investigates the usefulness of ontologies in the field of IE by implementing two knowledge-based systems that rely almost exclusively on unsupervised methods and large, general ontologies.

The first system implemented is designed to perform entity detection and recognition starting from natural language texts. The approach taken here unifies two major problems of IE - Word Sense Disambiguation and Named Entity Recognition into a single task - General Entity Recognition.

The second implemented system is designed to perform text classification. This system also uses a general, large ontology and a custom semantic distance function to assign scores to topics and topic concepts based on the ontology graph, and then rank them according to each topic's relevance to the analyzed document.

The thesis closes with conclusions on the implemented systems: benefits and disadvantages of the approach taken, implementation issues and their performance.

[15] http://www.cs.washington.edu/research/textrunner/
[16] http://www.mpi-inf.mpg.de/yago-naga/

II. Information Extraction related tools, methods and techniques

This chapter describes some of the necessary and/or basic tools and techniques needed to perform any Information Extraction related task. Some of the methods presented are actually basic tasks that must be performed before any other major task, and are not exclusively located in the Information Extraction sub-domain, but are used in larger domains such as NLP (Natural Language Processing), IR (Information Retrieval), Machine Learning and others.

The first section describes text pre-processing tasks like tokenization or stemming. Then, two machine learning algorithms are presented as they are essential in IE. Parsers are then presented as a tool for IE, in which are used to analyze sentences syntactically. After that, Coreference Resolution is presented as an important task to be done that can sensibly improve the entity extraction task. Last but not least, annotated generic English corpora are presented as a basis for many tasks and subtasks, such as training POS Taggers or Parsers, extracting Information Content values for words and concepts for Word Sense Disambiguation tasks, etc.

II.1. Text pre-processing

Text pre-processing is usually the first step that has to be done in NLP related tasks. The original text has different processing algorithms applied to it, in order to extract (or annotate) needed information about the text, portions of text or individual words.

II.1.1. Tokenization and sentence splitting

Tokenization is the process of splitting a text into individual words, phrases, symbols, or other meaningful elements called tokens. It is usually the first step applied in any NLP task, as it outputs a list of separated tokens that normally are fed into further pre-processing tasks or directly into major NLP tasks.

Tokenization, even though at first sight seems a simple matter of splitting words, quickly becomes problematic for a sentence like “Mr. John Little (b. 1974), C.T.O. of Apala Labs (05.2005-11.2008) … “. To begin with, there is a punctuation dot just after the first token “Mr”. In this case, the tokenizer must include the dot with the token, as opposed to treat the dot individually as a separated token like it should do with sentence ending dots (sentence stop). The opening parenthesis should be an individual token, but the dot after “b” should be included in “b.”. Then, “C.T.O.” is yet another token, this time containing three punctuation marks. An even more difficult problem arises on dates: should the tokenizer split the date into “05”, “.”, “2005” or “05.2005”? As can be seen, tokenization becomes a more difficult problem, accentuated by the fact that an error in this initial step will be very costly to detect and correct in later processing stages.

Currently there are several types of tokenization. The simplest way is to use whitespace/punctuation splitting at the cost of very poor performance. More advanced ways involve lists of regular expressions providing better performance (which do have the ability of detecting specific patterns if needed in a domain-text for example). The most successful way at present is to use statistical/machine learning models such Maximum Entropy or Hidden Markov Model that are given a tokenized training corpus and learn the language model on it.

The same issues apply on sentence splitting. Usually sentences end with a punctuation sign, either a dot, an exclamation or question mark, three dots, etc. However, there are several punctuation signs inside the sentence itself, and the sentence splitter should recognize them. The same statistical/machine learning models are also used to split sentences in a text with good success rates. Even more, to give the tokenizers the clearest input text possible, first sentence splitting is performed, and then individual sentences are given to the tokenizers. This ensures that the tokenizer does not get confused with sentence boundaries.

II.1.2. Stop words

Stop words are list of common words that do not hold value in a shallow NLP analysis, and as such are filtered out in initial stages of text pre-processing. The term “stop word” has been first attributed to Hans Peter Luhn for describing the removal of useless words in Information Retrieval tasks.

Lists of stop words are freely available, and can contain anywhere from tens of words to hundreds of words. Some lists are domain-related. For example, a general list would contain words such as “*a*”, “*by*”, “*the*” or “*she*”, while a chat-oriented list would contain words/strings such as “*:)*”, “*brb*” (be right back), “*gtg*” (got to go) or “*lol*” (laughing out loud).

Stop word removal in some applications is actually harmful, for example in tools that support phrase searching where removal of some linking words would lead a search engine to skip valid results.

II.1.3. Stemming

Stemming is the process of reducing derived words to their root form. It is a basic task that is usually performed at the beginning of most NLP problems, Information Retrieval, etc.

Currently there are many different approaches to stemming, with various performances:

Suffix Stripping Algorithms use a list of rules to transform an inflected word into its root. For example, the rule "if the word ends with '*ing*' → remove '*ing*'" will transform "*flying*" in "*fly*". This algorithm class provides performance as good as the linguist which programs the rules. However, there are many exceptional cases that must be hand-coded. Suffix stripping algorithms have average performance.

Lemmatization Algorithms start by determining the part of speech of the word, to try and apply different rules depending on that part of speech. This approach does depend greatly on the accuracy of the part of speech identification.

Brute Force Algorithms use a simple mapping between root forms and inflected forms. The process is very quick; a simple lookup of an inflected word will return its stem. However, for such a mapping to be held on a host machine, a huge amount of memory (or other storage form) would be needed. Also, if the inflected form does not exist in the table, no result will be given. Counting the number of words in the English language, it is unlikely the manual filling if such a mapping will ever be completed; even automatic filling with human supervision is too time consuming and the accuracy increase is minimal. On the other hand, if every inflected form would be input in such a mapping, the stemming accuracy would be 100%.

Stochastic Algorithms use probabilities to determine the word's root. This class uses machine learning algorithms that are trained on existing mappings between inflected and root forms of words. Stochastic algorithms try to achieve the highest probability of correctness of the stemmed word, internally using somewhat similar rules like the suffix stripping and lemmatization algorithms.

An improvement that can be made to stochastic algorithms is to consider the context in which each inflected word is found. This can be done by considering the words next to the inflected word – n-grams. An n-gram is a sequence of entities (most often words but can be syllables or characters) of size n. If n is 1, then we have unigrams. If n is 2 we have bigrams (most used), for n equals 3 – trigrams, and so on. The improved stochastic algorithm can look at the words preceding the inflected word (the preceding words are called qualifiers for the last word) to determine its sense, part of speech, if it is already stemmed, what stem is more appropriate, whether to strip or substitute suffixes, and so on, based on probabilities.

While n-gram analysis increases accuracy by a varying margin, it is argued that the programming effort involved and even more the training-retraining requirements of the model make it hard to maintain.

Hybrid Approaches to stemming mean using at least two existing techniques combined. The techniques can work in parallel or can be used in sequence: for example, a hybrid algorithm can first try a brute-force method that has mapped only exceptions; if the word is not found in the mapping, then the algorithm falls back on a standard suffix stripper.

In the present work the Porter Stemmer [2] is used for stemming support.

II.1.4. Part-of-Speech Tagging

Part-of-Speech Tagging (or POST for short) means identifying the part of speech that corresponds to a given word. POST needs to take into account the context (connected or related words in the same sentence or paragraph) of the word, considering that the same word in different contexts belongs to different parts of speech.

There is no standard list of parts of speech; there are 9 basic categories in English: noun, verb, adverb, pronoun, article, adjective, preposition, conjunction, interjection. There are however many more sub-categories. We can identify a noun as being a named or a common noun, being possessive, accusative, having a number, being animate or not, and so on. This typically increases the number of distinct parts of speech to above 100, different for every implementation of POS Tagger.

POST is useful in Information Retrieval, Text to Speech (for example the word object can be either a verb or a noun, depending on the accentuation: ***ob***ject(N) vs. ob***ject***(V)), Word Sense Disambiguation and other more complex tasks.

Algorithmically, there are many types of taggers developed. For example, there are Rule-Based POS taggers [3], Transformation-based taggers (Brill's tagger [4]), Stochastic (Probabilistic) taggers [5]. The best accuracy is obtained by stochastic tagging algorithms.

Supervised taggers use machine learning techniques to assign predefined classes to words. Most often Hidden Markov Models are used. For any chosen model, they need to be trained on an existing, pre-tagged corpus before being applied in practice (for example, a general-purpose, POS tagged corpus used for training is the Brown corpus[17]). After training, a table of probabilities is generated, based on part of speech sequences. For example, the word 'the' is in most cases followed by either a noun or an adjective, and never by a verb. Higher order models can estimate probabilities of entire sequences, not only pairs of words. Supervised taggers usually achieve around 95% accuracy.

[17] http://khnt.aksis.uib.no/icame/manuals/brown/

Unsupervised taggers use untagged corpora to derive probability tables. Such taggers extract similar patterns of words (based on a preset metric or other discriminative criteria) and infer part of speech categories for them. One notable example for this category is the Brill Tagger [6].

In the present work Stanford's POS Tagger [7] is used. It is a hybrid supervised tagger, using both preceding and following tag contexts via a dependency network representation and using lexical features like jointly conditioning on multiple consecutive words and modeling of unknown word features.

II.2. Machine learning approach and tools

Machine learning algorithms can determine by themselves an output, path of action or result when given an input, based on previous supervised or unsupervised training. Machine learning is currently a scientific domain in itself.

"*A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*" [8]

The purpose of machine learning is to learn to act intelligently upon complex inputs. Being impossible to map every single possible output course of action, machine learning algorithms must learn to generalize for them to produce an output dependent on any given input.

While there are many sub-classes and categories of machine learning algorithms, two major categories stand out:

- **Supervised Learning:** Decision trees, nearest neighbors, linear classifiers and kernels, neural networks, linear regression, bagging and boosting, feature selection.

  Supervised learning means that training is done on "supervised" data, data that has previously been annotated with the result the algorithm should output. After training, the resulting regression function (if the output is continuous) / classifier function (if the output is discrete) should provide reasonable output given any input data.

- **Unsupervised Learning:** Clustering, graphical models, EM, factor analysis, manifold learning.

  Unsupervised learning means giving unlabeled training data to the algorithm to find interesting patterns, identify classes of related data, etc. Based on this training, the algorithm can then process test data, and cluster / assign a new input as belonging to the most similar class determined in the training phase.

For the rest of this section we will investigate the Support Vector Machines (supervised algorithm) and the Conditional Random Fields (unsupervised algorithm).

II.2.1. Support Vector Machines

Support Vector Machines (SVMs) are a set of supervised learning methods used for regression and classification.

A SVM tries to obtain the optimal separation boundary of two distinct sets in a multidimensional vector space, independently on the probabilistic distributions of training vectors in the sets. The task is to locate the boundary that is most distant from the vectors nearest to the boundary in both of the sets. For nonlinear boundaries, the introduction of a kernel method is equivalent to a transformation of the vector space.

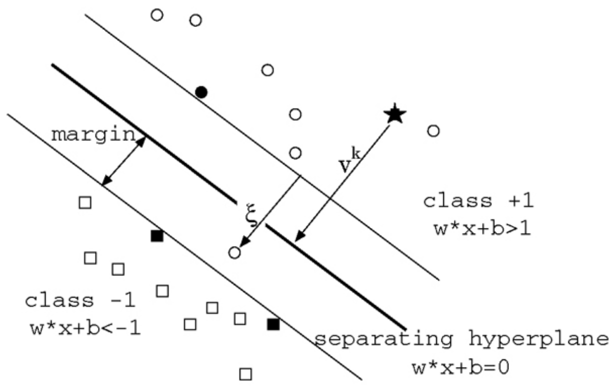


Figure 1. SVM, boundary

The task of this class of algorithms is to detect and exploit complex patterns in data. Typical problems are how to represent complex patterns (computational problem) and how to exclude unstable patterns (statistical problem).

Input is given in the form of data instances. Each instance is an *n*-dimensional vector.

$$D = \{(x_i, c_i) \mid x_i \in R^n, c_i \in \{-1,1\}\}_{i=1}^{m} \tag{1}$$

D is the training set, x_i is the *i*-th dimensional value and c_i is the class that the vector belongs to. The *n-1* dimensional hyperplane that best divides the data instances is expressed as:

$$w^T x + b = 0 \tag{2}$$

where *w* is the weight coefficient vector and *b* is the bias term. The *margin* is the distance between a training vector xi and the boundary:

$$\frac{|w^T x_i + b|}{||w||} \tag{3}$$

Introducing a restriction to this expression, we have:

$$min_i|w^T x_i + b| = 1 \tag{4}$$

The optimal boundary maximizes the minimum of (3). Considering (4), minimization for:

$$c_i(w^T x + b) \geq 1 \tag{5}$$

The optimization is done using Lagrange's indeterminate coefficient method. Given:

$$L(w, b, \alpha_i) = \frac{1}{2} w^T w - \sum_i \alpha_i [y_i(w^T x_i + b) - 1] \tag{6}$$

where $\alpha_i \geq 0$ are indeterminate coefficients. The partial derivatives are zero if w and b take optimal values:

$$\frac{\partial L}{\partial w} = w - \sum_i \alpha_i y_i x_i \quad and \quad \frac{\partial L}{\partial w} = -\sum_i \alpha_i y_i \tag{7}$$

Setting the derivatives to zero, extracting w, rewriting and substituting:

$$L(w, b, \alpha_i) = -\frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j x_i^T x_j + \sum_i \alpha_i \tag{8}$$

The problem is reduced to a quadratic programming problem, maximizing the right-hand side term while considering that the sum $(\alpha_i y_i)$ equals 0, for all $\alpha_i \geq 0$.

The quadratic problem is well known and applications exist that can solve it efficiently. Another note to make is that SVM splits the data into two classes. If we have more than two output classes, then a more complex technique is applied, running the model iteratively.

The SVM algorithm is implemented in several free and commercial packages, such as WEKA[18] , LIBSVM[19], SVM-Light[20] and others. Because of its versatility and ease of usage, in the present work WEKA [9] will be used for SVM support.

II.2.2. Conditional Random Fields – Linear-chain CRF

The problem of labeling sequences to a set of observations is often found in many NLP tasks (ex: labeling words in sentences with parts of speech, labeling words with chunk identifiers, etc.) Usually, either Hidden Markov Models (HMMs) [10] or finite-state machines are used.

Hidden Markov Models define a joint probability distribution p(*X*,*Y*) where *X* is a random variable ranging over observation sequences while *Y* ranges over the label sequences. This form of generative model cannot iterate over all possible observation sequences (as it needs

[18] http://www.cs.waikato.ac.nz/ml/weka/
[19] http://www.csie.ntu.edu.tw/~cjlin/libsvm/
[20] http://svmlight.joachims.org/

to) if independence assumptions are not made beforehand. Usually, it is assumed that an observation depends only on the label at a point in time, even though in reality observations depend on multiple levels and even depend on previous labels in time. A generative model like the HMM directly describes how the inputs are 'generated' by the outputs probabilistically.

The answer to this problem is to create a model that defines the conditional probability as p(Ylx) over label sequences given a certain observation sequence (x) instead of a joint distribution over observation and label sequences. Another way of saying is that while HMMs are generative models (modeling p(Y,X)), discriminative models like CRFs model p(YlX) that does not need to model p(X) which is usually intractable if it contains many highly dependent features. The main advantage of discriminative modeling is that it is better adapted to include many overlapping features.

A Conditional Random Field (CRF) is a discriminative model, represented as an undirected graphical model where nodes are random variables and the links between nodes are dependencies. A CRF defines a log-linear distribution over label sequences for a certain observation sequence [11]. Linear-chain CRFs relate to HMMs in that they are used for many of the same problems, but are more permissive about input and output assumptions. A CRF can be viewed as a HMM with general feature function that do not use necessarily constant probabilities to model the emissions and transitions. Furthermore, CRFs can contain an arbitrary number of feature functions and these feature functions may not be defined as probabilities.

If we start from a HMM, introducing feature functions for more compact notation and considering the need of one feature function f_{ij} for each (i, j) transition and one feature function f_{io} for each state-observation pair (i, o), then the HMM can be written as:

$$p(y,x) = \frac{1}{Z}\exp\left\{\sum_{k=1}^{K} \lambda_k f_k(y_t, y_{t-1}, x_t)\right\} \tag{9}$$

and considering that the joint probability of a state sequence y and an observation sequence x can be written as follows:

$$p(y,x) = \prod_{t=1}^{T} p(y_t|y_{t-1})p(x_t|y_t) \tag{10}$$

then the HMM can be written as:

$$p(y|x) = \frac{p(y,x)}{\sum_{y'} p(y',x)} = \frac{\exp\{\sum_{k=1}^{K} \lambda_k f_k(y_t, y_{t-1}, x_t)\}}{\sum_{y'} \exp\{\sum_{k=1}^{K} \lambda_k f_k(y'_t, y'_{t-1}, x_t)\}} \tag{11}$$

This distribution is a linear-chain CRF that includes features just for an element identity (for example a single word in a word sequence labeling problem). To use additional features (like adjacent words or prefixes or suffixes) the feature functions f_k should be made more general than just identity functions. Thus, a linear-chain CRF is written as:

$$p(y, x) = \frac{1}{Z(x)} \exp\left\{\sum_{k=1}^{K} \lambda_k f_k(y_t, y_{t-1}, x_t)\right\} \tag{12}$$

where *Z(x)* is the normalization function defined as:

$$Z(x) = \sum_{y} \exp\left\{\sum_{k=1}^{K} \lambda_k f_k(y_t, y_{t-1}, x_t)\right\} \tag{13}$$

CRFs are a good solution for a number of relational problems [12] because they allow dependencies between entities and include rich features of entities. Furthermore, CRFs avoid the bias problem that conditional Markov models based on directed graphical models exhibit [13]. On the other hand, CRFs are more difficult to implement, the training step is more complex and they are rather slow compared to HMMs and even to Maximum Entropy Markov Models.

Due to their characteristics and performance, CRFs are being used on an ever increasing rate for IE and NLP tasks. Commercial and open source implementations exist, like MALLET[21], MinorThird[22], CRFSuite[23], CRF++[24] or in Stanford's NER[25]. In the proposed system in this work, Stanford's NER is used for CRF support.

II.3. Parsers

Parsers, from a NLP point of view, perform syntactic analysis on texts in order to discover their sentences' grammatical structure. Parsing is done in respect to a formal grammar of choice.

There are many categories of parsers, but a general classification could identify a few distinct areas like *dependency parsing* vs *phrase structure parsing* or *shallow* vs *deep parsing*.

Dependency parsing reveals a sentence's structure as determined by a relation between a head (a word) and its dependents (other words, usually modifiers, objects or complements).

[21] http://mallet.cs.umass.edu/
[22] http://minorthird.sourceforge.net/
[23] http://www.chokkan.org/software/crfsuite/
[24] http://crfpp.sourceforge.net/
[25] http://nlp.stanford.edu/software/CRF-NER.shtml

Dependency parsing is concerned only with creating dependency trees and ignores other issues like word order for example. This makes them well suited for language invariant syntactic analysis (ex: for languages with free word order). Furthermore, dependency parsers have a high efficiency rating compared to phrase structure parsing and deep parsing.

The output of these parsers are dependency trees. The trees look similar to constituency trees (dependency grammar is equivalent to constituency grammar if there is one restriction of the constituency grammar – that in each phrase a word is set to be its head [14]) and usually the NLP field treats both tree types the same [15]. A dependency tree makes explicit relationships between words in terms of heads and dependents (see figure 2) while a constituency tree makes explicit syntactic constituents visible in a sentence (see figure 3).

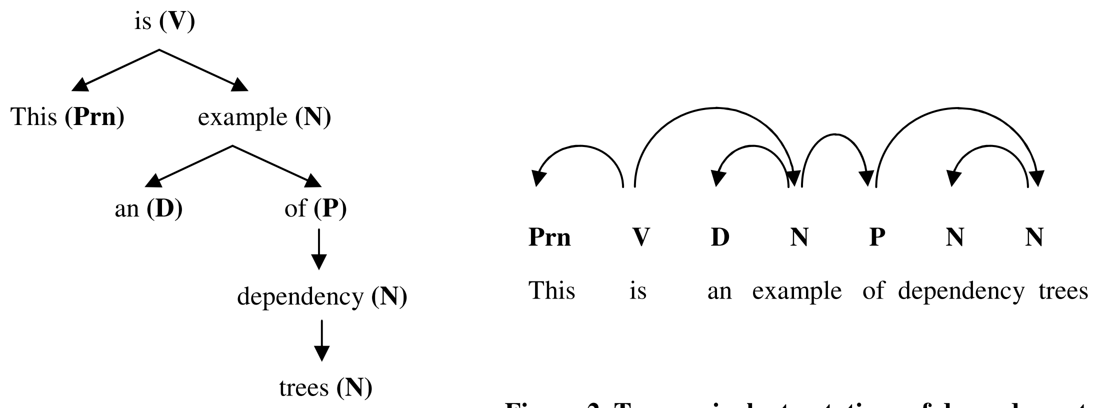


Figure 2. Two equivalent notations of dependency trees

The trees' nodes are words, while their links are relations between words.

Some examples of current dependency parsers: KSDEP [16] – uses a probabilistic shift reduce algorithm, MST [17] – implements an Eisner algorithm for projective dependency parsing.

Phrase structure parsing, coming from phrase structure grammar, usually divides phrases into a verb phrase (VB) and a noun phrase (NP) and then further refines each until reaching individual word level. Phrase structure parsing has been the most active parsing sub-domain due to the existence of the Penn Treebank.

Examples of current parsers include NO-RERANK [18] – based on lexicalized PCFG model; RERANK [19] – takes the first 4 results of NO-RERANK and using a MEM (Maximum Entropy Model) selects the most probable choice; BERKELY's Parser [20]; Stanford's [7] parser – an unlexicalized parser.

Shallow parsing analyses a sentence to identify its component groups (nouns, verbs, etc), but does not attempt to describe the sentences' internal structures or any other deeper features (as deep parsers do). In NLP applications shallow parsers are often used over deep parsers due to their considerable speed gain. *Deep parsers* use the concept of predicate argument structures in their parse evaluation. Such a structure is a graph that represents syntactic and/or semantic relations between words. Deep parsers provide theory specific syntactic and/or semantic structures [21].

A good example is the ENJU deep parser [22] using a HPSG grammar extracted from the Penn Treebank. It also uses a maximum entropy model that has been trained with a HPSG treebank derived from the Penn Treebank.

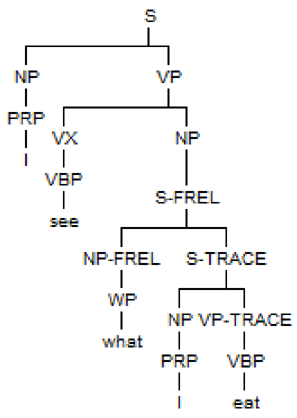


Figure 3. ENJU deep parser visual example (constituency tree) for the sentence 'I see what I eat' [26]

Most of the current parses are statistical parsers, incorporating machine learning algorithms. These kinds of parsers require training data before usage. The majority of parsers are trained using the treebank (parse tree collection) offered by Penn, and especially using the Wall Street Journal section of the Penn Treebank. There are other available treebanks, for example in the medical sector there is the GENIA Treebank [23]. The accuracy of parsers varies with the available training set's size and most importantly, domain. A state of the art parser trained on a generic treebank will perform worse that an older generation parser trained on a specific domain treebank.

In the present work Stanford's Parser is used to obtain the syntactic and the dependency trees.

II.4. Coreference resolution

Coreference resolution is the task of identifying expressions (words or sequences of words) that refer to other expressions in texts. This is an important task as it allows NLP applications to identify information that is given about each particular entity throughout the available text. In particular, coreference resolution is a critical component of an IE system.

Anaphora is a linguistic phenomenon in which an entity is interpreted using knowledge of previous entity or expression (defined as the antecedent) in a text. The anaphor and the

[26] Image obtained using the online ENJU 2.4 parser at http://www-tsujii.is.s.u-tokyo.ac.jp/enju/demo.html

antecedent are in a coreference relation, with one referring to the other. This creates confusion on the definitions of anaphor and coreference. Both are interrelated, but also have non-overlapping areas. There are coreference relations that are not anaphoric and the other way around: for the sentence "The best *teams* in NBA are better than *ours'*." the anaphoric relation is not coreferential, while for the sentence "The *capital of Romania* … in *Bucharest*…" the coreferential relation is not anaphoric.

Coreference resolution is a very difficult task due to the complexity inherent in natural language. There are many types of coreference, for example repetition coreference (I saw *a car*. *The car* was green), synonym (I lost *my bicycle*. I think *my bike* was stolen.), hypernym (Alex was stung by *a mosquito*. *The insect* then flew away), proper name (*Bill Gates* gave a speech. *Mr. Gates* said that ..), pronoun (I saw *Alex*, I told *him* to..) etc.

The coreference resolution task can be considered with three approaches: Supervised Machine Learning, Unsupervised Machine Learning and Knowledge-based approach.

The **Supervised Machine Learning** approach is a very active sub-domain, due to the increasing ML popularity. The systems usually work by finding anaphoric NPs (Noun Phrases) and then create chains by identifying the most likely antecedent(s) using predefined features [24]. Training is done on existing corpora (for example MUC – Message Understanding Conference – training and test data).

Feature sets are diverse and several techniques are used for comparing features: string match (*cat* matches *the cat*), alias (if an entity is an alias of another, depending on type, can be dates like *08.10* matches *August, 2010*, or *Clinton* matches *Bill Clinton*), distance between entities (number of words or sentences that separates two entities), pronouns features (true or false if entity is a pronoun or not), definite NP features (true or false if a NP is definite – if it starts with the word *the* – *the car*), demonstrative NP features (true or false if a NP is demonstrative – if it starts with *this, that, these, those*), number agreement (if both entities are singular or plural), semantic class agreement (use of WordNet for example to determine if both entities are persons, dates, objects, etc.), gender agreement (if both entities are a *he*, a *she* or *unknown*), proper name agreement (if both entities are capitalized), appositive features (if one entity is in apposition to the other – ex: *president of USA* is in apposition to *Lincoln* in "Lincoln, president of USA, …"), etc [25].

The choice of classifiers is also varied, starting from Hidden Markov Models, to decision tree learning algorithms [24], up to Conditional Random Fields [26].

Unsupervised Machine Learning approaches are an alternative to the well performing supervised approaches where annotated training corpora are not available. Even for

English, where corpora are plentiful, there are sub-domains that are poorly covered, and thus unsupervised coreference resolution systems are chosen.

The best known unsupervised system was proposed by Cardie and Wagstaff [27] [28] in 1999, viewing the problem as a clustering task. The clustering is done using a distance metric that is given by a set of incompatibility functions and other indicators. Iteratively, the algorithm starts the initially single-word clusters and merges them based on the distance metric. The initial system had 12 features. Eventual developments rose the number of features to almost 50, and one experimental system later tried the same approach with a feature set of over 300 obtaining improved results.

Knowledge-based approaches use a lot of diverse methods. There are all kinds of combinations between rule-based systems, heuristic systems, morphological, syntactic and semantic information provided by deep parsers, up to data and knowledge repositories (starting from word lists, dictionaries and gazetteers containing lists of names/ organizations/places to semantic information such as WordNet's hypernym tree).

For example, one such system [29] works by first using heuristics to extract only valid antecedent candidates and then choose pair candidates based on proximity and coreference type. Another approach is to use information extraction patterns to identify entity role: first identify NPs that are not anaphoric and then use case resolution to determine coreference. The remaining unsolved cases go through a series of manually coded extraction patterns that use knowledge sources [30].

One big advantage of knowledge-based approaches is that usually there is no need of annotated corpora; however, the downside is that manually created rules and heuristics are needed (sometimes difficult to upkeep), involving domain knowledge of the developers.

II.5. General purpose corpora

Corpora have started to be developed in the early 60s, as a natural need for a repository of accurate data for one task or another. Initially, the vast majority of work was done by hand. As time passed and computers evolved, the corpora started to benefit from semi-automatic and automatic annotation, further improving the accuracy but especially the time needed for manual correction. The size of corpora increased, as well as their specialization. Today there exists a large variety of large corpora for very diverse tasks, starting from the standard POS tagged corpora, treebanks, domain corpora (large collections of medical texts or news articles, or old language documents, etc), tokenized corpora used for chunker training, semantically annotated corpora annotated with ontology references, and so on.

Further on, two important corpora will be presented, the Penn Treebank [31] and the Brown Corpus. The Penn corpus provides a collection of syntactically parsed sentences while Brown provides POS tagged text for tagger training.

Penn Treebank

Treebank refers to a text that has been syntactically annotated and represented as a collection of tree structures. Such treebanks are usually created starting from texts that have been annotated with part-of-speech tags and syntactic structures.

As an alternative to the manually solution, where linguists annotate sentences with syntactic structure, a parser can be used to assign it. The parser alternative does not eliminate the human interaction – the annotation being required to be checked and if necessary corrected.

Penn Treebank annotates the phrase structure (it is also possible to annotate the dependency structure) and is very popular due to its large size and simplicity of the representation. It can be rapidly used to train parsers or other NLP related tools.

A simple example:

```
Cat hunts mice
 (S (NP (NNP Cat))
    (VP (VPZ hunts)
       (NP (NNP mice)))
    (. .))
```

Brown Corpus

In the mid-1960s, at Brown University, the first major corpus for computer analysis was developed – the Brown Corpus, made up of 1,000,000 words from random publications.

Almost a decade later, the tagging for the Brown Corpus was nearly completed. It was based on a handmade list of what categories of part-of-speech could co-occur at all. Initially, the first approximation was around 70% accurate. Subsequently, corrections have been made and the tagging accuracy improved to almost 100% (considering that there exists inter-annotator disagreement).

The Brown Corpus formed the basis for later part-of-speech tagging systems, such as VOLSUNGA and has been superseded by the much larger British National Corpus (100 million words).

The Brown corpus is used for many purposes, starting from POS training to using the sentences and words themselves to provide a reference for each word's information content when used with an ontology like WordNet.

III. Knowledge acquisition and representation

III.1. Ontologies as information repositories

An ontology is a form of knowledge representation and can be used to describe a domain using a set of concepts and the relationships between those concepts in the context of that domain. There are many definitions of ontologies; one would be that an ontology defines a common vocabulary for entities (humans or machines) who need to share information in a domain. It includes machine-interpretable definitions of basic concepts in the domain and relations among them [32].

An ontology is based on formal explicit descriptions of concepts, properties of each concept and restrictions on these properties. Description of concepts are applied to a specific context, they describe particular meanings. These descriptions are called concepts or classes. Properties or roles of these concepts describe various features and attributes of the concepts. While classes have the role to represent an entity into an ontology, a specific example of this entity is represented by instances.

Artificial intelligence, software engineering[27], biomedical informatics[28], Semantic Web or GIS Systems [33] [34] are just a few of the domains the ontologies are used in.

One of the purposes ontologies are developed for is to share the same language between entities. For example, there are different Web sites containing information from the same domain or providing e-commerce services in the same area – for example pharmaceutical information/e-commerce. If all these sites share and publish the same basic ontology of the terms they use, then software agents can extract information and can aggregate it from each of these different sites. The aggregated information will be used to answer user queries or will be sent as input data to other applications.

If an ontology is developed to represent different domains with specific needs (for example different domains whose models need to represent the notion of time) then the ontology can be simply reused by other groups. An ontology can also be extended to describe a specific domain of interest or can be integrated among other ontologies to describe portions of a larger domain.

When reusing existing ontologies or extending them, a formal analysis of terms is useful. Once a declarative specification of the terms of a domain knowledge is available, the analysis of domain knowledge is possible.

The major ontology components are classes, relations, attributes and individuals.

[27] For example, SUMO (http://www.ontologyportal.org/index.html) is used in commercial applications
[28] http://www.ebi.ac.uk/ontology-lookup/ontologyList.do

Classes are concepts, abstract groups of objects defining kinds of things, from general to specific. Classes can be subclasses of other classes. Classes classify other classes and/or individuals. For example, class `thing` is a superclass of class `vehicle`. In turn, `vehicle` is a superclass of class `automobile`, meaning `automobile` is a subclass of `vehicle`. This type of inheritance between classes forms a tree or a graph defining a domain. Classes can be instantiated by individuals. For example, the class `Hyundai_Accent` can be instantiated by an individual that is an `automobile` and has specific attributes.

Relations are ways in which classes and individuals can be related to one another. The set of relations describes the semantics of the domain. The most important types of relations are the *`superClassOf`*, *`type`* or *`subClassOf`*. This defines which objects are classified by which class. Based on these relations a hierarchical taxonomy is created and each object is the "child" of a "parent class". Other relations exist. For example, another possible relation is *`isPartOf`* (meronymy). `petrol_engine` is part of `car` since a car contains an engine. Relations link classes or instances to other classes, instances or literals. For example, the relation *`isBornOnDate`* links a `person` instance to a calendar date.

Attributes are aspects, properties, features, characteristics, or parameters that objects (and classes) can have and that relate them to other objects.

Individuals: instances or objects (the basic or "ground level" objects). A class is given specificity by instantiating it (creating an object that is a type of a specific class) so that the class is now unique by the values of the attributes it now has filled. There is an ongoing discussion whether individuals should be used instead on named classes, especially more so on rather small ontologies, where both approaches work. In theory [32], an individual should be used when it has properties that are different from other individuals of the same type, and that individual is further referred on by other individuals. Otherwise, named classes should be used. Individuals should be instantiated classes that are located at the bottom layer (leaf nodes – most specific) of an ontology.

There are other ontology components like restrictions (dependencies between classes restricting the set of valid assignments), rules (classic if-then constructs), events (triggers on changing values of attributes or relations), etc.

III.1.1. RDF – Resource Description Framework

RDF is a metadata model designed originally by the W3C[29] and became a W3C Recommendation in 2004. It is a standardized method of information modeling implemented in web resources, initially on top of XML for encoding metadata (metadata is

[29] http://www.w3.org/

data about data, like the date of the author of a news article, accompanying the article itself). RDF is designed to be read and understood by computers, not humans (though more human-readable formats exist).

RDF identifies resources using URIs (Web identifiers) and describes them using properties. For example:

```
<?xml version="1.0"?>
<rdf>
  <description about="http://www.stefandumitrescu.ro">
    <author>Stefan Dumitrescu</author>
    <homepage>http://www.stefandumitrescu.ro</homepage>
  </description>
</rdf>
```

where the *resource* is a website (www.stefandumitrescu.ro), having the *property* 'author' and the *property value* 'Stefan Dumitrescu'.

A *statement* is an association of a *subject* and an *object* linked by a *predicate*. The above example translates as 'Stefan Dumitrescu is the author of www.stefandumitrescu.ro' , with 'Stefan Dumitrescu' as subject, the website address as object, linked by the 'is the author' relation. We can also say that the *statement* is a *triple*, meaning we have a *subject*, a *predicate* and an *object*. The subject is normally either a URI or a blank node (anonymous resource). The predicate is also a URI, and the object can be either a URI, a blank node or a simple string literal.

RDF defines a specific vocabulary: rdf:type (the resource is an instance of a specific class), rdf:Property, rdf:Alt rdf:Bag rdf:Seq (containers), rdf:List (list), rdf:nil (an empty list), rdf:Statement, rdf:subject, rdf:predicate, rdf:object.

RDF can be serialized in different formats. The main format is XML, like in the example above. Notation 3 (N3) is also a major format for RDF, being a more compact non-XML notation, designed mainly for human readability.

In N3, the example above becomes (using `ds` as a namespace, definition omitted):

```
http://www.stefandumitrescu.ro
  ds:author "Stefan Dumitrescu"
  ds:homepage "http://www.stefandumitrescu.ro"
```

Another RDF format is Turtle (Terse RDF Triple Language), it is a superset of N-Triples (yet another RDF format) and also valid N3 format. Turtle does not extend RDF's graph model, unlike N3.

N-Triples uses plain-text serialization to store RDF information. It was designed to be a simpler format than both N3 and Turtle.

Ontologies can be stored in RDF format, and RDF itself can form the basis of more advanced knowledge representation languages, like OWL (Web Ontology Language).

III.1.2. WordNet

Princeton University's WordNet[30] [35] is a free electronic lexical resource containing dictionaries of nouns, verbs, adjectives and adverbs. It provides not only dictionaries but also organizes related concept from the individual categories into synsets (synonym sets). Currently the latest version of WordNet is 3.0, containing around 150000 words organized in around 115000 synsets.

The basic WordNet concepts are: *synsets*, *glosses* and *lemmas*. The *gloss* is an explanation or definition of a word in a text, basically a sense-disambiguated corpus. In addition to the definition itself, the *gloss* also contains additional explanations and examples. *Lemmas* are the words that belong to a synset. They represent the string text of the word from WordNet database.

The synset is the equivalent of a concept. A synset is, in essence, an ordered list of synonyms. The synonyms themselves are words that are in the same lexical category and are commonly used to express the same meaning. Synsets as well and the synset hierarchy (created by relations like is-A, part-of, etc.) represent the most used information in WordNet, bringing also semantic value over the standard lexical value a dictionary provides.

WordNet is currently the most commonly used lexical resource for word sense disambiguation. It encodes many senses for every word, and while this seems at first a solid base to use for the diverse tasks, it has been argued that there may be too many senses even for humans [36]. This issue might prevent Word Sense Disambiguation systems from performing at their best. Solutions have been proposed, like clustering methods that might be used to group similar senses together and reduce the total number to only a few, more manageable and distinct senses [37]. For English, accuracy is over 90% if we take coarse-grained senses (every word has few, clearly defined and separate senses), and about 59.1% - 69.0% for fine-grained senses (reported by Senseval-2[31]) (every word has a many senses covering many possible meanings). We must note that for fine-grained senses, the baseline algorithm is that of always choosing the first sense of every word, with accuracy ranging from 51.5% to 57%. This fine-grained baseline accuracy is a problem for most algorithms to even reach, let alone out-perform.

[30] http://wordnet.princeton.edu/
[31] http://www.senseval.org/

There is an ongoing discussion whether WordNet can be viewed as an ontology. From one point of view, the graph provided by the synsets and the hypernym relations between them can be viewed as such an ontology. But, an ontology in its definition does not allow inconsistencies that are present in WordNet, like redundancies in the hierarchy or common specializations for exclusive categories. However, the hierarchy has been automatically cleaned and imported, in one form or another, in several ontological systems, starting from WebKB-2[32] to DOLCE[33], DBpedia[34] or YAGO[35].

WordNet synset graph

WordNet's synsets together with the different relations between them, as discussed above, may be considered to form an ontology. Although under discussion, even if is not considered an ontology, then at the very least the tree-like graph it forms if we consider the hypernym relation (*`subClassOf`*) is a large taxonomy.

The formed graph can be used to add semantic value to nodes, and the links themselves are used as information in NLP, IR, IE algorithms, where the nodes are considered entities the algorithms work with.

For example, figure 4 presents a fragment of WordNet's hypernym tree. It can be seen that classes are linked up to more and more generic classes until the top of the tree `wordnet_entity`.

[32] Integration of WordNet 1.7 in WebKB-2, http://www.webkb.org/doc/wn/wnIntegration.html

[33] http://www.loa-cnr.it/DOLCE.html, also reference *Sweetening WORDNET with DOLCE* [145]

[34] DBpedia includes Wikipedia categories, the YAGO Classification scheme and WordNet Synset links, http://wiki.dbpedia.org/Datasets

[35] YAGO is built starting from Wikipedia and WordNet. Every entity in YAGO has at least one correspondence to a WordNet class through the *`type`* relation ex: `Ford_Focus` *`type`* `wordnet_car` [39]

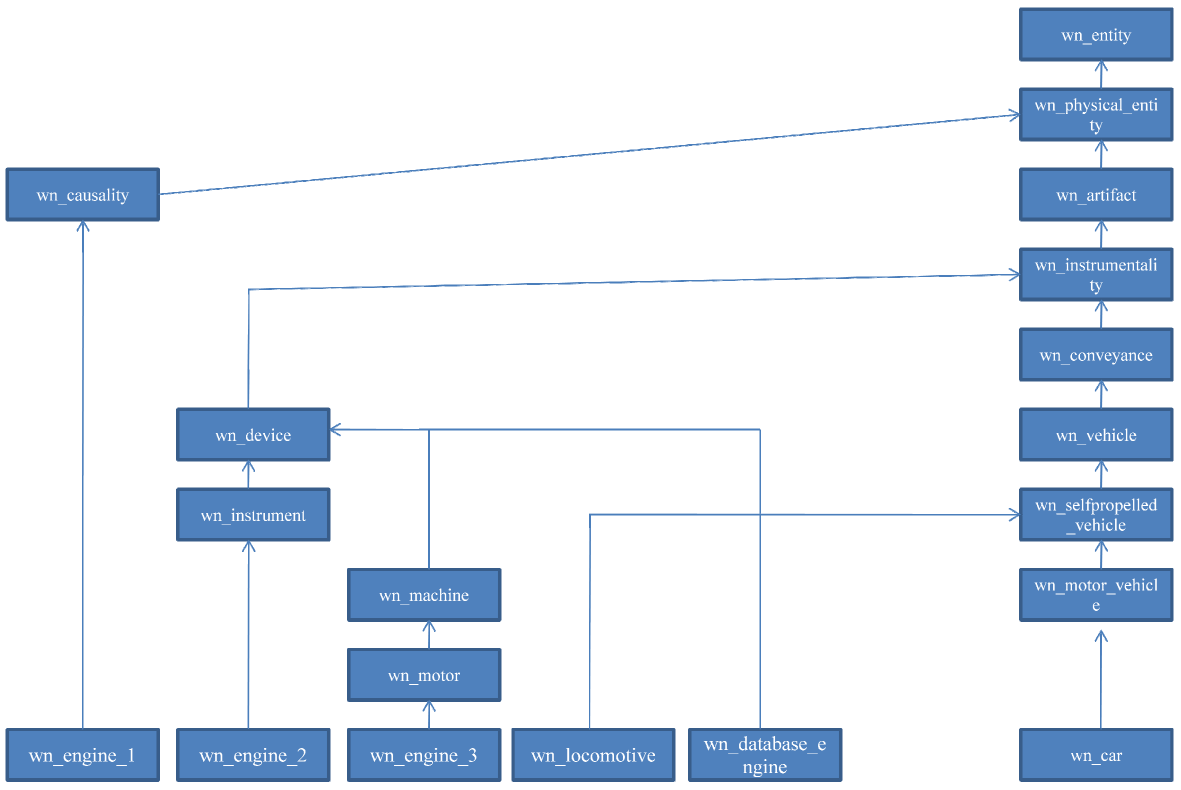


Figure 4. Example of synset partial hypernym graph

Semantic similarity measures

Semantic similarity is a measure of the closeness of relatedness of two concepts. While there are many existing ways to determine relatedness, we will present measures that use Information Content (concept introduced by Resnik in 2005 [38]). Information Content (IC) is a specificity measure for concepts. For example, concepts that are more specific have a higher IC associated value than more general concepts (ex: locomotive has a higher IC than device). The IC value is calculated depending on the frequency of concepts from the text. The process is as follows: the text (corpus) from which IC values are to be derived from is parsed, and for every concept found, its frequency as well as the frequency count of its ancestors are increased by one in WordNet. The ancestor hierarchy is a concept hierarchy where the links are WordNet relations (e.g.: for nouns we have *`subClassOf`* or *`partOf`* relations). Most often the hypernym relation is used (*`subClassOf`*).

An important issue comes from the corpora on which the IC values are determined. If the corpus is sense-tagged, it is easy to count the senses that have to be incremented for every word. However, if the text is not sense-tagged, then all the possible senses for every word are incremented, as well as their ancestors. In this scenario, the frequency of all the occurrences of a word is divided equally among the different possible senses.

After the frequency count is completed, for each concept in WordNet the IC value is computed as the negative log of the probability (frequency count) of the concept.

$$IC(s) = -\log(P(s)) \tag{14}$$

IC information is extracted from general corpora like the Brown or SemCor corpora.

We will investigate three different Information Content measures: Resnik's measure *res*, Lin's measure *lin*, Jiang and Conrath's measure *jcn*. All these measures take two synsets as inputs, and produce a real value that represents the similarity between the two synsets. They are all based on the idea of finding the least common ancestor (LCA), meaning finding the concept that subsumes both of the synsets in WordNet's synset hypernym hierarchy. If there is more than one LCA, the least general LCA is taken (the lowest in the hierarchy).

The Resnik measure (*res*) provides the basic metric that is used both for *lin* and *jcn* measures. The similarity value is the Information Content value of the synset's LCA.

$$sim_{res}(s1, s2) = IC(lcs(s1, s2)) \tag{15}$$

The *res* measure may provide the same value for different synsets that share the same LCA, and thus is not a very informative measure. Lin's measure attempts to improve the accuracy by incorporating information about the IC of each of the synsets.

$$sim_{lin}(s1, s2) = \frac{2 \times sim_{res}(s1, s2)}{IC(s1) + IC(s2)} \tag{16}$$

Jiang and Conrath provide an alternate distance metric instead, using the same elements as Lin:

$$dist_{jcn}(s1, s2) = IC(s1) + IC(s2) - 2 \times sim_{res}(s1, s2) \tag{17}$$

However, to transform *jcn* into a similarity measure, we can simply invert the distance formula:

$$sim_{jcn}(s1, s2) = \frac{1}{dist_{jcn}(s1, s2)} \tag{18}$$

While this formula provides a similarity measure instead of a distance measure between synsets, it does alter the value differences between sets of synsets due to the division.

These three measures types represent standard measures used for a long time in NLP applications, with consistent results.

III.1.3. YAGO

YAGO (standing for Yet Another Great Ontology) is a light-weight and extensible ontology with high coverage and quality [39]. YAGO was built as a very large, accurate (95+ accuracy) and simple to use ontology for machines including WordNet entities and hierarchy, and information extracted from Wikipedia like named entities (people, organizations, geographic locations, books, songs, products, etc.), and also relations among these entities.

For the chosen representation language, YAGO designers decided to extend RDFS to suit their particular needs. Although OWL is the current web standard, the motivation of not developing YAGO in OWL was because OWL Full is undecidable (it is an extension of RDF) and OWL Lite and DL, while decidable, place some restrictions on class definition and description (they are both extensions of a restricted view of RDF). RDFS, which is the basis for OWL can express such relations but can only provide limited semantics, thus the need to extend RDFS. In the YAGO model all objects are entities and two such entities can stand in a relation.

Example [39]:

`Albert_Einstein` *`hasWonPrize`* `Nobel_Prize`

`Albert_Einstein` *`bornOnDate`* 1879

"Einstein" *`means`* `Albert_Einstein`

In the first two statements entity `Albert_Einstein` stands in relations to entity `Nobel_Prize` and the date 1879. The third relation links a string to a class using the *`means`* relation. This enables the linking of any number of strings to an entity, helping to deal with name synonymy. Entities are instances of classes. For this relation, YAGO uses the type relation as in `Albert_Einstein` type physicist. Also, classes stand in a *`subClassOf`* relation to one another. Thus, we have `physicist` *`subClassOf`* `scientist`, which in turn is a subclass of another class, and so on until reaching the root node `entity` (every class is an indirect subclass of class `entity`).

Another important note to be made about YAGO is that for n-ary relations it uses facts about facts. Each fact (two entities that stand in a relation) is given a unique id. Thus, for example, we could express that Einstein was born in the city of Ulm, Germany in the year 1921 like:

#1: `Albert_Einstein` *`bornIn`* `Ulm,_Germany`

#2: #1 *`time`* 1921

YAGO uses two sources of information: WordNet and Wikipedia. From WordNet it borrows the hypernym hierarchy, while from Wikipedia it borrows entities and uses them as arguments to the relations implemented in YAGO. Each synset from WordNet is translated in a YAGO class. In cases where Wikipedia also contributes entities, WordNet is always given preference. Thus, WordNet defines the upper hierarchy, while Wikipedia contributes to the lower, most specific branches. These are also linked up using the *`subClassOf`* relation.

WordNet synsets have words with similar meaning. After YAGO creates an class from each synset, it uses every word in the synset to add *`means`* relations to the created class. For example, the word "car" belongs to the Automobile synset – YAGO creates the *Automobile* class, and the fact "car" *`means`* `automobile`.

There are meta-relations defined, like *`context`*, or *`extractedBy`*. These give information about the place the data was extracted, the confidence in the extraction, etc. It should be noted that there is a fixed number of relations built in YAGO. While this does not mean that YAGO is limited, it does create the need of extending YAGO with new relations, and brings up the debate whether the relations should be canonical (being pre-defined as a function with domain and range $f{:}D \rightarrow R$) or free (not defined). YAGO can be improved with the addition of new canonical relations.

YAGO stores its data in any of the XML, SQL and RDFS formats. This provides a great boost in accessibility.

In summary, YAGO stores more than 2 million entities with 20+ million facts about them. The facts are high quality, having been automatically extracted from two trusted information sources, Wikipedia and WordNet. Also, YAGO uses a simple, extendable, RDFS-compatible model. YAGO is important because it is a major step in providing large, accurate data sources. At the time of writing, the YAGO upper hierarchy was embedded in DBPedia[36] database, which is the only larger source of information than YAGO itself. However, community efforts like Freebase[37] will, if not already, create much larger information sources.

III.2. Information Extraction

The Information Extraction (IE) field has risen from the need for computers to understand the huge amount of information on the Web that is in raw unstructured text format. IE tries to extract textual facts into a relational structure, a knowledge base. The smart structuring

[36] http://dbpedia.org/About
[37] http://www.freebase.com/

of information into such knowledge repositories allows computers to answer user queries with actual answers instead of lists of candidate web sites.

The core task that defines IE is the extraction of facts form natural language, in one form or another, using an extractor that identifies entities and their linking relationships. For example, from the sentence “Einstein, born in Ulm (1879), went on to win the Nobel Prize in Physics in 1921” several facts can be extracted:

1. *Einstein* was born in *Ulm*.
2. *Einstein* was born on date *1879*.
3. *Einstein* won the *Nobel Prize in Physics*.
4. *Einstein* won Nobel Prize in Physics in *1921*.

While for humans these facts are extracted instantly and correctly, for a computer it is much more difficult. For example, Einstein in this case is the subject of every fact. Einstein is a reference in fact to the `Albert_Einstein` entity. The extractor must know that usually a number in brackets after a location means a date, so it can extract that the born date is the one in brackets. The extractor must ignore irrelevant words such as “went on” and pick “win” as the correct relation between Einstein and the `Nobel_Prize` entity. Furthermore, the `Nobel_Prize` entity is generic, and it has to be specified that there exists a `Nobel_Prize_in_Physics` instance characterized by the date of winning, in this case, 1921.

Normally, the standard triple format is used: Subject, predicate, Object. But how to store relations that have multiple arguments? For example fact 4 cannot be stored into a standard triple. One solution is to also store the id for each fact, and then fact 4 would be stored as #4 (*#3*, *`onDate`*, *1921*) where fact 3 would be: #3 (`Albert_Einstein`, *`wonPrize`*, `Nobel_Prize_in_Physics`); Another solution would be to individualize the Nobel Prize instance, meaning to add another fact like (`Nobel_Prize_in_Physics_12345`, *`type`*, `Nobel_Prize_in_Physics`) where the _12345 would be an unique identifier for the new entity that is a type of generic Nobel Prize in Physics. Then it would be easy to link Einstein to this specific instance (for fact 3) and specify that this instance was won in 1921 (for fact 4).

There is a choice of storage formats that influences the types of algorithms that can be run on the database in response to queries. There are representation formats on which inferences can be made; some formats are decidable but more restricted (OWL Lite, DL), some are not decidable but much more expressive (OWL Full). An extractor has to take every such aspect into consideration.

There are two major types of results obtained for the IE task. The major effort currently in IE is to analyze texts and extract canonical facts. A canonical fact is a fact that has its relation predefined in an ontology (the relation is one from a set of relations defined in that ontology) and its entities also belong to a specific generic entity (they can be either subclasses of a more generic class or instances of a specific class). There are advantages and disadvantages with this approach. For example, one has to predefine each relation of interest. This leads to a certain domain specificity degree, and also the time needed for every relation is linear to the number of required relations.

Initially, IE systems tried to extract information from very domain-specific sources like news articles or internet posts. More recently different systems have begun to be used on a larger domain base, with decent success [40] [41]. SOFIE – Self-Organizing Framework for Information Extraction [42] is a good example of the current generation of systems.

The second major approach is based on the premise that the Web contains very much information and the number of possible relationships that can be found is much larger that it is possible for humans to predefine and create models for each relation type. To be able to extract all possible relations (in the thousands as opposed to only a few hundred) some concessions have to be made. While the extracted entities and relations number is vastly superior to the traditional canonical approach, the facts themselves are not canonical, meaning they are just strings, without any predefined meaning behind them. The best example of this approach is TextRunner [43], having a collected knowledge base of millions of entities and thousands of extracted relations.

III.2.1. Open IE – TextRunner

The TextRunner system (developed by Michele Banko [43]) introduces a new term coined "Open Information Extraction", moving "away from architectures that require relations to be specified prior to query time in favor of a single data-driven process that discovers an unbounded number of relations whose identity need not be known in advance" [43].

TextRunner takes as input web documents (unstructured free text) and outputs sets of tuples containing entities that stand in a relation. TextRunner tries to extract as many relations as possible. It does not have a set of predefined relations, thus the tuples it extracts are string tuples (entities and relations are not canonical, meaning they are not predefined in an ontology, for example). Currently TextRunner has extracted a large number of tuples (more than 13 million after tuple reductions), spanning about 16000 distinct relations linking around 4.2 million entities.

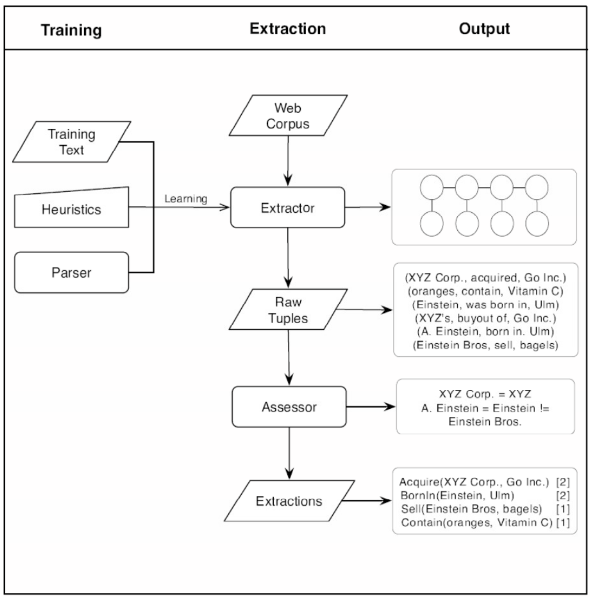
The system emphasizes on three points / problems:

- Automation – meaning that for a system to be useful it must extract as many relations as possible.
- Domain Independence – meaning that an IE system should handle texts from any domain.
- Efficiency – meaning that the system must scale to the size of the Internet, being able to process billions of documents.

The main feature of TextRunner is that it implements a unitary model of expressing relationships (independent of relationships themselves). This allows for a language model that can either be learned automatically or developed by hand that takes as input documents (domain independent) and outputs tuples that contain entities linked by relations. This feature addresses the three points above in that it removes the need for manual relation identification (reducing manual labor to a constant, independent on the relation set size) and it shifts the focus to relation discovery and extraction rather than the traditional entity discovery and relation identification from the pre-programmed set.

In the thesis that presents TextRunner [43], Banko shows that 95% of the patterns that are used to define binary relationships can be grouped into only a few generic patterns. Thus, most instances are verb-centric – about 37%, verbs + preposition at about 16%, infinitival phrases – 9%, noun phrases + verb – 1%.

TextRunner is composed of the **Learner** module, **Extractor** module, **Assessor** module and the **Query Processor** module.



Training
Extraction
Output
Training Text
Heuristics
Parser
Learning
Web Corpus
Extractor
Raw Tuples
Assessor
Extractions
(XYZ Corp., acquired, Go Inc.)
(oranges, contain, Vitamin C)
(Einstein, was born in, Ulm)
(XYZ's, buyout of, Go Inc.)
(A. Einstein, born in. Ulm)
(Einstein Bros, sell, bagels)
XYZ Corp. = XYZ
A. Einstein = Einstein !=
Einstein Bros.
Acquire(XYZ Corp., Go Inc.) [2]
BornIn(Einstein, Ulm) [2]
Sell(Einstein Bros, bagels) [1]
Contain(oranges, Vitamin C) [1]

Figure 5. TextRunner architecture [38]

The **Learner module** outputs an extraction model for relationships based on a training corpus and manually added heuristics. The model is language dependent (given it was trained on a certain corpus) but is relation independent.

In the first stage (of two) the Learner labels its own training examples based on heuristics as possible relation instances. In the second stage it uses the labeled data to train the Extractor module. The Learner uses a set of parse trees to train the extractor using a single self-supervised procedure instead of using a parser repeatedly. The relation instances

[38] Image taken from [43], page 24

(positive and negative examples) are modeled using features that do not depend on syntactic or semantic analysis during extraction. The output model does not contain relation specific features.

The **Extractor** is used to extract tuples for all possible relations found in a given text. TextRunner implements two extractors: the first considers relation extraction as a classification problem while the second as a sequence labeling problem.

The first Extractor implements the Naïve Bayes classifier [44] which tries to evaluate if chunks of text involving two delimiting entities form a relation. The classifier is trained using the examples previously labeled by the Learner. Possible relationships are found by examining the tokens (words) in the intermediate context created by a pair of given entities. The search is refined by using a phrase chunker to identify and eliminate unnecessary tokens such as adverbs or adjectives. The top most likely tuples are kept to be evaluated in the next module.

The second Extractor implements CRFs (Conditional Random Fields, presented in section II.2.2). The second order linear chain CRF is used to determine if token sequences are valid candidates for entity-relation-entity tuples. After entity identification, all combinations of two entities no more than a set word count apart (window size) are considered as borders for a possible relation. The tokens between the entities are labeled using the *BIO* encoding [45]. This encoding labels tokens as either *B* (beginning), *I* (intermediate – follows *B*) or *O* (out – not in the phrase).

There are some limitations though: the extractors extract only explicit relations from the text; the extractors extract word-based relations, not punctuation bases; relations must occur in the same sentence to be considered.

The **Assessor module** identifies and ranks instances that refer to the same object or indicate the same relation using different words. It implements an unsupervised algorithm [46]. It firsts normalizes the tuples, performs synonymy reduction and then ranks the resulting tuples.

Normalization is performed by simply stemming the words to their roots. Also, it removes tokens that can lead to over specification by using a set of head-finding rules developed by the parsing community [47]. This may introduce some problems, as possible needed words are lost (ex: “most people” is reduced to “people”). The next step is Synonym Resolution where the RESOLVER algorithm [46] is used to predict the likelihood that two strings refer to the same item based on string-similarity and shared relational attributes. The last step is the Assessment, where identical tuples are merged together. Given that if the number of tuples is large, memory and processing problems may occur, the MapReduce [48] framework is employed.

The **Query Processor** is the last module in TextRunner. It takes as input the tuples and outputs a distributed index, useful for fact retrieval based on user queries. The inverted index is created using Lucene[39] (an open source search engine). The Query Processor enables relational Web search, where nodes in the graph are entities and the edges that link two nodes are relationships between entities.

TextRunner is available online for testing at [40] where it allows searching the extracted tuples.

III.2.2. Canonic fact extraction – SOFIE

SOFIE - A Self-Organizing Framework for Information Extraction [42] was developed as a system for automated ontology extension. SOFIE parses text documents and extracts ontological facts, adding the facts back in the ontology it used to asses them.

The problems SOFIE comes up against are: Word Sense Disambiguation, Pattern Matching and Ontological Reasoning. SOFIE is interesting from quite a few points of view. First, for the WSD problem, if it detects as it parses the text that more evidence for a word sense accumulates against the previously selected sense, it reevaluates its choice for that word. Second, it can reason on the plausibility of the proposed extracted patterns and reject some of them. Third, SOFIE uses an ontology for reasoning, meaning it uses relation information (like relation domain and range, constraints, etc), proposes hypotheses that it tries to satisfy. It does all this by combining the three distinct problems in a single framework – translating the problems into logical clauses that need to be satisfied, in essence solving a weighted Maximum Satisfiability problem. The MaxSat problem (an extension of the standard Sat problem) is to determine the maximum number of clauses that can be satisfied for a given Boolean formula. The weighted MaxSat adds weight to the clauses and asks to determine the maximum weight obtainable for the given formula.

The motivation behind SOFIE was that even though large sources of information exist, like YAGO or DBpedia, they are still small compared to the information volume available on the web. Furthermore, the size of the ontologies (knowledge repositories) themselves should help with the effort of extracting even more knowledge with high accuracy.

The SOFIE model uses the following notations:

Facts are noted as `produced`(*Hyundai*,*Accent*) [1], meaning that the company Hyundai has produced the product (in this case a car) named Accent, with the truth value of 1 (can be either 1 or 0) in square parenthesis.

[39] http://lucene.apache.org/
[40] http://www.cs.washington.edu/research/textrunner/

SOFIE extracts pattern such as *patternOcc*("@ is in @", Bucharest@D1, Romania@D1) [1], meaning that it has found a pattern represented by the string "@ is in @" where @ denotes placeholders, with text entities Bucharest and Romania, both of them found in document 1 (noted as @D1). There is a restriction in place, meaning that if an entity is found several times in the same document it is considered to denote a single entity throughout that document – ex: "Java" found several times in one document will denote either the island or the programming language, never both.

The fact that states how likely it is for a text entity to refer to an ontological entity is called a disambiguation prior, that has a confidence value attached: *disambPrior*(Accent@D1, `Hyundai_Accent`, 0.6)[1]. Based on the disambiguation priors, SOFIE can propose hypotheses like one that states that an text entity is to be disambiguated as an ontological entity: *disambiguateAs*(Accent@D1, `Hyundai_Accent`) [?]. The truth value is unknown and thus noted with a question mark. The same type of hypotheses can be made for new facts that have yet to be verified: *locatedIn*(`Bucharest`, `Poland`) [?] or about patterns that may or may not express a relation: *expresses*("@ is in @", *locatedIn*) [?].

SOFIE also uses rules, which are first order predicate logic statements. The rules are used for general world knowledge, like expressing a fact that if a person has died on date X it cannot be born on a date later than X, or that if that person is born in a location, it cannot be born on any other location. For example, the latter rule is expressed (generically) by:

R(*X*,*Y*) ^ *type*(R,function) ^ *different*(*Y*,*Z*) => !R(*X*,*Y*)

There are also rules that link facts and hypotheses. For example, a rule that links the pattern occurrence P (string) to an actual relation R (ontological relation), with WX, WY meaning words (text entities) and X and Y ontological entities:

patternOcc(P,WX,WY) ^ *disambiguatedAs*(WX,*X*) ^ *disambiguatedAs*(WX,*X*) ^
R(*X*,*Y*) => *expresses* (P,R)

All the rules are hand-added to the system.

The aim of SOFIE is to find the maximum number of satisfiable rules that make hypotheses to be accepted as facts. The taken approach is to cast the problem as a weighted Max Sat (Maximum Satisfiability) problem, with variables as hypotheses and rules the first order predicate logic formula. SOFIE is given facts, hypotheses and rules and tries to find the combination of truth values for the hypotheses so that the maximum number of rules is satisfied.

First, the text is cleaned, tokenized and interesting entities are extracted. Between the entities (which might be names, locations, organization, dates, numbers, etc) the linking text is kept as a possible pattern, spanning no more than a preset window length. This step produces the following fact type: *patternOcc*(P, WX, WY) [1], where P is the pattern (text string), WX and WY are the interesting entities. Then all the interesting entities are evaluated and the following fact type is produced: `disambPrior`(WX, *X*, k) [1], where WX is the text entity, *X* is the ontological entity and k is a value that expresses the trust that WX actually refers to *X*. What is interesting here is that the system uses the ontology for this step. In the ontology there is a `means` relation that links strings to entities. In this way, if "Lincoln" is found in the text, then the ontology is searched for the means relation that has the string parameter equal to "Lincoln" and the ontological entity parameter is taken. This may produce more than one disambiguation priors, as "Lincoln" can be a person, a rifle or a car, and thus produce three facts with their own trust value.

The second step is the weighted MaxSat problem, which is NP-hard. The facts, hypotheses and rules are cast into clauses. However, due to the form of the specific rules used by SOFIE, a special customized algorithm named Functional Max Sat is used to "circumvent" the difficulty of solving an NP-hard problem yet also delivering performance. This approximation algorithm outputs solutions that are guaranteed to be in a certain range from the optimal solution. The approximation guarantee parameter can be tuned for faster run time but probably further from the optimum or longer run times but closed to the optimal solution. The output of this second step is a set of facts like `expresses`(P,R), `disambiguatedAs`(WX,*X*), meaning that SOFIE is certain that a pattern expresses an actual relation, and an entity from the text refers to an ontological entity. Based on these relations, new facts can be added: R(X,Y).

Results showed that SOFIE performs well, delivering 90%+ precision (meaning that 90%+ of the facts that it finds are correct). However the recall is very low (so it actually finds very few facts out of the total number of facts that could be found in a document), but comparable to current systems.

IV. Entity recognition and word sense disambiguation for Information Extraction

A major task in Information Extraction (as well as in Information Retrieval, Artificial Intelligence, etc) is entity recognition and disambiguation – entities are the subjects and objects of sentences, and clear identification is essential in order to achieve performance in this field.

Regarding entity type, there are two major categories: named and common entities. Named entities are usually persons, objects, places that are identified by a proper name (in most cases beginning with a capital letter). Common entities are usually normal nouns, adverbs, adjectives starting with lower case letter. For example: "*Today*, *John* is visiting the *city* of *New York*" – "*Today*", "*city*" are common entities and "*John*", "*New York*" are named entities.

The differences between entity types and characteristics have created two distinct tasks of NLP: the task of disambiguating the senses of words (Word Sense Disambiguation – WSD – applied generally to common nouns) and the task of recognizing and classifying named entities (Named Entity Recognition – NER – applied generally to proper nouns as well as other interesting entities like dates or numbers).

This chapter presents the tasks of WSD and NER separately, and then looks at common points and systems that see both tasks from a single point of view (a term coined GNER – General Named Entity Recognition).

IV.1. Word Sense Disambiguation

Entity disambiguation is the task of identifying *which sense (meaning) of an entity* (a simple or composed word) is used in a sentence, given the fact that words are affected by polysemy / homonymy problems. WSD allows computers to 'understand' the meaning of words and language. It is an essential problem that if solved (or better stated if a WSD approach would be developed that has human-like performance) would advance many fields, starting from commercial applications like Internet search performance increases (as a computer would 'understand' what a search string means), better automatic language translation, etc, up to the field of Artificial Intelligence where WSD would be a requirement for a 'reasoning computer' that could pass the Turing test.

Brief History of WSD

WSD first began as a problem for Machine Translation (MT) by Weaver, 1949. Later, Bar-Hillel (1960) tried to determine the sense of certain words in different sentences, but the attempt was a failure, deciding that there were no means to identify the correct senses and thus left the problem to the MT field. Bar-Hillel's report represented the basis for the ALPAC report (ALPAC, 1966), which is generally regarded as the direct cause for the abandonment of most research on machine translation in the early 1960's. The 1980s brought rule based systems, relying on hand crafted knowledge sources. Most of these years were spent on AI-based work, yielding promising yet almost unusable results in all but restricted domain fields. The major problem was the "knowledge acquisition bottleneck" [49] – the problem of acquiring very large amounts of knowledge. In the 1990s corpora were beginning to be developed to a large enough scale, and coupled with the increase in processing power of the new PCs, corpus based approaches began to appear [50]. The last decade brought hybrid systems that combine classic methods and newly available resources like the Web.

WSD Applications

Sense disambiguation is an "intermediate task" [51], necessary in some step or another to aid or to form the basis for many natural language processing tasks. Besides its main purpose for message understanding and communication, it is also used in instances where language understanding is not needed:

• *machine translation*, needed for automatic translation of foreign words that depend on the surrounding context;

• *speech processing*, where WSD is needed for correct phonetization of words in speech synthesis, segmentation and homophone discrimination in speech recognition;

• *simple text processing*, necessary for spelling correction, hyphenation, case changes, diacritics placement, etc;

• *information retrieval and hypertext navigation*, WSD is used to eliminate word occurrences from documents that use other senses for the given keyword;

• *content and thematic analysis*, WSD is needed to analyze the distribution of pre-defined categories of words;

• *grammatical analysis:* for example, in part of speech tagging.

Approaches

Currently, there are two major directions for WSD:

- *Supervised Disambiguation*, where machine learning approaches are used to train various classifiers; these systems encode custom features into feature-vectors, and, based on the provided labeled training data build models used to assign appropriate word senses;
- *Unsupervised Disambiguation*, where the learning system uses unlabeled corpora.

When evaluating a WSD approach based on the resources used, two main categories appear:

- *Knowledge-Rich*, where lexical resources like ontologies, thesauri or dictionaries are used;
- *Knowledge-Poor*, where no such resources are used, instead relying only on the corpus.

IV.1.1. Supervised Disambiguation

Supervised WSD uses machine learning (ML) techniques to determine a word's sense. As during the last decade new algorithms were developed and older ones improved, the ML pool offered increasing resources to researchers that began using more and more such algorithms. Currently, the vast majority of WSD systems is based on one or more ML algorithms and overall performs better than other non-supervised systems.

IV.1.1.1. Decision based WSD

This approach to WSD is among the first attempts to use ML type of algorithms. However this type of WSD was not very successful. Early attempts using decision trees in the 70s [52] and 80s [53] and decision lists [54] yielded rather poor results. The development of the C4.5 algorithm by Ross Quinlan [55], an algorithm now implemented and used in many ML tools and application suites such as WEKA [9] did improve the results obtained by using decision trees.

A decision tree is used as a predictive model to classify some input based on observations about it. The process starts from the root observation and moves on branches down to the leaves which represent the possible classification variants. The choice whether to move down a branch or another from any node within a tree is based on the result of the evaluation of the observation in that particular node.

For example, the C4.5 algorithm builds a decision tree starting from a set of training data, using the concept of information entropy (entropy is the measure of uncertainty associated with a random variable). Information entropy here refers to Shannon's entropy which measures the expected value of information contained in a word/message, using a standard unit of measure (ex: a bit). The training data is represented as feature vectors of the form $t_i = \{f_1, f_2, f_3, \dots, f_n, c_i\}$ where f_k represents the features of training item t_i and c_i is the class of t_i. In the process of building the tree, at each node the algorithm chooses the attribute that best splits the training set into two distinct categories by evaluating the normalized information gain obtained by using that attribute to split the data. The information gain is in this case the difference in entropy, and the attribute that maximizes this difference is chosen as the criterion for the node. The process repeats on the now smaller list of attributes until the tree is complete (all attributes have been used and are found in the tree – with a few base cases as exceptions to this rule).

The advantage of using decision trees is that they are simple for humans to understand, do not require extensive data pre-processing (like normalization, etc), handle both numerical and nominal data attributes (a nominal attribute is a ‘class’ attribute), the models are consistent in that they are robust and are statistically provable to obtain certain results and also due to the open nature of the algorithm, it’s progress can be followed step-by-step, unlike a neural network for example.

On the other hand, there are some disadvantages. The problem of generating the tree is a difficult task. During the tree building process, the decision to pick an attribute over another at each node is actually a problem of a local optimum usually solved by greedy algorithms. The addition of genetic algorithms shifts the local to a global optimum problem, yielding better results. Another problem would be that because of their features, some problems cannot be modeled very well as a decision tree, even though it would seem so at first sight. Problems such as overfitting [56] and attribute bias [57] arise.

Unlike a decision tree, a decision list is an ordered list of rules of the type if-then-else. These rules are determined from the training set. Given that each rule has a weight assigned, the list obtained by sorting the rules descending by their weights constitutes a decision list. For every word and its features, the list is checked and the rule that has the highest score matching the features of the word will give the sense of that word.

IV.1.1.2. Neural network WSD

An artificial neural network (ANN) is a mathematical model that tries to replicate the behavior of real biological neurons. An artificial neuron also tries to replicate a biological neuron by implementing a mathematical function such as that when it receives an input (it receives values on one or more connections from previous artificial neurons) it uses a function to evaluate the inputs and produce an output. The function could be, for example, the sigmoid function or the step function. Such a network interconnects a number of artificial neurons in different patterns. Depending on the type of ANN, its structure may change during the evolution of the network. An ANN can be trained on labeled examples to induce an input type → conditioned output behavior.

ANNs have been in constant development since the 40s [58]. A short classification of ANNs used in WSD based on the distinction between connection patterns between units and the way data is being propagated: *Feedforward Neural Network* – Without any loops or cycles, here the information moves in only one direction, from the input nodes through the output nodes. While this is the simplest of ANNs, it is one of the most used. *Radial Basis Function (RBF)* – the network has a hidden layer of radial units, each modeling a Gaussian response surface. RBF is a real-valued function which has built into a distance criterion with respect to origin or some other point called center. *Kohonen Self-organizing Neural Network (SOM)* – this network is characterized by a set of artificial neurons that learn to map points in an input space to coordinates in an output space – this is a form of unsupervised learning but is presented here for reference. The input space and the output space can have different dimensions and topologies. *Learning Vector Quantization Neural Network (LVQ)* – neural networks that consist of two layers. The first layer maps input vectors into clusters that are found by the network during training. The second layer maps merges groups of first layer clusters into the classes defined by the target data. An LVQ system is represented by prototypes of the classes parameterize, together with an appropriate distance measure, a distance-based classification scheme. *Recurrent Neural Networks – (RNNs)* – here the data flow is bi-directional. This property allows for a large number of variants of this base type, like fully recurrent networks, simple recurrent networks, hierarchical RNNs, etc.

The usage of ANNs for WSD can take many shapes. One is to consider the neurons as words. Then, during training, words that appear in context are activated together, thus linking the words in context to word senses [59]. Later uses of this method involve linking sense to current knowledge repositories [60]. Numerous attempts have been made to use ANNs to the task of WSD, some of them with good results [61] [62].

One of the disadvantages of using ANNs is that they require a lot of training data to output usable results, a problem even for the existing annotated corpora today. Also, they have a large number of human-adjustable parameters. While this can be seen as both an advantage and a disadvantage, these parameters make replicating previous work (previous results) difficult and thus makes comparing the performance of this class of systems even more so.

IV.1.1.3. Instance-based WSD

In this approach to WSD a classifier is built from example instances. This class of algorithms does not perform explicit generalization, instead evaluates each new instance to the previous instances from the training set (lazy-learning). The advantage of this type of learning is that it can adapt to new data instances, and does not have to re-train on the entire training set from scratch.

As instances are seen as feature vectors, they can be represented as points in an n-dimensional feature space.

Because the space is n-dimensional, a metric to evaluate the 'distance' between two instances is required. Many distance metrics can be employed, one of the simplest being the Hamming distance:

$$Ham(t_i, t_j) = \sum_{k=1}^{n} w_k I(t_{ik}, t_{jk}) \tag{19}$$

where t_i and t_j are instances, w_k is the weight associated to attribute k and $I(t_{ik},t_{jk})$ is the identity function that is 1 if $t_{ik} = t_{jk}$ or 0 otherwise.

Another simple distance (if it can be applied – working with real-valued features) is the n-dimensional Euclidian distance:

$$Euc(t_i, t_j) = \sqrt{\sum_{k=1}^{n} (t_{ik} - t_{jk})^2} \tag{20}$$

The k-Nearest Neighbor algorithm is the most basic algorithm in the instance-based learning class. A new instance is classified as belonging to a certain class based on the classes of its closest k neighbors – the class of the new instance is the class of the majority of neighbors.

The training of a kNN algorithm is simply storing each instance as an n-dimensional point. In the classification phase, a new point has its distance calculated and the majority of its closest k neighbors gives the class of this new instance. While simple, the algorithm does suffer from imbalances. Classes with many representatives will likely be selected more often simply because the probability that the majority of neighbors of any point belong to that particular class. Techniques to alleviate this issue exist, such as weighing the value of each neighbor based on the distance from the new instance. A value of $1/d$, where d is the distance (expressed as a real value in the particular case of real-valued features) is called linear interpolation, helping dampen the influence of an uneven data set.

The choice of k is human adjustable, low values favoring noise while higher values being less susceptible to noise but failing to make good distinctions among classes. Cross-validation is one technique used to estimate k.

Applied to WSD, the major difficulty is creating a good feature vector, meaning determining the best mix of attributes to describe an example instance. While this is a major issue with the vast majority of ML systems applied to WSD, it is even more relevant here as the kNN algorithm, for example, is very susceptible to imbalanced datasets. Features can be extracted in many ways: part-of-speech tags, stemmed form (for verbs), singular form (for nouns), words adjacent, head of phrases, tags extracted from the syntactic tree or the dependency tree, capitalization, distance to other relevant words, string similarity measures, number of other certain words, etc, can be used to create a feature vector. Several systems using kNN have been implemented [63] [64], with good results.

IV.1.1.4. Probabilistic Classifiers

The Naive Bayes (NB) classifier is the basic example of probabilistic classifiers, and at the basis of advanced probabilistic-based systems.

A NB classifier is a probabilistic classifier that uses Bayes' formula while making strong independence assumptions between features. The general model for a probabilistic classifier is:

$$p(C|f_1, \dots, f_n) \tag{21}$$

where C is the dependent class variable, with f_k the features for a particular instance. Estimating p will yield the class of that instance. However, when the number of features n is very large or the number of possible values for the features is very large, directly computing such probabilities is not practical. Using Bayes' theorem that states that:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{22}$$

and also using the independence assumption that a feature f_i is independent of f_j, meaning:

$$p(f_i|C, f_j) = p(f_i, C) \ with \ i \neq j \tag{23}$$

then rewriting iteratively the first formula, the conditional distribution over C (class variable) becomes:

$$p(C|f_1, \dots, f_n) = p(C) \prod_{i=1}^{n} p(f_i|C) \tag{24}$$

The obtained formula is now easily computable. A scaling factor can be applied to adjust p.

Applied to WSD, the NB classifier computes the probability of each sense of a given word to appear, given the feature vector for that particular word, assuming that there are no dependencies between individual features. *p(C)* can be estimated as the frequency of a certain sense while *p(f_i|C)* is the frequency of feature f_i when used in the sense *C*.

While this approach produces surprisingly good results in spite of the independence assumption [65] [66], and several systems have been created in the late 90s based on this method [67] [68], its use in recent systems is minimal, being superseded by better performing supervised approaches, such as random forests or SVMs.

IV.1.1.5. Support Vector Machines

Support Vector Machines (SVM) are a useful tool for many IE/IR/NLP applications, including WSD. Because of its importance, the SVM concept was presented in more detail in an earlier chapter. In short, a SVM tries to obtain an n-dimensional hyperplane that separates two instance classes. Based on the support vectors, a SVM constructs the hyperplane by maximizing the distance between the two classes.

In WSD, a SVM classifier is trained on a set of instances (a feature vector and the sense assigned to the word that yielded the feature vector) and used to classify new, unseen instances, similar to the NB classifier. However, as the SVM is a binary classifier, so to be used to determine several senses (several classes) either many SVM classifiers are trained in a one vs. all strategy or one vs. one strategy and then combined as to overall offer a multiple class answer.

Another way to use the SVM (different from the NB style) in cases where the number of features is too large to be tractable is to create custom kernel functions. In essence, a SVM computes the dot product between two vectors. The kernel function (which is in fact a similarity function between two vectors) is overridden to compute the dot product in a tractable way (meaning using heuristics that do not require the features to be explicitly expressed, for example). This ability makes the SVMs a versatile tool in many fields.

Applied to WSD [69], the SVMs have been shown to perform very well [70], usually better than other supervised approaches. The most frequently used approach here is the standard extraction of a feature vector for every instance word as well as its class (sense), train the classifier on as many training instances as possible, and then apply the created model to new instances.

IV.1.2. Unsupervised Disambiguation

Unlike supervised approaches, in most cases unsupervised methods attempt to cluster words together rather than identify a class for each word from a lexical resource containing structured words / word senses. Assuming that a word has a sense when surrounded by a certain context and another sense in another context, unsupervised disambiguation tries to cluster together words in common contexts.

Because of the lack of using an external resource (an ontology, taxonomy or dictionary) based on which to link words to word senses, most unsupervised WSD systems cannot be compared or have their performance clearly evaluated.

There are three main approaches: word clustering, context clustering and co-occurence graphs.

IV.1.2.1. Word Clustering

Word clustering approaches try to cluster words that are semantically close. One such approach [71] identifies words similar to a word w based on the information content of individual features like syntactic dependencies. To discriminate between word senses (which at this stage cannot be done as the similar words associated to w can represent any sense of w) a word clustering algorithm is applied. The similar words are sorted by their similarity to w. Next, w is placed as a root node in the currently empty 'sense' tree. Each similar word is then added to w (and in later stages to previous similar words) so that iteratively a tree is constructed. Finally, each similar word that is a child of w is considered as a distinct sense (as it contains under it further words that describe that particular sense).

Other methods that follow the same general idea have been developed. For example, instead of a tree, a matrix is created where the value of row i and column j is the similarity of words i and j, and then a clustering algorithm is applied on the matrix.

IV.1.2.2. Context Clustering

The basic strategy for unsupervised systems is context clustering. A feature vector, or in this scenario a better name would be a context vector, is used to represent each encountered word. The grouping of these vectors represents word sense clusters. The simplest form of such vectors is a standard frequency count of surrounding words. Given the n most frequent words in a text (pruning is usually performed to not consider words that are very infrequent – also done because the number of distinct words can be very large), the context vector of a word w_i would be an n-length array of values, on each position (dimension) having the frequency of that word measured in a certain window (a fixed number of sentences for example).

$$context\ vector\ of\ w_i = (freq_{w_1}^{w_i}, freq_{w_2}^{w_i}, \dots, freq_{w_n}^{w_i})$$

$$freq_{w_j}^{w_i} = frequency\ of\ word\ w_j\ in\ the\ window\ surrounding\ w_i$$

Having 'translated' each word into a context vector, similar such vectors can be grouped into clusters using different similarity metrics. One such metric is the cosine similarity, where the similarity between two vectors is their dot product divided by the norm of each vector. More specifically, it is the sum of the product of their individual dimensions divided by the root of the product between the individual sums of each vector's squared component. The lower the cosine value, the more similar the vectors are.

Applying different clustering algorithms on the collection of context vectors will yield sense clusters [72] [73]. Methods to improve results have been studied, such as applying Latent Semantic Analysis [74] on the matrix obtained from the context vectors (cooccurence matrix). This will reduce the dimensionality and arguably improve the context vectors by reducing ‘noise’ (less relevant words in context). Also, it is helping with the polysemy problem where similar words are so-called ‘merged’ during dimensionality reduction. Another method to improve results is to create better context vectors, such as adding features of the word itself or other external information like glosses taken from a repository such as WordNet [75].

IV.1.2.3. Co-occurrence graphs

Another approach to unsupervised disambiguation is to use co-occurrence graphs. Such a graph has words as nodes and the links between the nodes are syntactic relations extracted from the text in which the words occur [76]. For every word in a text, a graph is built starting from it.

Given the co-occurrence graph (also seen as an adjacency matrix), several algorithms can be applied. [76] proposes both a method to create the graphs as well as using a Markov clustering algorithm to cluster senses together.

Another proposed algorithm is HyperLex [77]. The graph in this instance is created between words that appear in the same paragraph. A word is added only once even if seen multiple times. The edges in this graph are assigned weights representing the co-occurrence of the linked words in the paragraph. The weight is based on the frequency of each word and the frequency of the co-occurrence of the two words, in such a way as two frequently co-occurring words have a weight closer to zero, and infrequently co-occurring words closer to one. Next, hub nodes are selected from the graph based on their connection degree. These hubs represent the ‘senses’ of the word, and are linked to the targeted word itself by virtual zero-weight edges. Next, the Minimum Spanning Tree algorithm is applied starting from the root word, linking all hubs. A score vector is then assigned to the word. The score vector contains on each position a score computed between the word and the hub whose index corresponds to the index in the score vector. This score is a distance based metric, calculating the distance in the MST tree between the word and each hub. Every word receives a score vector. Next, the score vectors are summed and then the hub most relevant to each position (word index) is chosen as the sense for that particular word.

PageRank [78] is another well known algorithm that has been applied to WSD. Given a graph, the degree of a node *$P(V_i)$* is:

$$P(V_i) = (1 - d) + d * \sum_{edges\, V_j \rightarrow V_i} \frac{w_{ji} * P(V_j)}{\sum_{edges\, V_j \rightarrow V_k} w_{jk}} \qquad (25)$$

where *d* is the damping factor and *w* is the weight of an edge.

Agirre [79] applied PageRank to WSD by considering weight *w* as being the co-occurrence probability between two words. Thus, the degrees of every node is computed and the top scoring nodes are picked as hubs, similar to HyperLex. Both HyperLex and PageRank obtained good results when applied to WSD, but still lower than standard supervised approaches.

Co-occurrence graphs can be extended in multidimensional space, where the assumption of a single link between co-occurring words is no longer a limiting factor. This is based on the assumption that that two or more words are usually combined to form a relationship of concepts in the context. Also, planar graph-based approaches fail to model collocations or multi-word terms. [80] proposed such a model. An edge in this multidimensional graph is called a hyperedge and is able to model the relation between multiple words – a hyperedge is a set of vertices. Thus, words are seen as vertices and relations between them as hyperedges. The degree of a vertex becomes the number of hyperedges it belongs to, and the degree of a hyperedge is the number of vertices it contains. Related nouns are grouped into hyperedges that are weighted by calculating *support* and *confidence* parameters. Both parameters are based on frequency functions of co-occurring words. Next, a variant of the HyperLex algorithm is used to select the hubs of the hypergraph, based on the degrees of the vertices. The results of [80] are average, achieving high entropy and purity performance (values that measure different aspects of a system’s performance) measures that outperform the most-frequent-sense baseline, however having low F-Scores.

IV.1.3. Knowledge-Based Disambiguation

Knowledge-based (KB) WSD is a class of methods that uses the knowledge drawn from lexical resources such as dictionaries or thesauri and also from the raw text that is analyzed. Some of the advantages of knowledge based disambiguation are that the scope is generally all-words sense disambiguation, as opposed to corpus based methods that usually restrict the set of candidate words for disambiguation; the target document can be from any source as opposed to supervised methods for example that require a similar annotated document to train on; KB methods do not require annotated documents making them very desirable to apply to other languages for example, or on domains where there are no large corpora to work with.

Regarding knowledge sources, dictionaries provide for every word contained in them a list of meanings, definitions and examples that help clarify the meaning of the word for each of its senses. A thesaurus adds basic relations between word meanings (ex: synonymy relation). The further addition of other relation types and the ordering of concepts in specific forms (like a tree or directed graph) define a semantic network. For example, WordNet organizes the noun synsets into a directed graph in respect to the Is-A relation (hypernymy). Moving to more complex examples, an ontology labels links between classes/entities. A graph is a good representation of an ontology, as the directed edges are the relations between the graph’s nodes, represented by classes/entities. For example, the YAGO ontology (presented earlier) has a graph structure (including cycles) while a tree-like structure extracted from WordNet (actually the WordNet hypernym tree) can be seen as the head of the ‘pyramid’, with every entity in the ontology linking to a leaf of the tree. An ontology has a defining schema. If more of these schemas can be merged, meaning that if correspondences between relations or classes/entities from two different ontologies can be found, then they can be seen as an aggregated ontology. For example, DBpedia[41] is a complex collection of a large number of domain ontologies linked together.

As a classification of knowledge based methods we enumerate the following classic approaches: the Lesk Algorithm, syntactic similarity measures, selectional preferences, other heuristic methods.

IV.1.3.1. Lesk Algorithm

This classic algorithm [81] is one of the first attempts at all-words WSD. Essentially a dictionary-based approach, the Lesk algorithm has provided not only a starting point and a general method of WSD but also a consistent and rather good baseline used for evaluating other systems.

The classic Lesk algorithm tries to identify word senses based on evaluating the overlapping among their sense definitions. Considering *n* words, each word having a number of senses, the algorithm evaluates all combinations of senses, for each combination measuring the overlap in the chosen senses’ definitions. The overlap is calculated as the number of words common in the senses of two or more words. The initial precision reported by Lesk in this initial attempt (1986) was around 50-70%. The dictionary resource used was Oxford’s Advanced Learner’s Dictionary.

An variant of the Lesk algorithm is the use of annealing. Given that the original algorithm tries to evaluate all possible combinations of senses, for many words with many senses this leads to an intractable problem due to the exponential nature of the problem. Simulated annealing is a function that reflects the overlap of a certain choice of senses, with the minimum value corresponding to the correct sense set. In an iterative manner, starting from the most likely sense of each word, one sense from a word is changed to another. The change is kept only if the function has a lower value (meaning a higher overlap). When there is no change in score, the iterations stop and the current sense set is chosen as correct.

Another variant is the simple Lesk algorithm. This approach disambiguates words in turn, instead of considering them all at once. The idea here is to choose the best sense of a word that overlaps with its context. Chosen senses for words do not influence the choice for the

[41] http://wiki.dbpedia.org/Datasets

senses of other words. [82] showed that the simplified Lesk algorithm showed an improvement of 16% over the original algorithm in Senseval-2's all words English task.

Yet another approach involves using augmented semantic spaces [83] where not only the definition of the target word is used, but also the definitions of related words (hyper/hyponyms, holo/meronyms, etc.). On the Senseval-2 task of English nouns this algorithm doubled the precision up to 32% of the original Lesk.

Overall, the best performing variant, both in performance and speed (due to the exponential nature of the original algorithm) is the simplified Lesk.

IV.1.3.2. Semantic Similarity

The semantic similarity approach is based on the premise that words sharing common context have similar senses. Thus, the task of WSD becomes the task of measuring the semantic distance between words and surrounding context.

The semantic measures are based on some type of semantic repository. The most often used such resource is WordNet. The table below offers a fast overview of semantic similarity measures, some of which were presented in an earlier section:

Table 1. Different semantic similarity measures

Measure	Expression	Notes
Resnik (1995)	$IC(C) = -\log(P(C))$ $Sim(C_1, C_2) = IC(LCA(C_1, C_2))$	*IC(C)* is the information content of a concept, usually quantified as the inverse log of the frequency in a corpus *LCA* is the least common ancestor of two concepts C_1, C_2
Agirre and Rigau (1996)	$ConDens(C) = \frac{\sum_i^{\#\ of\ senses\ of\ C} W_i}{desc(C)}$	*desc (C)* is the number of concepts in *C*'s hierarchy sub-tree (where *C* is root) W_i is the weight of a concept in the hierarchy expressed as the number of hyponyms of the concept adjusted by an empirically determined value The sense with the highest *ConDens* is the chosen sense when using this semantic measure in a system. The approach is similar to Lesk's, but using this measure to determine similarity.
Jiang and Conrath's (1997)	$Sim(C_1, C_2) = 2 * IC\big(LCA(C_1, C_2)\big) - IC(C_1) - IC(C_2)$	Improvement over the Resnik similarity measure
Lin (1998)	$Sim(C_1, C_2) = \frac{2 * IC\big(LCA(C_1, C_2)\big)}{IC(C_1) + IC(C_2)}$	Improvement over the Resnik similarity measure

Hirst and St-Onge (1998)	$Sim(C_1, C_2) = C - path(C_1, C_2) - k * d_c$	d_c is the number of direction changes C and k are adjustment constants Path(C_1, C_2) is the path length between the concepts
Leacock (1998)	$Sim(C_1, C_2) = -\log \frac{path(C_1, C_2)}{2 * D}$	D is the depth of the taxomony
Mihalcea and Moldovan (1999)	$Sim(C_1, C_2) = \frac{\sum_i^{H_{12}} W_i}{\log(desc(C_2))}$	H_{12} is the number of common words in the definitions of C_1 and C_2's hierarchies W_i is in this case the depth of the concept in the hierarchy

These measures can be applied in different ways. Due to the fact that in a sentence there can be many words that need disambiguation, the method of calculating overall similarities such as to take into account how any one word influences another is opening an array of different methods on how to actually apply these measures. Two distinct categories emerge when considering context: either local or global.

When considering local context, the general consensus is that a window of fixed size is inspected around each target word. For example, [84] applied the measures above to a data set from Senseval-2 using a context window of size k = 1, meaning adjacent words. The best scoring method was JCN (Jiang and Conrath's), with Hirst and St-Onge's being the most consistent among different words.

Global context relies mainly on another type of approach: lexical chains. A lexical chain is a list of diverse words (word distance is not important) from a text that are related and generate context and continuity in a discourse.

A lexical chain is usually created in the following manner: for each candidate word in a text (most usually nouns) find a suitable lexical chain and add the word to the chain, or else start a new chain. A word is added to a lexical chain if its semantic similarity to the words preexisting in the chain is above a threshold. Several systems that create lexical chains and thus the senses of words have been proposed: Galley et all. [85] obtained a 61% disambiguation precision for a SemCor corpus; also on a SemCor corpus Mihalcea [86] reported a 90% precision and 60% recall using chaining started from anchor words (words that can be reliably annotated with its corresponding meaning).

Using a graph algorithm for sequence data labeling, Mihalcea [87] obtained good results (55% precision) compared to the baseline set by the simplified Lesk algorithm (48.7% precision) on the all-words English Senseval-2 task.

IV.1.3.3. Selectional Preferences

The concept of selectional preferences means detecting the links and relations between words in text, thus constraining the possible meaning of those words. Relations between concepts emerge, based on an array of features like concept class (Picture – Color, where the noun representing the Color class means a color due to the usage in a Picture context), part of speech (verb – noun, where the noun's senses are restricted by the action expressed in the verb), etc.

As with most approaches, learning selectional preferences depends on the amount of training data. The more senses are encoded/annotated in a text, the larger amount of selectional preferences can be extracted and the better the performance of the WSD.

The simplest learnable constraints in a text can be word-to-word facts. Such a fact can be expressed as the frequency of count of two words that stand in a relation. Two words w_1 and w_2 that stand in relation R are expressed as $cnt(w_1,w_2,R)$. Extending to conditional probabilities, considering that w1 depends on w2 is expressed as:

$$P(w_1|w_2,R) = cnt(w_1,w_2,R)/cnt(w_2,R) \tag{26}$$

If considering semantic classes (suggested initially by Resnik in this thesis [88]) then selectional associations can be the measure of the semantic fit between a word w and a semantic class C. If the word is linked to the class by a relation R, then the conditional probability of the class C dependent on the word w is:

$$P(C|w,R) = cnt(w,C,R)/cnt(w,R) \tag{27}$$

where $cnt(w,C,R) = \sum_{every\ w_C \in C} \frac{cnt(w,w_C,R)}{cnt(w_C)}$, then the selectional association is:

$$assoc(w,C,R) = \frac{P(C|w,R) * \log\left(P(C|w,R)/P(C)\right)}{\sum_C P(C|w,R) * \log\left(P(C|w,R)/P(C)\right)} \tag{28}$$

Extending even further to class-to-class selectional preference Agirre and Martinez [89] propose a measure that tries to maximize the co-occurrence of the class of a target word with the class of its co-occurred word. The measure is rather complex and involves the calculation of several word-to-word conditional probabilities, and then choosing the maximum scoring choices.

[89] tested these measures and discovered that on the noun data set of SemCor the class-to-class disambiguation works significantly better than word-to-word and word-to-class, but still under the most frequent sense baseline. Other systems have been implemented, including an unsupervised WSD system [90] that learned selectional preferences without a sense-tagged corpus, but none exceeded the baseline.

IV.1.3.4. Heuristic Methods

The first and very basic heuristic method is the most frequent sense. For several reasons this heuristic is used as a baseline in WSD systems. It is based on the observation that while words have several meanings, there always is one meaning that is used more often than the others. Thus, for every word a most-frequent-sense can be determined by simply counting the senses frequency of that word on a corpus as large as possible. Because of its simplicity and ease of implementation, the most-frequent-sense is the baseline that other systems should exceed [91]. However, this heuristic does have limits. For example, if considering domains, the sense distribution for each word changes increasingly with the specialization of the domain. Also, the measure is dependent on the available annotated corpus used to extract sense information. The larger the corpus, the better the frequency distribution (considering that this method has a clearly established upper bound performance).

Another used heuristic is the one-sense-per-discourse [92], stating that a word tends to preserve the same sense in the entire discourse. This immediately simplifies the problem of WSD, meaning that from several appearances of a word, the clearest disambiguation (the chosen sense should have the highest confidence among other candidate senses) is chosen as the sense for every appearance of the word. In the rather small experiment, [92] showed a 98% correlation between the appearances of a word and its senses. The experiment covered only words with two senses. This result showed that the heuristic was rather good. However, in a later experiment where words were allowed more than two senses [93], the hypothesis that a word will mean the same thing obtained a poorer score, where a third of the words were found to have different senses in the same discourse. This leads to the conclusion that where fine-grained word disambiguation is concerned, this heuristic can actually decrease a system's performance.

Scaling down the one-sense-per-discourse heuristic yields the one-sense-per-collocation, where the assumption made is that a word keeps its sense when used in the same collocation. The correlation between a target word and its context is strong, with the strongest links being to the words closest to the target word. However, similar to the previous heuristic, the more senses a word has, the worse this assumption holds.

IV.1.4. WSD Bounds

The lower and upper bounds are measures that are needed to evaluate the performance of a system.

The lower bound is the minimum acceptable performance for any system. An example of such a measure is the random baseline, where the choices for senses are made randomly. Any system should outperform this baseline. A more difficult baseline is the first sense, where the most frequent sense of every word is always chosen. This lower bound is actually quite difficult to exceed.

The upper bound is the maximum performance that a system could obtain. The upper bound surprisingly is not a 100% score for every WSD system performance measure. The standard upper bound usually chosen is the ITA – Inter Annotator Agreement, the percent of word senses that human annotators (at least 2) agree upon given a text to be sense-disambiguated. For coarse grained tasks (few senses per word) the percent is rather high, reaching 90% [91]. For finer grained tasks (many senses per word, such as the senses in dictionaries or WordNet for example) the percent drops in the 60-80% range [94].

One of the big problems for WSD is the granularity of senses. For example, using WordNet for sense inventory, a Senseval-3 system obtained 65% accuracy on the all-words English task [95]. This performance raises questions on both sides: the performance is rather good given the upper bound ITA; on the other side the low ITA means that there might be a problem with the senses definitions themselves – if humans cannot exceed a certain percent then maybe the fine-grained WSD problem should be redefined.

The ITA upper bound raises interesting questions [96] such as what happens if a system exceeds the ITA bound, and is better than human annotators, especially for the fine-grained task where the ITA score is not very high.

A different upper bound is considered to be the 'oracle bound'. Such a system always knows the correct sense for every word out of the available senses. In systems implementing multiple WSD sub-systems, its accuracy is determined by the number of word instances for which at least one of the systems outputs the correct sense.

IV.1.5. Evaluation metrics

To evaluate a WSD system a few standard metrics are used:

Precision is percentage of words that are tagged correctly out of the words addressed by the system.

Recall is the percentage of words that are tagged correctly out of all words in the test set. Specifically to the WSD domain, recall is also called *accuracy*.

Coverage is the percentage of words that the system has evaluated out of all words in the test set.

Example: If a system has 100 words to evaluate, out of which it attempts only 75 and correctly disambiguates 50, then the precision *P* will be 50 / 75 = 66% while recall *R* will be 50 / 100 = 50%. This system's coverage *C* is 75 / 100 = 75%.

It can be seen that if coverage C is 100% then $P = R$, else, R will always be smaller or at most equal to P.

The classic F score in WSD is usually the F_1 measure defined as: $F_1 = \frac{2PR}{P+R}$, obtained from the general F_α measure, where $F_\alpha = \frac{1}{\alpha\frac{1}{P}+(1-\alpha)\frac{1}{R}} = \frac{(\beta^2+1)PR}{\beta^2 P+R}$, with $\alpha = \frac{1}{\beta^2+1}$. Choosing $\beta = 1$ balances precision and recall, obtaining the F_1 metric. This is a good measure for systems with less than 100% coverage. However the integrated F measure can hide a very bad precision or recall. If either P or R is almost 100% while the other is close to zero, then the F measure will still be around 50%.

As [96] summarizes, the F measure is not always a good indicator of system performance. [97] proposed an evaluation metric that if a system performs a wrong classification, then it should be penalized on the distance of choice it has made to the correct sense. If the chosen wrong sense is a fine-grained distinction of the correct sense then the system should be penalized less than if it had chosen a sense which was very far from the correct sense (a coarse grained distinction). Other methods have been proposed, but because most systems are evaluated on the precision, recall and F_1 measures, then subsequent systems will also use the same measures, to have a common ground on which to be able to compare to earlier attempts.

IV.2. Named Entity Recognition

Named Entity Recognition (NER) is a task in the area of Information Extraction that refers to the identification of certain entities in a text. A rather recent development as a stand-alone task [98], the Message Understanding Conference 6 (MUC, 6th edition, 1996) outlined the need to entity identification as a needed component for better IE systems. Initially termed Named Entity Recognition and Classification, the task handles recognition of persons, locations, organizations, etc. as well as certain numeric values such as dates or money amounts.

Even though the term was ‘officially’ used for the first time in 1996, works that undertook subsets of NER were published in the early 90s. Initial attempts to detect restricted categories of entities such as company names [99] slowly evolved to more and more complex systems, and with the formalization of the task in the MUC conference, NER research gained speed.

Before presenting the main approaches to NER we will present some of the aspects of the task. The first and most important aspect is the entity types searched for and used. As the name of the task suggests (*Named* Entity Recognition), the targeted entities are primarily proper names, most often proper nouns and/or capitalized words (ignoring common nouns that start a sentence).

Traditionally, there are three large categories that a named entity could be: a person, a location or an organization. These three categories were proposed as initial designators for entities at MUC conferences, and they have remained in use unchanged so far, most systems being developed to detect these three categories. There is also a forth category, which is the ‘other’ category, or the miscellaneous category, encompassing named entities that do not fall in the other three. Systems that further specify this very simple division exists, trying to detect fine-grained entity recognition to subclasses such as musician, poet, writer for the people category, or village, city, state, country, continent for the location category [100].

Other types of entities suitable for detection with a NER system are dates, time, money, percents. These types are accepted as candidates for the majority of systems; there are however some purpose-built systems that detect fringe-entities like phone numbers, email addresses [101], person titles (detect from “Dr. Eng. Smith” that person Smith has the Doctor and Engineer titles), movie or literature titles, job titles [102] and so on. Also, the biomedical domain, one of the most active sub-domains, has proposed a large number of systems that detect domain entities like proteins or drugs in medical text [103].

Extending the entity range even further, the “open” NER proposes the idea of unrestricted entity type, meaning a fine-grained recognition of entities, down to very specific categories like truck, car, sports-car, convertible, etc [104]. This approach requires the predefinition of the fine-grained categories thus requiring an ontology/taxonomy to represent them.

Another aspect regarding NER is the language. By far, English is the most studied language regarding NER systems. Recently however many languages attract attention, like German or Arabic. Special tasks in domain workshops like CONLL or MUC have Chinese or Japanese NER tasks.

Also, the genre of the analyzed text is important. The development of systems for specific genres such as scientific texts versus standard news articles (for example) shows a sensible performance difference [105], marking the text’s genre as an important aspect of a NER system.

IV.2.1. Classification of NER Approaches

IV.2.1.1. Supervised Learning

Just like in WSD, supervised learning is the method currently most often used in NER. It is relatively straightforward to apply, requiring the modelization of the problem to fit a certain supervised learning algorithm. As WSD, the core idea is to extract a number of features for every entity and then apply a classification algorithm. Virtually all classifier types that were suitable to model the NER problem were experimented with.

Initial attempts to model the NER problem as a Hidden Markov Model proved relatively successful [106]. A Markov Model is a stochastic model in which the Markov property is assumed: the present state of the system does not depend on the past or the future states. In a HMM the named entity recognition problem is seen as a Markov process with unobserved hidden states. While in a normal Markov Model the states are directly observable with the state transition probabilities as the only parameters, in the HMM the states cannot be seen, only the output is observable, which is dependent on the states generating it – the states through which the model passes are 'hidden'.

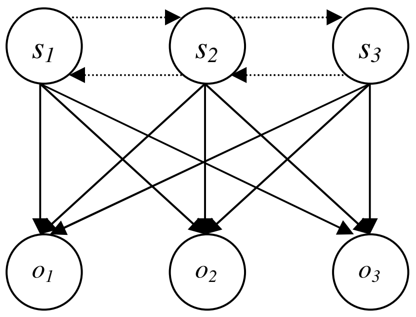


Figure 6. A Hidden Markov Model example

where s_i are the states the system passes through, o_i are the outputs of the system, the dotted arrows between the states are the state transition probabilities and the arrows linking the states to the outputs are the output probabilities.

A HMM system tries to find, given an output sequence the most likely set of state transitions and output probabilities. As such, given a set of labeled entities and a number of features for each one, the system determines the most likely parameters that output each label in turn, and then applies the trained model to new unseen entities.

Decision trees, presented in the WSD section can also be applied to NER. Learning to discriminate among features to obtain the label of an entity as the leaf in the learning tree was also applied with limited success [107]. This approach works best with a limited number of entity categories, and depends greatly on the extracted features.

Maximum Entropy Models, also known as multinomial logistic regression models, implement a regression model that generalizes logistic regression by allowing several possible discrete outcomes. Unlike generic regression models which estimate continuous values, when the output variables are nominal (categories), the ME model is used instead. This model does not place independence restrictions on the independent variables (input variables / entity features) like the Naïve Bayes, thus allowing a more realistic modelization and accepting sacrificing the tractability of the problem for large feature spaces. One of the first uses of the ME model applied to NER was in the 7th edition of the Message Understanding Conference [108].

SVMs, presented in detail in chapter II.2.1. are among the best classifiers used in the NER field. Together with CRFs (chapter II.2.2.) represent the state-of-the-art supervised learning techniques applied to NER. For example, the NER system used in this work based on CRFs is the Stanford Named Entity Recognizer[42] [109] that in the CoNNL 2003 English news dataset obtained 92.15% precision and 92.39% recall.

IV.2.1.2. Semi-supervised Learning

Semi-supervised learning started from the observation that to be successful, supervised systems require large training sets that are difficult to create. Thus, different techniques and approaches have been developed that circumvent to a degree the training-data size issue. One such method is the 'bootstrapping' where initial seeds are given to start the learning process. The seeds might take many forms, for example known names for organizations. The system then searches on a corpus of data for sentences containing the seeds and features surrounding them, extracting patterns. After a system is sufficiently certain that the pattern is valid, it applies that pattern to extract more organization names. After a large enough number of iterations, the system has collected a training corpus of sufficient size for the learning process. The main idea is that allowing a certain variation on the valid patterns, new entities and more importantly, new contexts will be discovered after a number of iterations.

There are many ways to detect such patterns. Regular expressions are one of the first techniques applied [110], along with using syntactic features such as part of speech and noun phrase analysis [111]. Mutual bootstrapping consists of a set of entities and contexts (patterns) that each grown in size in turn. Initially starting from a number of seeds, all found contexts are kept, and then using those contexts in turn new seeds are found, and so on. Such an algorithm is very sensitive to noise [112].

Other approaches use NER systems to generate initial seeds [113]. Use of syntactic relations instead of RegEx (regular expression) patterns is also a possibility. A very interesting showcase of the use of semantically related words is performed by [114] where allowing patterns containing words in the same semantic class a precision of 88% was achieved when applying their system on the web, starting from only 10 seed pairs.

IV.2.1.3. Unsupervised Learning

NER approaches using unsupervised learning are similar to the WSD approaches using the same unsupervised methods: attempt to group together entities that are similar. As such, entities that are found in similar contexts will be grouped together. Another way to evaluate the similarity of entities is to use their semantic type thus requiring lexical resources like WordNet. [104] attempted to use the semantic entity types gathered from WordNet, where

[42] http://nlp.stanford.edu/ner/index.shtml

to every top level WordNet class a certain topic was assigned to by counting the co-occurring frequency of the word in a corpus. Then, for every target entity in a document the context is analyzed and depending on the words found in the context, the most likely WordNet class is assigned.

Another unsupervised learning method is to detect named entity hyponyms and hypernyms. This is accomplished by identifying patterns that indicate hypo/hypernym relations (ex: “A such as B” indicates that A is a hypernym of B) [115].

IV.2.2. Named Entity detection and recognition techniques

This section reviews the major features used in NER systems to detect and tag the targeted entities. [116] provides a very good overview of the feature space. A feature is a unit of information about a targeted word/entity. It can be a number, a Boolean value, a nominal value or a string. For example, the POS tag is a nominal value because it is a value belonging to a restricted set of possible POS tags. The length of a word (the number of characters) is represented as a number. The fact that an entity is the first word in a sentence is represented a Boolean true/false value.

IV.2.2.1. Word Features

Word Case – usually a Boolean value signaling if the word is capitalized, if contains all uppercase or lowercase letters, or a mixture of both.

Punctuation – also usually a Boolean value signaling if the word ends with a punctuation mark, or if it contains punctuation marks inside the word (ex: “Dr. Smith”)

Digit – a Boolean value if the word is composed only of number; if the word contains numbers inside, etc; special patterns can indicate that a number is a date, a year, a zip code, a phone number, or more specific cases such as an IP address.

Part-of-Speech – the POS tag is usually represented as a nominal value.

Character – if it is a single character. In context maybe it represents first person pronoun, for example.

Word-form – there are several features that can be used, such as the word suffix and prefix (string values), singular form (plural→ singular); in case the word is a verb then stemming can be applied to obtain the root form, etc.

Word-type – this category also contains several possible features. For multi-word tokens the word count is a possible feature. The lowercase and the uppercase versions of the word, non-alphanumeric characters and n-grams are also good features. Pattern features are also found in this category. Pattern features can be used to encode character types such as using a character for all uppercase letters, another character for lowercase, yet others for all punctuation or numerical characters.

IV.2.2.2. List Features

List features form a different category of features altogether. Lists (also known as dictionaries or gazetteers) contain an enumeration of words belonging to some category. The simplest examples can be the list of months in the year, the list of popular English names, the list of capital cities, the list of countries, etc. Below are summarized possible list types:

General lists – these lists contain usual generic information, like the days in a week or stop-words. Other larger general lists include common nouns or common verbs.

Lists of Entities – in this category the lists contain actual named entities, such as city names, continent names, organization names, governments, shop names, airport names, etc.

List of Entity Cues – these lists contain words that are frequently found to indicate a certain entity type, such as “Dr.” indicating a person or “Inc.” indicating and organization.

An interesting aspect of the list pointed out by [116] is the way of using such lists. As simple match on one or more of the elements in these lists is too restrictive, other approaches have been used. The first approach involves stemming words (removing prefixes and suffixes) and reducing reasonably similar characters to the standard ASCII (or English) character set, meaning accepting letters as *ă* or *è* to *a* and *e*. Another approach is to use distance-based metrics between strings. There are several string distance measuring algorithms available such as the Hamming Distance, the Levenshtein Distance, Smith-Waterman Distance, Jaro-Winkler Distance, cosine similarity or even simple Euclidian Distance. Another interesting string distance is the SoundEx Distance which is actually a phonetic algorithm [117] that indexes sounds as they are pronounced in English. Therefore, based on the difference in the sound of two similarly sounding strings a distance metric can thus be used to pick fuzzy matches in a NER list.

IV.2.2.3. Corpus Features

This category contains document-wide dependent features.

Multiple occurrences or references – If in a document a word appears both as a common word and as a capitalized word, it is classified as common word that sometimes appears as ambiguous due to its position at the start of a sentence, for example [118]. Another feature in this category is entity coreference. The task of coreference resolution is a difficult task in itself; however, having a hypothetical system that can tell whether a target entity is actually a reference to a previously found entity would provide a large amount of information for the target entity.

Localized syntax – possible features include Boolean values that signal whether the target entity is in apposition with another adjacent entity, it is part of an enumeration; another feature can be the actual position of the entity relative to the sentence /phrase or document beginning.

Metadata – depending on the document, metadata can be extracted from the document itself. If the document is an e-mail, then usually the From: field is a good indicator of person names. If a document contains tables or figures, then the description below them can provide clues about the types of entities enumerated.

Frequency Measures – the simple word frequency count is another numeric feature useful for a NER system. The frequency can be normalized across all words and documents for example (apply standard TF-IDF). Frequency count can also be applied to non-standard words (multi-word tokens or very long words, for example). Such ‘special’ words will be considered for as candidate entities for the NER system.

IV.2.3. Evaluation Metrics

Evaluation of a NER system is a rather complex issue, because there are many cases of partial errors whose scores are debatable. Here, precision and recall have a different meaning than when used in the WSD context. A NER system will mark an entity by borders (thus delimiting the entity – single or multi-word – from the other adjacent words) and also specify the type of that entity. For example, some of the errors a NER system will output are missing to identify an entity or identifying an entity where none should be found; assigning a wrong type to an entity; misplacing the borders – either including other extra words or not including all the entity’s words; or any combination of the above errors.

A first evaluation method proposed by the MUC conference divides the attempt to score a system in two categories: finding exact entities (entity boundary) and finding exact categories for the entities. A positive score is assigned to the category choice if the category is correctly assigned, even if the boundaries are not exact. Similarly, a positive score is assigned if the boundary assignment is exact, regardless of the category assigned. For each of the two distinct aspects, the following measures are proposed: the number of correct entities (correct identification, both for boundary and category), the number of identified entities by the system and the number of possible entities in the solution. Precision is calculated as the number of correct entities divided by the number of number of identified entities by the system. Recall is the number of correct entities divided by the number of possible entities in the solution. An F-measure is proposed which is the harmonic mean of precision and recall for both aspects (boundary and category). The harmonic mean is used because it tends to minimize the influence of large and small values.

Another more complex evaluation method is the ACE[43] evaluation. Due to the fact that the NER task in the ACE setting involves finer-grained entity categories, coreferences, etc, it implements measures for partial matching and partial credit for errors and so on. Here, the initial score is 100% out of which a certain value is deducted for every mistake the system makes. For example, the score calculated for correct identification of entity category depends on the category type (ex: a correctly recognized person scores differently than a correctly identified location or organization); all the entities' aspects contribute to the 100 score. Partial score is taken for missed entities, border mismatch, category misclassification. For each error class more subtle rules are used: for example, for border mismatch, an allowable mismatch is if an entity's head matches on a minimum amount of characters. The ACE scoring method is on one hand very customizable and complete, but on the other, due to the number of customizable parameters it is difficult to implement and use as a common scoring method amongst NER systems not developed specifically for the ACE task.

The last method of NER evaluation is the simple, strict match. This evaluation method is used by CoNNL[44] to evaluate its systems. Here an entity is given points for correct recognition only if the borders are a perfect match and the chosen category is also correct. Precision is in this case the number of correct entities divided by the number of entities found by the system, while recall is the number of entities found by the system divided by the total number of entities.

IV.3. General Named Entity Recognition

In a very broad sense, both Word Sense Disambiguation and Named Entity Recognition have the task to identify and clarify the sense of words. Whether to determine if the word 'engine' in "The engine is broken" refers to a mechanical engine or a locomotive (WSD), or to determine that 'Santa Fe' in "I drove the new Santa Fe in Santa Fe" refers to a car and then a city (NER), in both cases the overall aim is to determine words' meanings. There are differences between WSD and NER, clearly defined in the ACE, MUC, CoNNL and in many other Natural Language Processing / Information Retrieval conferences. Because of those particular differences, currently the problem of identifying words is split into WSD and NER domains and almost never treated as a single entity.

Alfonseca & Mandahar try to define the WSD/NER recognition problem (and a unified approach) in their paper [104], where they define the term "General Named Entity

[43] Automatic Content Extraction, http://www.itl.nist.gov/iad/mig//tests/ace/ , with the latest edition in 2008
[44] Conference on Natural Language Learning, http://ifarm.nl/signll/conll/

Recognition” (GNER) in the context of an existing knowledge source used for sense repository. Given an ontology O having a set of concepts C (person, country, etc), a set of instances I of those concepts (Ann, France, etc) and a hypernymy function $h: C \cup I \rightarrow C$ that determines a taxonomy of instances and concepts, then the task of GNER is “the task of identifying, for an unknown concept or instance u, the correct concept $c \in C$ such that $h(u) = c$, i.e. consisting of finding the most accurate immediate generalization of u in the known hierarchy of concepts. “ [104].

The relation of GNER to NER (as seen by Alfonseca & Mandahar) is that NER is a restricted task compared to GNER, having a flat hierarchy and containing relatively few concepts whereas GNER has a taxonomy of fine-grained concepts. Regarding WSD, they also consider it a more restricted task than GNER. In GNER the task is to find the synset that matches the best meaning of the word, where WSD tries to find the synsets containing that particular lexical word. GNER is seen as a “task that covers, and is harder than both Named Entity Recognition and Word Sense Disambiguation”.

The system they propose is based on the work of Yarowksy [119] and Agirre [120] and involves first collecting topic signatures for every WordNet concept using an unsupervised algorithm. For every synset the algorithm generates a query containing the words in the synset, the hyponyms as positive keywords and words in other synsets that contain the same words as negative keywords. The query is sent to a search engine and the responses are analyzed, counting the frequencies of the words that appear in each initial word’s window context. After a cleaning and scoring step, the algorithm obtains a frequency count for co-occurring words for every WordNet concept (a topic signature). Alfonseca & Mandahar then use the topic signatures to calculate the similarity of new unknown concepts to the existing topic signatures using a top-down approach in the concept hierarchy. For an unknown concept u its topic signature is obtained using the same method as for the existing WordNet concepts. Then, at each level of the taxonomy the concept whose signature is closest to u’s is selected. If none of the selected concept’s children have a higher similarity score then the currently selected concept is the concept that is assigned to u. The similarity metric used is the dot product of vectors, here topic signatures.

They tested the system using a small, domain specific taxonomy and have obtained some interesting results. However, because there is no other similar system to compare them to, the overall system performance cannot be determined. They have discussed the problems of topic specificity (where for example, some concepts are too general for use, like the ‘person’ or ‘location’ concepts – too many sub-concepts linking to them), the context window size (small is apparently for this task better, because large windows introduce noise words), and so on. The system can be used to extend or even create an ontology (at least concerning the hypernym hierarchy).

Another closer ‘unified’ view of WSD and NER is the Super-Sense Tagger (a SST). A SST, just like NER and WSD is a Natural Language Processing task where significant entities (nouns, verbs, etc) are annotated with super-senses [121] from a taxonomy (most often WordNet). A super-sense is a higher level class from the taxonomy. Compared to WSD it is an easier task as the higher level senses are more fine-grained; compared to NER is a more difficult task as there are more super-senses than the usually very few categories a NER system deals with.

A SST system can have many forms of implementation. For example, Ciaramita and Altun [122] developed a system that annotates nouns and verbs with 41 WordNet super-senses. They modeled the problem as a sequential labeling task and have implemented a discriminatively trained Hidden Markov Model, showing better results than the baseline on SemCor and Senseval corpora. The baseline for this task is the super-sense of the most frequent synset for a target word. They obtained a 11% improvement on the SemCor corpus over the 66% f-score of the baseline, and a 6% improvement over the 64% baseline for the Senseval-3 corpus.

V. A General Entity Recognition (GER) System

The Web is currently the most used information source world-wide. New content is added every day, in ever increasing amounts. However, the vast majority of this content is added in an unstructured manner. Current search engines build increasingly larger indexes of websites to allow access to this content. But current Information Retrieval methods are starting to show their limits given the information amount or when subjected to very specific user queries, and new methods to quickly obtain information are requested. The Semantic Web promises relevant information delivered fast, and in the format the user desires. This means that computers need to 'understand' to some degree the information they store and process. The field of Information Extraction (IE) takes on the task of extracting information from existing sources, be they unstructured (free text, books, news articles), semi-structured (XML, structured web pages like Wikipedia) or structured sources (databases) and then translating this information in a computer understandable structured format that the machine can process. One basic form to store this information so that it can be easily processed by the computer is in the form of simple entity-relation tuples (subject-predicate-object).

As such, some of main tasks in IE are entity and relation detection and identification.

This chapter presents **an approach to a sub-task of IE,** namely **identification and correct assignation of predefined ontological classes to entities found in free text**. We present an **unsupervised**, **knowledge-rich system** that, **given natural text as input will extract relevant entities from it** (both common and named entities) **and will assign to each extracted entity a class found in an ontology**. From a certain point of view it may seem comparable to a fine-grained, partially targeted word sense disambiguation (WSD) problem [96], or even partially as the WSD sub-problem of word sense discrimination [123] .

The entire system is driven by the idea that entities are defined by their context. A single entity can mean multiple things, but when put in context its meaning becomes clear. Context in this case means some form of directional logical link from an entity to another, as each entity specifies every other to some degree. Extending a classic example [123], when saying "the *bank* in *Paris*", *bank* defines *Paris*, and *Paris* defines *bank*. Individually, "*Paris*" could mean the capital of France, the singer or even the historic Greek figure, and "*bank*" could mean a monetary institution, a school of fish, a flight maneuver or the side of a river. Put together, their meaning becomes clearer. Paris can no longer be a person, and bank can no longer be a flight maneuver. Adding another entity as "*accounts* opened at the *bank* in *Paris*" will then clearly specify every entity, including *bank* which represents a monetary institution and not the bank of a river possibly named Paris, even without looking at the words linking them such as verbs, prepositions, etc.

We rely on the fact that in order to connect entities together, this information has to already exist in some form of knowledge repository. Ontologies match our desired repository structure, as an ontology is at its core a type of graph that interconnects entities. In the system presented in this chapter we will use a generic, large scale ontology that encompasses both named and common entities – the YAGO ontology.

Before starting a formal description, to better understand the aim of the system we present the following example showing the inputs and outputs of the system.

Example: Let us consider an **input document** (unstructured, natural language text). For simplicity, let's assume the document is composed of only one sentence:

Document: "He replaced the <u>pipe</u> giving his <u>car</u> new <u>life</u> - his <u>Santa Fe</u> now runs quieter."

Given this document, the system will **identify interesting entities** (both named and common nouns – shown underlined) and **assign to each entity a suitable class from an ontology**. The system will **output pairs of entities identified in the text with their assigned entity from the ontology**:

"pipe" → `wordnet_exhaust_pipe_103303510`

"car" → `wordnet_car_102958343`

"life" → `wordnet_life_115140405`

"Santa Fe" → `Hyundai_Santa_Fe`

On the left we have interesting entities extracted from the text while on the right we have canonic entities existing in the YAGO ontology (presented in section III.1.3, YAGO also integrates WordNet classes thus covering both common and named entities). The system attempts in a generalized manner the task of WSD (handling common nouns, ex: determining that the correct sense of "car" here is identified by id #102958343 out of the other possible senses of the word "car") and NER (handling proper nouns, ex: determining that in this sentence "Santa Fe" represents a car, and not other similarly named entities like the town Santa Fe in USA).

The content of this chapter is structured as follows: we start with a system overview, referring to other partially similar systems, discussing the architecture of our proposed system and then presenting a formalization of our stated problem. At this point it is noticeable that the system can be divided into two major components: the Linker Algorithm (a custom graph algorithm we named "Linker Algorithm") and the supporting system. Even though the algorithm is the last processing step of the system it will be presented first because it needs to be characterized out of the context of the current system, from an abstract mathematical point of view. We then return to the supporting system, present its implementation and close the chapter with performance evaluation and conclusions.

V.1. System overview

Given free text in the form of sentences written in natural language, we aim to detect relevant entities and then identify them to matching classes in an ontology. For example, for the sentence "Einstein's theories are discussed by Kaku in his latest book." we would like to detect that Einstein is an entity and that it refers to Albert Einstein, Kaku is also an entity and it refers to physicist Michio Kaku, and also that the common nouns "theories" and "book" are identified as a scientific theory or at least a general theory and a literature book respectively.

We will attempt to do so using graph algorithms applied on an ontology which represents our knowledge source.

One useful feature of the system is that all results are accompanied by their semantic justification graph composed of the path between entities (including relation types and intermediate entities). This justification graph can be used for further result evaluation, manual or automatic, similar to [124] [125].

Our purpose could be interpreted as a fine-grained all-word disambiguation (WSD) problem, similar to some of the tasks presented in past MUC/SensEval[45] challenges. From a certain point of view, we aim at exactly that: given a text and a knowledge source, assign to each word a class from the knowledge source. However, there are differences: 1) for example while we look at both named entities and common entities (closer to targeted WSD), we do not take into account verbs or other modifiers, and focus only on nouns (named or common). 2) we use a generic ontology that contains millions of possible entities to choose from instead of a small, restricted set. 3) the aim of the system is for its results to be further used in conjunction with other methods or systems (ex: relation detection, machine learning methods) to provide, for example, full ontological facts. This is even more relevant as we allow for unknown entities (entities from the text that have not been assigned an entity from the ontology) to exist in valid result sets, thus allowing them to be used in new extracted facts to gather information about previously unknown entities.

We shortly review some of the closest methods and techniques our system is related to in the area of knowledge-based methods for WSD [123] [126].

One approach is to determine the overlap of sense definitions. Also known as the Lesk algorithm [81], the similarity between a pair of words is calculated as the highest overlap of words definitions. In some sense it is related to our algorithm problem: as the number of words increases linearly, the computational problem increases exponentially having to consider every possible sense combination for all entities. Another approach is by selectional preferences. These are constraints on the type of words that can stand next to

[45] http://www-nlpir.nist.gov/related_projects/muc , http://www.senseval.org

another. Word to word measures are computed using frequency count on a corpus or other methods [127]. For other, more interesting word to class or class to class (a problem we actually face when evaluating results), large corpora are parsed and frequency together with words' semantic classes provide a way to select a preferred class or word. This approach however yields poorer results than Lesk's algorithm [89] but is interesting for its class to class selection feature that could be applied as a final result selector for our system.

Another category is structural approaches, divided into similarity and graph-based methods. Similarity methods propose methods to assign a score to different words based on the structure of the graph, for example measuring WordNet hypernym edge distances between words (not much unlike our own scoring method) [128]. Many other metrics have been proposed, including distance and information content based metrics. The second category of graph-based methods exploits the structure of graphs itself. However, most of these approaches [129] [130] focus on lexical chains (structures of semantically related words), an approach different from ours.

Overall, knowledge-based systems usually have a somewhat poorer performance than fully supervised machine learning algorithms. However, they do benefit from a wider coverage due to the general, large knowledge sources they exploit [96].

Compared to Alfonseca & Mandahar's system for General Named Entity Recognition [104] (described in more detail in the previous chapter) the proposed system has as a common feature that it targets both common nouns and proper nouns. On the other hand, there are two major differences stemming from the knowledge source used and method. On one hand Alfonseca & Mandahar's system used WordNet to tag words while our proposed system uses YAGO as a tag repository. The current version of YAGO (v1) we use contains about 70.000 WordNet classes from about 2 million + classes, yielding a tag space almost 30 times larger. Another difference is that the former system tags named entities with WordNet tags (ex: Bucharest/`wordnet_city`) while the latter system tags named entities with actual instances of WordNet classes (ex: Bucharest/`Bucharest`). The second major difference is the way tags are assigned. While both systems are unsupervised and knowledge-rich in their approach, the former system uses frequency counts of co-occurring words to create WordNet topic signatures while the latter uses graph-based methods to determine the most likely tags for groups of related words.

V.1.1. Architecture

The proposed system is structured as presented in the figure below. The diagram shows a natural sequential flow of the component modules that operate on the input document. Step by step the document (Input) is split into sentences then tokens; the tokens are analyzed and merged, if necessary, into multi-word tokens (Module A). The ontology is consulted for possible entities that could represent the words identified in the document (Module B.1).

The influence of the extracted entities on each other is captured into an influence matrix (Module B.2). A graph is created based on the ontology itself and the possible entities extracted from it (Module C.1). The entire entity group is split into smaller manageable groups (Module C.2). A custom graph algorithm (Linker Algorithm) has been developed that, applied to each entity group, will detect the strongest connected entities (Module C.3), which is the final result of the system (Output).

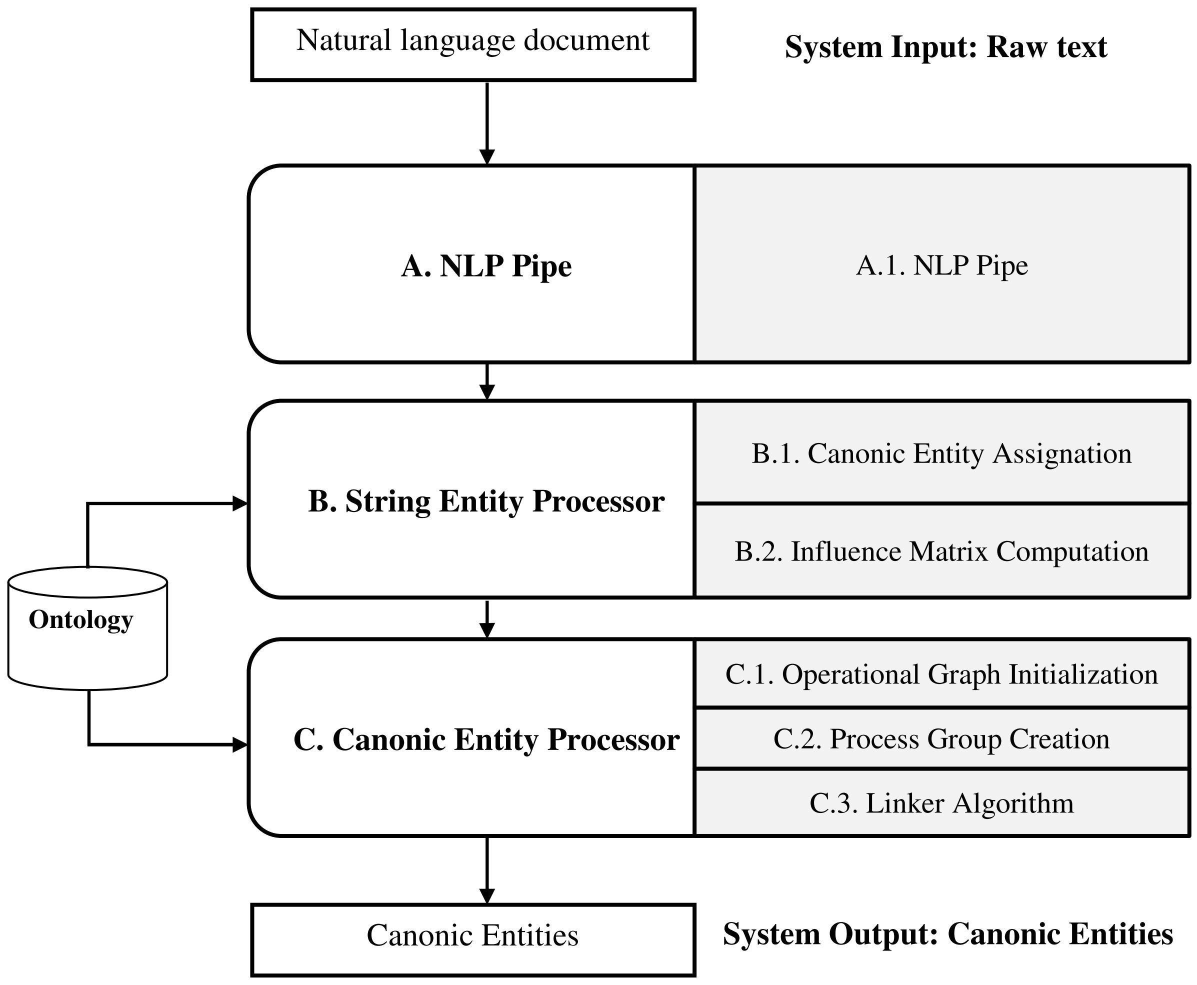

Natural language document
System Input: Raw text
A. NLP Pipe
A.1. NLP Pipe
B. String Entity Processor
B.1. Canonic Entity Assignation
B.2. Influence Matrix Computation
Ontology
C. Canonic Entity Processor
C.1. Operational Graph Initialization
C.2. Process Group Creation
C.3. Linker Algorithm
Canonic Entities
System Output: Canonic Entities

Figure 7. Logical architecture of the proposed system

V.2. Formalization

To understand the notations used in the rest of the chapter we will formalize the problem, explaining its individual components.

The input of the system is a natural language, free text **document (*DOCUMENT*)**, composed of a number of **sentences (*SENTENCE*)**.

$$DOCUMENT = \{SENTENCE_k \mid k \text{ is the number of sentences}\} \tag{29}$$

Each sentence is in turn composed of individual **tokens (*TOKEN*)**

$$SENTENCE = \{TOKEN_k \mid k \; is \; the \; number \; of \; tokens \; in \; the \; sentence\} \qquad (30)$$

Example: a valid *DOCUMENT* composed of two sentences could be “Einstein visited Ulm. The ship sailed towards the Bering Strait.”.

The first module of the system analyzes each sentence and its tokens, and extracts a number of **String Entities (*SE*)**. The String Entities are composed of either a single token or from multiple adjacent tokens (for example in the case of names that contain a first and a last name).

$$SENTENCE = \{SE_k \mid k \; is \; the \; number \; of \; string \; entities \; in \; the \; sentence\} \qquad (31)$$

Example: for $SENTENCE_2$ = “The ship sailed towards the Bering Strait.” we detect 7 tokens (individual words). After analysis we detect that token “ship” is a noun and create String Entity SE_1 = “ship”. We also detect that tokens 6 and 7 are proper nouns and can be merged into a single String Entity, SE_2 = “Bering Strait”. Thus this sentence can be represented as $SENTENCE_2 = \{SE_1, SE_2\}$.

We now introduce the concept of **Canonic Entity (*CE*)**, in contrast to the String Entity. While the String Entities extracted from sentences are just that, bounded sequences of characters, entities in the ontology will be called Canonic Entities. They are clearly defined, immutable entities linked by several relations between them that generate a certain semantic structure.

Next, we define the concept of a **Set of Canonic Entities (*PCE*)**. We assign to each String Entity SE_k a list of Canonic Entities that each could represent SE_k. For example we assign to String Entity SE_1 “Einstein” Canonic Entity CE_1 `Albert_Einstein`, but also CE_2 `Hermann_Einstein`, his father. Both are valid possibilities for “Einstein” as a first name is not specified. We define PCE_{SEk} the set of probable Canonic Entities *CE* assigned to a String Entity SE_k:

$$PCE_{SE_k} = \{CE_j \mid j \in [1, m_k]\} \qquad (32)$$

where m_k is the number of Canonic Entities identified for String Entity SE_k.

Example: For SE_1 = “Einstein” we have $PCE_{SE1} = \{CE_1, CE_2\}$ = {`Albert_Einstein`, `Hermann_Einstein`}

The purpose of the system is to assign a Canonic Entity to each String Entity identified in the document. Up to this point we have identified String Entities, and to each String Entity’s *PCE* we have added a number of possible Canonic Entities. However, String Entities can either be related amongst themselves or not (usually String Entities in a sentence are related amongst themselves but not to String Entities in other sentences). As it will be shown in a later section, we have the need to split our problem into smaller tasks, that is we do not want to process all String Entities at once, and due to the fact that we can identify groups of related String Entities we will focus on processing these independent groups separately. As each group is processed in exactly the same way, we have reduced the problem from considering all String Entities in a document to just a group of ***N*** related String Entities. For the remainder of the chapter, unless stated otherwise, ***N*** is the number of String Entities we have to deal with.

$$N = |\{SE_i \mid any\ SE\ is\ related\ to\ any\ other\ SE\ in\ this\ group\ \}| \qquad (33)$$

Thus, given our group of ***N*** related String entities, we define a **Result Set (*RS*)**:

$$RS = \{CE_k \big| CE_k \in \{PCE_{SE_k} \cup \emptyset\}, k \in [1, N]\} \qquad (34)$$

A Result Set *RS* always contains exactly *N* Canonic Entities CE_k, each one belonging either to its probable Canonic Entity set PCE_{SEk} assigned to String Entity SE_k or being unknown, as we allow for the possibility of new, unknown entities. The system will output for each group of String Entities not just one, but a sorted array of Result Sets RS_k (from which we will usually consider only the top scoring result).

Example: considering the previous example: "Einstein visited Ulm." We have String Entity SE_1= "Einstein" with PCE_{SE1} = {`Albert_Einstein`, `Herman_Einstein`} and SE_2 = "Ulm" with PCE_{SE2} = {`Ulm`, `Ulm_Montana`}. After processing, we can have several Result Sets:

RS_1 = {`Albert_Einstein`, `Ulm`}

(because the system knows that Albert Einstein was born in Ulm, Germany)

RS_2 = {`Herman_Einstein`, ∅}

(a valid Result Set comprising of Herman Einstein and **∅**, the empty placeholder, showing that the system did not find enough evidence to link Herman Einstein to any of the Ulm towns in PCE_{SE2}).

RS_3 = {∅, `Ulm_Montana` }

(another valid Result Set, showing that maybe there is a person named Einstein that is not Albert nor Herman, and the town in question is actually the Ulm in Montana)

etc.

It can be seen that on the i^th position of any Result Set there is either an empty placeholder or a Canonic Entity of the i^th String Entity, where i (1, ***N***).

We define ***RSA*** as the **array of Result Sets**:

$$RSA = \{\{RS_i, Score_i\} \big| i \in [1, Q]\} \qquad (35)$$

where $Score_i$ is the score associated to RS_i, a floating point value.

Example: considering the above example, the *RSA* would be:

$$RSA = \{ \{RS_1, 2.0\}, \{RS_2, 0.8\}, \ldots, \{RS_Q, 0.5\} \}$$

While Result Sets do not have scores, the containing Result Set Array assigns scores to each of its Result Sets.

Another important variable is the **size of the Result Set Array (*Q*).** The number of Result Sets in *RSA* will be used in the complexity evaluation of the algorithm.

Regarding the algorithm that produces these Result Sets, we need to introduce further notations:

We define the **input graph *(G)*** used by the algorithm. *G* is a weighted, undirected graph, with the following properties: 1. every vertex represents a Canonic Entity *CE* that belongs to a PCE_{SE}; 2. no links exist between the Canonic Entities belonging to the same *PCE*, only direct links to Canonic Entities belonging to other of the *N-1 PCEs.* These properties make *G* an *N*-partite graph.

We define the **number of vertices (*V*)**, as well as the **number of edges (*E*)** in *G*.

$$V = |V(G)| \text{ and } E = |E(G)| \tag{36}$$

where *V(G)* is the set of vertices in *G*, and *E(G)* is the set of edges in *G*.

These two integer values are also used in the algorithm's complexity evaluation. It is worth noting that *Q* will be determined in the algorithm, being a function of *N*, *V* and *E*.

V.3. Proposed custom graph algorithm – Linker Algorithm

As the graph algorithm represents a distinct contribution, as well as for the reason that the algorithm can be presented independent of any system, the algorithm's description, implementation and results will be presented in this separate section. The evaluation of the algorithm presented in this section will focus only on the algorithm's performance (runtime/memory/complexity/etc.) and not on the accuracy of the results when applied to the General Entity Recognition System (which will be presented after this section, integrated in the system).

The proposed algorithm is designed to solve the problem of **discovering the highest scoring sets of connected vertices within an *N*-partite weighted graph.** An *N*-partite graph is a graph that is logically divided into n partitions, having the property that there are no edges between vertices in the same partition.

The algorithm was developed in the context of our proposed system, where it serves the purpose of assigning ontological classes to entities extracted from text, given an ontology.

We abstract the ontology to a graph, the relations between classes in the ontology as weighted edges and the classes as the graph's vertices.

As the input of the algorithm in the context of our system is a group of *N* related String Entities (*SE*), each of them having associated a set of probable classes from the ontology (PCE_{SE}), we thus abstract the input of the algorithm to *N* sets of starting nodes. We consider the input graph on which the algorithm will work upon as previously created, with all the Canonic Entities from the input as included.

The output of the algorithm in the context of our system is an array of Result Sets (*RS*) sorted by their descending scores. A Result Set is an *N*-size set of Canonic Entities (*CE*), where the i^{th} *CE* is the chosen *CE* that represents the i^{th} *SE* from the algorithm's input (as presented in the formalization section V.2).

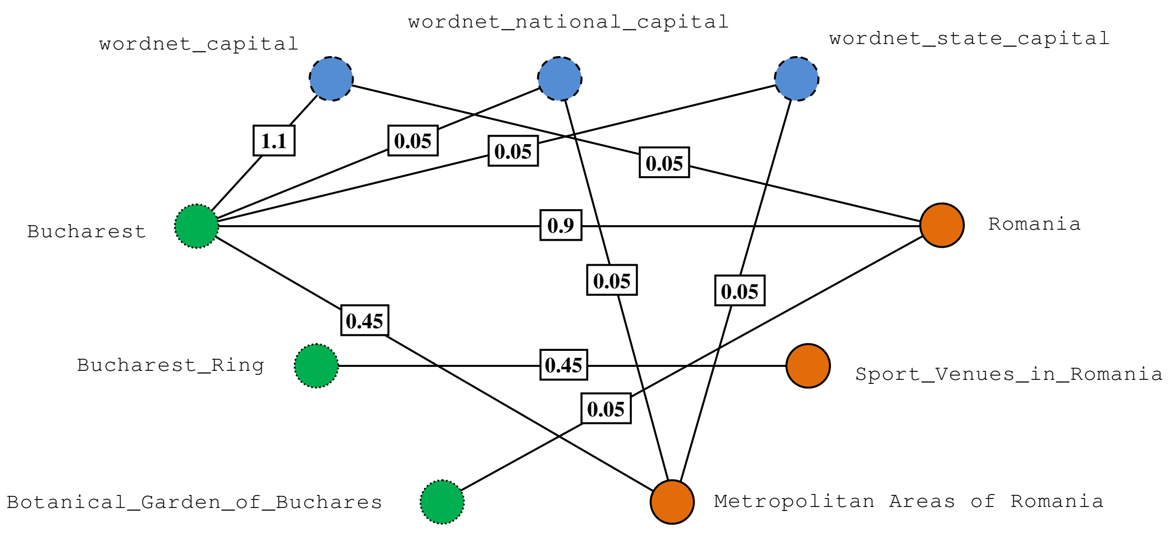
We now continue with an in-depth algorithm description, complexity analysis and evaluation.

V.3.1. Description

This section presents the algorithm in detail. It starts with an example, then divides the algorithm into smaller logical steps and presents them individually.

As previously stated, the algorithm aims at discovering sets of vertices, each belonging to a different partition. The sets are created based on the scores derived from the values of the weighted edges in the graph.

Consider the following example: Starting from the sentence "Bucharest is the capital of Romania" we identify "Bucharest", "capital" and "Romania" as interesting entities, and for each we assign 3 probable Canonic Entities, as shown in figure 8. Also, we create the graph linking them based on the ontology. Because in this section we focus on the algorithm only, the way the graph is created and the weights on its edges are not of importance (they will be presented in the next section V.4.).



where:

- ● are ontological entities that could represent SE_1 “Bucharest”
- ● are ontological entities that could represent SE_2 “capital”
- ● are ontological entities that could represent SE_3 “Romania”

Figure 8. Example of the graph decision problem

As can be seen, there are no links between vertices representing Canonic Entities of the same String Entity, the graph respecting the *N*-partite property, with *N*=3 in this particular case.

We now abstract the actual Canonic Entities for a shorter representation, and the graph becomes:

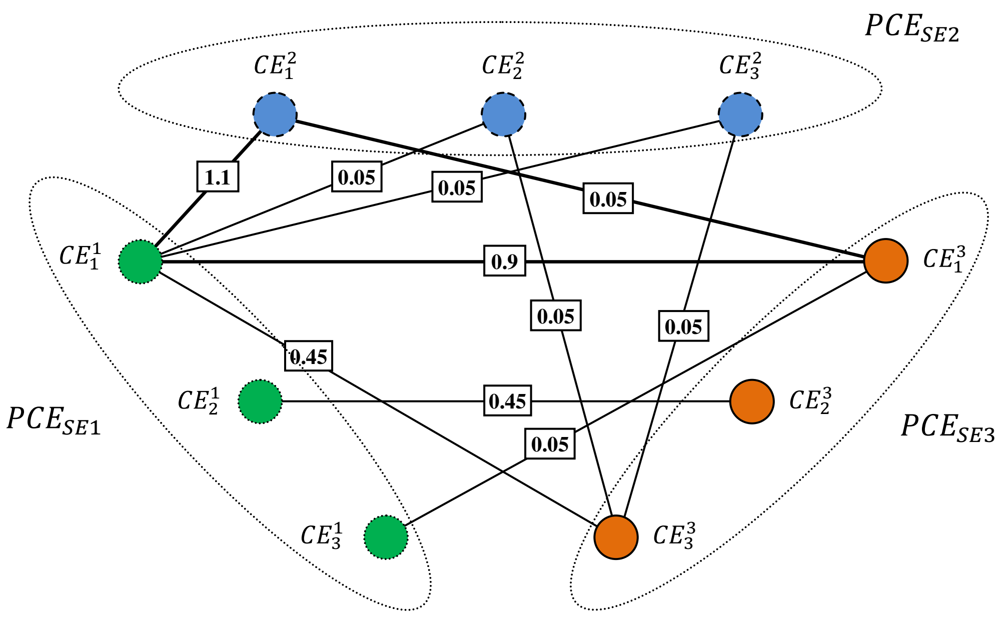


Figure 9. Abstraction of the 3-partite graph in figure 8

Thus, given an *N*-partite graph (input), find the sets of *N* vertices maximizing the score, each vertex belonging to one of the partitions (output is in the form of a Result Set Array containing multiple scored Result Sets). For example, in the figure above, we could have a valid Result Set Array:

$$RSA = \left\{\left\{\{CE_1^1, CE_1^2, CE_1^3\}, 2.05\right\}, \left\{\{CE_1^1, CE_2^2, CE_1^3\}, 0.95\right\}, \left\{\{CE_2^1, \emptyset, CE_2^3\}, 0.45\right\}, \dots \right\}.$$

As seen, the RSA holds sorted Result Sets and their scores. Also, Result Sets that are not 'complete' in the sense that for the third *RS* in our example *RSA*, we have on the second *CE* position a void element ∅, meaning that this 0.45 scoring *RS* is composed of CE_2^1 from PCE_{SE1} and CE_2^3 from PCE_{SE3}, and ANY Canonic Entity from PCE_{SE2}. Because there is no edge between either of the selected entities to any entity from PCE_{SE2}, we cannot assume an information link and thus any entity residing in PCE_{SE2} (or none of them) could be a valid choice for us. We allow this because if we find a strong link between two entities for example, then it is much more likely that the two entities are the correct *CEs* for their respective *SEs* than forcing a third, forth, etc entity to appear in the Result Set but with a weaker score.

The algorithm can be divided into four distinct sequential steps, each further detailed:

Step 1. Load and initialize data
Step 2. Perform DFS for every vertex in the graph
Step 3. Compute scores for the Result Set Array
Step 4. Merge non-overlapping Result Sets (optional)

A quick overview: first, data is loaded and processed in the format needed for the algorithm. Then, a depth-first search is performed starting from every vertex in the graph. This will discover possible solutions that will be added to the Result Set Array. After the graph search, scores are computed for every Result Set in the array. Next, if desired, a merging will take place between non-overlapping Result Sets to create the best scoring Result Sets.

V.3.1.1. Step 1 – Load and initialize data

This first step loads and converts the external data in the form needed for processing. The input of the algorithm is actually a long array of tuples, holding links between vertices and their weights. Also, the logical partitions are provided, in the form of an array holding sets of vertices.

In this step the link array is parsed. Because the original data is taken from a directed larger graph, if there is a link from CE_1 to CE_2, then there might be a link from CE_2 to CE_1 with a different weight. Because the algorithm runs on undirected graphs, every reverse link is deleted and its weight is added to the original link.

Then, the array containing all the vertices is created from the array containing the individual sets of vertices. This array is created for efficiency, because we already have all the vertices, just separated into sets. However this would require two operations instead of one when iterating over the vertices. We can view this new array as *V(G)*.

Also for algorithm efficiency, a hash map (key-value) is created for every vertex, containing the vertex as key, and the array of its neighbors as the value.

V.3.1.2. Step 2 – Perform a custom DFS for every vertex in the N-partite graph

This is the main processing step of the algorithm.

The main idea here is to determine sets of connected components. This is done by performing a custom depth-first search in the graph. But we need to determine all connected components containing each individual vertex. For this we need to perform the graph search starting from every point in the graph.

So for every vertex *CE_v* in the *V(G)* array created in step 1 we launch the **CDFS()** function.

```
function CDFS($CE_v$) {
  push $CE_v$ to path;
  for every neighbor $CE_n$ of $CE_v$: {
    if size(path)=1
      if edge($CE_v$,$CE_n$) is already visited
        continue to next neighbor;
    if $CE_n$ visited OR $CE_n$ does not belong to a unique $PCE_{SE}$
      continue to next neighbor;
    else
      mark edge($CE_v$,$CE_n$) as visited;
      CDFS($CE_n$);
  }
  if $CE_v$ has no valid neighbors AND size(path)>1
    addSolution(path);
  pop $CE_v$ from path;
}
```

Figure 10. Pseudocode for CDFS() function

CDFS() is a recursive depth-first search, storing the path from the initial vertex on each recurrent call, and stopping to add a new solution only when no neighbor vertices can be further added to the path. A new vertex can be added only if this vertex is not already present in the path and the vertex (which is a Canonic Entity *CE*) belongs to a probable

Canonic Entity set PCE_{SE} that no other vertex on the path belongs to. This ensures that we only add solutions that contain one Canonic Entity for every String Entity found.

Also another point of interest is the condition to continue the depth-first search only for the links that have not yet been visited. This heuristic drastically reduces the total number of graph searches and will be explained in the complexity analysis section.

Whenever the search encounters a valid vertex which has no further neighbors to explore, the **addSolution**() function is called, with the *path* parameter that holds the visited vertices so far.

The purpose of **addSolution**() is to add new, increasingly larger result sets (paths) to the possible solutions pool. It does this by performing two operations: subset detection and merging of solutions. Given a new solution *path*, and the already existing set of already added solutions, it will first be determined if *path* is not a subset of an existing solution, and then if *path* cannot be actually merged with an existing solution. This means that *path* can either be added as a new solution, discarded because it is a subset of an existing solution or merged into an existing solution.

Example: Let us consider a more complex example with four String Entities SE_A, SE_B, SE_C, SE_D, each with its associated PCE_{SE}. Let's assume that for SE_A, PCE_{SEA} is {*A1*, *A2*}, for SE_B, PCE_{SEB} is {*B1*, *B2*} and similar for SE_C and SE_D. A result set *RS* will have in this case length 4, because we have 4 String Entities. On each position, *RS* must necessarily have either a null value or a vertex (Canonic Entity) belonging to that respective PCE_{SE}. For example, a void *RS* can be visualized as {∅,∅,∅,∅} (where "∅" means null value, interpreted as "any" Canonic Entity), a completely filled *RS* as {*A1*, *B2*, *C1*, *D1*} and a partially filled *RS* as {*A1*, ∅, ∅, *D2*}. In our example, let's say we determined *path* as {*A1*, *B1*, *C1*, ∅}. If when iterating over the existing *RSA* we find that there is already a *RS* like {*A1*, *B1*, *C1*, *D1*} then we drop our proposed path *RS* because it is a subset of the existing one. If we find that path is not a subset of any existing *RS*, we search if we can merge path to any of the existing *RS* instead. For example, if we find {*A1*, ∅, *C1*, *D1*}, then *path* can be merged to it (because *A1* and *C1* are common to both, and the ∅ from the second position from the existing *RS* can be replaced by B1 from *path*, and the ∅ from the forth position from *path* can be replaced by *D1* from the existing *RS*) and produce the merged solution {*A1*, *B1*, *C1*, *D1*}. If *path* cannot be merged, it is added as a new *RS* to *RSA*.

```
function addSolution(path){
  RS = new empty result set;
  fill RS's appropriate slots with CEs extracted from path
  for every $RS_i$ in result set array RSA: {
    if isSubSetOrEqual(RS,$RS_i$)
      exit function without any changes;
    if canMerge(RS,$RS_i$)
      $RS_i$ = mergeRS(RS, $RS_i$);
      exit function;
  }
  add RS to RSA;
}
```

Figure 11. Pseudocode for addSolution() function

The **addSolution**() function will create an empty result set, populate it with entities found in CDFS's *path*, and then check every other result set in our result set pool to see if the current result set is either a subset or an equal or if it can be merged with any of them.

Determining if a *RS* is a subset of another *RS$_i$* (**isSubSetOrEqual**()):

```
function isSubSetOrEqual(RS, RSi){
  for j = 0 → N {
    if RS[j] ≠ RSi[j] AND RS[j] ≠ ∅
      return false;
  }
  return true;
}
```

Figure 12. Pseudocode for isSubSetOrEqual() function

The function checks iteratively all *N* positions of both *RS*s. If it finds a position of *RS* that is different from *RS$_i$* and that position is not ∅ then *RS* is not a subset of *RS$_i$*.

Determining if *RS* can be merged with another *RS$_i$* (**canMerge**()):

```
function canMerge(RS, RSi){
  commonElements = 0;
  for j = 0 → N {
    if RS[j] ≠ ∅ AND RSi[j] ≠ ∅
      if RS[j] = RSi[j]
        commonElements++;
      else
        return false;
  }
  if commonElements > 0
    return true;
  else
    return false;
}
```

Figure 13. Pseudocode for canMerge() function

This function iteratively counts the number of common elements in both *RS*s. If there is a position that is not ∅ and the respective elements are different, then the *RS*s cannot be merged. Finally, if the function cannot find at least one common element, the *RS*s cannot be merged because they are actually distinct, non-overlapping.

Both of these seemingly minor functions are important in the algorithm as the algorithm spends a great deal of time in them.

Overall, this step of the algorithm creates the initial *RSA*, but without any scores associated to the *RS*s contained within.

V.3.1.3. Step 3 – Compute scores for the Result Set Array

After **CDFS()** was run on every vertex in the graph, we have a *RSA* that contains many Result Sets, but with score zero. The score calculation is left for a later stage of the algorithm and not included in the **CDFS()** because it would add too much unneeded overhead if it would be computed on-the-fly. Also, it was not needed for Result Set generation. The Result Sets having been obtained, this step now computes the scores for each one of them.

```
function computeScores() {
  for every RS in RSA: {
    for $CE_i$ in RS (i = 1 → N): {
      if $CE_i = \varnothing$
        continue to next CE;
      for every neighbor $CE_j$ of $CE_i$
        if $CE_j \in$ RS AND position of $CE_j$ > position of $CE_i$ in RS
          $Score_{RS}$ += weight of edge between $CE_j$ and $CE_i$
    }
  }
  sort(RSA);
}
```

Figure 14. Pseudocode for the computeScores() function

The **computeScores**() function iterates over all *Q* Result Sets *RS* in *RSA*. For each *RS*, it searches iteratively each of its *N* positions. Each position might hold a *CE* or be void. If it is not void, it searches for every neighbor of *CE* not already visited. If a neighbor is found, it then adds the weight of the edge between them to the *RS*'s score.

After computing the scores, the Result Set Array is sorted descending by the score of each *RS*.

At the end of this step we have a sorted Result Set Array filled with possible vertex choices, each belonging to a logical partition. At this point, the first *RS* (or first *RS*s) can be used as the solution to the current problem.

V.3.1.3. Step 4 – Merge non-overlapping Result Sets

This step is optional. To see the opportunity/necessity of this step, consider the following possible *RSA* output by the algorithm for an *N*=6 logical partition problem:

$$RSA = \begin{Bmatrix} RS_1 = \{A1, B1, C1, \emptyset, E1, F1\} & Score_{RS_1} = 4.0 \\ RS_2 = \{A2, B1, \emptyset, \emptyset, \emptyset, \emptyset\} & Score_{RS_2} = 3.0 \\ RS_3 = \{A3, B1, C1, \emptyset, \emptyset, \emptyset\} & Score_{RS_3} = 2.5 \\ \ldots & \ldots \\ RS_i = \{\emptyset, \emptyset, \emptyset, D1, E2, \emptyset\} & Score_{RS_i} = 0.6 \\ RS_j = \{\emptyset, \emptyset, C2, \emptyset, \emptyset, F2\} & Score_{RS_j} = 0.6 \end{Bmatrix}$$

Figure 15. Example Result Set Array

For this example it is clear that the best choice is RS1 with the highest score 4.0. However, if we look down the list of other result sets, we can see that there are other RSs that could be useful. As ∅ stands for the ANY placeholder, meaning that a ∅ spot can hold any vertex in its logical partition, maybe it would be useful to **combine non-overlapping Result Sets**, like RS_i and RS_j for example into a single *RS*. It is immediately apparent that the combination of RS_2, RS_i and RS_j would yield a higher scoring *RS* containing {*A2*, *B1*, *C2*, *D1*, *E2*, *F2*} with a score of 3.0 + 0.6 + 0.6 = 4.2, higher than RS_1. Another option would be to combine RS_3 with RS_i yielding {*A3*, *B1*, *C1*, *D1*, *E2*, ∅} with score 2.5+0.6 = 3.1.

Even though this step will present a new Result Set Array with different *RS*s (larger) that have higher scores, it does not add new information. The creation of new, larger *RS*s could even be seen as confusing. Intuitively, the vertices presented in a *RS* are thought to be linked each one of them (as they are in essence connected components in a graph). However, presenting a new *RS* composed of two distinct, non-overlapping *RS*s would create the impression that all vertices contained form a connected component, which is not true. But in the context of our system, this step is actually required as we aim to detect most likely *CE*s for every *SE*, meaning that a *SE* that is represented by a *CE* is better that being represented by ANY *CE* in its set.

Having motivated the opportunity/necessity of the merging, we come to an interesting problem. What combination is the highest scoring? Or in other words, how to determine the combinations of Result Sets that lead to the best scores? This in itself is a problematic question.

Example: Consider the following two cases, identical sets but with different scores for the forth Result Set:

$$RSA = \begin{Bmatrix} RS_1 = \{A1, B1, \emptyset, \emptyset, \emptyset, \emptyset\} & 1.0 \\ RS_2 = \{\emptyset, \emptyset, C1, D1, \emptyset, \emptyset\} & 1.0 \\ RS_3 = \{\emptyset, \emptyset, \emptyset, \emptyset, E1, F1\} & 1.0 \\ RS_4 = \{\emptyset, \emptyset, \emptyset, D2, E2, \emptyset\} & \mathbf{1.0} \end{Bmatrix} \qquad RSA = \begin{Bmatrix} RS_1 = \{A1, B1, \emptyset, \emptyset, \emptyset, \emptyset\} & 1.0 \\ RS_2 = \{\emptyset, \emptyset, C1, D1, \emptyset, \emptyset\} & 1.0 \\ RS_3 = \{\emptyset, \emptyset, \emptyset, \emptyset, E1, F1\} & 1.0 \\ RS_4 = \{\emptyset, \emptyset, \emptyset, D2, E2, \emptyset\} & \mathbf{3.0} \end{Bmatrix}$$

Case 1 Case 2

The best scoring set in the first case would be *RS`* = RS_1 + RS_2 + RS_3 with score 3.0. However, modifying the score of *RS4* in the second case would produce a higher scoring result *RS``* of just $RS_1$ + $RS_4$ = 4.0. Even though RS` has all positions filled and RS`` has to void places, RS`` scores higher and should be chosen. From this example it can also be seen that higher-scoring Result Sets can be created from any number of non-overlapping Result Sets.

One brute-force approach would be to consider all combinations of Result Sets. However, that would lead to factorial complexity, a worst case complexity scenario. We propose the following algorithm:

```
function mergeNonOverlappingRSA() {
  initialize new $RSA_{final}$;
  for $RS_i$ in $RSA$ (i = 0 → $Q$) {
    initialize $RSA_{RSi}$;
    for $RS_j$ in RSA (j = i+1 → $Q$) {
      if $RS_j$ and $RS_i$ are non-overlapping
        add $RS_j$ to $RSA_{RSi}$
    }
    createSolutionTree($RS_i$, $RSA_{RSi}$);
    $RS_{final}$ = getBestSolution($RS_i$);
    add $RS_{final}$ to $RSA_{final}$;
  }
  $RSA$ = sort($RSA_{final}$);
}
```

Figure 16. Pseudocode for mergeNonOverlappingRSA() function

The general idea is that we want to create for each Result Set in *RSA* a weighted tree with all the other Result Sets that are non-overlapping, and then search the tree to determine the highest scoring branch. This approach, while not factorial in complexity, will provide an optimum solution for each *RS*.

In more detail, **mergeNonOverlappingRSA**() will create a new empty Result Set Array RSA_{final}. For each Result Set RS_i in the original *RSA* it will create a new Result Set Array RSA_{RSi} for RS_i (the algorithm will thus create *Q* new smaller *RSA*s by the end). It will then populate RSA_{RSi} with all the other Result Sets RS_j in RSA that are non-overlapping with RS_i. Then, it will launch the **createSolutionTree**() function, presented below, that will create a tree with weighted edges, and Result Sets as nodes. Then **getBestSolution**() function will traverse the tree searching for the highest scoring path from the root RS_i to a leaf RS, and return the merged results as a new Result Set RS_{final}. RS_{final} is then added to the new RSA_{final} array. The last step is to sort RSA_{final}, and replace RSA with the new, larger Result Set Array.

We now present **createSolutionTree():**

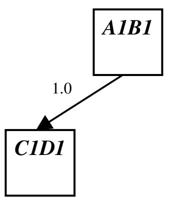
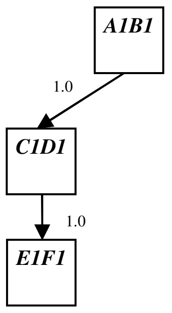
```
function createSolutionTree ($RS_i$, $RSA_{RSi}$) {
  $RS_{merged}$ = merged $RS$ from root to current $RS_i$;
  for $RS_j$ in $RSA_{RSi}$ {
    if $RS_j$ and $RS_{merged}$ non are non-overlapping
      create edge between $RS_i$ and $RS_j$ with weight $Score_{RSj}$;
      createSolutionTree($RS_j$,$RSA_{RSi}$ without $RS_i$ … $RS_j$);
    }
}
```

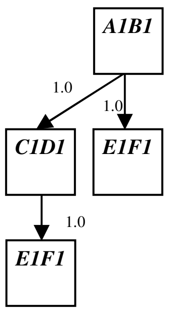
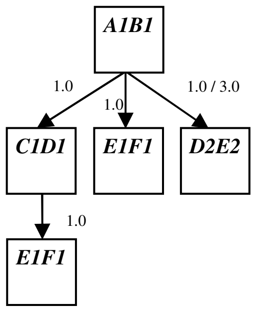
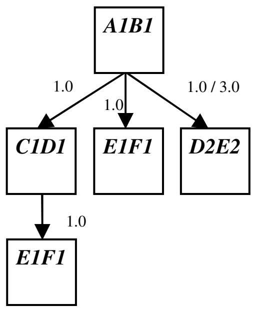
Figure 17. Pseudocode for createSolutionTree() function

This function will recursively create a tree having as root the initial Result Set in the **mergeNonOverlappingRSA**() function. It takes two parameters, the *RS* node and the *RSA* corresponding to that node.

In the following example we use Case 1 or 2 presented above, where we abstract *RS* notation for easier reading – instead of {*A1*, *B1*, ∅, ∅, ∅, ∅} we write just {*A1*, *B1*}. For this example scores are not important, just the items in the Result Sets to see how they can be merged.

Table 2. Example of tree creation for merging non-overlapped Result Sets

Depth	*RS*	*RSA*	Observations	Tree
1	*A1B1*	{*C1D1*, *E1F1*, *D2E2*}	Search *RSA* for non-overlapping children of *A1B1* Find children: *C1D1*, *E1F1*, *D2E2* For child *C1D1* create link *A1B1* → *C1D1* Call function with (*C1D1*, {*E1F1*, *D2E2*})	
2	*C1D1*	{*E1F1*, *D2E2*}	Search *RSA* for non-overlapping children of *C1D1* Find children: *E1F1* For child E1F1 create link *C1D1* → *E1F1* Call function with (*E1F1*, {*D2E2*})	

3	*E1F1*	{*D2E2*}	Search *RSA* for non-overlapping children of *E1F1* Find no children and return to parent (depth 2) Find no children and return to parent (depth 1) For child *E1F1* create link *A1B1* → *E1F1* Call function with (*E1F1*, {*D2E2*})	
2	*E1F1*	{*D2E2*}	Search *RSA* for non-overlapping children of *E1F1* Find no children and return to parent (depth 1) For child *D2F2* create link *A1B1* → *D2F2* Call function with (*D2F2*, {∅})	
2	*D2F2*	{∅}	Search *RSA* for non-overlapping children of *D2F2* Find no children and return to parent (depth 1) Find no children and return to parent (depth 0) Exit recursion tree	

The new merged Result Sets are obtained by traversing the tree from root to leaf. For the example above, the first merged RS is obtained by starting from *A1B1*, moving down through *C1D1* to *E1F1*, yielding RS_1 = {*A1*, *B1*, *C1*, *D1*, *E1*, *F1*}. RS_2 will be {*A1*, *B1*, ∅, ∅, *E1*, *F1*} and RS_3 will be {*A1*, *B1*, ∅, *D2*, *E2*, ∅}.

As presented in the table, the **createSolutionTree**() function will create a tree containing on each level a non-overlapping *RS*. It should be noted that on every step, we pass the Result Set Array parameter to the next function call with all elements up to RS_j (as in the pseudocode above) to avoid duplication of results. This will actually halve the solution space (tree) obtained. Having the tree constructed, the **getBestSolutionFunction**() will perform a DF search in the tree and obtain the highest scoring path. In the example above, the trees are identical for cases 1 and 2 with only the weight of edge *A1B1* → *D2E2* being different. If **getBestSolutionFunction**() is called in case 1, it will return a *RS* containing {*A1*, *B1*, *C1*, *D1*, *E1*, *F1*} with score 3.0 and in case 2 it will return a *RS* containing {*A1*, *B1*, **∅,** *D2*, *E2*, ∅} with score 4.0.

After every Result Set in the original *RSA* has been merged with the best combination of non-overlapping Result Sets, the final operation of **mergeNonOverlappingRSA()** is to sort the newly created *RSA*.

V.3.2. Complexity analysis

The complexity of the algorithm is determined by inspecting each component in turn.

Step 1 of the algorithm handles loading data and processing it. The first operation performed is to transform the directed graph in an undirected graph. This is performed by inspecting every edge and checking if there is another reverse edge. If so, the weights are combined and the reverse edge is dropped. Considering that the graph has *E`* edges and will be reduced to *E* edges (as defined at the beginning of this chapter), we can approximate the number of operations to $O(E^2)$.

Next, a hash map is created, containing for every vertex an array of its neighbors. This is required because we create a dictionary of edges that will make edge retrieval an constant time O(1) operation in the next step. This requires iterating over every vertex (we have *V* vertices in the graph). For each vertex we traverse the edge list containing *E* edges. The entire operation implies $O(VE)$ complexity.

The second step of the algorithm is where the actual depth first searches are performed, starting from every vertex.

The depth-first search in itself is a $O(V+E)$ operation [131], because we have previously created the edge dictionary (hash map) so that neighbor retrieval is now an O(1) operation. However, when reaching a vertex where no new nodes can be added, the **addSolution()** function is called. The graph has *V* vertices, so the function will be called *V*-1 times.

The **addSolution()** function performs two operations in respect to its input parameter which is a potential solution (*RS*). It first tries to detect whether the potential solution is a subset of another existing solution, then whether the potential solution can be merged with another existing solution. This implies iterating over the Result Set Array that has an increasing number of solutions. Finally, *RSA* will contain *Q* elements, so we will consider the worst-case scenario where we have to iterate over *Q* elements. Thus, **addSolution()** will iterate over *Q* Result Sets, for each comparing position by position (*N* positions corresponding to the *N* partitions of the graph) whether the candidate *RS* is a subset of exiting *RS*s. Both **isSubSetOrEqual()** and **canMerge()** are $O(N)$ functions. This implies that **addSolution()** will have a complexity equal to $O(N+Q(N+N)) = O(2QN+N)$.

Therefore the total complexity of a **CDFS()** call will be $O(V+E*(2QN+N)) = O(2QNE + NE +V)$. Considering that in worst case scenario we will have *V* **CDFS()** calls, then the total complexity of step 2 will be $O(2QNEV+NEV+V^2)$.

Step 3 handles computing scores for the Result Sets obtained in step 2. This requires iterating over the *Q* Result Sets. For each Result Set, for each position (from a total of *N* positions, as RS is a *N*-length array), a list of neighbors is obtained (using the edge dictionary in O(1) time). This list of neighbors is then iterated over. However, considering the worst case scenario, where we have a complete k-partite graph, we need to iterate close to *V* neighbors. This implies that **computeScores()** will have a complexity of O(*QNV*) so far.

Next, we need to sort the Result Set Array. For this reason we use a merge sort algorithm because it is a stable sorting algorithm as the average and worst time complexity are both O(*QlogQ*) in our interpretation (as opposed to Quick Sort for example[46] which has a worst time complexity of O(Q^2) when the list is sorted, even though in average is also a O(*QlogQ*) algorithm).

After sorting, **computeScores()** will have total complexity of O(*QNV* + *QlogQ*).

In step 4 we create a tree of possible Result Set combinations. We have *Q* Result Sets to look at. We will analyze the worst case scenario where we have *Q* Result Sets that all are non-overlapping between them.

Example: Consider we have *Q* = 5 Result Sets, noted as *A*, *B*, *C*, *D* and *E* (assuming for simplicity that *A* = {*A*, ∅, ∅, ∅, ∅}, *B* = { ∅, *B*, ∅, ∅, ∅}, etc.). We have *N* = 5 Result Set length (also for simplicity we will allow Result Sets having only one Canonic Entity). To create merged sets we can have any combination of *A* .. *E*. A valid RS would be *A* ∪ *B*, *C* ∪ *D* ∪ *E* or even *A* ∪ *B* ∪ *C* ∪ *D* ∪ *E*. The created tree would look like this:

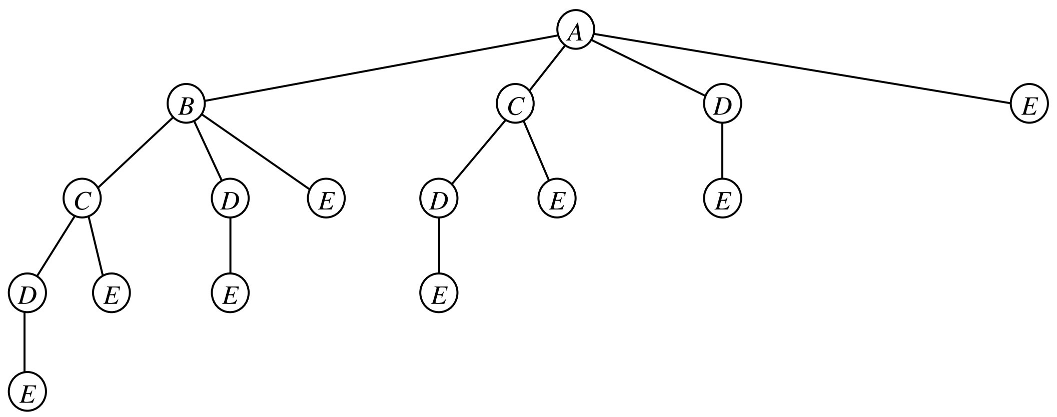


Figure 18. Worst case scenario tree construction for step 4 merging function

The **createSolutionTree()** function complexity can be calculated as follows, considering that in any step, in worst case scenario, a node will have *Q-d* children (*d* = depth in tree), and

[46] http://en.wikipedia.org/wiki/Randomized_algorithm

will visit them each sequentially, with a *Q-d-*i_{th_child} children array. We can write this function as follows:

$$T(n) = N + \big(N + T(n-1)\big) + \big(N + T(n-2)\big) + .. + \big(N + T(1)\big)$$
$$T(n) = nN + T(n-1) + T(n-2) + .. + T(1) \tag{37}$$

where the initial *N* is for obtaining RS_{merged}, and each parenthesis is a "for" iteration containing an O(*N*) for checking if RS_j and RS_{merged} are non-overlapping and then calling the recursive function again with a sequentially decreasing RSA. *T(1)* = *O(1)* = 1. Simplifying notation yields:

$$T(n) = nN + \sum_{i=1}^{n-1} T(i) \tag{38}$$

Isolating *T(n-1)* and expanding recursively:

$$T(n) = nN + [T(n-1)] + \sum_{i=1}^{n-2} T(i) =$$
$$= nN + \left[(n-1)N + \sum_{i=1}^{n-2} T(i)\right] + \sum_{i=1}^{n-2} T(i) = (n + n - 1)N + 2\sum_{i=1}^{n-2} T(i)$$
$$\dots \tag{39}$$
$$T(n) = (n + n - 1 + n - 2 + .. + 1)N + 2^{n-1}T(1) = N\sum_{i=1}^{n} i + 2^{n-1}T(1)$$

Considering that *n* can be at most *Q*, and *T(1)* = 1, ignoring constants and approximating, *T(n)* becomes:

$$T(n) = N\frac{n(n+1)}{2} + 2^{n-1} \cong NQ^2 + 2^{Q-1} \tag{40}$$

The dominant term is 2^{Q-1} and is half of the sum of all k-combinations of *Q* elements (which is 2^Q) considering all other operations are O(1), which is we would need to do to check all possible combinations of Result Sets by brute force.

As a side node, even though worst case complexity is almost as bad as having to generate all possible k combinations of Q elements, in average, in our problem setting, this is a non-issue because rarely we have more than one or two possible Result Sets to combine with – meaning we create a tree of depth 1 or 2 with only a couple of branches – bringing the average complexity down into almost constant time O(*kN*) (because having only one or two children implies 1 or 2 calls to the function that needs to check for non-overlapping which takes O(N) for each child).

The **getBestSolution()** function has complexity O(V'+E') as it is a depth-first search. However, in the worst case scenario the tree has 2^{Q-1} vertices with 2^{Q-1}-1 edges. This means the complexity of this function is O(2^Q).

The total complexity of **mergeNonOverlappingRSA()** is thus: O(Q*(NQ + NQ^2+2^{Q-1} + 2^{Q-1})+Qlog$Q$) = O(NQ^2 + NQ^3 + 2^QQ + $Q$log$Q$). The total complexity obtained is the largest of any step. However, in real life scenarios this function is executed quickly, the largest influence having the Q^3 term.

Observation: We have generally sacrificed storage space for speed. Considering that in general we work with relatively small number of edges, vertices and Result Sets that in current computers occupy only a fraction of the total available amount of RAM, the choice for speed over storage is obvious. This is why, for example, in step 1 we create an edge dictionary even though we already have the graph links stored as a simple array, or in step 3 we use merge sort instead of quick sort.

We now calculate the total complexity of our algorithm:

Step 1: O(E^2 + VE)

Step 2: O($2QNEV$ + NEV + V^2)

Step 3: O(QNV + $QlogQ$)

Step 4(optional): O(NQ^2 + NQ^3 + 2^QQ + $Q$log$Q$)

The total complexity of the algorithm will thus be the sum of each individual step, as each step is executed sequentially.

The experiments that follow show that core processing time (step 2 + 3) is very fast (especially for real-life graphs applied to our problem setting), on the order of less than 500ms per graph. It can be seen that in the forth optional step there is an exponential term in the complexity: 2^Q which takes the problem from the polynomial to the exponential complexity domain. However, as Q is a variable dependent on N, E and V (basically depends on the graph density – in our case defined as the ratio of E over V as we can have several edges between any two vertices), it will be shown that in real life scenarios Q will be small, and the 2^Q term will play a less significant role than other polynomial terms. The following "Experiments" section will further elaborate on the complexity and performance of the proposed graph algorithm.

V.3.3. Experiments

To evaluate the performance of the algorithm we will focus on runtime of diverse types of graphs, varying the input parameters *N*, *E* and *V*, which will directly influence *Q*.

The evaluation will follow two types of graphs: random generated graphs that show how performance and parameters vary, and real-life graphs extracted from actual texts, to see the algorithm's actual performance in practice.

The algorithm (as the entire system) was built in Java 1.6 64bit. It was implemented as a single threaded application, even though all steps can be easily parallelized. Steps 1, 3 and 4 can be directly parallelized as they process data that is not dependent on other data. Step 2 can also be parallelized by synchronizing thread access for I/O on the Result Set Array. The algorithm was not parallelized because analysis is easier and more relevant when not considering threads as a parameter.

The experiments were conducted on a normal PC, powered by a single-core Pentium 4 processor (Cedar Mill, with Hyper Threading disabled) at 2.8 GHz. A low-end machine was chosen specifically for the algorithm not to take advantage of operating system or java compiler automatic pseudo-parallelization that happens when running single-threaded applications on multi-core processors.

V.3.3.1. Evaluation on random-generated complete graphs

The first type of evaluation the algorithm will be subjected to is a worst-case scenario complete graph. A complete graph is a graph in which every vertex is connected to every other vertex (in our case excluding connections between partition vertices to maintain k-partite property). We generate the graph with an equal number of vertices in each of its *N* partitions. Similarly, an equal number of edges will link the vertices of any two partitions.

We use the term 'complete' graph somewhat abusive, because we refer to the graph as complete in the sense that every partition is linked to every other partition, even though individual vertices from a partition could have any number of links to vertices of another partition (including zero links meaning isolated vertices). We will thus use the term 'complete' graph in this section keeping in mind the note above.

We implemented a random graph generator. This generator varies the *V* and *E* parameters and benchmarks the algorithm for every variation. We generate complete graphs by first generating *N* times *V*/*N* vertices representing the *N* partitions. This generates an equal amount of vertices for each partition. Then we iteratively generate between every two partitions a number of 2**E*/(*N**(*N*-1)) links between elements of the two partitions. Because a complete graph has V*(V-1)/2 links, in total we obtain a number of *E* randomly generated links, but evenly distributed between partitions.

For our tests, we fixed *N* = 5, as it is the average number of partitions we estimate the algorithm will handle; the number *N* itself is not very relevant as the results will scale accordingly. More important parameters are the total number of vertices *V* and the total number of edges E in the graph. We vary *E* from 100 edges to 1000, and *V* from 5 vertices (meaning one vertex per partition) to 500 (meaning 100 vertices per partition).

We first observe the number of Result Sets obtained varying *E* and *V* as specified.

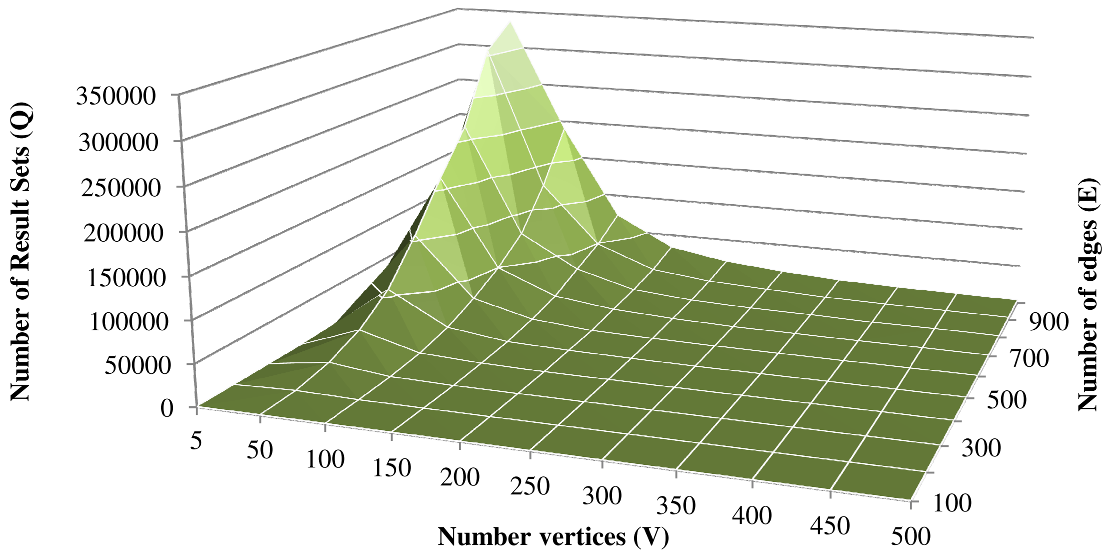


Figure 19. Result Set size variation on complete graph

The surface above immediately shows two things: 1. the algorithm has an almost exponential tendency for a large number of edges and a small number of vertices, and 2. for the vast majority of our test cases, the Result Set *Q* number remains fairly small and constant.

It can be seen that *Q* depends greatly on the ratio of edges to vertices. The higher this ratio is, the higher the number of Result Sets generated. This happens because if in a graph where there are several edges between any two vertices the algorithm will evaluate all combinations of edge paths in the graph between vertices, yielding several Result Sets having the same vertices but with different scores. In the figure above we can see that the highest number of Result Sets is found for graphs with the fewest number of vertices interconnected by the largest number of edges. We now look at the time required for the algorithm to finish steps 2 and 3 (step 1 is performed <5ms each time, and thus irrelevant, and step 4 is optional and mimics step 2 in trend so it is omitted).

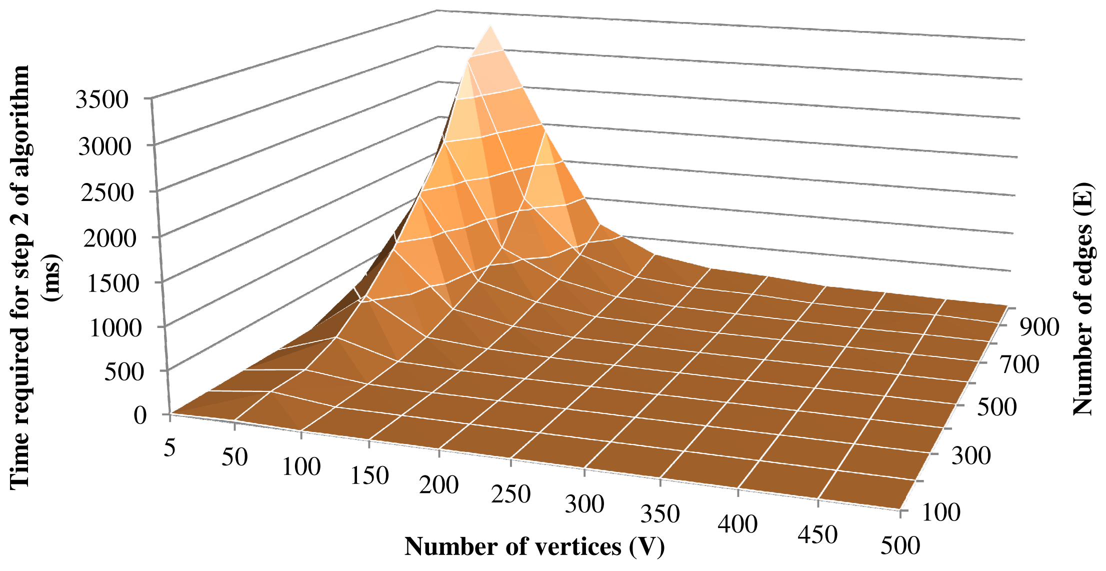


Figure 20. Algorithm step 2 time variation for complete graph

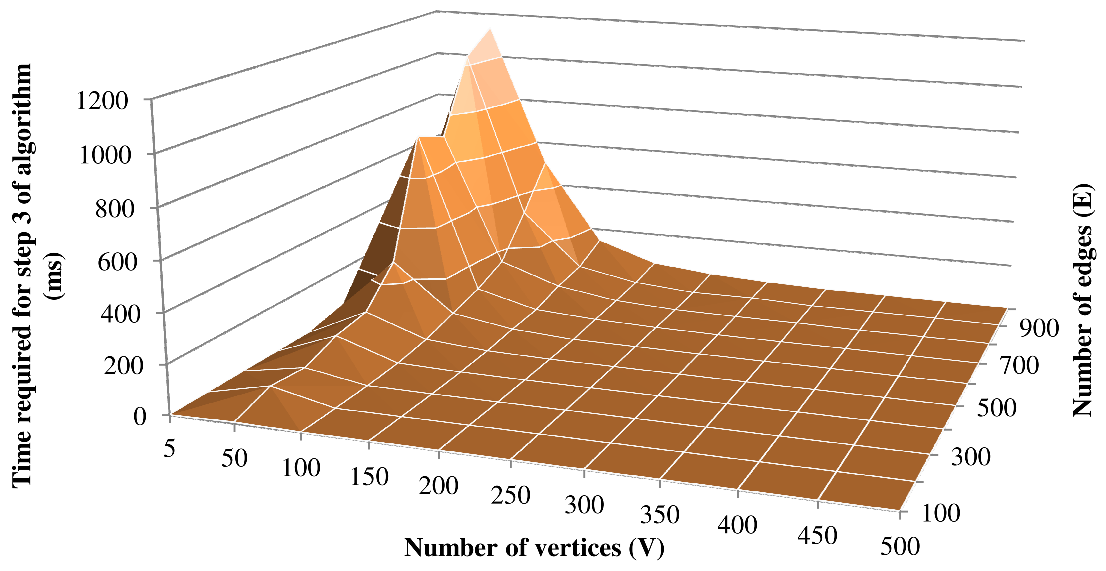


Figure 21. Algorithm step 3 time variation for complete graph

We see the same evolution for both steps 2 and 3 as in the Result Set number figure. This is a first indication that the size of the Result Set Array *Q* is the most important factor deciding run time. Also, based on the variation of *E* and *V*, *Q* is obviously varying greatly on the structure of the graph. As the graph is denser, with fewer vertices but more interconnections, *Q* is growing very fast. The strong correlation between run-time with *Q* implies that *Q* is the deciding factor in algorithm complexity, more than *E* or *V*. Overall, we see that the average running time for most of the test cases is just a few milliseconds.

We now look into more detail to the algorithm time performance and the size of the Result Set Array for a fixed number of V = 50 vertices for 5 partitions, with the number of edges varying from 50 to 1000.

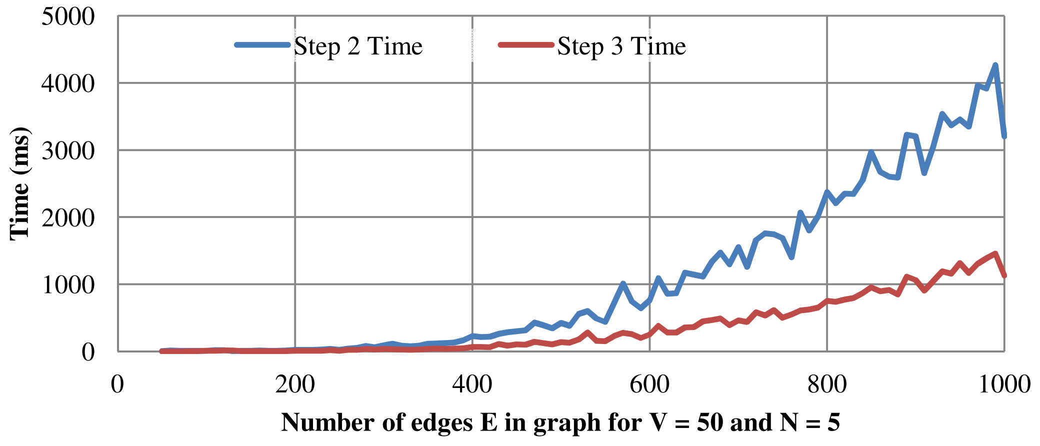


Figure 22. Time measurement when varying the number of edges

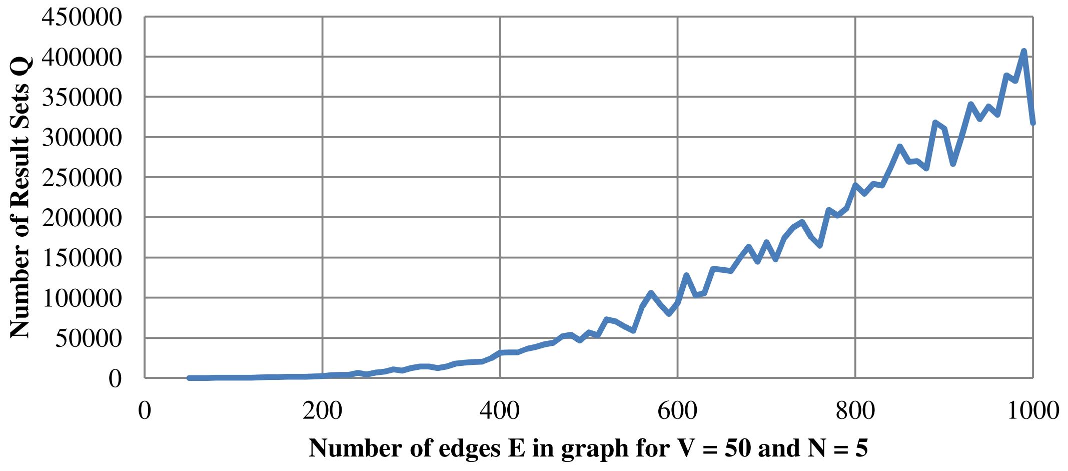


Figure 23. Result Set Array size when varying the number of edges

Considering the complexity of steps 2 and 3 (Step 2: O($2QNEV + NEV + V^2$) and Step 3: O($QNV + QlogQ$)) we see that the most important variable appears to be Q, as it tends to grow exponentially in certain cases, even though E and V increase linearly. Q is basically a function of three parameters: E, V, and the graph's structure, and is determined in step 2 of the algorithm. However, both steps vary linearly depending on Q, so we can conclude the algorithm has an almost linear complexity in average cases (we based this on the worst case scenario involving the largest solutions possible. This was due to the fact that the custom depth first search has to discover every possible maximal length solution due to the nature of the graph).

V.3.3.2. Evaluation in a real world scenario

The evaluation of the algorithm is better performed in the scenario it was designed for, which means a varying number of vertices, edges and uneven partitions and links between partitions. All of these parameters are determined by the characteristics of the text the algorithm is applied on – in this section the graphs are generated from a collection of text documents using the General Entity Recognition system itself to extract and process the String Entities. The algorithm is given a graph in the form of links between vertices and the logical partitions of these vertices. The vertices themselves represent entities extracted from the ontology, and the edges between them are the paths in the ontology between these entities.

As such, to determine the graph that the algorithm will be run upon we first need to determine the entities and links in the ontology which in turn are found by analyzing the words that make up the sentences in documents. We need to determine the *N*, *E* and *V* parameters, as well as the structure of the graph represented by the edges between vertices.

The *N* parameter is the number of String Entities that are related. Usually, *N* is the number of nouns in a sentence, because we target only nouns (common and proper) and usually we consider all nouns as related in simple sentences.

Next, the number of vertices *V* is determined by the number of Canonic Entities found in the ontology for every String Entity. The number of vertices is thus strongly linked to the method of discovering possible matches in the ontology and to the ontology itself (larger ontologies will hold more probable entities that could mean the same thing). The method of Canonic Entity assignation to a String Entity is explained in the section describing the General Entity Recognition system itself.

The number of edges *E*, and the structure of the graph itself is also directly linked to the ontology, as an edge between two vertices in our graph is actually a path (either direct or composed of other intermediate entities) between the two entities in the ontology. The method of generating the graph starting from the ontology will also be explained in the GER system section.

We are interested in real life evaluations of these parameters to be able to run the algorithm and analyze its performance. Keeping in mind that we intend for our system to be used in the context of Information Extraction from the Internet, we try to evaluate the parameters by analyzing three sources of information: Wikipedia pages, news articles and blogs.

For this evaluation we will use 50 Wikipedia pages, 50 news articles and 50 blog entries. While the average words per Wikipedia article is 590[47] we chose articles that had at least

[47] http://en.wikipedia.org/wiki/Wikipedia:Size_comparisons

5000 words, as we wanted to create a category of longer documents. The length of the news articles and blog entries is usually around 400-800 words.

For Wikipedia pages a wiki parser[48] was used to parse some of the more popular and longer articles online. The extracted HTML tree was cleaned and a text-only version of the pages was obtained.

For the news articles we have used BBC[49] as a news source, but filtered the pages as text-only versions by another online site[50]. 50 news items were extracted and placed in simple .txt files. For the blog entries we used the random function on blogspot.com to extract 50 names of English language blogs. For the text-only translation viewtext.org was given as parameter the newest post in every blog. The obtained HTML was parsed and only the text section (excluding title, etc) was kept and stored in .txt files, in the same format as above.

Next, we ran the entire system on each of the documents sequentially. For each document, it was first split into sentences, sentences into tokens, tokens were merged where necessary to create String Entities, to which Probable Canonic Entity sets were added. Then, the intermediary ontology was created and paths between related entities were identified. The set of edges links and the Canonic Entities themselves were given to the algorithm for processing.

We ran the algorithm for each of the three categories of items. We present the algorithm's results in the table below showing the average number of String Entities found that are related and implicitly processed together and associated parameters:

Table 3. Algorithm input parameters average grouped by document category

Property	Wikipedia	Blogs	News	All Combined
Number of sentences per document	268.7	29.6	31.2	109.85
N average (average # of related String Entities)	6.7	5.7	6.9	6.4
Named SE average per partition (proper nouns)	2.5	2.0	2.2	2.2
Common SE average per partition (com. nouns)	4.2	3.7	4.7	4.2
Distribution of Named vs. Common SE per part.	37%/63%	35%/65%	32%/68%	34%/66%
Average # Canonic Entities per document	107121	7896	12017	42375
Average Canonic Entities per Named SE	119.9	87.1	113.4	106.8
Average Canonic Entities per Common SE	18.4	14.2	15.0	15.9

From the experimental results obtained we can draw some conclusions:

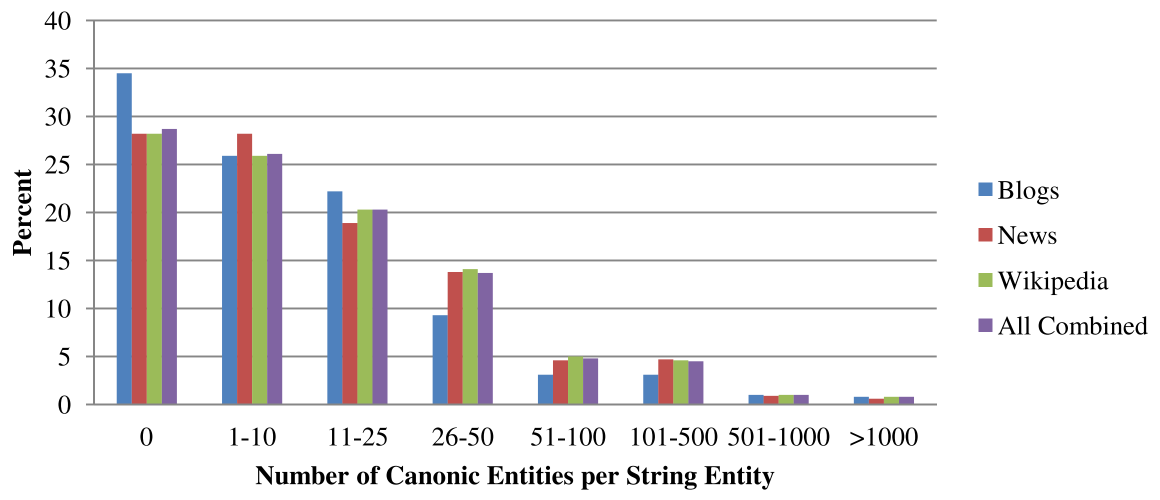
- The average number of sentences per document illustrates the average size of documents chosen. While the news from BBC and blog posts are rather short,

[48] http://sweble.org/wiki/Sweble_Wikitext_Parser
[49] www.bbc.com
[50] Example: http://neilbryson.net/newsfeed/single.php?url=/go/rss/int/news/-/news/world-africa-13371638

visited Wikipedia pagers are longer, reaching an average of 268 sentences per document.

- The *N* average is the average number of related String Entities. News documents have the highest average of 6.9 *SE*s per partition. Out of these total average numbers of SEs we have also calculated the number of String Entities representing common words or named entities. It can be seen that most named entities are found in Wikipedia articles (37% out of all String Entities are named entities). However, the distribution is rather similar for every category. This is an important parameter, as it shows that most of the vertices in the graph are common words, together with the observation that links in the graph between common words are links of type `subClassOf` which are of lower importance. In our problem setting, the existence of named entities is essential as named entities are strongly interconnected compared to the common entities that usually have only a single `subClassOf` link to a single other entity, up one level in the WordNet hypernym tree. Also this parameter is important for the algorithm itself, as graphs with many named entities will have a much higher number of edges and vertices, and thus more Result Sets and longer processing times.
- The average number of Canonic Entities per document is a statistical parameter to show approximately the number of CEs identified and passed to the algorithm per document. The bigger the documents, the more Canonic Entities are identified in the ontology.
- The last two values are the average number of Canonic Entities identified for each named or common String Entity. It can be seen that Wikipedia articles (with news articles close behind) have String Entities that are more 'popular', meaning YAGO knows more probable classes per String Entity than for blog articles, both for named and common entities. This reflects the fact that blogs have words that are less common, and thus less probabilistically to exist in the ontology, just as expected.

Another interesting parameter is partition distribution. If we analyze the number of Canonic Entities per each String Entity (named and common combined), we observe the following distribution:



The figure above shows that most String Entities (almost 80%) have less than 25 Canonic Entities associated. This means the algorithm will most usually run with partitions that have less than 25 vertices. Interestingly, we see a surprising number of String Entities that have no Canonic Entity associated. This means that during the Canonic Entity association phase (later described in the General Entity Recognition system section) we have found no suitable candidate. As seen, many partitions will thus have no vertices whatsoever. On the other end of the chart, we see that there are partitions that have more than 1000 vertices. These String Entities are always proper nouns representing persons, locations, etc. For example, searching the ontology for candidates for String Entity “United States” can match 3171 possible Canonic Entities (it matches many more as a substring, but only 3171 remain after the cleaning step – explained later in the GER system section), while searching for common String Entities like “book” yield a much lower 44 possible Canonic Entities for example (including the 6 senses of the word “book” itself available in WordNet, along with other possible classes like `cook_book` or `picture_book`).

In the following table we present the results obtained for the three categories analyzed. We have the *N*, *E*, *V* and *Q* average for each individual category. Also we have recorded the algorithm processing time for each step individually for every group of String Entities. Each step is then averaged by summing the time and dividing it to the number of groups.

Table 4. Algorithm average run-time grouped by document category

Property	Wikipedia	Blogs	News	All Combined
N average	6.7	5.7	6.9	6.4
E average	585.6	345.9	680.3	537.2
V average	114.7	91.3	134.4	113.4
Q average	7505	458	6871	4945
Step 1 (ms)	950	139	241	433

Step 2 (ms)	5412	107	1192	2237
Step 3 (ms)	916	6	220	380
Step 4 (ms)	14410	177	3733	6107
Step 2 + 3 (ms)	6328	114	1412	2618
Step 1 + 2 + 3 + 4 (ms)	21688	291	5145	9041

The table shows some interesting results. For example, while the Wikipedia documents have in average slightly smaller graphs than news items, the number of Result Sets Q is larger, and also the processing time is significantly higher. This shows that the graph structure is more complex for Wikipedia documents. Blogs on the other hand have lower numbers of vertices per graph as well as almost half the number of edges, thus making the average computational time (step 2+3) very small, 114ms.

About the runtime, the Wikipedia documents have came up with some very long and difficult sentences, that have taken in some cases around 1-2 minutes and also very few sentences that were so complex (so many related named String Entities with many vertices in each partition) that processing time exceeded 5 minutes. Clearly such extreme running times for just a set of entities make the system unusable even in offline processing. However graph complexity is independent of the algorithm that will be run upon it and depends directly on the previous computation. Thus, we include all times in the average document run-time. Because of these border cases with very high complexity the average for Wikipedia for step 2 for example is almost 5 times bigger than for the news. However, for the vast majority of cases run-time is around 50-300 ms, faster than the time it takes to parse the sentence (which is around 500ms for the Stanford Parser).

The results show an algorithm adapted for the particular scenario of k-partite graphs that exhibits good performance given the exponential nature of the problem.

V.4. Integrating the Linker Algorithm into the General Entity Recognition System

In the previous chapter we have investigated the proposed Linker graph algorithm as out-of-context as possible, as it can be abstracted and presented as a stand-alone algorithm for solving a particular graph problem. This chapter will present the entire General Entity Recognition (GER) system with which the algorithm is integrated with.

As seen in the system architecture diagram presented at the beginning of this chapter, the Linker Algorithm is actually the last sub-module in the process-flow of the system. Each module with its sub-modules will be presented in order, showing the required steps and proposed methods to obtain a set of Canonic Entities starting from a text document.

V.4.1. Module A - NLP Module

This first module handles document preparation for processing. It is basically a full NLP pipe. The NLP pipe is the standard treatment applied to texts, meaning sentence detection, tokenization, noun transformation from plural to singular form, Part-Of-Speech tagging, parsing, Named Entity recognition. For most of these tasks we use Stanford CoreNLP package[51].

Initially, the document is split into sentences. This is done by Stanford's SentenceAnnotator (the splitter uses a Maximum Entropy model to detect sentence boundaries). This ensures with rather high accuracy that sentences like "The book is written by J.R.R. Tolkein." are correctly identified and not split whenever encountering a comma or other punctuation. Each sentence is then split into tokens by Stanford's tokenizer. Next each token is further processed. Each token contains the *original string*, its *singular form* (for nouns), its *Part of Speech*, *word type* (whether it is a common word, a proper noun or punctuation mark), *named entity type* (if it is a proper noun, then what kind of proper noun it is).

The original string is the token itself. The singular form of common nouns is obtained using the Inflector class[52] (Java implementation) of JBoss[53] Community DNA open middleware. This is a required step as entities in the ontology are all in the singular form. The Part-Of-Speech tag is obtained from Stanford's POS Tagger [132] (a Maximum Entropy model). The detection of common words is done using a free English dictionary, with added heuristics (such as ignore capitalized words and words that start the sentence – they are always capitalized and the dictionary is not a sufficiently accurate measure for these cases). Punctuation marks tokens are detected using regular expressions.

If a word is not a common word (words not found in the noun dictionary, capitalized words that start the sentence, words which are recognized as having a NNP POS (Nominal Noun Proper)) then it might be a named entity. A named entity type can either be a person, a location, an organization or other. We have taken these 4 major categories because they are the standard used for named entity recognizers including Stanford's Named Entity Recognizer [109]. This NER uses a Conditional Random Field implementation to detect whether a proper noun could be one of the four broad categories above. It is useful for the system in a later phase as it allows the cutting down of unlikely Canonic Entities for each String Entity.

[51] http://nlp.stanford.edu/software/corenlp.shtml
[52] http://docs.jboss.org/modeshape/0.4/apidocs/org/jboss/dna/common/text/Inflector.html
[53] http://www.jboss.org/

After token processing, the sentence is parsed using Stanford's Parser [133], storing the syntactic tree and the dependency tree.

The final step of this module is to reiterate over the tokens sequentially and extract String Entities. String Entities are basically noun tokens with added properties. A String Entity can be composed of a single token (ex: "government") or span several ("ex: "John E. Smith") in cases of proper nouns. Also, each String Entity inherits the entity type and named entity type properties from its tokens. So, a String Entity might be a common word, punctuation (this category is ignored in this stage and further), a unit of measure or a named entity. In case it is a named entity, it could be a person, a location, an organization or another type designated by 'unknown'.

As a summary, this module receives a document and outputs extracted String Entities.

Example: let us consider the sentence "Alan Mulally has just announced the new Focus with a 1.6 liter engine". String Entities are made of single or multiple neighboring tokens. From the example sentence, we identify nouns (proper or common) and other interesting words as: Allan, Mulally, Focus, 1.6, liter, engine. After identification we join named entities together if they could represent a full name. This is done with the help of Stanford's NER and heuristically when finding proper nouns that have not been marked by the NER (automatically setting the named entity type to 'unknown'). If neighboring named entity tokens are in the same sequence (as marked by the NER) then we merge tokens together to create a single String Entity. Otherwise we let each token be an individual String Entity. If units of measure are detected (based on a dictionary) they are always linked together with their values. As such, we obtain String Entities: "Allan Mulally", "Focus", "1.6 liter" and "engine".

V.4.2. Module B - String Entity Processor Module

This second module takes as input the processed sentences from the NLP pipe. It performs two relatively distinct operations on it. First, to each String Entity it retrieves a set of appropriate Canonic Entities from the ontology. Second, it computes an influence matrix that measures the influence of each String Entity on every other based on each sentence's dependency tree.

V.4.2.1. Canonic Entity Assignation

Each sentence comes out of the NLP pipe with a set of identified String Entities. For each of these entities we must assign a set of possible Canonic Entities that the String Entity might represent. We obtain these Canonic Entities from the ontology we use. Using YAGO's `means` relation we can link strings to ontology classes (Canonic Entities).

For example, for String Entity “engine” we ask YAGO to give us all the entities that *contain* the *substring* “engine”. In this particular case, YAGO return for this initial phase 2259 facts, in the form:

Table 5. Example of classes YAGO returns for the query about “engine”

Subject (string)	Relation	Object (YAGO class)
"Alfa Romeo Flat-4 **engine**"	*means*	Alfa_Romeo_Flat-4_engine
"automobile **engine**"	*means*	wordnet_automobile_engine_102761557
"diesel **engine**"	*means*	wordnet_diesel_103193107
"**engine**"	*means*	wordnet_engine_103287733
"**engine**"	*means*	wordnet_engine_103288003
"**engine**"	*means*	wordnet_engine_111417561
"**engine**"	*means*	wordnet_locomotive_103684823

We search for substrings and *not* exact strings because each ontology class has at least one *means* relation, meaning that there are more ways to define a single entity. For example, Canonic Entity United_States is referenced by “U.S”, “Stati Uniti d'America“ or “American Civilization“ (to name a few) through the *means* relation. Also the reverse holds true, meaning that we can define with the same word more YAGO classes. As illustrated in the table above, the word “engine” can mean either a motor engine or a locomotive with equal probability. Searching for substrings means we do not miss any probable Canonic Entity.

Having obtained for each String Entity a set of probable Canonic Entities, we now perform a cleaning step for each String Entity. The first thing the cleaner does is that for each Canonic Entity a String Entity has associated, it ensures that all the individual words of the String Entity are found in the Canonic Entity’s name. For example, for String Entity “New York” Canonic Entity York is removed as a probable entity because it does not contain “New” in its name. This strategy removes a large number of false positive entities but also could sometimes remove a few true positives.

Then, depending whether a String Entity is a common or named entity, it cleans WordNet/non-WordNet Canonic Entities appropriately. As presented in the previous section describing YAGO, the ontology has an almost tree-like topology, with the WordNet hypernym hierarchy on top, then wikicategory classes and then other entities linking to them. This means that common words (nouns for example) can only mean WordNet classes, and named entities only lower level, non-WordNet classes. For the above example, we drop Canonic Entity Alfa_Romeo_Flat-4_engine because it is an individual and not a generic entity. The reverse is done for named entities, where all matching WordNet entities are dropped. In YAGO, entities starting with “wordnet_” and followed by an id are high-level entities (ex: wordnet_engine_103287733).

The cleaning step allows for more heuristics to be applied. For example, a good heuristic we apply to reduce the number of possible Canonic Entities per String Entity is to not allow named entities to contain years or other numbers enclosed in parentheses in their names (for example we discard directly all the individual highways identified by numbers, ex: `Interstate_I50`); another heuristic is to not allow too long Canonic Entities names: if the number of words in the Canonic Entity's name is three times more that the number of words in the String Entity then we discard it (this is done because of the way YAGO was created from Wikipedia, which tends to have many long names); another good heuristic we observed is to remove wikicategory entities altogether, because they are generic concentrators, are neither suited for common nouns or for named entities.

The final cleaning step is applied only to String Entities that are named entities (proper nouns) and is basically a collection of heuristic cleaning rules. A first heuristic is to look for commas in its entity name to identify if it could be a location. For example, for String Entity "California", using the *means* relation in the ontology we find `Farmersville,_California`. Because the word "California" is after the comma, we drop the Canonic Entity because it is a location *in* California and not an entity that could represent California itself. However, for String Entity "Farmersville", Canonic Entity `Farmersville,_California` is a valid candidate ("Farmersville" is before the comma). In YAGO locations can be identified by the comma separating the city/town/village to its region/state/country.

Another heuristic in this final step is that by using the information provided by the NER in module A we have a general idea whether the String Entity is a Person, a Location, an Organization or in the Other category. This fact alone helps to reduce the number of possible Canonic Entities greatly, and most importantly by removing entire types of *CE*s it prevents the detection of many false-positive Result Sets. To understand how this named entity category cleaning happens, it is useful to know that any entity in the ontology has at least one *type* relation (the type relation in YAGO is basically the generic *Is-A* relation), specifying what type of entity it is. For example, for String Entity "Paris" YAGO returns, among others, *CE*s `Paris` and `Paris_Hilton`. They are of the following *type*:

`Paris` *type*(→) `wordnet_municipality_108626283`

`Paris_Hilton` *type*(→) `wordnet_artist_109812338`

To determine if a *CE* is of a certain type, we have marked a few *CE*s by hand in the ontology as class determiners. For example, any *CE* that links to `wordnet_person_100007846` is a Person, and any *CE* that links to `wordnet_location_100027167` is a Location.

To determine the type of any entity we simply need to walk up the WordNet hypernym tree to see if we reach any of the marked entities. For example, `Paris_Hilton` is a Person, based on the path `Paris_Hilton` *type*(→) `wordnet_artist_109812338` *type*(→) `wordnet_creator_109614315` *type*(→) `wordnet_person_100007846`. The same procedure applies for `Paris` to determine it is a Location. If during the upward walk through the hypernym tree no marked entity is found, then the Canonic Entity is considered to fall in the Other category.

This cleaning step ensures that a named String Entity will not contain Canonic Entities of another distinct type. However, it should be noted, that for completeness (because sometimes YAGO misses to label a *CE* as a type altogether, even though for a human it is obvious it should be labeled as a Person/Location/Organization), the Other category is never removed for any String Entity. This has the effect that named *SE* "Paris" that has been identified by the NER as a Location will contain *CE*s that are of type Location *and* Other (stripping down only Organization and Person).

Even after this cleaning step, we are usually still left with many probable classes. For example, for String Entity "California" we clean more than 90% of the 12000 possible Canonic Entities and we are still left with around 1000 that each could be correct. There are even cases where after cleaning there could be more than 5000 entities. However, the majority of words are common nouns, and they have less than 25 Canonic Entities associated, only named entities having more, usually around 100-500.

V.4.2.2. Influence Matrix Computing

The NLP pipe provides a set of String Entities but it does not provide a measure of influence of an entity over another that we need to take into account in the processing module. Thus, given the set of String Entities and a sentence's dependency tree[54] (obtained in module A) we need to compute a matrix of String Entity – String Entity influence.

For example, for the sentence "The new Hyundai Accent is equipped with a 1.6 liter engine delivering 110 hp." the NLP pipe provides us with String Entities: "Hyundai Accent", "1.6 liter", "engine" and "110 hp".

The obtained dependency tree looks like the following (type-of-dependency, governing entity, dependant entity):

```
1. det(Accent-4, The-1)
2. amod(Accent-4, new-2)
3. nn(Accent-4, Hyundai-3)
4. nsubjpass(equipped-6, Accent-4)
5. auxpass(equipped-6, is-5)
6. det(engine-11, a-8)
7. num(engine-11, 1.6-9)
8. nn(engine-11, liter-10)
9. prep_with(equipped-6, engine-11)
```

[54] Presented in section II.3., more information about dependency trees at http://nlp.stanford.edu/software/stanford-dependencies.shtml

```
   10.partmod(engine-11, delivering-12)
   11.number(hp-14, 110-13)
   12.dobj(delivering-12, hp-14)
```

Figure 24. Dependency tree example

For each String Entity we traverse the tree looking for connections to other String Entities. For our example sentence, we discover the following:

```
computeInfluence for : Hyundai Accent-4 - 1.6 liter-10
   Value:1.0 subject relation but with proxy engine-11
computeInfluence for : Hyundai Accent-4 - engine-11
   Value:1.0 subject relation.
computeInfluence for : engine-11 - 1.6 liter-10
   Value:1.0 direct link for nn(engine-11,liter-10)
computeInfluence for : engine-11 - 110 hp-14
   Value:0.5 proxy for engine - 110 hp using proxy delivering-12
```

Figure 25. Connections found between String Entities

We determine for each String Entity a head word to use as a match in the dependency tree (for multi-word tokens like “Hyundai Accent” we use Accent-4 as the representative for the String Entity and hp-14 for “110 hp” – the number after the dash is the word’s position in the sentence, as words can be repeated in the same sentence with different meanings and influences).

We heuristically assign three distinct influence values between entities. The 1.0 value means strong connection, 0.5 means somewhat connected and 0.1 means no direct link, but used for context.

In the above example Accent-4 is the subject of the sentence and is thus directly linked to engine-11 through dependency link #4 (of type *nsubjpass*, meaning a noun subject relation in passive tense) and 9 (of type *prep_with* meaning the “with” preposition). This link will score 1.0 meaning a strong link between them. The same score is assigned to the link between “engine” and “1.6 liter” due to their direct link (link #8). We also check for so-called “proxy” relations, meaning non-direct links between entities.

We perform a breadth-first search with maximum depth 2 and look for certain link types between entities. For example, entity “engine” is linked to entity “110 hp” by the word “delivering” (links #10 and #12).

We obtain the following matrix:

```
Influence Matrix: (ROW-Subject) has property (COLUMN-Object)
                  Hyund  1.6 l  engin  110 h
   Hyundai Accent: ---    1.0    1.0    0.1
   1.6 liter     : 0.1    ---    0.1    0.1
   engine        : 0.1    1.0    ---    0.5
   110 hp        : 0.1    0.1    0.1    ---
```

Figure 26. Influence Matrix example

As can be seen, the matrix is not symmetrical. Hyundai Accent influences engine, but engine does not influence Hyundai Accent. This is critical in the algorithm processing module as many false-positives result sets can be avoided. Also, the matrix is stored as a spare matrix, because of the large amounts of zero influences among entities (in larger documents there can be at least one thousand String Entities identified).

A note worth mentioning is that the system's performance is strongly connected to this influence matrix. At times, the dependency tree fails to generate correct dependencies and thus the final results will be rather poor due to missed links between entities. However, we assume that parser we use (Stanford's Parser) is among the best currently available. The dependency tree generation, like the syntactic tree generation is a hard problem in itself.

V.4.3. Module C - Canonic Entity Processor Module

The String Entity Processor module provides sentences with delimited String Entities, each entity having associated a set of probable Canonic Entities, as well as an influence matrix between the String Entities themselves. Based on this data, the Canonic Entity Processor module will provide sets of Canonic Entities sorted by descending scores.

Example: for the simple sentence "Bucharest is the capital of Romania" having String Entities "Bucharest", "capital" and "Romania", we expect the Canonic Entity Result Set with the highest score to be (`Bucharest`, `wordnet_capital_108518505`, `Romania`).

However, there are two steps that need to be performed prior to applying the algorithm. First we need to create the graph on which to run the algorithm, and then we need to split the entire array of String Entities into smaller sets that can and should be processed independently. The splitting is needed because it is impractical (performance-wise) to run the algorithm on tens or hundreds of entities in one run, and also because many entities do not have any connection to each other and then the algorithm will yield many result sets having the same score but with combinations of related entity sets that can all be valid. This issue will be further detailed in the Process Group creation sub-section.

V.4.3.1. Operational Graph initialization

The operational graph is the structure from which the input graphs that the algorithm will be run on are derived, so it is the first to be created. It is a large, directed, unweighted graph. It should not be confused with the smaller *N*-partite input graphs that the algorithm is run on, as these graphs are created based on this operational graph every time a group of related String Entities is encountered. Figure 29 presents an example of an operational graph.

As the ontology can be viewed as a graph, then the operational graph itself is actually an ontology, a smaller section of the original ontology. It contains all the probable Canonic Entities from every String Entity as well as YAGO's top level WordNet hypernym tree. In essence, the graph created here is a stripped down sub-graph of YAGO itself.

We need to create this graph and not directly use YAGO due to the following reasons:

1. YAGO is too large to be stored into main memory for our current available machines. After some tests, we determined a machine with at least 12GB of RAM would be needed to load the essential YAGO core in main memory as a graph. Efficient processing (time-wise) cannot be performed using YAGO as a relational or XML database stored on slow media like a hard disk.
2. We prune unneeded links from the graph itself. Not only does this increase performance because the search space is drastically reduced, but it is needed because of the way YAGO stores facts about entities. For example, if we ask YAGO about Canonic Entity `Paris`, we will obtain the facts that `Paris` is of type `wordnet_city_108524735` as well as `wordnet_location_100027167`. The nature of the proposed algorithm requests that we use classes that are most specific, and we need to drop `wordnet_location_100027167` because a location is more general than a city (city is in fact a hyponym of location in WordNet's hypernym tree).

The first step in graph creation is the addition of YAGO's entire top-level hypernym tree. This will generate a graph that contains around 65000 entities and 73000 relations between them. As can be noticed, the hypernym tree is in fact an acyclic graph containing only *`subClassOf`* links starting from the most specific WordNet classes up to the final root class `entity`. A class can have links that skip a few levels up in the tree. This is why the 'tree' has more links than the number of entities. However, it is conceptually easier to speak of this acyclic graph as a tree, and we will maintain this convention throughout this chapter.

The second step is to add all the Canonic Entities assigned to every String Entity that is a named entity. String Entities that are common words have already been included in the first step – they can only have WordNet entities. So, for each named String Entity, every Canonic Entity is added to the graph. However, we add not only the Canonic Entity itself, but other Canonic Entities from YAGO that have connections to the original entity. We find these related entities and links by performing a breadth-first (BF) search on the YAGO ontology starting from the initial Canonic Entity. We limit the BF search to a maximum depth of 3. Thus, for each Canonic Entity belonging to a String Entity we add a sub-graph starting from the Canonic Entity. We restrict the links and entities in this sub-graph with a few rules. First, we allow only 51 out of the almost 100 relations YAGO knows. We do this because the rejected relations are not relevant for our search, for example relation *`hasBudget`* between a movie and its production budget will yield no further links as the budget is a number. Second, we ignore certain entities. For example, `wordnet_physical_entity_100001930` is too general to be of any use and all entities will eventually link to it. Third, each Canonic Entity is a *`type`* of a WordNet class. We accept only the most specific links between the entity and WordNet. For example for Canonic Entity `Albert_Einstein` we accept only the link to `wordnet_physicist_110428004` and not the link to `wordnet_person_100007846` as physicist is more specific than a person. We thus link each entity to the most specific WordNet class. An entity can be linked though to more WordNet specific classes, if the classes themselves are not one-other's hypernym up to a certain height (because all classes eventually meet at the root node).

The operational graph is created after these two steps, containing all the initial Canonic Entities and possible entities that may link them, as well as the complete WordNet tree to which every entity must have at least one link to.

V.4.3.2. Process Group creation

The Process Group creation is a needed step before the algorithm itself is run. A document can contain many sentences, and each sentence can contain many String Entities. For example, a normal Wikipedia page can have around 500 identified String Entities and a longer page more than 2000. The proposed algorithm provides solution sets that are as long as the original String Entity set, and this would lead to very long processing times if the algorithm would be run on hundreds of String Entities at once. Because we consider that String Entities are not related between sentences or even inside longer phrases, based on the influence matrix we obtain small sets of related String Entities that are processed separately.

Example: let's consider two sentences S_1, S_2, each having two String Entities, SE_A and SE_B for S_1, SE_C and SE_D for S_2. In this example the String Entities from S_1 are not related to the String Entities from sentence S_2. Now, for each String Entity we have discovered three probable Canonic Entities. The algorithm has also discovered the following link: *A1-B1, A2-B2, A3-C3, C1-D1, C2-D2* and *C3-D3*, every link having the same value/score. We now have two options: Option A – consider all String Entities together, Option B – consider only groups of related String Entities at a time. A Result Set is as long as the number of String Entities provided.

For Option A. N=4, so we expect Result Sets of length 4. The algorithm provides 9 Result Sets, all having the same score, being the combination of all the links discovered.

For Option B. we have two groups, each with N=2. This in turn generates RS_1, RS_2 and $RS3$ for the first sentence, and RS_4, RS_5, and RS_6 for the second. In total we have 6 Result Sets.

Considering both options it is immediately clear that the result sets from A. are just combinations of Result Sets from B, so basically Option A provided a larger number of Result Sets that are not more informative than those generated by Option B. at the expense of more processing time and more memory used. As a generalization, processing entities that are not related will inevitably generate an exponential number of combinations between the independent groups, as the algorithm tries to maximize overall Result Set score. Furthermore, the average number of related String Entities is usually less than 10, compared to the total number of String Entities in a document which can be orders of magnitude larger (on which it becomes impractical to run such an algorithm). Graphical example:

$$\begin{tikzcd}
 & & SE_A \leftarrow \{A1, A2, A3\} \\
 & \text{Sentence } S_1 \arrow[ur] \arrow[dr] & \\
 & & SE_B \leftarrow \{B1, B2, B3\} \\
\text{Document } D \arrow[uur] \arrow[ddr] & & \\
 & & SE_C \leftarrow \{C1, C2, C3\} \\
 & \text{Sentence } S_2 \arrow[ur] \arrow[dr] & \\
 & & SE_D \leftarrow \{D1, D2, D3\}
\end{tikzcd}$$

Links between Canonic Entities: *A1-B1, A2-B2, A3-C3, C1-D1, C2-D2, C3-D3*

How String Entities are considered:

A. All together (1 group, N=4):	B. Separate groups (2 groups, N=2 for each):
$RS_1 = \{A1, B1, C1, D1\}$	$RS_1 = \{A1, B1\}$
$RS_2 = \{A1, B1, C2, D2\}$	$RS_2 = \{A2, B2\}$
$RS_3 = \{A1, B1, C3, D3\}$	$RS_3 = \{A3, B3\}$
$RS_4 = \{A2, B2, C1, D1\}$	$RS_4 = \{C1, D1\}$
$RS_5 = \{A2, B2, C2, D2\}$	$RS_5 = \{C2, D2\}$
$RS_6 = \{A2, B2, C3, D3\}$	$RS_6 = \{C3, D3\}$
$RS_7 = \{A3, B3, C1, D1\}$	
$RS_8 = \{A3, B3, C2, D2\}$	
$RS_9 = \{A3, B3, C3, D3\}$	

Another strong argument of group creation is that this setup of partial results suits well to parallelization, where independent processors can handle independent entity groups, because the operational graph is read-only and thus can be shared between threads each handling its own group of related String Entities. Furthermore, the vast effort of graph searching would be wasted as entities that are not related will likely not have connecting paths between them.

Thus, we need to find the smallest independent groups of String Entities. This is achieved by applying the flood-fill algorithm on the influence matrix *Inf*. First we create a copy of the influence matrix where every value that is non-zero is replaced with a 1.0 (a black/white table). We find the first non-zero element Inf_{ij} (which in the matrix means that entity in row *i* influences entity in column *j*) and start zeroing any element that it influences or is being influenced by while in the mean time adding these elements to a new Process Group ("flooding" the connected elements). This flood-fill is a breadth-first graph search on the matrix. We repeat the process until the entire matrix is zeroed out and we have obtained all the independent groups of String Entities.

In practice, we have observed that most often almost all entities in a sentence are related, even if only for context (exception being long phrases that contain more sentences not separated by usual punctuation). This basically narrows down the problem to working on a single sentence at a time.

V.4.3.3. Linker Algorithm

This sub-module is applied to each Process Group independently. The input here is a Process Group containing String Entities that each has a list of probable Canonic Entities associated, and the operational graph from which to derive the input graph for the algorithm. So, two phases are identified: first the input graph is obtained from the operational graph, and then the algorithm is run on it. The output is a list of sorted Result Sets.

The creation of the input graph for the current Process Group is based on the following algorithm:

```
function obtainInputGraph () {
  initialize empty graph G;
  for $SE_i$ in the current Process Group (i = 0 → N) {
    for every $CE_i$ in $PCE_{SEi}$ {
      initialize BFiterator for BF starting from $CE_i$ on the op. graph;
      while (BFiterator) {
        $CE_q$ = BFiterator.getCurrentEntity();
        If ($CE_q \in PCE_{SEj}$ with $i \neq j$) {
          edgeWeight = getEdgeWeight($CE_i$, $CE_q$);
          add to G vertices $CE_i$ and $CE_q$ (if not already added);
          add to G edge between $CE_i$ and $CE_q$ with weight = edgeWeight;
        }
      }
    }
  }
  return G;
}
```

Figure 27. Pseudocode for obtainInputGraph() function

In essence, function **obtainInputGraph**() performs a BF search from every *CE_i* belonging to a *SE* on the operational graph. Whenever encountering a vertex *CE_q* that belongs to a different *SE,* it calculates the score of the path between the two Canonic Entities and adds them and the weighted edge to the input graph *G*. The function that calculates the path score is presented next:

```
function getEdgeWeight ($CE_i$, $CE_q$) {
  path = path from $CE_i$ to $CE_q$ in the operational graph;
  score = getMatrixInfluence($CE_i$,$CE_q$) / path.getDistance();
  if path contains changes of direction
    score = score * penalizationCoefficient;
  if path contains links of type "subClassOf" or "type"
    score = score * penalizationCoefficient;
  // other possible heuristics
  return score;
}
```

Figure 28. Pseudocode for getEdgeWeight() function

The **getEdgeWeight**() function calculates a score between two Canonic Entities in the operational graph based the path between them. First, the score is calculated as the value in the influence matrix between the String Entities representing them, divided by the distance between them. For example if a two String Entities are strongly connected (influence matrix value of 1.0) but their representing Canonic Entities are found to be linked by a path of length 2 (meaning an intermediate Canonic Entity), then the initial score will be 1.0/2 = 0.5.

Next, we apply some heuristics, such as the one used by Hirst and Onge [134] in their semantic distance measure for WordNet, penalizing the changes of direction in the path from one entity to the other, or discriminating between relation types. For example if between two named entities there are 2 or more links of type `subClassOf` or type then we penalize the score. A link like `Entity_A` *type*(→) `wordnet_village` *subClassOf*(→) `wordnet_city` *type*(←) `Entity_B` is not very informative, and can lead to erroneous results, linking two entities just because they are of the same general type in this instance. The penalization coefficient is a variable, set heuristically at 0.5, halving the score whenever encountering an unwanted path type. The resulting score is then returned as the weight the edge between the two Canonic Entities will have in the input graph.

In summary, **obtainInputGraph**() performs several BF searches on the operational graph to build an undirected weighted input graph.

System Example: Revisiting the initial example of this chapter, we illustrate here the process for the sentence "He replaced the pipe giving his car new life - his Santa Fe now runs quieter.". String Entities "pipe", "car", "life" and "Santa Fe" are extracted in the first module of the system. Next, in module B the influence matrix is computed and Canonic Entities from the ontology are associated to each *SE.* Next, in module C the operational graph is created. First the WordNet hypernym tree is added, then every *CE* from each *SE* is taken as a starting point for a breadth-first search in the ontology, and every neighbor of the *CE* is added to the operational graph, up to a depth of 3. The figure below shows this operational graph (only a small section). We consider that we only have a single Process Group containing all the String Entities.

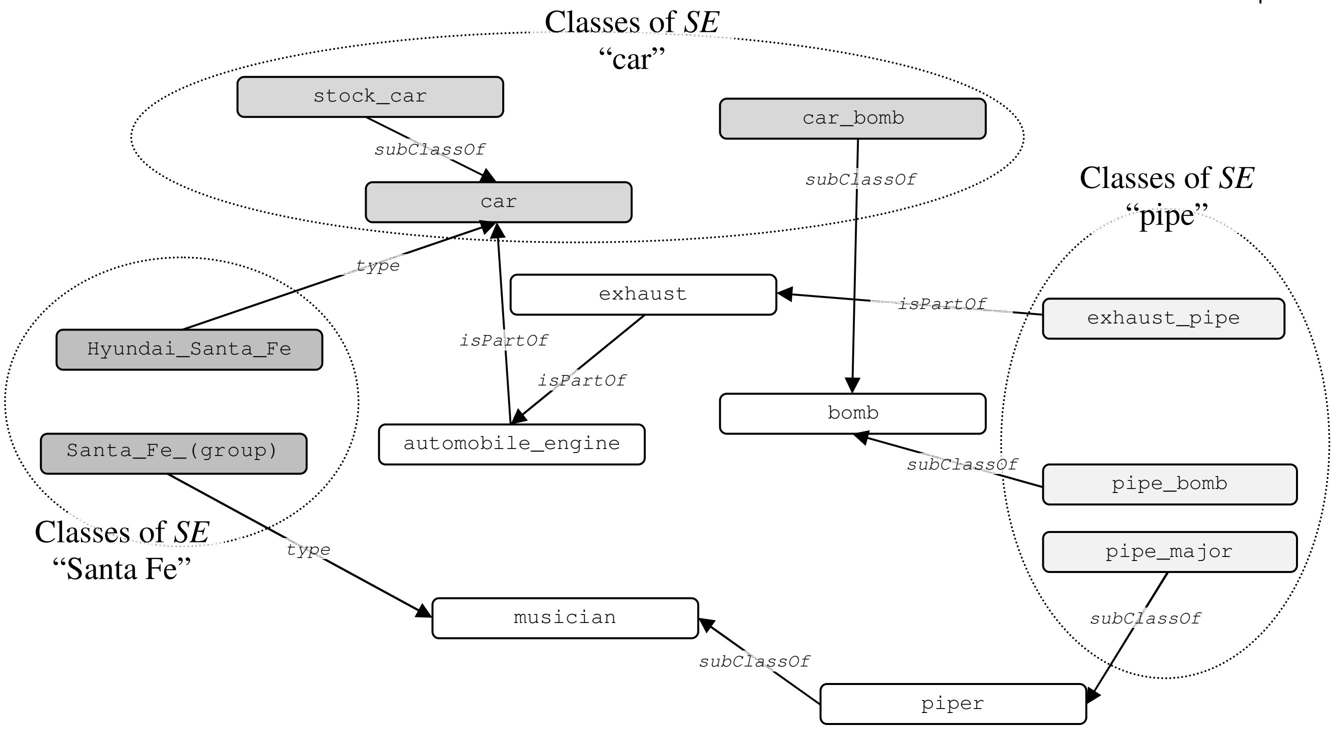


Figure 29. Operational graph

The **obtainInputGraph**() function is run on the graph for the set of String Entities. *SE* "life" is not shown in the image as no path was found from any of its *CE* set to any *CE* of other *SE*s. Paths exist, but they are of length greater than 3, and thus ignored. For every starting CE the breadth search is performed and a graph like the one in the following figure is obtained:

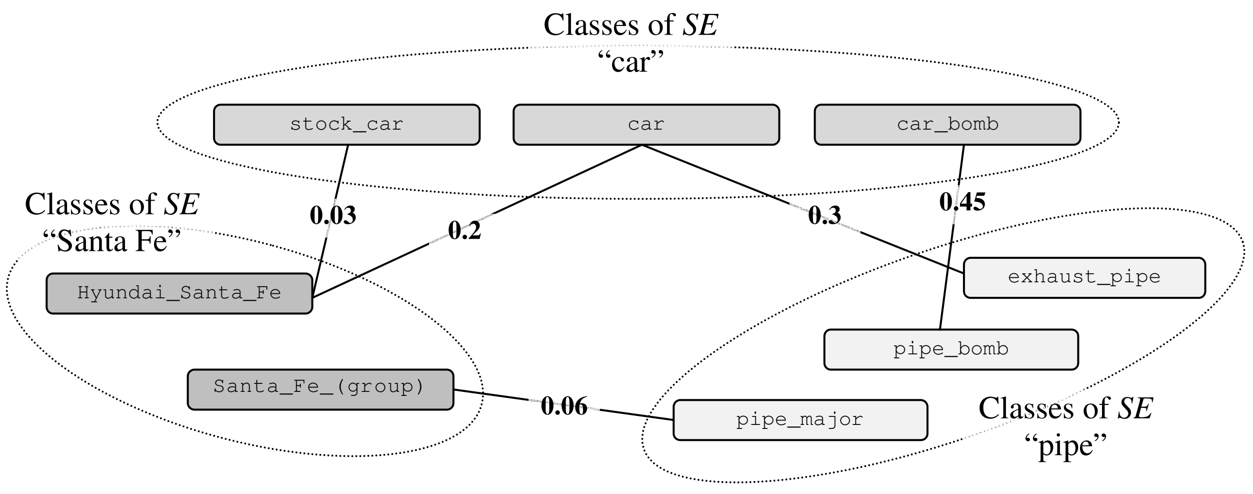


Figure 30. Input graph derived from the operational graph

The obtained input graph is much smaller and simpler. The only vertices are the starting Canonic Entities. The undirected weighted edges represent the paths between the Canonic Entities from the operational graph. The input graph created in this manner is actually a k-partite graph. For example, even though *CE* `stock_car` does not have a direct link to any other *CE* belonging to a *SE*, it is still connected to `Hyundai_Santa_Fe` through `car`. In the input graph thus choosing `stock_car` over car is a valid choice, if the combined score of the chosen entities would be higher than using directly `car`.

After obtaining the input graph, the Linker Algorithm is applied. It runs the four steps presented in the previous section in sequence.

First it searches the graph for any duplicate links and creates the edge dictionary and the vertex hash set.

Then, in the second step, it performs a custom depth-first search. At the end of each search, whenever adding another vertex is not possible, respecting the constraint that a solution cannot have two vertices belonging to the same partition, it adds the path obtained until that point to the Result Set Array. It is added to the array if the solution is not a subset of another Result Set, in which case it is discarded, or it cannot be merged with any other Result Set.

After the search space has been exhausted, for every solution it computes its score based on the weights of the edges, and then sorts the Result Set Array (only required if the following step 4 is not applied).

The last step is to obtain a merged Result Set Array, where non-overlapping Result Sets are combined to create the most specific Result Sets possible. This larger Result Set Array is finally sorted, and represents the solution to the problem of detecting the best choice of Canonic Entities that represent the String Entities extracted from the text document.

V.5. System evaluation

We start the GER system evaluation first from a computer hardware point of view. We have run the complete system on a standard 2.8 GHz, 64 bit machine with 8 GB of RAM. RAM is largely needed to store the models used by the Stanford CoreNLP and other support tools, at almost 3 GB in total. The developed system itself uses at maximum another 2-3 GB, for everything from the syntactic and dependency trees, influence matrix to the operational graph and edge dictionaries for the algorithm.

Due to the splitting of string entities into independent Process Groups, the algorithm computationally performs very well, because usually in a single set there are no more than 4-8 entities, a number for which processing is almost instant, even though there usually are anywhere from a few tens to a few thousand of probable entities (vertices in the graph) for each String Entity in the individual set. Even better, due to the independent nature of process groups, they can be run in parallel without any algorithm modification (as explained in a previous section).

A point needed to be made, the bottleneck of the system in terms of run-time is the ontology interface. Even though a query is answered in milliseconds, there are thousands of these calls to the database. From obtaining the *PCE* for every String Entity to creating the graph by starting a BF search on the ontology from every *CE* in every *PCE* (that could potentially have thousands of additional *CE*s discovered), the need to access the hard drive for the vast majority of them (the database cache is almost useless here as almost every new query is different from the previous ones) is actually by far the slowest part of the system (more than 95% run-time is lost here).

V.5.1. Evaluation methodology

Evaluation of the system's results from an accuracy point of view is a somewhat difficult task as we have found no other systems to compare ours with because of our particular setting: we cannot apply the system to reference test corpora like the ACE 2003/2004 or other similar Sens/SemEval corpora because we rely on a large generic ontology and not on a subset of entities, and we handle both named and common entities (basically the proposed system does not fit completely into any of the ACE/MUC/SemEval tasks). Also, we cannot restrict our working entity set because the system is working better the larger the entity set is and the connections number within it. Our knowledge base is actually our entire search and result space. The larger the number of entities and relations, the larger the number of resulting assigned classes. However, we can manually create a set of tests and measure our system's performance against them.

As such, in our problem setting, we measure the accuracy of the Canonic Entities assignation in the following manner: for example, for the sentence “Smith was born in Farmersville, a small town in California.” we extract String Entities (“Smith”, “Farmersville”, “town”, “California”). For the Result Set (∅, `Farmersville,_California`, `wordnet_town`, `California`) we assign a 4/4 (100% accuracy) score because it matched all the preset entities: 1. it correctly identified that Smith could be any person (“∅” meaning that either YAGO does not contain any possible canonic entities for “Smith” or more likely that no links have been found between any canonic entities representing Smith to any other entities), 2. “Farmersville” is correctly identified by Canonic Entity `Farmersville,_California`, 3. “town” is correctly identified by `wordnet_town` and 4. “California” is correctly identified by `California` Canonic Entity. If for example instead of ∅ it would have been Canonic Entity `John_Smith`, then accuracy would have dropped to 3/4 (75%), because even if there is some long, improbable, low scoring path between John Smith and Farmersville, such as `John_Smith` *`bornIn`*(→) `San_Francisco` *`type`*(→) `city` *`type`*(←) `Farmersville,_Califonia`, for a human there is no logical link, because we know (or at least agree by general consensus or by probabilistic reasoning) that no generic John Smith was actually born in a small town in California named Farmersville. We thus evaluate the system against human judgment on which Canonic Entities should correctly represent String Entities. Accuracy is calculated as the number of correctly assigned Canonic Entities to String Entities divided by the total number of String Entities.

V.5.2. Evaluation set and standard creation

The evaluation set consists of 40 sentences, each with minimum 3 String Entities and a maximum of 14. The sentences belong to Wikipedia snippets and news article phrases. In total, the 40 sentences contain in total a number of 211 String Entities. This averages to 5.3 String Entities per sentence. The distribution of named versus common String Entities is a bit different from the 34%/66% obtained from the document test set for algorithm performance, to 45%/55% in our sentence set (2.4 named String Entities and 3.1 common String Entities per sentence or 95 named and 116 common String Entities).

To obtain a “gold standard” (a test set considered as 100% accurate by humans) a small application was created to reduce the time needed to create the standard. For each of the 221 String Entities the application returns every YAGO Canonic Entity that could represent that entity. So, for every String Entity extracted, its Possible Canonic Entity set was obtained, but sorted in a tree-like manner by entity type, for annotator ease of usage and speed. For example, when searching for String Entity “Maryland” YAGO returns exactly 1776 Canonic Entities (before cleaning – the same cleaning method as the one described in the system in a previous section) like:

```
-- "Maryland" --
Maryland_Route_396
Maryland_Institute_College_of_Art
Maryland_(automobile)
USS_Maryland_(SSBN-738)
Maryland_Symphony_Orchestra
Maryland_Exiles
…
Maryland
…
```

As there are four major entity types (for named entities only), four checkboxes were implemented: Person, Location, Organization and Other. Clicking on any will show only Canonic Entities of that type (all Canonic Entities have the type property linking them in the WordNet hypernym tree, thus being able to detect the type of entity). For example, checking Location will show only entities like `Maryland` or `Sandy_Spring,_Maryland`. This is done by checking the *`type`* facts belonging to each Canonic Entity. For example, for `Maryland` we discover that Maryland is of *`type`* `wordnet_district_108552138`. When further investigating `wordnet_district_108552138` we find it is actually a *`subClassOf`* `wordnet_region_108630985` which in turn is a *`subClassOf`* `wordnet_location_100027167`. Any entity that links up to `wordnet_location_100027167` is a Location entity. Similarly for Person and Organization. If it is does not reach any of these three predefined entities, it falls into the Other category.

However most of the entities are of type Other. For entities that are unknown for the person creating the standard, clicking on an entity will display in an adjacent window the entity's properties. For example, at first sight entity, a person does not know what `Maryland_Exiles` mean. Clicking on the entity in the list will show up the following properties:

```
Maryland_Exiles
type → wikicategory_US_rugby_union_teams
type → wordnet_team_108208560.
describes ← http://en.wikipedia.org/wiki/Maryland_Exiles
```

From this information alone it is obvious for a person that when speaking whether Maryland won this season we are actually talking about the rugby team named Exiles, and the fact that String Entity "Maryland" in that sentence should be represented by `Maryland_Exiles` (rugby team) and not, for example, by `Maryland` (location). The arrow displays the relation direction. A right pointing arrow indicates that `Maryland_Exiles` is the subject. An inverse relation means `Maryland_Exiles` is the object.

For common String Entities we come up on another problem. While for named entities the problem was the large number of them, for common entities the issue of sense becomes the main problem. For example for String Entity “bank” (in a sentence where bank is used in the economic context) we find Canonic Entities like:

```
wordnet_agent_bank_108418316
wordnet_bank_100169305
wordnet_bank_102787772
wordnet_bank_108462066
…
wordnet_bank_113368318
…
wordnet_banker_109837720
…
wordnet_piggy_bank_103935335
…
```

In this type of list we easily figure out the correct entity, which is (for our example) a bank, without any modifiers. However, here comes the problem – there are 8 senses for `wordnet_bank_#id`. For this problem the most obvious choice (as we don’t have access to the glosses in WordNet) is to move up the hypernym tree. The same mechanism of hovering or clicking on a named entity will now show for a common entity the following:

```
wordnet_bank_100169305
         → 0>subClassOf wordnet_flight_maneuver_100170844
         → 1>subClassOf wordnet_maneuver_100059552
         → 2>subClassOf wordnet_evasion_100059127

wordnet_bank_102787772
         → 0>subClassOf wordnet_depository_103177349
         → 1>subClassOf wordnet_facility_103315023
         → 2>subClassOf wordnet_artifact_100021939

wordnet_bank_108462066
         → 0>subClassOf wordnet_array_107939382
         → 1>subClassOf wordnet_arrangement_107938773
         → 2>subClassOf wordnet_group_100031264

wordnet_bank_109213434
         → 0>subClassOf wordnet_ridge_109409512
         → 1>subClassOf wordnet_natural_elevation_109366317
         → 2>subClassOf wordnet_geological_formation_109287968
… etc …
```

From this display it is obvious for a person that the correct choice is the second one: `wordnet_bank_102787772`, considering that its direct hypernym is a depository

Using this small application that automates YAGO discovery, the three persons annotating the sentences took only a few seconds to a minute per String Entity to select the correct meaning (as opposed by just navigating a very large list of possible entities which would have taken a long time). This created the "gold" standard needed on which to test the system against.

Before moving on to evaluate the system using this standard, it should be noted that the task of annotating is difficult, in the sense that different people annotate differently. For example, when annotating the simplest of sentences: "The car has an engine", for String Entity "engine" we obtain the following possible Canonic Entities:

```
wordnet_internal-combustion_engine_103579982
wordnet_automobile_engine_102761557
wordnet_gasoline_engine_103424630
wordnet_aircraft_engine_102687423
wordnet_gas_engine_103422771
wordnet_engine_103287733
…
```

Given these choices, which is the correct one? In essence all could be correct as they are just more or less specific types of engine. The sentence itself does not say that the car is a diesel or a petrol, so maybe the types of engine that specify that should be rejected as valid Canonic Entities. Given the lack of any additional information in the sentence, the car could actually be powered by a steam engine or even a jet engine (in the case of land-speed record vehicles). However, the choice between `wordnet_engine_103287733` and `wordnet_automobile_engine_102761557` is less clear, as the term "car" in usual usage is actually an automobile. Because of the same lack of information, we do not know if it is an automobile, but common logic says it is, based solely on that almost all the times when we use the term "car" we are referring to an automobile.

This issue was resolved by letting the annotators choose *multiple* correct choices. However, to enforce some strictness, a Canonic Entity was considered valid if two of the three annotators marked it as correct.

It should also be noted that String Entities that were found to not have any correct Canonic Entity were marked as *null*, meaning that the system should not pick any *CE* to represent that *SE*. This happens in two cases, first if YAGO does not know about an entity (YAGO was created on a Wikipedia dump from 2009 and there are official persons in the news that were unknown then, thus impossible to appear in Wikipedia and therefore YAGO), or if the String Entity denotes an generic entity (ex: sentence from a blog entry: "Ann walks among the houses, … ", where Ann is just a normal person, that should not have a corresponding Canonic Entity in the ontology).

We estimate an ITA (inter-annotator agreement) for the current task of around 60% (given that the annotators were not related to the NLP/linguistic field). Similar results were obtained for fine-grained tasks, for example [94] reports an ITA on WordNet senses between 67% and 80%. The most common issue was which and how many of the selected Canonic Entities to be allowed in the "gold" standard considering that annotators sometimes picked several general and specific entities as correct. The ITA was calculated the number of times that at least two annotators came up with the same correct Canonic Entity set per String Entity divided by the total number of String Entities.

V.5.3. Testing the system

The system was run, and we evaluated the first Result Set for every processed group of String Entities (the system outputs a descending sorted array of Result Sets – in this case we only looked at the first *RS*). We obtained an arguably low/average performance of 22.3% for this strict evaluation method.

Performance is affected because in many cases we run into one or both of the following issues:

Issue 1: the system cannot yet discriminate between similar scoring Result Sets with similar entity types. Given the sentence "Alan Mulally has just announced the new Focus with a 1.6 liter engine." with string entities "Alan Mulally", "Focus", "1.6 liter" and "engine") and the first two scoring Result Sets:

```
RS1: 2.0 (ANY, Ford_Focus_WRC, wordnet_liter,  wordnet_automobile_engine)
RS2: 2.0 (ANY, Ford_Focus, wordnet_liter, wordnet_automobile_engine)
```

As can be seen, the only difference between the two Result Sets (both scoring equally at 2.0) is that "Focus" could be either a Ford Focus WRC or a generic Ford Focus vehicle. Both entities are present in YAGO with the same type of links, and no information can differentiate one over the other. Because YAGO does not know that the WRC Focus is actually a modified type of standard Focus, then it will treat both entities as equal possible representatives for String Entity "Focus".

Issue 2: the ontology lacks information in the form of relations between entities, and the system biases certain links to compensate for the lack of this information by penalizing a few link types. The act of finding a suitable coefficient for penalization, as the entire heuristic penalization method itself is just an attempt to "correct" the choices the system makes, usually with different degrees of success – a certain coefficient will generate good *RS*s for a sentence and break other previously-well performing sentences. In quite a few Result Set Arrays we find Result Sets with the correct choice for Canonic Entities having a score just a bit lower than the best scoring *RS*, because of the penalization coefficient. Just like issue 1, this issue is unavoidable.

We propose two more 'forgiving' evaluation methods, in which we relax allowed results.

The first of the two evaluation methods implies ignoring issue 1. This means we look to see if in any of the *equal top-score* Result Sets we have correctly identified Canonic Entities. To exemplify this relaxation, if we take the Ford Focus example above, we would get for that sentence a (4/4) 100% accuracy, because even though the system's default choice is RS_1 which only evaluates to 3/4 (75%) accuracy, we inspect also RS_2 because it has the same top score, and we detect that RS_2 actually provides a better 4/4 (100%) accuracy.

The relaxation of the second issue means that we allow searching for correct results in lower scoring Result Sets. Result Set scores usually are not distributed linearly (meaning Result Sets have many slightly different scores) but tend to be distributed in a step-like manner (meaning that we have relatively few different scores, implying many Result Sets having the same score). Because of this property, we allow searching for correctly identified Canonic Entities in Result Sets having the second- and third-best scores.

The table below shows the accuracy obtained when using these new evaluation criteria.

Table 2. Accuracy of system against a manually created standard

Evaluation method	**System Accuracy**
Strict evaluation (first *RS* only)	47 / 211 (22.3%)
Evaluation w/o issue 1	76 / 211 (36.0%)
Evaluation w/o issue 1 & 2	89 / 211 (42.2%)

The results show that when evaluating on somewhat more relaxed criteria, the initial accuracy almost doubles, from the initial figure of 22.3% to 42.2%. The last figure itself is quite impressive, meaning that in almost half of the cases the system was able to determine the matching Canonic Entities within the first few top scoring results.

While our initial overall results with this system are average, we can conclude on some points:

First, the results depend heavily on the type and composition of sentences tested. For sentences with entities in areas of the ontology with higher information density, results are usually better, because of the increased link number and not necessarily because of the scoring function. This function is an important performance affecting factor: we have used a distance-based function, which is sensitive to information density fluctuation, a problem practically unavoidable in large general ontologies.

Second, calculated accuracy depends even more on the human created standard to which results are evaluated against. But currently we can only evaluate the proposed system on such a standard. The standard was created by people reviewing possible classes extracted from YAGO manually and assigning them as correct answers to each String Entity. Even so, misunderstandings have been rather common between the annotators because of the large number of apparently correct classes. Also, a standard baseline was very difficult to establish. Standard baselines like random-sense or first-sense are hard to implement because we work with both named and common entity identification, meaning we do not have a 'first sense' as we could have had if evaluating only common nouns for example. Also, because of the number of seemingly good responses (especially for named entities) among a very large number of possible classes, a random baseline would yield uninformative results. For example, String Entity "Hyundai" could mean the ship building company, the auto company or any of its 30+ car models, all being named entities. Though not comparable, for a general overview, SemEval 2007[55] yielded results in the 50%-60% performance range for fine-grained tasks (with a maximum 10% above the baseline for the best system for their 465 tagged words), underlining the task's difficulty.

Third, context is highly important. For example for sentence "Hyundai has launched a new car named Santa Fe.", with string entities "Santa Fe", "car" and "Hyundai", we obtain the Result Set (`Hyundai_Tucson`, `wordnet_car_102958343`, `Hyundai_Santa_Fe`) scoring 2/3 accuracy because the system thinks that "Hyundai" could mean `Hyundai_Tucson` which is a car similar to its partner *CE* `Hyundai_Santa_Fe`, instead of the arguably correct `Hyundai_Motor_Company`. However, for the sentence "Hyundai has launched the new Santa Fe." we obtain (`Hyundai_Motor_Company`, `Santa_Fe_Industries`), because of the conceptual link between Santa Fe Industries and Hyundai Motor Company as they are both industries, and missing the link to the auto vehicles because of insufficient evidence for Santa Fe being a car;

Forth, the proposed algorithm efficiently makes the most of the information available to it. Where links are available, it finds all possible connections, evaluates them all in a single pass instead of processing an exponential number of entity combinations, and based on the scoring method, creates the result best sets given the available information.

V.6. Conclusions

In this chapter we have presented a knowledge-based system that presents a viable algorithm and encouraging first results for entity identification and correct class assignation from ontologies. We aim to show that ontologies can be used for more than just standard classification of the entities they contain, and that the structure itself of such large generic ontologies can be used to generate added value. Furthermore, we have presented an algorithm that provides fast results in a single pass for the current problem of evaluating the best combination of every possible entity assignation. Using a standard combinatorial approach where each entity would be tested against every other, the problem would quickly grow unsolvable even for a few entities.

[55] SemEval 2007 - http://nlp.cs.swarthmore.edu/semeval/index.php

As a conclusion we note the major issues that influence performance to a large degree:

1. Dependency tree generation. In most cases the tree is correctly generated, but it also happens that the parser misses or incorrectly assigns dependencies between words that lead to a poor starting point for the influence matrix creation.

2. Matrix creation rules. The matrix is generated by parsing the dependency tree. As rules are heuristically created, new rules or improved versions can be implemented.

3. Scoring function. Same as the matrix creation, the scoring function has been heuristically chosen. As with existing similarity measures for WordNet for example, variations of the scoring function applied in the same algorithm can be created for improved system performance. While we used a distance-based scoring method, which by default suffers from large variations in information density [96], it does provide a good performance and is applicable to both named and common entities, even though named entities are linked in a random graph of direct links while common entities are linked in a hypernym tree. For this reason a conceptual-density [125] measure is arguably risky to implement.

4. Knowledge source. The most important factor in the system's performance by the largest margin is the ontology used. For sentences where there is a large information amount about a subject, results will be surprisingly good, while lower information densities will yield poor results. As time passes and knowledge sources get richer, even without any change to the system, its performance will increase.

VI. A knowledge-based approach for document classification

VI.1. Introduction

This chapter presents a knowledge-based, unsupervised approach to the problem of document classification in respect to a set of topics.

The system we propose takes as input unclassified text documents and a set of possible topics, and outputs the n-best possible topics for each processed document. It uses the ontology as a knowledge source on which it applies graph algorithms to detect and create a partial sub-graph illustrating the relations between the concepts that characterize each document. Thus, our solution avoids the use of machine learning algorithms in the main processing phase, while only employing such algorithms in the document pre-processing phase for sentence identification, token splitting and named entity recognition (standard NLP pipeline).

The proposed approach is presented as an implemented, working system that uses the YAGO ontology as its knowledge source in order to perform unsupervised, natural language document classification. We also engage in a discussion on the benefits and problems of using ontologies for such a task.

The system presented in this chapter, while using some of the same methods and tools as the GER system presented in the previous chapter, represents a distinct contribution with a different goal.

VI.2. Domain Literature Review

The domain of text classification is, at present, dominated by machine learning and statistical methods, with knowledge engineering methods trailing behind [135]. While a large variety of approaches can be observed, the best performing systems consistently use algorithms like SVM (Support Vector Machine) to achieve consistent and good results, a class of supervised machine learning (ML) algorithms.

ML algorithms like SVM, Naive Bayes or Maximum Entropy are relatively simple to understand and use, and unlike knowledge engineering methods, they do not require large knowledge-bases to be manually pre-defined by engineers. Also, this category of systems is not domain related, unlike most knowledge engineering approaches which are focused on sub-domains (as it happens, for example, in the medical domain where compact parts of consecrated ontologies are adopted for certain medical specializations). The functioning of these algorithms usually requires the “translation” of the documents into feature vectors. Common construction of feature vectors involves term frequency, document frequency, term frequency and inverse document frequency combined, information gain, term strength, and chi-statistic [136] [137].

Latent Semantic Indexing (LSI, also known as Latent Semantic Analysis or LSA) has been used in conjunction with WordNet or other domain ontologies to reduce the dimensionality of feature vectors [138] [139]. The main idea of LSI is that there is a semantic structure between words in a document that can be discovered and used to group similar documents into similar space structures using statistical analysis. Using LSI means that after document preprocessing, the document vector is obtained (in the form of d={$(keyword_i,weight_i)|i$=1..n}, its dimensionality is reduced using LSI and then it is compared to every category vector topic. The category vector that is closest to the document vector is the topic assigned to that document. [140] showed an slight increase in performance when using LSI and an ontology as opposed to simply using a Naïve Bayes classifier (or equivalent) and an ontology. In [141] we developed a text classification method where the LSI technique was combined with a WordNet-based text analysis. However, while LSI is effective in mitigating word similarities, it is quite difficult to maintain such a system when the document size varies and any modification of the initial set of documents requires the entire semantic space to be reconstructed [142].

Concerning the ontology-based approaches of text classification, it can be observed that domain ontologies are most often used [143]. Domain ontologies are usually small and contain very specific facts about a domain, like certain group of illnesses for the medical domain, names and hierarchy of wines for the oenological domain or car parts for the automotive domain. When applied to a collection of texts from a certain area, a domain ontology focusing on that area will be much more effective than a general ontology. However, for diverse collections of documents, the use of domain ontologies is no longer possible.

VI.3. System Implementation

This section discusses system architecture and implementation. The system can be logically divided into three major modules: Processor, Analysis and Evaluator.

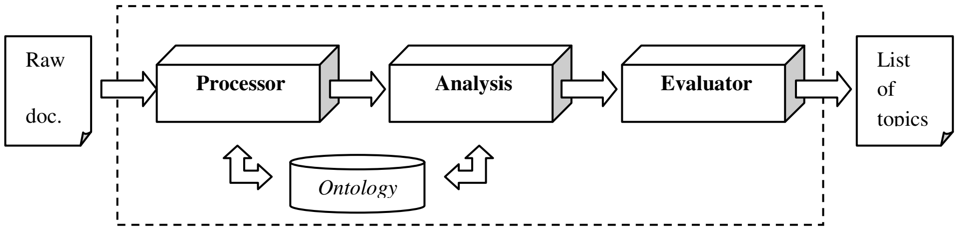


Figure 31. Document classification system architecture

A quick overview of how the system works: First, at initialization phase, the topic list is constructed. Then a document is fed to the Processor where it is parsed and tokens are extracted from it, along with other useful information, as word frequency and word type, form, etc. The tokens are analyzed and String Entities are created based on these tokens. A String Entity is a simple string representing a token or multiple connected tokens (for first and last names or for composed nouns, etc) – we use the definition of String Entity from section V.2. In the Analysis module, the String Entities are searched for in the ontology and possible Canonic Entities (also defined in section V.2. – as a side-note, throughout this chapter we may omit writing their trailing IDs if they are not relevant) are associated to each String Entity. A String Entity can be represented by a Canonic Entity from the ontology. Based on YAGO, a graph containing every Canonic Entity of every String Entity is created. Based on this constructed graph, links are found between topics and String Entities. Thus, topics are scored depending on these links. After all String Entities have been processed, the topics are sorted by their descending score in the Evaluator module. The topic with the highest score is the document's proposed topic. Below, we present each module, starting with the initial topic list creation.

VI.3.1. Topic list creation

The topic list creation is not a module in itself, but rather an essential initialization step, hand-built into the system.

We assume the system will deal with a fixed number N of topics. In our case, $N = 50$, as we use the LA' 94 news articles data collection[56] for evaluation. For each of the topics, we create an array holding a variable number of topic concepts (TC). A topic concept is actually a simple word, concept, idea. Thus, several topic concepts are needed to define one topic.

A topic concept has a name (a simple word – a string), a weight (a real value number) and a score (also a real value number). From an implementation point of view, as the topic concept cannot be represented by a simple string – its name, it contains an array of classes from our knowledge source, along with a weight of the class itself representing how relevant that class is for the topic concept. We use YAGO as the knowledge source, so the array contains YAGO entities.

Example: Given topic #53 (topic ids start from #41 to #91 in our test collection) "Genes and Diseases", we create the following 5 topic concepts:

[56] LA94 news articles collection, http://trec.nist.gov/data/docs_eng.html

```
Topic top = new Topic(“53 Genes and Diseases”);              Topic #53

tc = new TopicConcept("gene",1.0);
      tc.addWord("wordnet_gene_105436752", 1.0);             Topic Concept 1
      tc.addWord("wordnet_genotype_107941405", 1.0);
top.addNewTopicConcept(tc);

tc = new TopicConcept("disease",1.0);
      tc.addWord("wordnet_disease_114070360", 1.0);          Topic Concept 2
      tc.addWord("wordnet_illness_114061805", 1.0);
      tc.addWord("wordnet_disorder_114052403", 1.0);
      …
top.addNewTopicConcept(tc);

tc = new TopicConcept("body",1.0);
      tc.addWord("wordnet_body_105216365", 1.0);             Topic Concept 3
      tc.addWord("wordnet_torso_105549830", 0.5);
      …
top.addNewTopicConcept(tc);

tc = new TopicConcept("human",0.7);
      tc.addWord("wordnet_homo_102472293", 1.0);             Topic Concept 4
top.addNewTopicConcept(tc);

tc = new TopicConcept("syndrome",1.0);
      tc.addWord("wordnet_syndrome_114304060", 1.0);         Topic Concept 5
top.addNewTopicConcept(tc);
```

The above code shows the structure of a topic and how it is created. The topic in question has 5 topic concepts. The topic concepts have different weights associated. For example, in this case topic concept “human” is assigned a weight of 0.7 instead of the maximum weight of 1.0, meaning that if encountered it is less relevant than other topic concepts. Each topic concept has at least one representative class from YAGO. For example, topic concept “body” has more YAGO entities associated, out of which `wordnet_torso_105549830` has weight 0.5, meaning is not as relevant to the topic concept as for example `wordnet_body_105216365` is.

As such, each topic has a number of weighted topic concepts, each topic concept being defined by a number of weighted YAGO entities. The weighing allows a fine-grained control over entity/concept influence. The weights associated are heuristically chosen, in increments of 0.1. It can be observed that topic concepts actually encode simple words/concepts. Thus, common concepts like “corruption”, “government”, “military”, “fruit”, “food”, etc will be shared among several topics.

The topic list initialization is among the most important aspects of the system, as it plays a central role in the system's performance. As can be seen, the topic concepts have been pre-programmed into the system manually.

This was done for a number of reasons, the primary one being that Word Sense Disambiguation is a yet unsolved problem and current systems do not perform at a sufficient performance level (as for example POS taggers that have 95-98% accuracy) to be included as trusted modules in an application (as discussed in the previous chapters). Fine-grained WSD performance is even worse than standard coarse-grained WSD, and we are working in a fine-grained environment. Because in this phase we chose the correct YAGO entities to represent the topic/topic concepts, meaning we performed the WSD manually, this will mostly alleviate the problem of WSD when analyzing the documents later.

Another reason is that the system is purpose-built for document classification. This means that in real world usage the number of categories (topics) is rather small and constant. We argue that given an initial effort to define the categories in an appropriate manner, then the system can be run as-is without any human intervention, except maybe adding another topic when necessary, a task that is done very fast.

The topic list was introduced programmatically in the system. However, the YAGO entities and the initial topic suggestions were done automatically. We wrote a small helper application that iterates over all topics, extracts the words, uses WordNet to suggest synonyms and YAGO to suggest named entities where necessary. Human intervention was required to add new topic concepts, delete or adjust weights of existing topic concepts, and to remove YAGO entities that are irrelevant for topic concepts. So, in some sense, the topic list was created semi-automatically. Ironically, for the 50 topics evaluated, the work needed by a human annotator was actually far less (a few hours) that the time needed to write the helper application.

VI.3.2. Processor Module

This initial module takes in a natural language text document and outputs a list of String Entities, each one having associated a set of probable classes (Canonic Entities) from the ontology.

From a functionality point of view, this module is similar to the preprocessing module of the GER system presented in the previous chapter. As such, using Stanford's CoreNLP[57], the text document is first split into sentences. Each sentence is further split into individual words (tokens). The tokens are then analyzed and their part of speech is determined, their form (singular or plural – if the word is in its plural form, it is transformed to its singular form) and whether the token is a named entity or a simple common word (using an English

[57] Stanford's CoreNLP package can be found at : http://nlp.stanford.edu/software/corenlp.shtml

dictionary to recognize common words). The named entities (which are first recognized by their 'NNP' part of speech tag[58]) are assigned a general category by the Stanford NER (part of Stanford's CoreNLP suite), such as Location, Organization, Person or Other. Punctuation tokens, numbers and dates are ignored.

Next, individual tokens (nouns only) are analyzed to see whether they form multi-word tokens. After this step, we will refer to the tokens as String Entities, as a String Entity may span multiple adjacent tokens. Then to each String Entity is associated a set of Canonic Entities from the ontology.

String Entity processing is done differently for named entities and common entities.

Named String Entities

Named entities are grouped together based on their category. If named entity tokens are adjacent and have the same category tag, they are joined into a single String Entity.

Example: For example, even if punctuation is omitted, the entities in the fragment "By/- visiting/- Sydney/L Ann/P Marie/P has/- …" will be correctly processed into two String Entities, even if all three named entities are together. "Sydney" will be the first extracted String Entity because it has tag L – Location while the adjacent named entities will be grouped into String Entity "Ann Marie" because they both are of type P – Person.

Next, for a named String Entity YAGO is queried using the `means` relation. This relation provides an entry in the ontology by having a list of strings that point to a Canonic Entity. These strings are not unique and a string can mean several Canonic Entities. Similarly a Canonic Entity can be represented by several different strings. So, for each named String Entity of the form "$word_1$ .. $word_n$" YAGO is queried as 'Select all Canonic Entities where relation is `means` and the string argument is "%$word_1$%$word_2$% .. %$word_n$%" ', where % means any character or string. A query for String Entity "Ann Marie" would look like "%Ann%Marie%" and would return Canonic Entities `Ann-Marie`, `Ann_Marie`, `Princess_Ann_Marie`, etc.

Furthermore, the category tag is considered. Using the method presented in the previous chapter, the type of any Canonic Entity can be determined. In summary, because we have only four large categories, out of which one is Other, we mark three Canonic Entities in the ontology, one for Person, one for Location and one for Organization. As any extracted non-WordNet *CE* has a `type` relation that links to a WordNet *CE*, following a few links up in the hypernym hierarchy will find one of the three *CE*s marking the type. If none of these is found, then the initial *CE* is of type Other. So, after the query for the String Entity is completed and all matching *CE*s are obtained, each *CE* is assigned a type out of the possible four. If the assigned type is not of the type of the String Entity itself, the *CE* is

[58] These are Penn Treebank style POS tags. http://www.comp.leeds.ac.uk/ccalas/tagsets/upenn.html

discarded. This ensures that if for example String Entity “Sydney” is determined as a Person, then all Canonic Entities that can represent Sydney and are locations or organizations are discarded. This processing usually halves the number of possible Canonic Entities associated to any named String Entity.

Common String Entities

The processing for common entities is a bit more computationally and I/O intensive. To determine if we deal with multi-word common String Entities we search for nouns separated by maximum of two non-noun, non-verb tokens. We obtain patterns like (noun$_1$ noun$_2$), (noun$_1$ word noun$_2$) or (noun$_1$ word word noun$_2$). The linking words cannot be punctuation marks, numbers, any other nouns or verbs or else the obtained pattern is discarded. Furthermore, the two nouns must be in the same noun phrase (determined by the syntactic tree of the sentence).

Next, the two nouns are searched for in the WordNet section of the ontology (in YAGO WordNet is represented as the hypernym hierarchy to which all other entities must link to by at least one `type` relation). All common Canonic Entities that contain both nouns are kept for further analysis. These initial *CE*s are iterated over. For each *CE* the linking words between the nouns have to match the linking words in the String Entity. For some *CE*s there might be extra words before the first noun or after the second. These words must match external words of the String Entity. If there are no remaining *CE*s then the String Entity is not multi-word, and the initial noun is kept as the only token. Class assignation for single word String Entities is a bit different that for multi-word, and will be described after an example of multi-word common String Entities.

Example: A sentence extracted from a news article: “Failure to take into account some of the effects predicted by the second law of thermodynamics has led to the failure of the initial prototype.”. Nouns are extracted sequentially. When reaching “law” we analyze nouns at a distance of maximum two, finding “thermodynamics”. A pattern is obtained (law of thermodynamics). The pattern is valid, as both nouns are in the same noun phrase (NP):

```
(NP
   (NP (DT the) (JJ second) (NN law))
   (PP (IN of)
       (NP (NNS thermodynamics))
   )
)
```

YAGO is queried in the form of “%law%thermodynamics%” and we obtain several Canonic Entities, like `wordnet_law_of_thermodynamics`, `wordnet_second_law_of_thermodynamics`, `wordnet_third_law_of_thermodynamics`. The first *CE* is kept, as the linking word “of” appears both in the *CE* and the *SE*. The second *CE* is also kept because the first word “second” (here not a noun), even though before the initial noun “law”, appears both in the

CE and the text before the *SE*, matching exactly. The third *CE* is discarded because the word "third" does not appear in the words in the text before the *SE*. As there is at least one valid *CE*, the String Entity will now encompass both nouns, and it will be formed by two nouns linked by a preposition "law of thermodynamics".

If a multi-word String Entity is found, then the valid *CE* list is kept as probable Canonic Entities that each could represent the *SE*. However, because of the way YAGO is structured, (common words are linked in a tree structure – the WordNet hypernym tree), we keep not only the initial list of *CE*s, but also each of these *CE*s' direct hypernyms.

Example: For the remaining *CE*s: `wordnet_law_of_thermodynamics`, `wordnet_second_law_of_thermodynamics` we determine their hypernyms. For `wordnet_law_of_thermodynamics` the direct hypernym is `wordnet_law` (a certain `wordnet_law` from the seven possible *CE*s `wordnet_law` (seven senses of the word law, each having the same word/name but different ids to differentiate them – not printed here because the actual ids are irrelevant for this example), which we keep). For `wordnet_second_law_of_thermodynamics` the direct hypernym in this case is actually `wordnet_law_of_thermodynamics`, which is already added.

This heuristic ensures a larger coverage for the purposes of this system, even if for this sentence we have actually found the actual, most specific Canonic Entity.

For single-word String Entities the treatment is a bit different. Here, we keep any Canonic Entity that contains the word itself.

Example: For String Entity "engine", both classes `wordnet_automobile_engine` and `wordnet_engine` are accepted, even though `wordnet_automobile_engine` is more specific than simply `wordnet_engine`.

After this step, all String Entities, both named and common, multi-word or single-word will have associated a set of possible Canonic Entities that could represent them.

As String Entities are identified and processed in sequence, a master frequency array list is kept, recording the frequency of identified String Entities.

One assumption that is being made in this module is that if a String Entity is identified and counted, then if another identical word-by-word String Entity is found, it is automatically considered "processed" and the frequency of the first String Entity is increased by one. The assumption is that a String Entity will always refer to the same thing in the current analyzed document.

This output in the form of a list of String Entities with their frequency and associated Canonic Entities is further fed to the Analysis Module.

VI.3.3. Analysis Module

This second module takes as input the list of String Entities with their associated classes and as output it assigns a score to every topic concept (but does not score the topics holding the topic concepts – a task for the Evaluator module).

First, the ontology graph on which to calculate the scores is created. Every Canonic Entity associated to every String Entity contributes to this ontology. Given we use YAGO as the knowledge source, the graph we are creating is actually a fragment of YAGO itself. Starting from every Canonic Entity a depth-first search is performed in YAGO, and all encountered entities and relations up to a depth of 2 are added to the new ontology graph (if they are allowed to be added as we allow only certain relevant relation types).

At this point, the topic list contains topics that have a score of zero, and all of the topic concepts themselves have a score of zero. While the score of a topic (the final score) is determined in the following module, the scores of the topic concepts (on which the document-topic assignation will be made) are obtained in this module.

The score assignation for topic concepts is done in the following manner: from every Canonic Entity assigned to every String Entity a breadth-first search is performed on the created ontology. If during the search, the visited entity is actually an entity in a topic concept, then the topic concept's score is increased accordingly. The formula by which the value of this increase is calculated is:

$$fullScore = \frac{log^2(1 + freq_{SE})}{2^{d_{ce}} * freq_{SE}} \tag{41}$$

where d_{ce} is the distance in the search from the originating Canonic Entity, and $freq_{SE}$ is the frequency of the String Entity to which the originating Canonic Entity belongs. This is a distance based score. The score is multiplied by the logarithm of the frequency of the String Entity processed to dampen the influence of the same String Entity repeated several times, while the division by the frequency of the *SE* directly is to add to the full score only the increment of the logarithm of the frequency corresponding to this particular instance of *SE*. If a *SE* is encountered 10 times (for example), then each time it is analyzed it will add the tenth part of its total value of $\log^2(1+10)$ to a certain topic. The need to add several times a small increment instead of adding the entire value a single time and then ignoring duplicate *SE*s will be explained later on.

The breadth-first search is limited at a depth of three. Paths longer than three have an almost zero information value and are ignored.

Two aspects need mentioning so far. The first aspect is to show how the problem of Word Sense Disambiguation is handled, or better said, partially avoided by making some compromises, and second, the influence of context in determining the scores (or so far, the lack of influence).

To exemplify, let's consider we have the common String Entity "artillery", for which we have a number of possible Canonic Entities:

```
String Entity "artillery"
Associated Canonic Entities:
     Canonic Entity #1: wordnet_artillery_102746365
     Canonic Entity #2: wordnet_artillery_108389297
     Canonic Entity #3: wordnet_artillery_plant_112395289
     Canonic Entity #4: wordnet_artillery_shell_102746595
     Canonic Entity #5: wordnet_artillery_fire_100994449
```

As can be observed, for String Entity "artillery" YAGO knows two different artillery entities (#1 & #2, having the same name but different trailing IDs meaning different senses of the same word), along with three other possible representative entities. So, this far, the system does not know whether the word "artillery" means either a plant, a shell, artillery fire, or which of the two senses of artillery is the correct one (if any). Because of the fact that the topic list was created manually, and each topic has associated topic concepts that have "disambiguated" entities (meaning the topic concepts are described by the correct YAGO entities), starting a breadth-first search from each of the five Canonic Entities will yield the following:

```
Canonic Entity: wordnet_artillery_102746365
      No matches;
Canonic Entity: wordnet_artillery_108389297
      Match 0.1725 e: wordnet_army_108191230 for Topic Concept "military"
         For Topic "47 Russian Intervention in Chechnya"
         For Topic "48 Peace-Keeping Forces in Bosnia"
         For Topic "66 Russian Withdrawal from Latvia"
Canonic Entity: wordnet_artillery_plant_112395289
      No matches;
Canonic Entity: wordnet_artillery_shell_102746595
      No matches;
Canonic Entity: wordnet_artillery_fire_100994449
      No matches;
```

The BF search from `wordnet_artillery_102746365` is performed up to the maximum distance of three, but no entity in any topic concept of any topic is found. This yields the "No matches" message. Only the second sense of the word "artillery" finds at a depth of 2 the topic concept "military" that has been defined for more than one topic. This example shows both aspects.

The first sense of "artillery" `wordnet_artillery_102746365` does not match any topic concept, meaning that sense is never used, as opposed to `wordnet_artillery_108389297` who actually contributes to three topics. This shows that because the topic concepts contain the correct, 'disambiguated' entities (and only those entities), then even if in this document processing phase we do not perform any disambiguation and keep all possible senses, only the correct sense will contribute to a topic. However, this does not remove the possibility that the word was used in the first sense of the word and not in the second, so it does introduce some false-positive results, but we estimate a much lower number if we just used direct word matching without regard to senses.

On an implementation note, the fact that for example the first sense does not reach any topic concept entity while the second sense does, is because that the WordNet hypernym tree is actually a tree with clearly separated senses, so for a BF search started from the first sense to arrive at the `wordnet_army_108191230` entity belonging to topic concept "military" it would have to climb almost to the top of the tree and then back down, a path of very long length. To a lesser degree this happens to named Canonic Entities, as different *CE*s describing a *SE* will usually be linked by different entities in different parts of the ontology graph (below the WordNet tree stands the majority of the ontology in the form of a graph).

The second aspect needing discussion is the fact that, as seen in the example, entity `wordnet_artillery_108389297` contributes to three topics. While this is normal, as topic concept "military" is common to more than one topic, to attribute the same score to topic concepts belonging to different topics might not prove to give correct results. The following example will focus on this specific aspect:

Consider the sentence: "There are suspicions that apples treated with … might lead to an increased risk of developing a condition similar to …". Among the String Entities identified there is *SE* "apples" and *SE* "condition". Consider that during our processing of the sentence we have reached *SE* "condition" and, following the algorithm presented above, we start the BF search in the ontology from the its possible Canonic Entities. The BF search from a *CE* of *SE* "condition" has encountered the *CE* `wordnet_illness`. In the WordNet hypernym tree `wordnet_condition` is a direct hyponym of `wordnet_illness`. To make the example easier to read and to keep track of, we drop the ids following each entity as we consider that each entity is the correct one (for example, out of the possible 8 senses / 8 *CE*s of *SE* "condition" we chose the correct one, meaning, as reported by WordNet: (n) condition (an illness, disease, or other medical problem) "a heart condition"; "a skin condition").

At this point the BF search was completed, and the `wordnet_illness` entity belonging to two different topics was reached: topic #71 "Vegetables, fruit and cancer" and topic #53 "Genes and diseases". As per the algorithm described above, both topics having the topic concept "disease" containing, among others, entity `wordnet_illness`, should have a similar score increase.

However, for humans the context allows to discriminate to which of the topics the word "condition" should contribute more. It is relatively easy, even from the fragmented sentence, to determine that "condition" is probably more relevant to topic #71 than #53. A simple clue might be that "apples" are fruits (topic #71), and the rest of the words do not imply anything related to genes (for topic #53).

We aim to follow this simple logic to discriminate between the different topics when considering a topic concept belonging to all the topics involved.

The algorithm is the following: for a String Entity that is currently under analysis, we define a window of *q* *SE*s to analyze, before and after the current *SE*. We heuristically define $q = 5$, as we observed good results given our test documents, but it can take any other value. However, *SE*s in this window must be in the same sentence as the current *SE* or at most in the previous or following sentence.

In our example, for simplicity, consider that in the window of *q* *SE*s we have found only *SE* "apples". At this point we ask if maybe "apples" is a clue that might differentiate between topics for the currently analyzed *SE* "condition". So, we perform a BF search to see if "apples" might contribute either to topic #71 and/or #53. From an implementation point of view, because *SE* "apples" was found before *SE* "condition", the BF was already performed and the results cached, so performance-wise the BF is not repeated, its cached results are directly used instead. Actually, in the system's implementation a BF is performed for every *CE* of every *SE* keeping the results into memory, then taking each SE in sequence to analyze its impact.

We determine that *SE* "apples" has a link to topic concept "fruit" of topic #71 "Vegetables, fruit and cancer" by the conceptual link of depth 2 in the ontology `wordnet_apple` *`subClassOf`* (→) `wordnet_pome` *`subClassOf`* (→) `wordnet_fruit`, where `wordnet_fruit` is an entity in topic concept "fruit" with weight 1.0. However "apples" does not have any link to topic #53.

So, at this point, we know that *SE* "condition" should contribute to both topics #71 and #53 (reaching topic concept "disease", present in both topics), but knowing that *SE* "apples" has a link to topic #71 allows us to make the assumption that the scores assigned should not be equal, but that topic #71's topic concept "disease" should receive a higher score than topic #53's topic concept "disease". As such, topic #71 will receive the full score defined earlier, while #53 will receive a smaller score. It should be noted that when saying topic score, for this module we do not assign scores to topics but to the topic concepts of each topic. The topic scores are calculated in the final module based on the scores of their topic concepts.

In a more complex scenario, there might be cases where a *SE* can contribute to several topics (reaching common topic concepts like "death", "money", "government", "disease", etc), with some of the neighboring *SE*s analyzed in the *q* window supporting some of these topics, and some other *SE*s pointing to other of the identified topics. To solve this issue we simply count the number of "supporters". So, given a number of *n* topics that the currently analyzed *SE* contributes to, for each of these topics we assign a temporary supporter count variable. For every Canonic Entity of every String Entity in the *q* windows, if during the BF for that *CE* an entity is found in one of the topic concepts belonging to one of the *n* topics, we increase that topic's supporter count by one.

With this strategy we can count how many supporters each topic has, and we can actually sort the topics descending according to these values. On the now sorted *n* topics a score assigning strategy can be applied, so that similar topic concepts belonging to different topics can take different scores.

Given a number of *n* topics, each having a certain supporter count, we assign scores as follows: for the top scoring topics (because there might be topics with equal supporter count) we assign the full score as defined previously; for the second set of highest scoring topics we assign only half of the full score; for the third set of highest scoring topics half of the half of the full score, and so on.

$$topicScore = \frac{fullScore}{2^{topicPosition-1}} \tag{42}$$

where *topicScore* is the score assigned to a certain topic (out of the *n* possible topics), *fullScore* is the score calculated by the formula defined previously, and *topicPosition* is the position of the topic in the sorted topic list. More topics can share the same position if they have the same number of supporters.

Example: if we consider that a String Entity has found 5 topics to which it should contribute (reaching a common topic concept present in all 5 topics, a likely scenario as some general topic concepts are often shared between topics), and after analysis of the window of *q SE*s surrounding the current *SE*, it has found a variable number of topic supporters for each topic. The scores will be assigned as follows:

Table 6. Example table showing the score percentage assigned to each topic based on its supporter count

Topic #	Supporter count	Score assigned (% of *fullScore*)
#1	0	25%
#2	3	100%
#3	1	50%
#4	1	50%
#5	3	100%

Using this strategy, the topic concepts of topics #2 and #5 will receive full score, topic concepts of topics #3 and #4 will receive half while the topic concept belonging to topic #1 will receive a quarter of the full score, even if we are talking about the same topic concept for each of the five topics.

A short review of this module: First, the ontology on which the analysis will be performed is created by applying a limited depth Breadth First search on the YAGO ontology starting from every Canonic Entity belonging to every String Entity. After the ontology is created, each String Entity is analyzed in order of appearance. The topic concepts of individual topics it contributes to are determined based on a BF search on the created ontology. If during the search an entity is encountered that belongs to a topic concept of a topic, that topic concept is added to the list of topic concepts that should have their score increased. A list of *n* topics is thus created. To discriminate between the same topic concept belonging to different topics, a strategy is employed: first, a window of *q* String Entities in the text document that appear before and after the current *SE* are analyzed. In the same fashion, a BF search starting from each of their individual Canonic Entities is performed to see to which (if any) of the *n* topics it can reach. If a topic is reached (meaning one of its topic concepts), then the topic has its "supporter count" variable increased. After all the BF searches are performed, the topics are sorted descending by the supporter count variable. The topics that have the highest single value of supporters award their topic concept the full score. Then, topics with the next highest supporter count award half the score, and so on, halving the score on each lower value of supporter count. Using this method, each String Entity will be analyzed sequentially and it will contribute (if possible) to one or more of topics by increasing the score of their topic concepts.

After all the *SE*s are analyzed, the topic list will contain topics that have non-zero scores to some of their topic concepts.

Example: In the example below, a document was analyzed and the score composition of topic concept "food" from topic #41 "Pesticides in Baby Food" is shown, as each matching String Entity adds a small increase to the final score of 8.5574:

```
concept [food/1.0] 8.5574:  wordnet_food_107555863/1.0:8.5574
 >> Add 1.0*3.76=3.76 to 3.76 from wordnet_food_107555863/42
 >> Add 0.25*0.69=0.1725 to 3.9324 from wordnet_game_107650449/1
 >> Add 0.5*2.89=1.445 to 5.3774 from wordnet_meat_107649854/17
 >> Add 0.5*1.1=0.55 to 5.9274 from wordnet_fish_107775375/2
 >> Add 0.5*0.69=0.345 to 6.2724 from wordnet_cheese_107850329/1
 >> Add 0.5*1.95=0.975 to 7.2474 from wordnet_seafood_107776866/6
 >> Add 0.25*2.48=0.62 to 7.8674 from wordnet_shellfish_107783210/11
 >> Add 0.25*0.69=0.1725 to 8.04 from wordnet_beef_107663592/1
 >> Add 0.25*0.69=0.1725 to 8.2124 from wordnet_delicatessen_107594406/1
 >> Add 0.25*0.69=0.1725 to 8.384 from wordnet_vegetable_107707451/1
 >> Add 0.25*0.69=0.1725 to 8.5574 from wordnet_pork_107668702/1
```

It should be noted that in the examples above we give only WordNet classes (e.g. class `wordnet_beef_107663592`) from YAGO because the relations between them are easier to understand. However, YAGO's imported WordNet hierarchy contains only around 65.000 classes from the more than 2 million entities known. Named entities contribute just as much as (and in some cases even more than) the common entities that our system uses.

VI.3.4. Evaluator Module

The evaluator module takes as input the topic list with their scores, and evaluates them. The output is a sorted list of probable topics for the currently analyzed document.

Given we already have for each topic the scores of its topic concepts, one method is to simply add the scores of the topic concepts and call this sum as the final score of the topic, then just sort the topics using this value. However, because of the way the system works, there are documents that have very many common words that contribute more to other topics and not the correct one. This is partly because of the structure of YAGO and WordNet, partly because of the problem of word sense disambiguation that is rather slightly circumvented and not solved, partly because of the way the topics were defined.

To allow a degree of variation to an otherwise strict method of scoring, we assume the following strategy:

1. We calculate the general score for each topic by adding the scores of its topic concepts.
2. We evaluate the first 4 highest scoring topics, and we calculate the average of the score differences between each topic, which will call the *error margin*.
3. If the score of the best topic is greater that the score of the second topic plus the error margin, we assume that the first topic is the correct topic. If the second topic is within the error margin of the first, we count for each topic the number of topic concepts that have a score greater than 0. The topic we believe is correct is the topic that has the best *coverage*. The coverage of a topic is the percent of non-zero topic concepts.

This heuristic was introduced because sometimes the correct topic is the second or third best scoring, with a score almost equal to the top scoring topic. We allow for the second scoring topic to precede the first if a larger percent of the topic's topic concepts are reached (non-zero) based on the assumption that a topic that characterizes a document should have most if not all of the words in the topic at least one time in the document.

Example: Given two topics that score almost equal, with topic #2 scoring slightly lower but being the correct choice, if topic #1 has 4 out of 5 concepts greater than zero, and topic #2 has 4 out of 4 topic concepts greater than zero, then we choose topic #2, because topic #1 has only 4/5 = 80% coverage while topic #2 has 4/4 = 100% topic concept coverage).

VI.4. Evaluation

Before evaluating the results, a quick description of the data collection on which we tested is needed. We used the LA94 TREC Information-Retrieval Text Research Collection[59], representing a sampling of news articles published by the Los Angeles Times in 1994. The collection includes 828 such articles, which are classified over 50 topics. The articles are small to medium-sized news (200 to 1500 words) on different topics such as entertainment, movies, television, music, politics, business, health, technology, etc.

In order to compare our results against a standard method of text classification used today, we have implemented a SVM based system for text classification. The system is built in Java and uses core functionality from WEKA [9]. Each document is parsed, and a feature vector is extracted. The vector is further elaborated upon, eliminating stop-words, using lowercase tokens, setting a minimum term frequency for allowed terms, pruning periodically, using a stemmer, and finally applying a TF*IDF transform. The SVM is then trained on the document collection, and evaluated using a random-seed, 10-fold cross validation. We have tried to build the evaluator system as best as possible using the latest feature vector techniques and the best classifier for this job, the SVM.

Table 7. Comparison between the proposed system and a standard SVM state-of-the-art method

System	**Performance (correctly classified documents)**
Proposed KB-approach ontology-centric system	570 / 828 (68.84%)
SVM comparison system	661 / 828 (79.83%)

The SVM comparison system at this moment performs better, by a margin of almost 11%. However, our proposed system achieves a respectable performance of 68.84% using only the ontology as a source of information. We designed this system as proof-of-concept, to test the possibility of using ontologies as the core of a text classification system, and to see the performance degree of such an approach.

While the system proves effective even at this stage, we believe that its performance can be greatly improved. During the implementation, we have noted a series of improvements that should significantly boost performance:

The first and most important issue affecting performance is the topic list creation. Depending on the description of each topic (meaning the topic concepts of each topic), performance is greatly affected.

[59] LA94 news articles collection, http://trec.nist.gov/data/docs_eng.html

Table 8. A short comparison between overall system performance grouped by topic, before and after topic tweaking, for 5 out of the 50 total topics

Topic #	Topic	# of docs	Performance bef. tweaking	Performance after tweaking	Difference
#50	Revolt_in_Chiapas	105	98 / 105 (93.3%)	**99** / 105 (**94.28**%)	+ 0.98%
#43	El_Nino_and_the_Weather	11	4 / 11 (36.36%)	**6** / 11 (**54.54**%)	+ 18.18%
#80	Hunger_Strikes	56	9 / 56 (16.07%)	**18** / 32 (**34.61**%)	+ 18.54%
#70	Death_of_Kim_Il_Sung	33	**28** / 33 (**84.84**%)	22 / 33(66.66%)	- 18.18%
#58	Euthanasia	49	14 / 49 (28.57%)	**31** / 49 (**63.26**%)	+ 34.69%

This table shows some of the performance gains after manually tweaking some of the topic's concepts. For example, while adding context to topic #50, performance is very slightly improved by almost 1%, while for topic #80 the correct topic classification rate is doubled to 34%. By concept tweaking we mean editing individual topic concepts. For example, for topic #80 we had the initial topic concepts of 'hunger' and 'strike'. After adding context, meaning topic concepts "government", "demonstration" and "cause" (each with a slightly lower weight than the initial two topic concepts), the detection rate greatly increased.

However, after also tweaking topic #58, the performance negatively affected topic #70's recognition rate. This is due to the adding to topic #58 (and others) of the concept 'kill' which was already present in more topics, including topic #70. This means that the word "kill" will now score for topic #58 also. The multiplicity of the same topic concept in many topics, while unavoidable, does negatively impact performance. It should also be noted after tweaking, each topic has grown from 2-3 topic concepts to an average of 4 topic concepts, few topics having more than 6 topic concepts.

Another valuable insight from this before/after comparison is related to the ontology information content. We have found out that there are sometimes lacks in information in the ontology, while in other places there is an abundance of it. For example, we had trouble finding YAGO classes to describe the concepts for topic #76 "Solar Energy": while we have `wordnet_energy_111452218` for the "energy" concept, for the "solar" concept there is no simple, general `wordnet_solar_#` class, just classes like `wordnet_solar_cell_104257986`, `wordnet_solar_dish_104258138` or `wordnet_solar_house_104258438`. While YAGO (and WordNet) contains `wordnet_solar_energy_111509697` (which we have also used to describe the topic), because it is multi-word, for it to positively match we need to have the entire "solar energy" expression in the text. This means that in the documents where the word "solar" is found, if it is not followed by "cell", "dish", "house" or "energy", it will not be counted. This issue accounts for the proposed system's <15% detection rate for this particular topic.

Another topic list related point is that the list needs to be created partially by hand. While usually this is not desired due to the required human intervention, we argue that the number of possible topics for any classification is manageable, ranging from a few tens to usually no more than a few hundreds, a relatively simple task for even one person. For the 50 topics we had, it took no more than a few hours to initially create the list (while assisted by the computer, using only topic concepts found in the topic name), and a bit longer for the topic tweaking (using topic concepts extracted from each topic's description which is a few sentence-long summary, also available to us in the LA 94 collection, but not used directly in the system) which increased overall system performance from an initial 55.79% (462 correctly identified documents out of the total 828) to the current 570 / 828 (68.84%).

Another idea to be implemented in a future revision, is that we could use the ontology as not only a semantic similarity map, but also use the relations themselves as useful information. That would mean identifying subject – object entities and then match the verb that links them to a specific relation in the ontology. This would provide a stronger link between concepts, and an algorithm could judge whether to take into account certain entities or not based on the relations between them. However, at present, the task of relation extraction is an even more difficult problem that text classification. Relation extraction systems do exist, but are difficult to implement and use, and they require very particular conditions to run under – thus currently impractical to use.

VI.5. Conclusion

We believe that ontologies, especially general ontologies represent a powerful yet somewhat underused tool for the text classification problem. The structure of the ontology itself contains information that can be used in the form of concept closeness, synonymy, hypernymy, relation types, etc. As time passes, it is inevitable that general ontologies will become larger and larger, thus providing better results even using the same algorithms.

However, the use of ontologies does impose some limitations and problems. For example, information density in an ontology varies greatly, meaning some concepts will be defined in more detail than others, that in turn leading to uneven topic recognition accuracy. This problem is usually addressed by using domain ontologies. However, for example, for news articles a domain ontology is mostly useless considering the method we have applied in this article, where we do not use the ontology as a simple hierarchical taxonomy, but as a concept semantic similarity map.

We propose a text-classification approach that achieves a good performance rating using only an ontology as its information source, and graph algorithms with a custom scoring method. The system uses the links available in the ontology to assign a score to the semantic similarity between concepts. Future work on the subject will include implementing some of the suggestions in the previous chapter, as well as an attempt to use a supervised ML algorithm to self-create the topic list's concepts instead of manually tweaking them, and evaluate performance between this system's versions.

VII. Conclusions

The field of Information Extraction (IE) is a relatively young area of research that holds many possible rewards. Information Extraction means extracting structured information from unstructured and semi-structured sources by a computer. Making the computer 'understand' the data it is processing will yield improvements in many areas, like better Internet search engines that identify words' meanings, automatic multimedia annotation that leads to more accurate information delivery, knowledge discovery from existing knowledge sources (like predicting events based on entity identification and the heterogeneous links between them), up to the field of Artificial Intelligence where a computer that would try to pass the Turing test would first need to understand the question it is being asked and then to reason a response adapted to the meaning of the question.

One of the main research problems of IE is entity identification and classification, an essential step in any IE system. This research problem is actually split in two distinct tasks: Word Sense Disambiguation and Named Entity Recognition.

Word Sense Disambiguation is the task of identifying the senses of words in context. It usually deals with common nouns (but can target also verbs, adverbs, adjectives, etc). For example, in the sentence "John is the engine that keeps our business going", 'engine' is not a mechanical engine or a synonym for locomotive, but refers to something used to achieve a purpose. Depending on the number of senses considered, WSD can be a coarse-grained (and easier) task having only a few possible senses per word, or a fine-grained (and thus more difficult) task having several senses per word, encoding subtler distinctions. The senses are references in a sense repository, usually a dictionary or a taxonomy (like WordNet).

Named Entity Recognition is the task of identifying and classifying interesting entities in context. It usually deals with proper nouns, meaning names of persons, locations, organizations, etc, but can also target other entities such as dates, numbers and so on. As with the WSD task, NER can be a coarse-grained task where only a few basic types of entities are recognized (ex: the standard major three categories: persons, organizations and locations) or fine-grained (having more categories, for example instead of location, having city, state and country categories). For example, in the sentence "I drove the new Santa Fe through Santa Fe" a good NER system might recognize the first "Santa Fe" as an car (in case of a coarse-grained system, recognize it as a named entity labeled "Other") and the second "Santa Fe" as a city (in case of a coarse-grained system, recognize it as a location). A NER system would use either a flat-list or a taxonomy to store possible entity categories.

Because there are obvious differences between NER and WSD the tasks remain strongly separated, with only little research in systems having a unitary view over both tasks. A General Entity Recognition system would try to tag both common and proper nouns with appropriate labels. Here, the labels would also come from a taxonomy encoding a hierarchy of classes. For example, for the sentence “Hyundai Accent has a 1.6 liter engine delivering 110 hp.” such a system would tag “Hyundai Accent” as a car, “1.6” as a quantity, “liter” as a unit of measure (liter), “engine” as an engine (having the sense of mechanical engine), “110” as a quantity and “hp” as a unit of measure (horse-power). This requires techniques from both WSD and NER. For example, the identification of entity boundaries is a NER-specific task (“Hyundai Accent” forms a single entity), while the correct identification of the fact that “engine” is used with the sense of mechanical engine and not a locomotive or other is a WSD-specific task.

The thesis presents a system that extends the task of General Named Entity Recognition (as defined in [104] : to tag every interesting entity – both named or common noun – with a WordNet sense) to identifying interesting entities and matching them to the most likely canonic classes in a large, general ontology. For the purposes of the thesis and the system we have used the YAGO ontology [39], holding among its 2+ million entities and 20+ million links between them all WordNet senses in the form of a hypernym tree. Therefore, the proposed system first identifies interesting words (defined as String Entities) and then attempts to assign to each one a class (defined as Canonic Entity) from the YAGO ontology.

We define the notions of String Entity and Canonic Entity as follows: a String Entity is a bounded sequence of characters, a single or multi-word token (ex: “chair”, “USA”, “relativity theory” or “Charles Darwin”), while a Canonic Entity is a class (entity/individual) from an ontology (ex: `wordnet_chair`, `United_States_of _America`, `wordnet_relativity_theory` or `Charles_Darwin`).

For example, for the sentence “The new Hyundai Accent has a 1.6 liter engine that delivers 110 hp” the system identifies “Hyundai Accent” as a multi-word String Entity and assigns it the YAGO Canonic Entity `Hyundai_Accent`, “liter” as the WordNet Canonic Entity (integrated in YAGO) `wordnet_liter`, “engine” as `wordnet_automobile_engine` (and not just the more generic `engine` Canonic Entity for example) and “hp” as `wordnet_horsepower`.

The approach of the system is to try to find for each String Entity the best matching Canonic Entity (or tagging the String Entity as `unknown`), taking into consideration the context of each entity. The approach taken first analyzes each sentence from a NLP point of view, performing token splitting, Part-of-Speech-Tagging, applying a Named Entity Recognizer of proper nouns (for indication of the general class of that noun : location / organization / person / other), obtaining the syntactic and the dependency tree for the sentence itself. From the dependency tree an influence matrix is created where the values represent the connection strength between any two String Entities. Next, for each String Entity the most probable Canonic Entities are found in the ontology (a String Entity can have anywhere from a few Canonic Entities to more than 1000). Then, a sub-graph is created starting from every Canonic Entity of every String Entity by exploring the YAGO ontology and adding relevant neighbors. Based on this directed unweighted sub-graph, every set of related String Entities (meaning they have a non-zero value in the influence matrix) is analyzed, creating a smaller directed and weighted graph. This latter graph is formed by finding limited-distance paths in the initial sub-graph to other Canonic Entities, weighing the connection strength between them using a distance-based metric and adjusting the score using values from the influence matrix. Thus, for each set of related String Entities a k-partite graph is formed. Such a graph has the property that it is divided into k-partitions in which there are no edges between the vertices belonging to a partition. For the purposes of the system, a partition represents a String Entity, and the vertices in the partition are the Canonic Entities associated to that String Entity. On this special graph type we propose a custom algorithm that finds the best combination of Canonic Entities respecting the k-partite property – picking at most one Canonic Entity for every String Entity (partition) in the graph, based on edge scores. The last step of the algorithm is to merge non-overlapping solutions (several non-overlapping connected components in the k-partite graph) to provide the highest scoring solution possible.

The approach taken here is based on graph algorithms and ontologies. This unsupervised, knowledge-rich approach yields interesting results. While the performance figures are not themselves very high, the problem undertaken is very difficult. Though not comparable, a task that resembles the General Entity Recognition approach is the Word Sense Disambiguation - English nouns fine-grained disambiguation task. Here, senses (usually WordNet senses) are associated to nouns. However, wherein the coarse-grained task there are no more than 2-3 senses per word, the fine-grained task has no limit, having sometimes more than 5-8 senses per word. This apparently slight increase in possible senses for a word has a major impact on disambiguation performance: for coarse grained tasks (few senses per word) the percent is rather high, reaching 90% [91]. For finer grained tasks (many senses per word, such as the senses in dictionaries or WordNet for example) the percent drops in the 60-80% range [94], not to mention that the Inter Annotator Agreement Rate was under the same circumstances at most 85% showing that even humans have a difficult time agreeing on word senses. The system we propose has to identify the correct tag (or corresponding Canonic Entity in the ontology) from not just a few but sometimes hundreds of possible choices, meaning the search space is much larger, as well as having more entities to deal with – both named and common.

Also, the thesis presents another system which is based on largely the same tools and techniques (ontologies and graph algorithms) but differently applied to the problem of text classification into predefined topics. Currently, this problem is usually solved by machine learning algorithms like the well-performing Support Vector Machine. While such supervised algorithms are sound, the problem of input data fed to them is not yet solved.

Depending on the feature vectors created (or other internal features like type of kernels used), machine learning algorithms provide better or worse results. The problem of supervised text classification has been studied in depth and while good results have been achieved, there seems to be a limit on the performance of such machines. New approaches should be developed and used either by themselves or with current state-of-the-art approaches in order to improve classification performance. With the system presented in this work we investigate an alternative approach that tries to leverage the information contained in large scale, general ontologies and apply it to the problem of text classification with encouraging results.

In the following paragraphs a short summary of the workings of this knowledge-rich system is presented:

Before the system is used, there is a phase of semi-manual topic crafting. This task is performed only once to define the topics in a way the system can understand. Here we introduce a few notions: as a topic can be defined by a word (ex: "Science" – general topic) up to several words / a sentence (ex: "Pesticides in baby food" – more specific topic), we define a *topic* as being composed of *topic concepts*. A topic concept is the encoding of a word/sense from that particular topic. For example in the example "Pesticides in baby food" we have 3 explicit topic concepts: "pesticides", "baby" and "food". Now, because we are working with an ontology, the topic concepts have to be 'translated' to that knowledge source. As such, each topic concept is itself composed of several weighted ontological classes. For example, topic concept "baby" can be expressed as the ontology class `baby` indicating a human infant (and not another sense of the word) as well as the ontological class `child` (in the same sense – a very young human as defined in the ontology). Because we are working with topic concepts expressing ideas, a certain generality must be maintained, in the form of synonymy. The classes are weighted to indicate that, for example, class `baby` is better suited to topic concept "baby" than class `child`, but `child` should also be allowed in our scenario as an indication of topic concept "baby".

This topic crafting is performed semi-automatically by one or more persons that add or remove ontological classes to the topic concepts that form topics. Using this approach we partially avoid the problem of Word Sense Disambiguation that appears in any NLP system, as will be explained shortly.

As we designed this system for text topic classification we assume that the topic number is relatively low (less than 100 topics for example). Even though the semi-manual topic creation would seem an undesired feature of the system, for our test collection of 50 topics of 848 Los Angeles Times (from year 1994) news articles, the topic creation phase took a short time to complete, and the benefits of this small initial topic 'disambiguation' would increase the performance of the system, avoiding many false-positives due to sense mismatch.

The first working step of the system is text pre-processing, where the standard NLP treatment is applied. Sentences are split into POS tagged tokens, multi word named entities and common entities are grouped together into larger String Entities (similar to the previous GER system proposed in this thesis).

After this step, every String Entity is assigned a number of Canonic Entities (classes from the ontology that could represent the String Entity). For example, for the String Entity "baby" extracted from a natural language text, the ontology reports 6 Canonic Entities (ontological classes) as `baby_#` (where the number following the word is an identifier to differentiate between the senses of baby). Because we perform no WSD, each and every Canonic Entity is kept as a possible representative of that String Entity. This means for our example that for the String Entity "baby" we keep all 6 senses of baby, including unlikely senses like "S: (n) baby (a project of personal concern to someone) *"this project is his baby"*".

After assignation of Canonic Entities to String Entities is completed, for each String Entity we start to sequentially look for classes from any of our topic concepts in the vicinity of every Canonic Entity for our current String Entity. This means that we perform a custom depth-first search starting from every Canonic Entity assigned to every String Entity in the ontology graph. If during the search we encounter a class that has been marked as belonging to a topic concept, we increase the score of that topic concept using a custom distance-based function (implementing some of the ideas like direction change penalization in the WordNet semantic similarity metric of Hirst and St-Onge (1998)).

At this point we are faced with two aspects: first, we keep every sense for every common String Entity and every individual for every named String Entity. This means that interpretation and sense errors should be overwhelming. However, due to the ontology's structure, it is not the case. Because we manually crafted the topic concepts before running the system, the limited depth first search initiated from every Canonic Entity from a String Entity will hit only the correct class of a topic concept. Returning to the "baby" example, if we initiate a search from each of the 6 senses for baby, only the correct sense of human infant would reach topic concept "baby" because of its direct link; all the other senses would have to go up to a top-level class (like root class entity) and then back down, a path too long and directly discarded. This is why even if we keep all possible Canonic Entities without discrimination, because of the manual choosing of the correct target class and because of the ontological paths linking different senses or individuals, we avoid many false-positive results and thus partially the problem of automatic WSD.

The second aspect needing discussion is how to solve the problem of topic concepts that are repeated amongst different topics. For example topic concept "military" belongs both to topics "Peace-keeping forces in Bosnia" and "Insurgency in Middle East" and a word like "army" would contribute to both topic concepts of both topics. Using context words (both previous and following words) we search for them if they appear in any of the topics. The distance from the target word and the frequency count in our limited window allows for ranking the scores assigned to a topic concept that belongs to multiple topics, thus differentiating between topics and directly increasing classification performance.

The final step is to heuristically score every topic based on their topic concepts' scores and assign the most likely topic to that text document.

As an overview, the system uses a general ontology as a graph in which it calculates custom distance-based scores between pre-defined topics and the words in text document. Based on the ranking of these scores the system offers the user a sorted list of topics. The system shows performance averaging close to the standard supervised classification system implementing the SVM with TF*IDF approaches, using only distances between classes in the ontology.

Contributions

In summary, the thesis makes the following contributions:

- An approach to General Entity Recognition using knowledge based methods and unsupervised algorithms. Based on a large, general ontology, the implemented system assigns ontological classes to text-extracted entities. Furthermore, it is a fine-grained system – its search space of ontological classes (that can be assigned to extracted entities from the text) is very large. For the implemented system the YAGO ontology was used, having 2+ million entities.
- The GER system has a unified approach based on an ontology seen as a semantic graph. It treats both named entities (proper nouns) and common nouns equally - basically it covers both the tasks of Named Entity Recognition (applied to proper nouns) and Word Sense Disambiguation (applied to common nouns) in a single pass.
- Varied methods and heuristics to reduce the complexity of IE/NLP problems: most likely ontological classes assignation heuristics for text entities to minimize future search space; sparse text entity influence matrix based on dependency trees; splitting of text entities into separate process groups based on influence matrix to vastly reduce the processing effort needed; algorithm designed to handle process groups in parallel (one process group per thread/core).
- A well-performing graph algorithm tuned to the problem of determining the best scoring sets of vertices in a weighted undirected k-partite graph. It is abstracted and can be applied to any problem that can be reduced to these specifications. It is shown to perform when applied to the General Entity Recognition task with dense graphs.

- An unsupervised system designed for text classification using general ontologies. Using partially annotator-corrected topics, such a system can obtain a relatively close score to the current state-of-the-art supervised classification standard (SVM machines), opening a new possible approach to this problem.
- A context-aware intelligent scoring method based on a custom semantic similarity distance function. This allows differentiated scores to be assigned to similar topic concepts that belong to different topics and thus increases topic classification accuracy.
- A topic scoring method implementing the concept of topic coverage. This method is applied only after the scores of topic concepts have been assigned. It allows topics with higher topic concept coverage but with lower scores (the lower scores have to be within the error-margin of the best score) to take precedence in the final sorted topic list.
- A survey of current tools, techniques and approaches in the domains of NLP processing, Word Sense Disambiguation and Named Entity Recognition.
- Interesting insights, benefits and limitations in the use of large-scale, general ontologies applied to the Information Extraction related problems treated in this thesis as shown by the two implemented systems. These less-investigated aspects are discussed and conclusions are offered.

VIII. References

[1] Ceri, S., & Brambilla, M. (2010). Search for Knowledge. In *Search Computing, LNCS 5950.* Berlin.

[2] Rijsbergen, C. v., Robertson, S., & Porter, M. (1980). New models in probabilistic information retrieval. In *British Library Research and Development Report no. 5587.* London: British Library.

[3] Voutilainen, A. (1995). A syntax-based part of speech analyser. *Seventh Conference of the European Chapter of the Association for Computational Linguistics*, (pp. 157-164). Dublin.

[4] Brill, E. (1995). Transformation-Based Error-Driven Learning and Natural Language Processing: A Case Study in Part-of-Speech Tagging. *Computational Linguistics , 21*, 543-565.

[5] Brants, T. (2000). TnT - A Statistical Part-Of-Speech Tagger.

[6] Brill, E. (1995). Unsupervised learning of disambiguation rules for part of speech tagging. *Third Workshop on Very Large Corpora*, (pp. 1-13).

[7] Klein, D., & Manning, C. D. (2003). Accurate unlexicalized parsing. *ACL.*

[8] Mitchell, T. M. (1997). *Machine Learning.*

[9] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA Data Mining Software: An Update. *SIGKDD Explorations , 11* (1).

[10] Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications. *Proceedings of the IEEE, 77*, *2.*

[11] Lafferty, J., McCallum, A., & Pereira, F. (2001). Conditional random fields: probabilistic models for segmenting and labeling sequence data. *International Conference on Machine Learning.*

[12] Sutton, C., & McCallum, A. (2007). An Introduction to Conditional Random Fields for Relational Learning. In *Introduction to statistical relational learning.* MIT Press.

[13] Brill, E. (2003). Processing Natural Language without Natural Language Processing. In *Lecture Notes in Computer Science.* Springer Berlin / Heidelberg.

[14] Gaifman, H. (1965). Gaifman, Haim. In *Information and Control* (pp. 304-307).

[15] Covington, M. A. (2000). A Fundamental Algorithm for Dependency Parsing. *39th Annual ACM Southeast Conference*, (pp. 95-102).

[16] Miyao, Y., Sagae, K., & Tsujii, J. (2007). Towards framework-independent evaluation of deep linguistic parsers. In *Grammar Engineering across Frameworks* (pp. 238-258).

[17] Pereira, R. M. (2006). Online learning of approximate dependency parsing algorithms. *EACL.*

[18] Charniak, E. (2000). A maximum-entropy-inspired parser. *NAACL.*

[19] Charniak, E., & Johnson, M. (2005). Coarse-to-fine n-best parsing and MaxEnt discriminative reranking. *ACL.*

[20] Petrov, S., & Klein, D. (2007). Improved inference for unlexicalized parsing. *HLT-NAACL 2007.*

[21] Sagae, K., Miyao, Y., Matsuzaki, T., & Tsujii, J. (2008). Challenges in mapping of syntactic representations for framework-independent parser evaluation. *Workshop on Automated Syntatic Annotations for Interoperable Language Resources.*

[22] Miyao, Y., & Tsujii, J. (2008). Feature forest models for probabilistic HPSG parsing. *Computational Linguistics.*

[23] Kim, D., Ohta, T., Teteisi, Y., & Tsujii, J. (2003). GENIA corpus — a semantically annotated corpus for bio-texmining. In *Bioinformatics.*

[24] Soon, W., Ng, H., & Lim, D. (2001). A machine learning approach to coreference resolution of noun phrases. In *Computational Linguistics* (pp. 521-540).

[25] Yang, X., Zhou, G., Su, J., & Tan, C. L. (2004). Improving noun phrase coreference resolution by matching strings. *Proc. of the 1st Int'l Joint Conference on Natural Language Processing.* Hainan.

[26] Mccallum, A., & Wellner, B. (2004). Conditional models of identity uncertainty with application to noun coreference. *NIPS-17.* Vancouver.

[27] Cardie, C., & Wagstaff, K. (1999). Noun phrase coreference as clustering. *Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, (pp. 82-89). Maryland.

[28] Gainaru, A., Dumitrescu, S. D., & Trausan-Matu, S. (2010). NLP Toolbox. *COMM2010 - IEEE* , 22-26.

[29] Zhou, G., & Su, J. (2004). A High-Performance Coreference Resolution System using a Constraint-based Multi-Agent Strategy. *20th International Conference on Computational Linguistics (COLING-2004)*, (pp. 522-528). Geneva.

[30] Bean, D., & Riloff, E. (2004). Unsupervised Learning of Contextual Role Knowledge for Coreference Resolution. *Human Language Technology Conference / North American Chapter of the Association for Computational Linguistics Annual Meeting (HLT/NAACL-04)*, (pp. 297-304). Boston.

[31] Marcus, M. P., Santorini, B., & Marcinkiewicz, M. A. (1993). Building a Large Annotated Corpus of English: The Penn Treebank. *COMPUTATIONAL LINGUISTICS , 19* (2), 313-330.

[32] Noy, N., & McGuinness, D. (2001). *Ontology Development 101: A Guide to Creating Your First Ontology.* Stanford Knowledge Systems Laboratory.

[33] Dumitrescu, S. D., Smeureanu, A., Diosteanu, A., & Cotfas, L. (2010). Adaptable Network Management System Using GIS and network ontology. *9th RoEduNet IEEE International Conference*, (pp. 310-315).

[34] DIOSTEANU, A., COTFAS, L., SMEUREANU, A., & DUMITRESCU, S. D. (2010). Multi-Agents and GIS Framework for Collaborative Supply Chain Management Application. *9th RoEduNet IEEE International Conference*, (pp. 157-162). Sibiu.

[35] Miller, G. A., Beckwith, R., Fellbaum, C. D., Gross, D., & Miller, K. (1990). WordNet: An online lexical database. 235-244.

[36] Navigli, R. (2006). Meaningful Clustering of Senses Helps Boost Word Sense Disambiguation Performance. *44th Annual Meeting of the Association for Computational Linguistics joint with the 21st International Conference on Computational Linguistics*, (pp. 105-112). Sydney.

[37] Snow., R., S., P., Jurafsky., D., & Ng, A. Y. (2007). Learning to Merge Word Senses. *Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, (pp. 1005-1014). Prague.

[38] Resnik, P. (1995). Using information content to evaluate semantic similarity in a taxonomy. *14th International Joint Conference on Artificial Intelligence.* Montreal.

[39] Suchanek, F. M., Kasneci, G., & Weikum, G. (2007). Yago - A Core of Semantic Knowledge. 16th international World Wide Web conference (WWW 2007).

[40] Pasca, M., Lin, D., Bigham, J., Lifchits, A., & Jain., A. (2006). Names and similarities on the web: Fact extraction in the fast lane. *Proceedings of the Association for Computational Linguistics.*

[41] Suchanek, F. M., Kasneci, G., & Weikum, G. (2008). YAGO: A large ontology from Wikipedia and WordNet.

[42] Suchanek, F. M., Sozio, M., & Weikum, G. SOFIE: A Self-Organizing Framework for Information Extraction. *18th International World Wide Web conference.* 2009.

[43] Banko, M. (2009). *Open Information Extraction for the Web.* PHD Thesis, University of Washington, Washington.

[44] Banko, M., Cafarella, M. J., Soderland, S., Broadhead, M., & Etzioni, O. (2007). Open information extraction from the web. *Proceedings of IJCAI.*

[45] Ramshaw, L. A., & Marcus, M. P. (1995). Text chunking using transformation based learning. In *CoRR.*

[46] Yates, A., & Etzioni, O. (2007). Unsupervised resolution of objects and relations on the web. *Proceedings of the Conference on Human Language Technologies / North American Chapter of the Association for Computational Linguistics.*

[47] Collins, M. (1999). *Head-Driven Statistical Models for Natural Language Parsing.* PHD Thesis, University of Pennsylvania, Pennsylvania.

[48] Dean, J., & Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. *OSDI.*

[49] Gale, W. A., Church, K. W., & Yarowsky, D. (1993). A method for disambiguating word senses in a large corpus. (26).

[50] Ide, N., & Véronis, J. (1998). Word Sense Disambiguation: The State of the Art. *24*.

[51] Wilks, Y., & Stevenson., M. (1998). Word Sense Disambiguation using Optimised Combinations of Knowledge Sources. *COLING-ACL'98.* Montreal, Canada.

[52] Kelly, E. F., & Stone, P. J. (1975). *Computer recognition of English word senses.* North-Holland Pub.

[53] E., B. (1988). An experiment in computational discrimination of English word senses. In *IBM J. Res. Devel* (Vol. 32, pp. 185–194).

[54] Rivest, R. L. (1987). Learning decision lists. In *Machine Learning 2* (Vol. 3, pp. 229–246).

[55] Quinlan, J. R. (1993). *Programs for Machine Learning.* San Francisco: Morgan Kaufmann.

[56] Bramer, M. (2007). *Principles of Data Mining* . Springer London.

[57] Deng, H., Runger, G., & Tuv, E. (2011). Bias of importance measures for multi-valued attributes and solutions. *21st International Conference on Artificial Neural Networks.*

[58] McCulloch, W., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. In *Bull. Math. Biophys.* (Vol. 5, pp. 115-133).

[59] Veronis, J., & Ide, N. (1990). Word sense disambiguation with very large neural networks extracted from machine readable dictionaries. *13th International Conference on Computational Linguistics*, (pp. 389–394). Helsinki, Finland.

[60] Tsatsaronis, G., Vazirgiannis, M., & Androutsopoulos, I. (2007). Word sense disambiguation with spreading activation networks generated from thesauri. *International Joint Conference on Artificial Intelligence*, (pp. 1725–1730). Hyderabad, India.

[61] Mooney, R. J. (1996). Comparative experiments on disambiguating word senses: An illustration of the role of bias in machine learning. *Conference on Empirical Methods in Natural Language Processing*, (pp. 82–91).

[62] Towell, G., & Voorhees, E. (1998). Disambiguating highly ambiguous words. *Computational Linguistics* , 125-145.

[63] Hoste, V., Hendrick, I., Daelemans, W., & Van Den Bosch, A. (2002). Parameter optimization for machine learning of word sense disambiguation. *J. Nat. Lang. Eng.* , *8* (4), 311-325.

[64] Decadt, B., Hoste, V., Daelemans, W., & Bosch, V. D. (2004). GAMBL, genetic algorithm optimization of memory-based WSD. *3rd International Workshop on the Evaluation of Systems for the Semantic Analysis of Text (Senseval-3)*, (pp. 108–112). Barcelona, Spain.

[65] Zhang, H. (2004). The Optimality of Naive Bayes. *FLAIRS.*

[66] Caruana, R., & Niculescu-Mizil, A. (2006). An empirical comparison of supervised learning algorithms. *23rd international conference on Machine learning.*

[67] Bruce, R., & Weibe, J. (1999). Decomposable modeling in natural language processing. *Computational Linguistics , 2* (25), 195–207.

[68] Ng, T. H. (1997). Getting serious about word sense disambiguation. *ACL SIGLEX Workshop on Tagging Text with Lexical Semantics: Why, What, and How?*, (pp. 1-7). Washington D.C.

[69] Murata, M., Utiyama, M., Uchimoto, K., Ma, Q., & Isahara, H. (2001). Japanese word sense disambiguation using the simple Bayes and support vector machine methods. *2nd International Workshop on Evaluating Word Sense Disambiguation Systems (SENSEVAL-2)*, (pp. 135–138). Toulouse, France.

[70] Keok, L. Y., & Ng, H. T. (2002). An empirical evaluation of knowledge sources and learning algorithms for word sense disambiguation. *Conference on Empirical Methods in Natural Language Processing*, (pp. 41–48). Philadelphia, PA.

[71] Lin, D. (1998). Automatic retrieval and clustering of similar words. *17th International Conference on Computational linguistics*, (pp. 768–774). Montreal, P.Q., Canada.

[72] Pedersen, T., & Bruce, R. (1997). Distinguishing word senses in untagged text. *Conference on Empirical Methods in Natural Language Processing*, (pp. 197-207). Providence, RI.

[73] Savova, G., Pedersen, T., Purandare, A., & Kulkarni, A. (2005). *Resolving ambiguities in biomedical text with unsupervised clustering approaches.* Minneapolis, MN: UMSI.

[74] Dumais, S. T. (2004). Latent semantic analysis. *Annual Review of Information Science and Technology , 38* (1), 188-230.

[75] Purandare, A., & Pedersen, T. (2004). Improving word sense discrimination with gloss augmented feature vectors. *Workshop on Lexical Resources for the Web and Word Sense Disambiguation*, (pp. 123–130). Puebla, Mexico.

[76] Widdows, D., & Dorow, B. (2002). A graph model for unsupervised lexical acquisition. *International Conference on Computational Linguistics*, (pp. 1-7). Taipei, Taiwan.

[77] Veronis, J. (2004). Hyperlex: Lexical cartography for information retrieval. *Comput. Speech Lang. , 18* (3), 223–252.

[78] Brin, S., & Page, M. (1998). Anatomy of a large-scale hypertextual Web search engine. *Conference on World Wide Web*, (pp. 107-117). Brisbane, Australia.

[79] Agirre, E., & Stevenson, M. (2006). Knowledge sources for WSD. *Word Sense Disambiguation: Algorithms and Applications* , 217–251.

[80] Klapaftis, I. P., & M., S. (2007). UOY: A Hypergraph Model For Word Sense Induction & Disambiguation. *Workshop on Semantic Evaluations (SemEval)* .

[81] Lesk, M. (1986). Automatic sense disambiguation using machine readable dictionaries: How to tell a pine cone from an ice cream cone. *5th SIGDOC*, (pp. 24-26). New York.

[82] Vasilescu, F., Langlais, P., & Lapalme, G. (2004). Evaluating variants of the Lesk approach for disambiguating words. *Conference on Language Resources and Evaluation*, (pp. 633–636). Lisbon, Portugal.

[83] Banerjee, S., & Pedersen, T. (2002). An adapted Lesk algorithm for word sense disambiguation using WordNet. *Conference on Computational Linguistics and Intelligent Text Processing*, (pp. 136–145). Mexico City, Mexico.

[84] Patwardhan, S., Banerjee, S., & Pedersen, T. (2003). Using measures of semantic relatedeness for word sense disambiguation. *Conference on Computational Linguistics and Intelligent Text Processing*, (pp. 241–257). Mexico City, Mexico.

[85] Galley, M., & McKeown, K. (2003). Improving word sense disambiguation in lexical chaining. *International Joint Conference in Artificial Intelligence*, (pp. 1486–1488). Acapulco, Mexico.

[86] Mihalcea, R., & Moldovan, D. (2000). An iterative approach to word sense disambiguation. *Florida Artificial Intelligence Research Society*, (pp. 219–223). Orlando, US.

[87] Mihalcea, R. (2005). Large vocabulary unsupervised word sense disambiguation with graph-based algorithms for sequence data labeling. *Joint Human Language Technology and Empirical Methods in Natural Language Processing Conference*, (pp. 411-418). Vancouver, Canada.

[88] Resnik, P. (1993). *Selection and information: A class-based approach to lexical relationships.* Ph.D. Thesis, University of Pennsylvania.

[89] Agirre, E., & Martinez, D. (2001). Learning class-to-class selectional preferences. *5th Conference on Computational Natural Language Learning*, (pp. 15-22). Toulouse, France.

[90] McCarthy, D., Carroll, J., & Preiss, J. (2001). Disambiguating noun and verb senses using automatically acquired selectional preferences. *International Workshop on Evaluating Word Sense Disambiguation Systems*, (pp. 119–122). Toulouse, France.

[91] Gale, W., Church, K., & Yarowsky, D. (1992). Estimating upper and lower bounds on the performance of word-sense disambiguation programs. *Annual Meeting of the Association for Computational Linguistics*, (pp. 249–256). Newark, U.S.A.

[92] Gale, W., Church, K., & Yarowsky, D. (1992). One sense per discourse. *DARPA Speech and Natural Language Workshop*, (pp. 233–237). Nwe York, U.S.A.

[93] Krovetz, R. (1998). More than one sense per discourse. *Workshop on Evaluating Word Sense Disambiguation Systems.* Sussex, England.

[94] Palmer, M., Dang, H., & Fellbaum, C. (2007). Making fine-grained and coarse-grained sense distinctions, both manually and automatically. *Journal of Natural Language Engineering , 13* (2), 137-163.

[95] Snyder, B., & Palmer, M. (2004). The English all-words task. *Senseval-3.*

[96] Navigli, R. (2009). Word sense disambiguation: A survey. *ACM Computer Survey , 41* (2).

[97] Resnik, P., & Yarkowsky, D. Distinguishing systems and distinguishing senses: new evaluation methods for word sense disambiguation. *J. Nat. Lang. Eng , 5* (2), 113-133.

[98] Grishman, R., & Sundheim, B. (1996). Message Understanding Conference - 6: A Brief History. *International Conference on Computational Linguistics.*

[99] Rau, L. F. (1991). Extracting Company Names from Text. *Artificial Intelligence Applications of IEEE.*

[100] Lee, S., & Geunbae Lee, G. (2005). Heuristic Methods for Reducing Errors of Geographic Named Entities Learned by Bootstrapping. *International Joint Conference on Natural Language Processing.*

[101] Witten, I. H., Bray, Z., Mahoui, M., & J., T. W. (1999). Using Language Models for Generic Entity Extraction. *International Conference on Machine Learning. Text Mining.*

[102] Cohen, W. W., & Sarawagi, S. (2004). Exploiting Dictionaries in Named Entity Extraction: Combining Semi-Markov Extraction Processes and Data Integration Methods. *Conference on Knowledge Discovery in Data.*

[103] Tsuruoka, Y., & Tsujii, J. (2003). Boosting Precision and Recall of Dictionary-Based Protein Name Recognition. *Conference of Association for Computational Linguistics in Natural Language Processing in Biomedicine.*

[104] Alfonseca, E., & Manandhar, S. (2002). An Unsupervised Method for General Named Entity Recognition and Automated Concept Discovery. *International Conference on General WordNet.*

[105] Poibeau, T., & Kosseim, L. (2001). Proper Name Extraction from Non-Journalistic Texts. *Computational Linguistics in the Netherlands.*

[106] Bikel, D. M., Miller, S., Schwartz, R., & Weischedel, R. (1997). Nymble: a High-Performance Learning Name-finder. *Conference on Applied Natural Language Processing.*

[107] Sekine, S. (1998). Description of the Japanese NE System Used For Met-2. *Message Understanding Conference.*

[108] Borthwick, A., Sterling, J., Agichtein, E., & Grishman, R. (1998). NYU: Description of the MENE Named Entity System as used in MUC-7. *7th Message Understanding Conference.*

[109] Finkel, J. R., Grenager, T., & Manning, C. (2005). Proceedings of the 43nd Annual Meeting of the Association for Computational Linguistics (ACL 2005). 363-370.

[110] Brin, S. (1998). Extracting Patterns and Relations from the World Wide Web. *Conference of Extending Database Technology. Workshop on the Web and Databases.*

[111] Collins, M., & Singer, Y. Unsupervised Models for Named Entity Classification. *Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora.*

[112] Riloff, E., & Jones, R. (1999). Learning Dictionaries for Information Extraction using Multi-level Bootstrapping. *National Conference on Artificial Intelligence.*

[113] Cucchiarelli, A., & Velardi, P. (2001). Unsupervised Named Entity Recognition Using Syntactic and Semantic Contextual Evidence. (M. Press, Ed.) *Computational Linguistics* , 123-131.

[114] Pasca, M., Lin, D., Bigham, J., Lifchits, A., & Jain, A. (2006). Organizing and Searching the World Wide Web of Facts—Step One: The One-Million Fact Extraction Challenge. *National Conference on Artificial Intelligence.*

[115] Evans, R. (2003). A Framework for Named Entity Recognition in the Open Domain. *Recent Advances in Natural Language Processing.*

[116] Nadeau, D., & Sekine, S. (2007). A survey of named entity recognition and classification. *Lingvisticae Investigationes , 30* (1), 3-26.

[117] Raghavan, H., & Allan, J. (2004). Using Soundex Codes for Indexing Names in ASR documents. *Human Language Technology conference - North American chapter of the Association for Computational Linguistics. Interdisciplinary Approaches to Speech.*

[118] Mikheev, A. (1999). A Knowledge-free Method for Capitalized Word Disambiguation. *Conference of Association for Computational Linguistics.*

[119] Yarowsky, D. (1992). Word-sense disambiguation using statistical models of roget's categories trained on large corpora. *COLING*, (pp. 454-460). Nantes, France.

[120] Agirre, E., Ansa, O., Hovy, E., & Martinez, D. (2000). Enriching very large ontologies using the www. *Ontology Learning Workshop, ECAI.* Berlin, Germany.

[121] Ciaramita, M., & Johnson, M. (2003). Supersense Tagging of Unknown Nouns in WordNet. *EMNLP.*

[122] Ciaramita, M., & Yasemin, A. (2006). Broad-Coverage Sense Disambiguation and Information Extraction with a Supersense Sequence Tagger. *Empirical Methods in Natural Language Processing (EMNLP).*

[123] Manning, C., & Schutze, H. (1999). Foundations of Statistical Natural Language Processing.

[124] Navigli, R. (2006). Consistent validation of manual and automatic sense annotations with the aid of semantic graphs. *Computational Lingusitics , 32* (2), 273-281.

[125] Navigli, R. (2006). Experiments on the validation of sense annotations assisted by lexical chains. *11th Conference of the European Chapter of the Association for Computational Linguistics*, (pp. 129–136). Trento, Italy.

[126] Mihalcea, R. (2006). Knowledge-based methods for WSD. *Word Sense Disambiguation: Algorithms and Applications* , pp. 107-131.

[127] Hindle, D., & Rooth, M. (1993). Structural ambiguity and lexical relations. *Computational Lingusitics*, *19*, pp. 103-120.

[128] Rada, R., Mili, H., Bicknell, E., & Blettner, M. (1989). Development and application of a metric on semantic nets. *IEEE Transactions on Syst. Man Cybernet.*, *19(1)*, pp. 17-30.

[129] Navigli, R., & Velardi, P. (2005). Structural semantic interconnections: A knowledge-based approach to word sense disambiguation. *IEEE Transactions on Pattern Analysis and Machine Intelligence , 27* (7).

[130] Mihalcea, R., Tarau, P., & Figa, E. (2004). Pagerank on semantic networks, with application to word sense disambiguation. *20th International Conference on Computational Linguistics*, (pp. 1126-1132). Geneva, Switzerland.

[131] Cormen, T. H. (2009). *Introduction to Algorithms, 3rd ed.* Cambridge, England: MIT Press.

[132] Toutanova, K., Klein, D., Manning, C., & Singer, Y. (2003). Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. *HLT-NAACL 2003*, (pp. 252-259).

[133] Toutanova, K., & Manning, C. D. (2000). Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger. *Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-2000)*, (pp. 63-70).

[134] Hirst, G., & Onge, D. S. (1998). Lexical chains as representations of context for the detection and correction of malapropisms. *Fellbaum*, (pp. 305–332).

[135] Nedjah, N., França, F. M., & Souza, A. F. (2009). Intelligent Text Categorization and Clustering. *Studies in Computational Intelligence , 64*.

[136] Ikonomakis, M., Kotsiantis, S., & Tampakas, V. (2005). Text Classification Using Machine Learning Tehniques. *WSEAS Transactions on Computers , 4*, 966–974.

[137] Sebastiani, F. (2002). Machine Learning in Automated Text Categorization. *ACM Computing Surveys , 34(1)*, 1-47.

[138] Moravec, P., M., K., & Snasel, V. (2004). LSI vs. wordnet ontology in dimension reduction for information retrieval. 18-26.

[139] Lv, L., Liu, Y. S., & Liu, Y. (2006). Realizing English text classification with semantic set index method. *Journal of Beijing University of Posts and Telecommunications , 29* (2), 22-25.

[140] Yang, X.-Q., Sun, N., Sun, T.-L., Cao, X.-Y., & Zheng, X.-J. (2009). The Application of Latent Semantic Indexing and Ontology in Text Classification. *International Journal of Innovative Computing, Information and Control , 5* (12(A)), 4491-4499.

[141] Brut, M., Dumitrescu, S., & Sèdes, F. (2010). A Semantic-Oriented Approach for Organizing and Developing Annotation for E-learning. *IEEE Transactions on Learning Technologies* .

[142] Lee, Y.-H., Tsao, W.-J., & Chu, T.-H. (2009). Use of Ontology to Support Concept-Based Text Categorization. *Lecture Notes in Business Information Processing , 22(6)*, 201-213.

[143] Gu, H., & Zhou, K. (2006). Text Classification Based on Domain Ontology. *Journal of Communication and Computer , 3* (5).

[144] Yang, X., Zhou, G., Su, J., & Tan, C. L. (2004). An NP-Cluster Based Approach to Coreference Resolution. *20th International Conference on Computational Linguistics.* Geneva.

[145] Gangemi, A., Guarino, N., Masolo, C., & Oltramari, A. (2003). Sweetening WORDNET with DOLCE. *AI Magazine , 24*.

Appendix

Example run of the proposed GER System

This annex presents a test run of the system on a document composed of two sentences. We present the steps of the system and we discuss the results.

Input document: "Currently, **heart disease** and **stroke** are the leading **causes** of **death worldwide** and according to **World Health Organisation estimates** will kill almost 24 million **people** by 2030. The metabolic **syndrome**, associated with an increased **risk** of **type** 2 **diabetes** and cardiovascular **disease**, affects about one fifth of the **world**'s **adult population**."

The GER System starts by splitting the document into sentences. Then each sentence goes through part-of-speech tagging, parsing, tokenization and a standard NER system to label named entities with one of the probable 3 classes (person, location and organization). The entities we are interested in are common and proper nouns (shown in bold above).

We obtain the following information:

Sentence 1: "Currently, **heart disease** and **stroke** are the leading **causes** of **death worldwide** and according to **World Health Organisation estimates** will kill almost 24 million **people** by 2030."

Syntactic tree for sentence #1

(on which to determine POS taggs and NP groups)

```
New sentence ID: 1
(ROOT
  (S
    (ADVP (RB Currently))
    (, ,)
    (S
      (NP
        (NP (NN heart) (NN disease))
        (CC and)
        (NP (NN stroke)))
      (VP (VBP are)
        (NP
          (NP (DT the) (VBG leading) (NNS causes))
          (PP (IN of)
            (NP (NN death) (NN worldwide))))))
    (CC and)
    (S
      (PP (VBG according)
        (PP (TO to)
          (NP (NNP World) (NNP Health) (NNP Organisation))))
      (NP (NNS estimates))
      (VP (MD will)
        (VP (VB kill)
          (NP
            (QP (RB almost) (CD 24) (CD million))
            (NNS people))
          (PP (IN by)
            (NP (CD 2030))))))
    (. .)))

advmod(causes-10, Currently-1)
nn(disease-4, heart-3)
nsubj(causes-10, disease-4)
conj_and(disease-4, stroke-6)
nsubj(causes-10, stroke-6)
cop(causes-10, are-7)
det(causes-10, the-8)
amod(causes-10, leading-9)
nn(worldwide-13, death-12)
prep_of(causes-10, worldwide-13)
prepc_according_to(kill-22, to-16)
nn(Organisation-19, World-17)
nn(Organisation-19, Health-18)
pobj(kill-22, Organisation-19)
nsubj(kill-22, estimates-20)
aux(kill-22, will-21)
conj_and(causes-10, kill-22)
quantmod(million-25, almost-23)
number(million-25, 24-24)
num(people-26, million-25)
dobj(kill-22, people-26)
prep_by(kill-22, 2030-28)
```

Dependency tree for sentence #1

(on which to calculate the Influence Matrix)

Token list for sentence #1

(Format: ID, token, type of word, POS tag, NER tag, original token, stem, singular form)

```
T1000 [Currently] (UNKNOWN,RB,O) O[Currently] S[current] I[currently]
T1001 [,] (PUNCTUATION,,,O) O[,] S[Invalid term] I[,]
T1002 [heart] (COMMON,NN,O) O[heart] S[heart] I[heart]
T1003 [disease] (COMMON,NN,O) O[disease] S[diseas] I[disease]
T1004 [and] (COMMON,CC,O) O[and] S[and] I[and]
T1005 [stroke] (COMMON,NN,O) O[stroke] S[stroke] I[stroke]
T1006 [are] (COMMON,VBP,O) O[are] S[ar] I[are]
T1007 [the] (COMMON,DT,O) O[the] S[the] I[the]
T1008 [leading] (UNKNOWN,VBG,O) O[leading] S[lead] I[leading]
T1009 [causes] (COMMON,NNS,O) O[causes] S[caus] I[cause]
T1010 [of] (COMMON,IN,O) O[of] S[of] I[of]
T1011 [death] (COMMON,NN,O) O[death] S[death] I[death]
T1012 [worldwide] (COMMON,NN,O) O[worldwide] S[worldwid] I[worldwide]
T1013 [and] (COMMON,CC,O) O[and] S[and] I[and]
T1014 [according] (UNKNOWN,VBG,O) O[according] S[accord] I[according]
T1015 [to] (COMMON,TO,O) O[to] S[to] I[to]
T1016 [World] (UNKNOWN,NNP,ORGANIZATION) O[World] S[world] I[world]
T1017 [Health] (UNKNOWN,NNP,ORGANIZATION) O[Health] S[health] I[health]
T1018 [Organisation] (UNKNOWN,NNP,ORGANIZATION) O[Organisation] S[organis] I[organisation]
T1019 [estimates] (COMMON,NNS,O) O[estimates] S[estim] I[estimate]
T1020 [will] (COMMON,MD,O) O[will] S[will] I[will]
T1021 [kill] (COMMON,VB,O) O[kill] S[kill] I[kill]
T1022 [almost] (COMMON,RB,O) O[almost] S[almost] I[almost]
T1023 [24] (UNKNOWN,CD,NUMBER) O[24] S[Invalid term] I[24]
T1024 [million] (COMMON,CD,NUMBER) O[million] S[million] I[million]
T1025 [people] (COMMON,NNS,O) O[people] S[person] I[person]
T1026 [by] (COMMON,IN,O) O[by] S[by] I[by]
T1027 [2030] (UNKNOWN,CD,DATE) O[2030] S[Invalid term] I[2030]
T1028 [.] (PUNCTUATION,.,O) O[.] S[Invalid term] I[.]

New sentence ID: 2
(ROOT
  (S
    (NP
      (NP (DT The) (JJ metabolic) (NN syndrome))
      (, ,)
      (VP (VBN associated)
        (PP (IN with)
```

Syntactic tree for sentence #2

(on which to determine POS taggs and NP groups)

```
          (NP
            (NP (DT an) (VBN increased) (NN risk))
            (PP (IN of)
              (NP
                (NP (NN type) (CD 2) (NN diabetes))
                (CC and)
                (NP (JJ cardiovascular) (NN disease))))))
      (, ,))
    (VP (VBZ affects)
      (PP (IN about)
        (NP
          (NP (CD one) (NN fifth))
          (PP (IN of)
            (NP
              (NP (DT the) (NN world) (POS 's))
              (NN adult) (NN population))))))
    (. .)))
```

Dependency tree for sentence #2

(on which to calculate the Influence Matrix)

```
det(syndrome-3, The-1)
amod(syndrome-3, metabolic-2)
nsubj(affects-18, syndrome-3)
partmod(syndrome-3, associated-5)
det(risk-9, an-7)
amod(risk-9, increased-8)
prep_with(associated-5, risk-9)
nn(diabetes-13, type-11)
num(diabetes-13, 2-12)
prep_of(risk-9, diabetes-13)
amod(disease-16, cardiovascular-15)
prep_of(risk-9, disease-16)
conj_and(diabetes-13, disease-16)
num(fifth-21, one-20)
prep_about(affects-18, fifth-21)
det(world-24, the-23)
poss(population-27, world-24)
nn(population-27, adult-26)
prep_of(fifth-21, population-27)
```

Token list for sentence #2

(Format: ID, token, type of word, POS tag, NER tag, original token, stem, singular form)

```
T2000 [The] (UNKNOWN,DT,O) O[The] S[the] I[the]
T2001 [metabolic] (UNKNOWN,JJ,O) O[metabolic] S[metabol] I[metabolic]
T2002 [syndrome] (COMMON,NN,O) O[syndrome] S[syndrome] I[syndrome]
T2003 [,] (PUNCTUATION,,,O) O[,] S[Invalid term] I[,]
T2004 [associated] (UNKNOWN,VBN,O) O[associated] S[associ] I[associated]
T2005 [with] (COMMON,IN,O) O[with] S[with] I[with]
T2006 [an] (COMMON,DT,O) O[an] S[an] I[an]
T2007 [increased] (UNKNOWN,VBN,O) O[increased] S[increas] I[increased]
T2008 [risk] (COMMON,NN,O) O[risk] S[risk] I[risk]
T2009 [of] (COMMON,IN,O) O[of] S[of] I[of]
T2010 [type] (COMMON,NN,O) O[type] S[type] I[type]
T2011 [2] (UNKNOWN,CD,NUMBER) O[2] S[Invalid term] I[2]
T2012 [diabetes] (UNKNOWN,NN,O) O[diabetes] S[diabet] I[diabete]
T2013 [and] (COMMON,CC,O) O[and] S[and] I[and]
T2014 [cardiovascular] (COMMON,JJ,O) O[cardiovascular] S[cardiovascular] I[cardiovascular]
T2015 [disease] (COMMON,NN,O) O[disease] S[diseas] I[disease]
T2016 [,] (PUNCTUATION,,,O) O[,] S[Invalid term] I[,]
T2017 [affects] (COMMON,VBZ,O) O[affects] S[affect] I[affect]
T2018 [about] (COMMON,IN,O) O[about] S[about] I[about]
T2019 [one] (COMMON,CD,NUMBER) O[one] S[on] I[one]
T2020 [fifth] (COMMON,NN,O) O[fifth] S[fifth] I[fifth]
T2021 [of] (COMMON,IN,O) O[of] S[of] I[of]
T2022 [the] (COMMON,DT,O) O[the] S[the] I[the]
T2023 [world] (COMMON,NN,O) O[world] S[world] I[world]
T2024 ['s] (PUNCTUATION,POS,O) O['s] S[Invalid term] I[']
T2025 [adult] (COMMON,NN,O) O[adult] S[adult] I[adult]
```

```
T2026 [population] (COMMON,NN,O) O[population] S[popul] I[population]
T2027 [.] (PUNCTUATION,.,O) O[.] S[Invalid term] I[.]
```

Tokens that can be merged are merged (ex: “World”, “Health” and “Organization” are merged into a single String Entity “World Health Organization”). Based on the dependency trees for every sentence, the asimetric influence matrix is created:

```
Influence Matrix: (ROW-Subject) has property (COLUMN-Object)
             hea  dis  str  cau  dea  wor  Wor  est  per  syn  ris  typ  dia  dis  wor  adu  pop
heart             0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1   0    0    0    0    0    0    0    0
disease       1   ---   1   0.1   1    1    1    1    1    0    0    0    0    0    0    0    0
stroke       0.1  0.1  ---  0.1  0.1  0.1  0.1  0.1  0.1   0    0    0    0    0    0    0    0
cause        0.9   1   0.9  ---  0.9   1   0.9  0.9  0.9   0    0    0    0    0    0    0    0
death        0.1  0.1  0.1  0.1  ---  0.1  0.1  0.1  0.1   0    0    0    0    0    0    0    0
worldwide    0.1  0.1  0.1  0.1   1   ---  0.1  0.1  0.1   0    0    0    0    0    0    0    0
World Hea    0.1  0.1  0.1  0.1  0.1  0.1  ---  0.1  0.1   0    0    0    0    0    0    0    0
estimate     0.1  0.1  0.1  0.1  0.1  0.1   1   ---   1    0    0    0    0    0    0    0    0
person       0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  ---   0    0    0    0    0    0    0    0
syndrome      0    0    0    0    0    0    0    0    0   ---  0.9  0.1  0.1  0.1  0.1  0.1   1
risk          0    0    0    0    0    0    0    0    0   0.1  ---  0.9   1   0.9  0.1  0.1  0.1
type          0    0    0    0    0    0    0    0    0   0.1  0.1  ---  0.1  0.1  0.1  0.1  0.1
diabetes      0    0    0    0    0    0    0    0    0   0.1  0.1   1   ---   1   0.1  0.1  0.1
disease       0    0    0    0    0    0    0    0    0   0.1  0.1  0.1  0.1  ---  0.1  0.1  0.1
world         0    0    0    0    0    0    0    0    0   0.1  0.1  0.1  0.1  0.1  ---  0.1  0.1
adult         0    0    0    0    0    0    0    0    0   0.1  0.1  0.1  0.1  0.1  0.1  ---  0.1
population    0    0    0    0    0    0    0    0    0   0.1  0.1  0.1  0.1  0.1   1    1   ---
```

The matrix shows the influence of each entity on every other. For example the influence of “heart” on “disease” in the first sentence is 1.0 because “disease” is determined by “heart” (ex: Question: what type of disease? Answer: A *heart* disease. This is what is meant by the influence of an entity over another)

```
computeInfluence for : disease-4 - heart-3
   Dependency val 1.0 direct link for nn(disease-4,heart-3)
```

while “hearth” is not determined by “disease” and therefore receives the context score of 0.1.

Next, each String Entity (single or multiple joined tokens) gets assigned a number of possible classes from the YAGO ontology. The assignation process was described in section V.4.2.1. For example, String Entity “heart” gets the following probable canonic entity set $PCE_{SEheart}$ containing 33 items:

```
wordnet_heart_valve_105395098, wordnet_heart_valve_103507857,
wordnet_bullock's_heart_111694866, wordnet_bullock's_heart_107761461,
wordnet_heart_block_114362593, wordnet_line_of_heart_113906936,
wordnet_bleeding_heart_111910271, wordnet_heart_disease_114103288,
wordnet_biauriculate_heart_105389310, wordnet_heart_104857490,
wordnet_heart_attack_114112855, wordnet_heart_failure_114112255,
wordnet_broken_heart_107534847, wordnet_valvular_heart_disease_114112466,
wordnet_artificial_heart_102745492, wordnet_heart_murmur_114334814, wordnet_heart_105388805,
wordnet_congenital_heart_defect_114469014, wordnet_bleeding_heart_109859818,
wordnet_heart_104624826, wordnet_heart_105919263, wordnet_heart_113865904,
wordnet_heart_107651905, wordnet_heart_103507048, wordnet_athlete's_heart_105389182,
wordnet_congestive_heart_failure_114112719, wordnet_heart_surgery_100675219,
wordnet_heart_cherry_112642435, wordnet_heart_cherry_107757602,
wordnet_artichoke_heart_107718920, wordnet_rheumatic_heart_disease_114142983,
wordnet_heart_urchin_102319829, wordnet_coronary_heart_disease_114102631
```

As a side comment, out of the 33 entities we can spot 7 `wordnet_heart_#` (underlined). In this scenario we are interested in `wordnet_heart_107651905` (marked with italics) which WordNet describes as: (n) heart, pump, ticker (the hollow muscular organ located behind the sternum and between the lungs; its rhythmic contractions move the blood through the body) *"he stood still, his heart thumping wildly",* being actually the second most used sense for the word “heart”. However, the computer does not know at this time which, if any, Canonic Entity is the correct choice.

Next, the Operational Graph is created. Initially the Operational Graph consists of only the WordNet hypernym graph to which every possible Canonic Entity of every String Entity is added (even if it forms a disconnected graph). Next, a breadth-first search is performed on the YAGO graph starting from the just added Canonic Entities to a maximum depth of 3, adding encountered neighbors and edges.

Separately, Process Groups are created using a flood-fill algorithm on the influence matrix. From this point forward each Process Group is treated separately as they are all independent.

We consider Process Group #1 as the group containing the entities of the first sentence.

At this point, we need to create the *N*-partite graph for this Process Group. For sentence 1 we have 9 String Entities so $N = 9$. Starting at every Canonic Entity belonging to every String Entity, a breadth-search is performed on the Operational Graph to see what other Canonic Entities of interest are in the neighborhood. For example, from the entities belonging to String Entity “heart”, starting from `wordnet_heart_murmur_114334814` we reach `wordnet_disease_114070360`. To this link we assign a score (described in sections V.4.3.1. and V.4.3.3.) based on the distance, influence and types of relations between the two end-point entities.

```
wordnet_heart_murmur_114334814 -> wordnet_disease_114070360 [1003] 1:2
infl: 0.1 score: 0.05   wordnet_disease_114070360 [isPartOf]
wordnet_symptom_114299637 [subClassOf] wordnet_heart_murmur_114334814
```

Also starting from the other endpoint we find:

```
wordnet_disease_114070360 -> wordnet_heart_murmur_114334814 [1002] 1:2
infl: 1.0 score: 0.5   wordnet_heart_murmur_114334814 [subClassOf]
wordnet_symptom_114299637 [isPartOf] wordnet_disease_114070360
```

Using this process paths are found between Canonic Entities of interest. Every time such a path is added an edge is created in the N-partite graph. In this example, the undirected edge between `wordnet_heart_murmur_114334814` and `wordnet_disease_114070360` will have a score of 0.55 (0.05 + 0.5). Only paths between Canonic Entities belonging to different String Entities (partitions) are added.

Next, the Linker Algorithm is run (described in section V.3.). After step 3 of the algorithm we have the following Result Sets (only the first 3 are shown):

```
RS1 : 1.25
      heart             [1002]: wordnet_heart_murmur_114334814   (33)
      disease           [1003]: wordnet_disease_114070360    (51)
      stroke            [1005]: ANY    (16)
      causes            [1009]: wordnet_probable_cause_105824514    (9)
      death             [1011]: ANY    (29)
      worldwide         [1012]: ANY    (0)
      World Health Organisation [1018]: ANY    (1)[World_Health_Organization]
      estimates         [1019]: ANY    (4)
      people            [1025]: ANY    (42)
RS2 : 1.1
      heart             [1002]: wordnet_heart_disease_114103288    (33)
      disease           [1003]: wordnet_cardiovascular_disease_114057371    (51)
      stroke            [1005]: ANY    (16)
      causes            [1009]: ANY    (9)
      death             [1011]: ANY    (29)
      worldwide         [1012]: ANY    (0)
      World Health Organisation [1018]: ANY    (1)[World_Health_Organization]
      estimates         [1019]: ANY    (4)
      people            [1025]: ANY    (42)
RS3 : 0.7333333333333333
      heart             [1002]: wordnet_heart_murmur_114334814    (33)
      disease           [1003]: wordnet_blood_disease_114189204    (51)
      stroke            [1005]: wordnet_ischemic_stroke_114166358    (16)
      causes            [1009]: ANY    (9)
      death             [1011]: ANY    (29)
      worldwide         [1012]: ANY    (0)
      World Health Organisation [1018]: ANY    (1)[World_Health_Organization]
      estimates         [1019]: ANY    (4)
      people            [1025]: ANY    (42)
…
```

We then run step 4 or the Linker Algorithm which is supposed to merge non-overlapping Result Sets. In this case, no improvements are found.

This output is the result of the GER system. It has the String Entities on the left side, their ID in square brackets and then the suggested Canonic Entity. Following the canonic entity is the number of Canonic Entities it had to choose from (how large the PCE_{SE} of each String Entity is). When there is just one possible Canonic Entity, like in the case of "World Health Organization", the corresponding entity is shown in square brackets but unless there is evidence to support it (connecting links in the graph) it is not selected by default, instead the system preferring to say it does not know.

The highest scoring Result Set of this example (score 1.25) can be analyzed as follows:

For String Entity "heart" it has missed the intended result `wordnet_heart_107651905`, instead choosing `wordnet_heart_murmur_114334814` because of the strong link to `wordnet_disease_114070360`. Interestingly, consulting the debugging log of the system, there is no path (of length equal or less than 3) from `wordnet_heart_107651905` to any other Canonic Entity. Because of the structure of the WordNet hypernym tree, in this scenario, there was no way for the system to discover to correct Canonic Entity.

For String Entity "disease" it has chosen the correct Canonic Entity. However this choice was made on the partially wrong path to `wordnet_heart_murmur_114334814`.

For the String Entities "stroke", "death", "estimates" and "people" the GER system did not find any information path so did not know what Canonic Entity to choose from.

For String Entity "worldwide" the system did not find any possible Canonic Entity that could represent it. This happens for words unknown to WordNet or YAGO, or if the cleaning step of the Canonic Entity assignation procedure cleans out all the Canonic Entities.

Last, for String Entity "World Health Organization" even though its PCE_{SE} only contains one entity `World_Health_Organization` (which is actually the correct one), because it does not find any information path, it prefers not to choose it.

Overall, in respect to the way we defined accuracy for the GER system, for this sentence the system would receive a score of 1/9 = 11% accuracy, for correctly matching only one of the 9 interesting String Entities.

Moving on to the second sentence:

Sentence 2: "The metabolic **syndrome**, associated with an increased **risk** of **type** 2 **diabetes** and cardiovascular **disease**, affects about one fifth of the **world**'s **adult population**."

Before discussing the results for the second sentence, it is interesting to note that because "metabolic" and "cardiovascular" are seen as adjectives ("JJ" part-of-speech tag) they are not included in the analysis even though for us, humans, they are relevant.

The same steps as for the first sentence are taken to create the *N*-partite graph based on the Operational Graph. The Linker Algorithm is run (here *N* = 8). We present two Result Sets after step 3 of the algorithm:

```
RS1 : 0.9833333333333334
      syndrome         [2002]: ANY    (15)
      risk             [2008]: ANY    (4)
      type             [2010]: ANY    (14)
      diabetes         [2012]: ANY    (5)
      disease          [2015]: ANY    (51)
      world            [2023]: wordnet_world_102472987    (16)
      adult            [2025]: wordnet_adult_109605289    (7)
      population       [2026]: wordnet_population_108179879    (9)
 …
RS4 : 0.65
      syndrome         [2002]: wordnet_syndrome_114304060    (15)
      risk             [2008]: ANY    (4)
      type             [2010]: ANY    (14)
      diabetes         [2012]: wordnet_diabetes_114117805    (5)
      disease          [2015]: wordnet_genetic_disease_114151139    (51)
      world            [2023]: ANY    (16)
      adult            [2025]: ANY    (7)
      population       [2026]: ANY    (9)
 …
```

After running step 4 (merging non-overlapping Result Sets) we find that merging the first *RS* and the forth *RS* is possible, as they are non-overlapping, summing the final score accordingly and producing the highest Result Set possible:

```
RS1 : 1.6333333333333333
      syndrome         [2002]: wordnet_syndrome_114304060    (15)
      risk             [2008]: ANY    (4)
      type             [2010]: ANY    (14)
      diabetes         [2012]: wordnet_diabetes_114117805    (5)
      disease          [2015]: wordnet_genetic_disease_114151139    (51)
      world            [2023]: wordnet_world_102472987    (16)
      adult            [2025]: wordnet_adult_109605289    (7)
      population       [2026]: wordnet_population_108179879    (9)
```

This Result Set can be interpreted as follows:

String Entity "syndrome" has chosen `wordnet_syndrome_114304060` correctly, identifying it as a pattern of symptoms indicative of some disease, the second most common sense of syndrome in WordNet.

String Entities "risk" and "type" were not identified.

String Entities "diabetes" was correctly identified as `wordnet_diabetes_114117805`. The length 2 information path to disease is the following:

```
wordnet_diabetes_114117805 -> wordnet_genetic_disease_114151139 [2015]
l:2 infl: 1.0 score: 0.5   wordnet_genetic_disease_114151139 [subClassOf]
```

```
wordnet_polygenic_disorder_114075199 [subClassOf]
wordnet_diabetes_114117805
```

String Entity “disease” was incorrectly identified as `wordnet_genetic_disease_114151139`. Even though `wordnet_genetic_disease_114151139` is actually a sub class of `wordnet_disease_114070360` (the expected correct choice), for this sentence specificity is not better because the sentence makes no reference to a genetic disease. The information path linking “syndrome” to “disease”:

```
wordnet_syndrome_114304060 -> wordnet_genetic_disease_114151139 [2015]
l:2 infl: 0.1 score: 0.05   wordnet_genetic_disease_114151139
[subClassOf]  wordnet_disease_114070360 [isPartOf]
wordnet_syndrome_114304060
```

String Entities “world”, “adult” and “population” are the correct choices for this example. Interestingly, out of the 16 possible Canonic Entities for “world”, `wordnet_world_102472987` was chosen.

```
wordnet_population_108179879 -> wordnet_world_102472987 [2023] l:2 infl:
1.0 score: 0.5   wordnet_world_102472987 [isMemberOf]
wordnet_people_107942152 [subClassOf] wordnet_population_108179879
```

This presents some level of ambiguity, because of the 8 senses WordNet has for “world” (included in the 16 possible Canonic Entities we assigned), the chosen sense was the second, referring to the world as a group of people. While maybe the first sense of “world” would have been better (world seen as everything that exists on earth), the second sense is also correct; arguably, world seen as the physical Earth globe would also be correct, captured in another Canonic Entity of the form `wordnet_world_#`. So in this scenario we have multiple correct answers.

The score for this Result Set is 5/8 = 62% correct, a quite good result considering the very large search space available.

Overall, for the example document we had 1 correctly identified Canonic Entity out of 9 for the first sentence/Process Group and 5 out of 8 for the second sentence/Process Group. The overall accuracy is thus 6(1+5) / 17(9+8) = 35%. The scores for this example were calculated using the strict method (first method presented in the evaluation section V.5 of the system), meaning we considered only the first Result Set provided by the system and any Canonic Entity that was not the expected Canonic Entity would be judged as a mismatch.