**WIRTSCHAFTSUNIVERSITÄT WIEN**

WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

EFMD EQUIS ACCREDITED

# Masterarbeit

**Titel der Masterarbeit:**
Generic Relation Extraction

**Verfasserin/Verfasser:**
Weber Gerald

**Matrikel-Nr.:**
0125536

**Studienrichtung:**
Wirtschaftsinformatik

**Beurteilerin/Beurteiler:**
PD Dipl.-Ing. Mag. Dr. Albert Weichselbraun

Ich versichere, dass:

ich die Masterarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe.

ich dieses Masterthema bisher weder im In- noch im Ausland (einer Beurteilerin/einem Beurteiler zur Begutachtung) in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

diese Arbeit mit der vom Begutachter/von der Begutachterin beurteilten Arbeit übereinstimmt.

Datum

Unterschrift

# Contents

# Abstract

Due to the vast extent of the Word Wide Web it becomes more and more complicated to find context based information. Because of the redundancy of information in the web, search engine queries result an unmanageable number of documents containing possible answers. Inspection of these results requires to much time and is not practicable in many cases. Extracting these relations from the documents in advance and storing them in a simplified manner would reduce the search effort tremendously. This thesis deals with the development of an extraction system capable of finding and storing domain independent relations from text.

# Zusammenfassung

Aufgrund der unglaublichen Ausmaße des Internets wird es immer schwieriger kontextbasierte Informationen aufzuspüren. Die Redundanz des Systems liefert für einfache Anfragen oft eine unüberschaubare Anzahl an Ergebnisdokumenten, deren Durchsicht auf der Suche nach der Antwort viel Zeit in Anspruch nimmt. Könnte man die Zusammenhänge extrahieren anstatt Dokumente als Ergebnis zu liefern, würde das den Suchaufwand verkleinern. In dieser Arbeit wird ein System zur Extraktion von Relationen beschrieben mit dem Ziel domänenunabhängige Zusammenhänge zwischen Objekten schnell aufzufinden und einheitlich abzuspeichern.



**Key Words**
Open information extraction, relation extraction, natural language processing, Stanford NLP

# 1 Introduction and Motivation

Since the invention of the World Wide Web (WWW), mankind attempt to harness the knowledge caught in billions of web sites. The Web started as a tool for military/scientific institutions and reached an expansion of 26 million pages in 1998. Since then the Internet has reached the billion mark in 2000 growing further to hit the one trillion unique URLs milestone in 2008 according to Google [1]. So, many people use search engines like Bing, Google, Yahoo, etc. to query the Web. Due to its astonishing size, most query terms occur in different contexts which have to be narrowed down by describing the search terms in more detail. Mostly, to put the terms into context, known relations of the query terms are used (e.g. working with, born at, living at for people search).

Therefore, systems that learn to extract relations between entities would be required to find entities and their different contexts. Search engines are not designed to behave like a natural language processing (NLP) system, which is a key component for such extraction tasks. While some mature solutions for text analysis exist, most of them are commercial like Thompson Reuters OpenCalais[1] with its CalaisViewer[2].

Nevertheless, some systems evolved in recent years, capable of extracting relationship between entities. Some of them are restricted to a domain whereas others are able to find relations in text from any domain. While none of the systems is freely available our motivation is to create a model of an Open Information Extraction (Open IE) system, that is capable of detecting named entities (NE) and relations between them from domain-independent text. Furthermore we want to implement a prototype and inspect its performance in terms of speed, recall and precision. While we want to reduce manual activities to a minimum, a graphical application helps the user to distinguish between relevant and irrelevant relations found by our system. This task is required to train a classifier, which finally identifies relevant relations automatically.

The reminder of this thesis is organized as follows. Chapter 2 (*Generic Relation Extraction Sytems*) provides a description of a selection of Open IE systems to get an understanding how different systems described in literature try to solve the task of Open IE. Chapter 3 (*Building our own Generic Relation Extraction System*) introduces a model for open domain relation extraction. The design of the system as well as experimental results are described there in detail. Finally, this thesis concludes with conclusions and outlook on potential future work in Chapter 5 (*Conclusions and Future Work*).

[1] `http://www.opencalais.com/`
[2] `http://viewer.opencalais.com/`

# 2 Generic Relation Extraction Systems

To understand how an Open IE system works we present systems which are either using state-of-the-art technology to accomplish the task or systems that were pioneers in this field of science. Due to the the usage of different corpora, it is not easily possible to compare the presented results of the following systems directly. Nevertheless, the study will provide a good insight into the problems that occur while trying to extraction relations from text.

## 2.1 Dual Iterative Pattern Relation Expansion (DIPRE)

To build an unprecedented source of information, Sergei Brin designed in 1998 the semi-automatic Web relation extraction system. As one of the pioneers in the field of relation extraction, the algorithm is explained by extracting relations between authors and title of books from the Web, starting with a small seed set of these pairs. The idea is to find (all) occurrences of (author, title) of the starting set in the web and learn the patterns describing these occurrences to find other (author, title) pairs which will result in a larger set of books. The new results lead to further patterns, that are used to start over and looks for more occurrences of the generated patterns. After some iterations, almost all books and patterns will be learned. The algorithm is illustrated in Figure 2.1.

This method is called DIPRE - Dual Iterative Pattern Relation Expansion *"which relies on a duality between patterns and relations" [2].*

### 2.1.1 The Problem

Apart from finding the books on the web and extracting the patterns, the major problem is *"to maximize the coverage and minimize the error rate" [2].* So we want to find as many as possible books to identify (<author, title> pairs) and reduce the number of false hits to a minimum. While *"a recall of 20% may be acceptable" [2]*, *"an error rate over 10% would likely be useless for many applications" [2].* The small recall is only practicable in

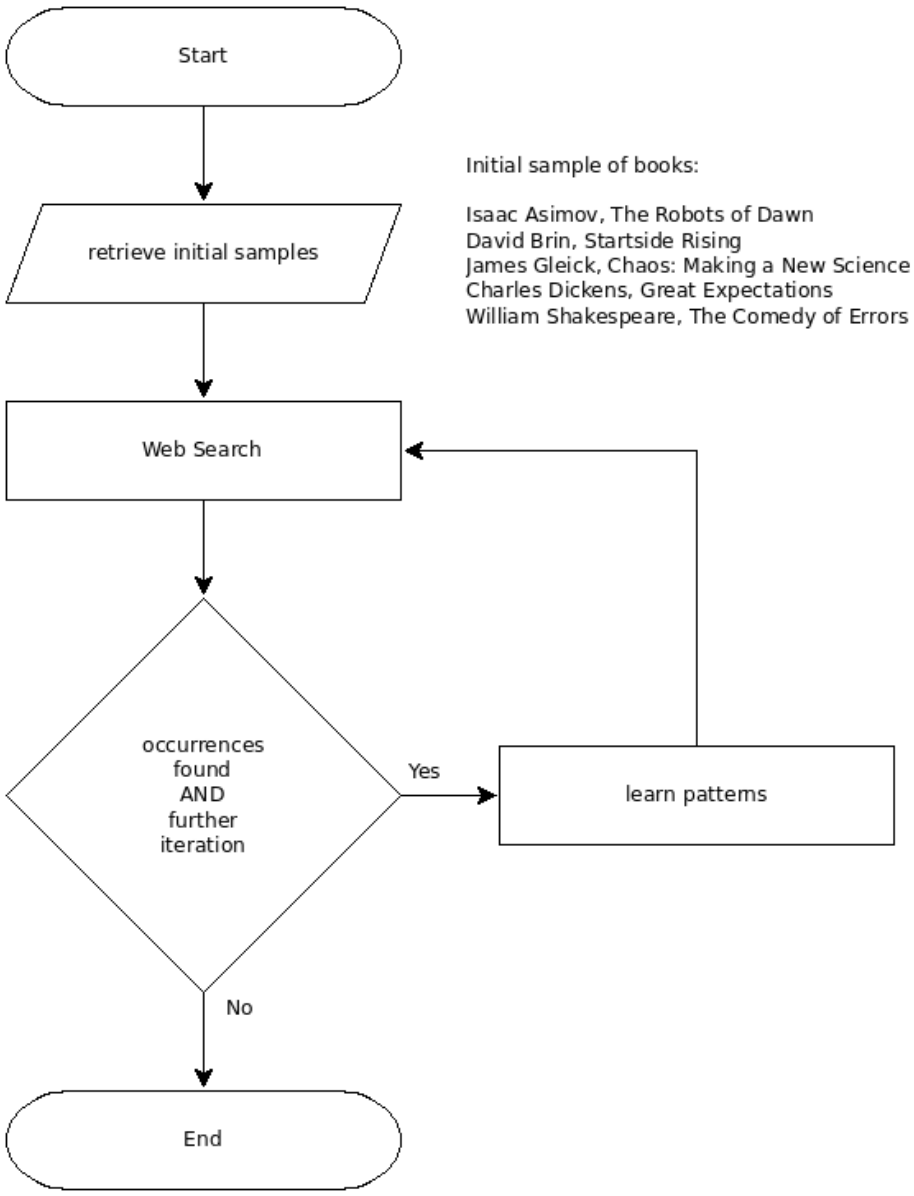


Figure 2.1: DIPRE flowchart

the case of a sufficiently large database (the Web) were redundant books can appear. So the message is, that it is not required to find every occurrences of a <author, title> pair on the Web (= low recall), but it is crucial to find only correct matches (= high precision).

The main problem is the generation of the patterns from the discovered <author, title> values. The pattern should be as concrete as possible to prevent matching of bogus tuples and general enough to match new values [2]. The suggested solution by Brin is called the Dual Iterative Pattern Relation Extraction (DIPRE) algorithm, which has the following skeleton:

1. Provide a small set of values of the desired relation (e.g. *William Shakespeare, The Comedy of Errors* for a *author, title* relation)
2. Start a Web search to find all appearances of the provided values. The context and the Uniform Resource Locator (URL) should be stored too.
3. Generate patterns to match the detected values
4. Extract a set of values from the stored occurrences that match the generated patterns
5. Start over with step 2 until the set is large enough

So while some bogus tuples can distract the pattern generation which would lead to more invalid tuples, the expansion of the patterns has to bypass a security mechanism. In this case, each pattern has to match multiple patterns to count as a valid pattern. Therefore the pattern generation is the main topic here which *"is a nontrivial problem and there is the entire field of pattern recognition devoted to solving the general version of this problem" [2].* To reject patterns which would cause detection of nonbooks, the specificity of every pattern is calculated. Patterns are kept in case that $specificity(pattern) * number\ of\ books\ found > threshold$.

### 2.1.2 Experiments

For experiments, 24 million web pages were available as part of Stanford WebBase. Starting with an initial set of five selected (author, title) pairs, 199 book occurrences were discovered and 3 patterns generated. Note, that no web pages from Amazon were involved. A run with these patterns over matching URL's produced 4,047 unique <author, title> pairs [2]. Searching these books on 5 million web pages discovered only 3,972 occurrences which was disappointing. Nonetheless, 105 patterns were generated (24 of them were bogus) for these books. The next iteration delivered 9,127 unique (<author, title>) pairs and 242 were faulty pairs - the author was *Conclusion* and was therefore removed. This was the only human interference in the whole process. The final iteration with the remaining 9,127 books on a subset consisting of 156,000 documents produced 9,938 occurrences and generated 346 patterns. Scanning over the same set of documents produced 15,257 unique books with very little bogus data [2].

### 2.1.3 Conclusion

As you can see, DIPRE is a remarkable tool, which started with a small set of five books and expanded it to a list of over 15,000 books. The difficulty with DIPRE is to prevent the algorithm from accepting too much patterns and drifting in a direction which has nothing in common with the initial task [2]. Sadly, the generation of the patterns is not part of Brin's paper.

## 2.2 TEXTRUNNER

TEXTRUNNER is a system operating on domain independent unstructured text [3]. Therefore, Banko et al. extract n-ary tuples from web crawls like other systems operating on raw text. TEXTRUNNER faces systems like DIPRE, SNOWBALL and KNOWITALL, but has additional advantages that makes it unique:

1. TEXTRUNNER operates in batch mode. So it creates a huge amount of data on a web crawl once whereas other systems start the web crawl on demand. This approach allows pre-computing of results giving the system the possibility to response faster to user queries [3]. Other systems start downloading and indexing of data only in response to a user query.
2. While TEXTRUNNER does not need a target relation or schema to start, it has the ability to acquire new relations as they occur in new articles [3]. Other systems that start with a given set of seed pairs are more or less caught in their schema. Nevertheless the distraction of a system with starting seeds is less probable.

The output of TEXTRUNNER has three unique characteristics in comparison to other text-centric extractors:

1. It returns binary relations instead of 3-ary or larger tuples. These binary relations usually consists of two nouns and a linking relation.
2. Due to the binary relation, the domain of text-derived data may differ from the domain found in relations because of the missing context.
3. In contrast to extraction systems like WEBTABLES [4], where every relation found is based on a single occurrence on the web, TEXTRUNNER extracts only relations which occur multiple times because the linguistic extractors of TEXTRUNNER are quite fallible.

### 2.2.1 Architecture

The architecture of TEXTRUNNER consists of three key modules:

1. Self-Supervised Learner: This learner can operate without hand-tagged data and outputs a classifier distinguishing between relevant and irrelevant tuples [5].
2. Single-Pass Extractor: This extractor generates candidate tuples for each sentence by processing the entire corpus in a single pass. The tuples labeled as relevant by the classifier are retained.
3. Redundancy-Based Assessor: Finally, each remaining tuple is assigned a possibility providing information about the correctness of extraction [5]. The probabilistic model of redundancy was introduced by Downey et al., 2005 [6].

The following subsections describe the modules in more detail.

#### 2.2.1.1 Self-Supervised Learner

The learner first start with an unsupervised learning of its training data. Then, it trains a Naive Bayes classifier with the labeled data which is used by the third module: The Extractor. While a parser would not be successful on Web-scale, the Learner uses one to identify a set of trustworthy extractions which are used as training examples for a Naive Bayes classifier [5].

#### 2.2.1.2 Single-Pass Extractor

With a single pass over the corpus, each word and each sentence will be tagged with its most probable part-of-speech. Using this information and a lightweight noun phrase chunker the extractor can identify entities. Relations are found by examining the text between the entities. Each noun is annotated with a probability to express the certainty of a noun being part of an entity. Tuples with lower probable entities are likely to be discarded by the extractor. Finally, for each candidate tuple the classifier stores correct tuples and rejects erroneous ones [5].

#### 2.2.1.3 Redundancy-Based Assessor

Inside the extraction sequence, TEXTRUNNER normalizes the relation by omitting non-essential modifiers from text (e.g. 'was originally developed by' -> 'was developed'). After extraction, TEXTRUNNER merges identical relations and counts the number of distinct sentences from which each extraction was extracted. In this way, the model can estimate the correctness of a tuple depending on the number of its occurrences.

### 2.2.2 Experimental Results

To measure the efficiency of TEXTRUNNER it is compared to a traditional IE system. In this case KNOWITALL is used to extract facts from a nine million Web page corpus.

While this system needs a set of start seeds, ten relations were selected in advance to start the process. Over this test set of nine million pages, the system's error rate and number of correct extractions are (see Table 2.2).

| | Average Error rate | Correct Extractions |
|---|---|---|
| TEXTRUNNER | 12% | 11,476 |
| KNOWITALL | 18% | 11,631 |

Figure 2.2: Error rate comparison: 33% lower than with KNOWITALL [5]

### 2.2.3 Conclusion

Even though the Open IE system TEXTRUNNER does not use Named Entity Recognition (NER) and syntactic or dependency parsers to detect relations, it reaches the precision of state-of-the-art systems like KNOWITALL by using a self-supervised learner.

## 2.3 Open Information Extraction System using Conditional Random Fields (O-CRF)

This paper written by Michele Banko and Oren Etzioni in 2008 introduces a new relation-independent extraction paradigm called *Open Information Extraction*. This paradigm is designed for massive, heterogeneous corpora like the Web [7]. They further describe a model called O-CRF with higher precision and recall than TEXTRUNNER, which is the predecessor of O-CRF.

For an unsupervised way of learning, Klein and Manning *[8]* showed, *"that unlexicalized parsers are more accurate than previously believed, and can be learned in an unsupervised manner" [7]*. Therefore, Banko et al. replaced the Naive Bayes Classifier used to decide whether a relation is trustworthy or not by an algorithm using Conditional Random Fields (CRF).

### 2.3.1 The Nature of Relations in English

To build an relation independent retrieval system, we have to understand how relations in the English language are expressed. For this task, the authors extracted relation-independent lexico-syntactic patterns and counted their occurrence [7]. By doing so, they found out, that 95% of all relations could be covered by one of the following eight categories of patterns (Table 2.3):

| Relative Frequency | Category | Simplified Lexico-Syntactic Pattern |
|---|---|---|
| 37.8 | Verb | $E_1$ Verb $E_2$<br>X *established* Y |
| 22.8 | Noun+Prep | $E_1$ NP Prep $E_2$<br>X *settlement with* Y |
| 16.0 | Verb+Prep | $E_1$ Verb Prep $E_2$<br>X *moved to* Y |
| 9.4 | Infinitive | $E_1$ to Verb $E_2$<br>X *plans to acquire* Y |
| 5.2 | Modifier | $E_1$ Verb $E_2$ Noun<br>X *is* Y *winner* |
| 1.8 | Coordinate$_n$ | $E_1$ (and\|,\|-\|:) $E_2$ NP<br>X-Y *deal* |
| 108 | Coordinate$_v$ | $E_1$ (and\|,) $E_2$ Verb<br>X, Y *merge* |
| 0.8 | Appositive | $E_1$ NP (:\|,)? $E_2$<br>X *hometown* Y |

Figure 2.3: *"Taxonomy of Binary Relationships: Nearly 95% of 500 randomly selected sentences belongs to one of the eight categories above" [7]*

### 2.3.2 Relation Extraction

In contrast to an traditional relation extraction system, where the relation is an input to the extraction task and therefore known in advance, an open information extraction system must find out the entities participating in a relation as well as the copulas connecting the entities [7]. Furthermore the Open IE process does neither know which type the participating entities are nor that these are named entities (e.g. in an relation called HEADQUARTERS(x, y) where one entity could be a location and the other one a company name). These problems have led the authors to developing a method using Conditional Random Fields (Lafferty et al., 2001) which *"are undirected graphical models trained to maximize the conditional probability of a finite set of labels Y given a set of input observations X" [7].*

The goal is to use CRF for relation extraction as it is used for other tasks in NLP like word segmentation, NER, POS tagging. Recently, CRF was also used for relation extraction (see Culotta et al.,2006).

### 2.3.3 Training

Training the CRF classifier is self-supervised. By applying *"relation-independent heuristics to the Penn Treebank it obtains a set of labeled examples in the form of relational tuples" [7].* These tuples are used to train a CRF model, capable of identifying tokens indicating relations between entities. Nevertheless, there are some limitations known for the O-CRF system like

1. O-CRF can only extract explicitly mentioned relations from text
2. relation extraction is primarily word-based instead of using additional features like punctuation

### 2.3.4 Extraction & Results

While this system should operate on Web-scale, entity recognition is accomplished in a single pass like TEXTRUNNER does. Subsequent to the recognition, O-CRF applies an algorithm called RESOLVER (see Yates and Etzioni, 2007) to find relation synonyms.

To compare O-CRF and O-NB (used in TEXTRUNNER) both algorithms had to identify relations within a set of 500 sentences. As you can see from Table 2.4 O-CRF nearly doubles recall and slightly elevates precision in comparison to O-NB.

| | O-CRF | | | O-NB | | |
|---|---|---|---|---|---|---|
| Category | P | R | F1 | P | R | F1 |
| Verb | 93.9 | 65.1 | 76.9 | 100 | 38.6 | 55.7 |
| Noun+Prep | 89.1 | 36.0 | 51.3 | 100 | 9.7 | 55.7 |
| Verb+Prep | 95.2 | 50.0 | 65.5 | 95.2 | 25.3 | 40.0 |
| Infinitive | 95.7 | 46.8 | 62.9 | 100 | 25.5 | 40.6 |
| Other | 0 | 0 | 0 | 0 | 0 | 0 |
| **All** | **88.3** | **45.2** | **59.8** | 86.6 | 23.2 | 36.6 |

Figure 2.4: Open Extraction by Relation Category: O-CRF has increased recall and precision compared to O-NB [7]

In a further experiment, O-CRF was compared to a traditional relation extraction system (referred as R1-CRF), which is trained with manually tagged training data that has a higher quality than O-CRFs self-supervised training data, but is more expensive to generate [7]. This is a comparison between the self-supervised, unlexicalized O-CRF and a supervised, lexicalized extraction system. Furthermore, the extraction task was reduced to four different relations - birthplaces, award winners, corporate acquisitions and inventors. While R1-CRF was trained separately on each of the four relations, O-CRF ran all tests without specialized training. Table 2.5 shows the results of this task.

| | O-CRF | | R1-CRF | | |
|---|---|---|---|---|---|
| Relation | P | R | P | R | Train Ex1 |
| Acquisition | 75.6 | 19.5 | 67.6 | 69.2 | 3042 |
| Birthplace | 90.6 | 31.1 | 92.3 | 64.4 | 1853 |
| InventorOf | 88.0 | 17.5 | 81.3 | 50.8 | 682 |
| WonAward | 62.5 | 15.3 | 73.6 | 52.8 | 354 |
| **All** | **75.0** | 18.4 | 73.9 | **58.4** | 5930 |

Figure 2.5: Comparison of precision and recall between O-CRF and a traditional relation extraction system R1-CRF [7]

Finally, Table 2.5 shows, that a large number of hand-labeled training data is required to achieve the precision of O-CRF. The low recall of O-CRF (18.4%) can partly be concerned to an lack of lexicalized features as well as POS errors. Furthermore O-CRF is not able to discover most of the synonyms in comparison to R1-CRF.

### 2.3.5 Conclusion

While O-CRF does disclaim NER to find possible relations in text, it reaches remarkable results by learning a model from relation-independent, lexico-syntactic patterns. In summary, O-CRF is the preferred tool at the expense of recall, when [7]

- kind of relations are unknown and
- number of relations is vast.

Otherwise, if a higher recall is required, traditional relation extraction would be advisable.

## 2.4 SNOWBALL and STATSNOWBALL

The Department of Computer Science of the Columbia University in New York developed a relation extraction system which is capable to work on large plain-text collections. This system address the problem of reducing the manual work required to train traditional extraction systems for new domains. One of the papers [9] describing SNOWBALL mention the creation of powerful graphical user interfaces for training a system which would still require a domain expert to do so. Another suggestion would be the usage of a large manually tagged corpus for training operations. Manually annotating a large corpus in turn involves a large amount of labour too. Closest to SNOWBALL are systems that use unlabeled corpora for training purposes and use bootstrapping for the extraction process. Bootstrapping is an approach using a training set to prepare a classifier for its real extraction task.

### 2.4.1 SNOWBALL

The main components of the SNOWBALL system are shown in Figure 2.6:

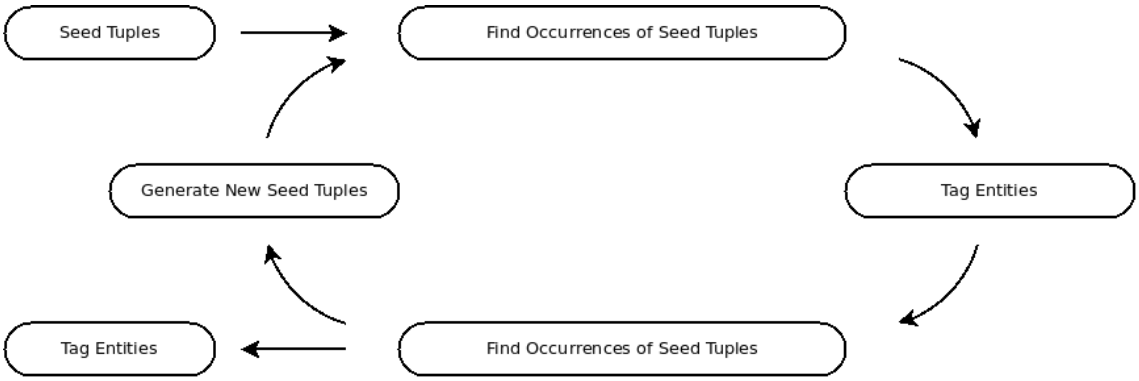


Figure 2.6: SNOWBALL's main components [9]

SNOWBALL starts an extraction process with some example tuples. In this case, we only consider organization-location tuples <$o$,$\ell$ >. In comparison to Section 2.1, one such tuple could be <*Microsoft, Redmond*>. While DIPRE (Section 2.1) does not use any kind of NLP like POS or NER, the first improvement of SNOWBALL *"is that* SNOWBALL*'s patterns include named-entity tags" [9].* Therefore, a SNOWBALL pattern would not simply connect any strings like *Microsoft* and *Redmond* together, but annotates them with ORGANIZATION respectively LOCATION annotation. So, the example patterns generated by SNOWBALL would look like in Table 2.7:

| |
|---|
| <ORGANIZATION>*'s headquarters in* <LOCATION> |
| <LOCATION>*-based* <ORGANIZATION> |
| <ORGANIZATION>, <LOCATION> |

Figure 2.7: Example of generated tags by SNOWBALL [9]

For the task of relation extraction, the system creates patterns, which will be compared to relation candidates. A pattern in the SNOWBALL system *"is a 5-tuple <left, tag1, middle, tag2, right>, where tag1 and tag2 are named-entity tags, and left, middle and right are vectors associating weights with terms" [9].* For example, the SNOWBALL pattern *<{<the, 0.2>}, LOCATION, {<-, 0.5>,<based, 0.5>}, ORGANIZATION, {}>* would match *"the Irving-based Exxon Corporation," [9]*, with the preceding article *the* (left context) followed by the location *Irving*. The middle context is assembled by *"-"* and *"based"* whereas the end of the matching string is represented by an organization *Exxon Corporation*. Marginal variations of the matching phrase would match too, but to a smaller degree. To calculate a match between a pattern and a candidate *S* (has to match the location and organization tags), SNOWBALL first constructs *"three weight vectors* $l_S$*,* $r_S$ *and* $m_S$ *from S by analyzing the left, right, and middle contexts around the*

*named entities" [9].* The weight of each term in the vectors are based on the frequency of their occurrence in the corresponding context. The weights are normalized and scaled to represent their relative importance. *"After extracting the 5-tuple representation of string S,* SNOWBALL *matches it against the 5-tuple pattern by taking the inner product of the corresponding left, middle and right vectors" [9].*

Therefore SNOWBALL generates 5-tuples for all seed tuples. The resulting 5-tuples are clustered together by the similarity calculated by a matching function. If the candidate pattern excess an defined minimum similarity threshold, it will be added to the extraction patterns. So, the improvement over DIPRE is the pattern and tuple evaluation which forms the key part of the SNOWBALL system [9].

### 2.4.2 STATSNOWBALL

In contrast to the related SNOWBALL system, **STATSNOWBALL** uses a statistical extraction model (Figure 2.8):

The first part $P_1$ represents the input. Like in SNOWBALL, the system can deal with seeds that contain no relation specific keywords $(e1, e2, ?)$ and with seeds that contain such keywords $(e1, e2, key)$. Therefore, an initial model can be provided. In the second part $P_2$ which is called statistical extraction model, the input seeds and the optionally provided input model are used to learn the extractor. *"We apply the $\ell_2$-norm regularized maximum likelihood estimation (MLE) at this step" [10].* The learned model is then used to extract new relation tuples. In the third step of $P_2$ (generated patterns) new extraction patterns are generated and used to improve the model. The systems iteratively runs through these four steps until no new tuples are found and no additional extraction patterns are generated. The third and final part of the system $P_3$ is only required for Open IE to present the output. Therefore it is treated as an optional post-processing step.

The main advantages of STATSNOWBALL over SNOWBALL in summary are:

- STATSNOWBALL is capable of traditional relation extraction as well as Open IE
- improvements in the evaluation and selection of new extraction patterns
- the weights of newly generated patterns are automatically calculated by Markov Logic Networks (MLN)
- easily extensible

### 2.4.3 Conclusion

Based on the improvements over SNOWBALL, a relation search-engine called *Renlifang* has been established on the Chinese Search market. The English counterpart is called *Entitycube* and is accessible via `http://entitycube.research.microsoft.com/`.

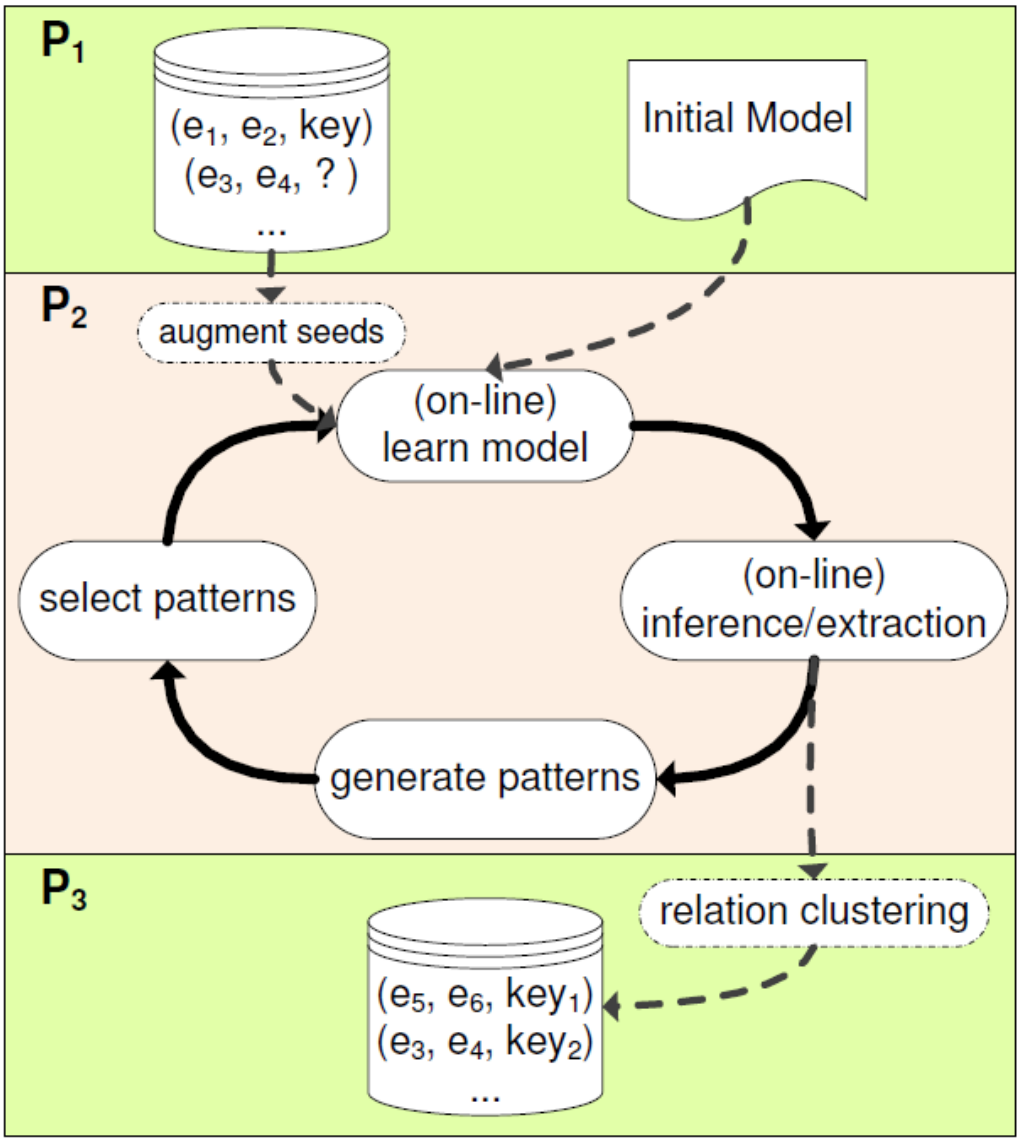


Figure 2.8: StatSnowBall's main components [10]

## 2.5 KnowItNow

Some NLP systems rely on Web search engines like Google to support their computations tasks. While these search engines are limiting the traffic, this task can become very slow with a high number of queries. So Google introduced an API to handle programmatic queries and limits the number they can issue. KNOWITALL is one system using an search engine to optimize its results. Whereas the first step, the generation of candidate facts by using domain-independent extraction patterns runs offline, the plausibility of the candidate facts is calculate by the use of hit-counts from search engine results. The measure is called *point-wise mutual information* (PMI) statistics which requires a number of queries for each candidate. With a raising number of candidates the time required for an experiment will go off. To overcome these problems of Web-based IE systems, the authors developed a search engine called Bindings Engine (BE) which finds bindings by providing variabilized queries [11]. Instead of querying the web for each task, the BE will return a list of occurrences to the question: *ProperNoun(Head(<NounPhrase>))*.

### 2.5.1 The Bindings Engine (BE)

Traditional Web-based IE systems would query a search engine to find an answer to the question *cities such as* by

1. fetching all documents returning to the query *cities such as*
2. searching in all documents for the query text and estimating if the noun phrase following *cities such as* matches the correct linguistic type
3. returning the string in case the type matches

While the querying a search engine is relatively fast, retrieving all documents of the result set is slow. There is a difference between a Web search engine and a private search engine which operates locally, but at scale this will not solve the problem. Therefor, the BE is designed to prevent the third step by dealing with queries containing

- typed variables (e.g. NounPhrase)
- string-processing functions

as well as standard query terms [11]. To achieve high speed at search time, the BE builds a so called *neighborhood index* containing the context of the typed variables to prevent fetching each document. Finally, each row in the index is connected to a document containing the string. While the search will be pushed by using the inverted index, a drawback could be the high disk space consumption. The decision between the BE and a privately operated search engine like Nutch[1] ended in favor of BE. The crucial speed comparison experiment with a 60 million Web pages covering corpus ended with the fact, that BE is 134 times faster than the Nutch index with at same configuration.

[1] `http://lucene.apache.org/nutch`

### 2.5.2 Experimental Results

While KNOWITNOW is limited by its index, KNOWITALL is limited by the search engines quota. Under these restrictions, four relations were extracted from both systems: Unary relations like *Country, Corporation* and binary relations like *CeoOf(Corp, Ceo), CaptialOf(Country, City)*. While both systems use the OpenNLP POS tagger, KNOWITALL applies it to documents returned by Google queries whereas KNOWITNOW applies it to already crawled web sites. For the unary relations KNOWITNOW discovers 70-80% of the instances found by KNOWITALL at a precision of 0.9 . This result is accomplished while this relations only have a limited number of correct instances. In contrast to the binary relations, where this number can be enormous (e.g. *CeoOf* because of an high number of corporations and CEO's as well as the low frequency of these instances. To achieve a comparable recall for the binary relations with KNOWITNOW, an experiment showed an approach with a corpus of 400 million pages. KNOWITALL needs only 300,000 Google queries to achieve the same recall at a precision of 0.9 . Nevertheless, KNOWITALL requires more than three days to fire these queries whereas KNOWITNOW finishes the task in several minutes.

### 2.5.3 Conclusion

So, given a large corpus of 400 million pages, KNOWITNOW can achieve a comparable recall as KNOWITALL at the same precision. while KNOWITNOW can operate on smaller corpora too at the cost of reduced precision/recall. Nevertheless, it has an impressive extraction rate by the use of a Bindings Engine.

# 3 Method - A Generic Relation Extraction System

## 3.1 Introduction

Based on the insights gained from other Open IE systems described in Section 2, we tried to implement a prototype for open relation extraction. Open in a way that it should reveal new relations and not be limited to a pre-defined set of relation types. The simplest way to start would be to imitate DIPRE (Section 2.1) with its <*author, book*> relation or Snowball (Section 2.4) which starts with <*organization, location*> tuples to find patterns that describe relations between supplied entities.

However, our prototype will stick to binary relations between entities and can learn how relevant relations look like. A graphical tool to inspect the revealed relations should be developed too. For testing purposes we use a Thompson Reuters corpus containing over 800.000 news articles starting from 20-08-1996 till 19-08-1997 with an average length of 1400 characters, varying from 1 to 54740. This corpus is not annotated and relations are not known in advance. Each article is assigned to one or more industries, regions and topics.

The remainder of this section describes the development of a prototype starting with the design (Section 3.2) which contains a detailed description of all required parts. Before we run the prototype, the test setup (Section 4.1) will be described. Finally, the evaluation and results will be presented in Section 4 which concludes this chapter.

## 3.2 Design

This section describes the design of the prototype used to extract relations from open domain documents.

The first part of the model in Figure 3.1 is called *NLP Package* (Section 3.2.1) and summarizes sparse text processing components used to split the articles into sentences (*Sentence Splitter*) and further to words (*Word Tokenizer*), generate the base form (*Lemma Generator*), assign part-of-speech (POS) tags (*POS Tagger*) and recognize the entity type (*Named Entity Recognizer - NER*) for each chunk of the text.

The articles selected from the corpus form the input of the *NLP Package*. The output of this component is the article annotated with POS and NE tags which builds the input for the *Relation Extractor* (Section 3.2.2). Its first component is the *NE Extractor* which is responsible for the determination of candidate sentences, the discovery of entity types and their handling in the *Pattern Extractor* which extracts and stores POS patterns representing a relation between two entities. In such cases, the corresponding tuple will be extracted by the *Tuple Extractor* and stored in the database (Section 3.2.3). The *Pattern Filter* (Section 3.2.4) is subsequent to the extraction and allows an user to inspect and disable bad patterns manually by using the *Pattern Inspector* (Section 3.2.4) (manual filtering) or by using the *Pattern Classifier* (Section 3.2.5) to automatically disable bad patterns (automatic filtering). Though, the task of manual pattern selection is only required once, to learn the classifier for automatic pattern discrimination.

The following sections describe the design parts and their components in more detail.

### 3.2.1 Natural Language Processing (NLP) Package

For the implementation of the prototype, we have chosen **Stanford CoreNLP** because it provides good documentation and clear API design. Furthermore Stanford implements Hidden Markov Models, Maximum Entropy and Conditional Random Fields for NLP processes which are state-of-the art algorithms for NLP. The package contains the following required components for our prototype:

- **Sentence splitter** divides each article into an array of sentences
- **Word Tokenizer** breaks up text into individual objects. The used PTBTokenizer *"Fast, rule-based tokenizer implementation, initially written to conform to the Penn Treebank tokenization conventions, but now providing a range of tokenization options over a broader space of Unicode text" [12].*
- **Lemma Generator** *"computes the base form of English words, by removing just inflections (not derivational morphology)" [12].* Based on a finite-state transducer it only works on noun plurals, pronoun case and verb endings. It does not support comparative adjectives or derived nominals.
- **Part-Of-Speech (POS) Tagger** assigns parts of speech to each token of a text. The Stanford POS tagger is an implementation of a log-linear part-of-speech tagger [13] [14] and uses the Penn Treebank [15] tag set for English to annotate tokens. Additionally to the three included English language taggers (trained on different corpora), other models for Arabic, Chinese and German are available too. The fastest supplied model *left3words-distsim-wsj-0-18.tagger* achieves an accuracy of 96.92% on Penn Treebank WSJ sections 22-24, however the *bidirectional-distsim-wsj-0-18.tagger* model improves to 97.32% but is very slow. So for our prototype, we would go with the *left3words-distsim-sj-0-18 model*. Nevertheless, with enough

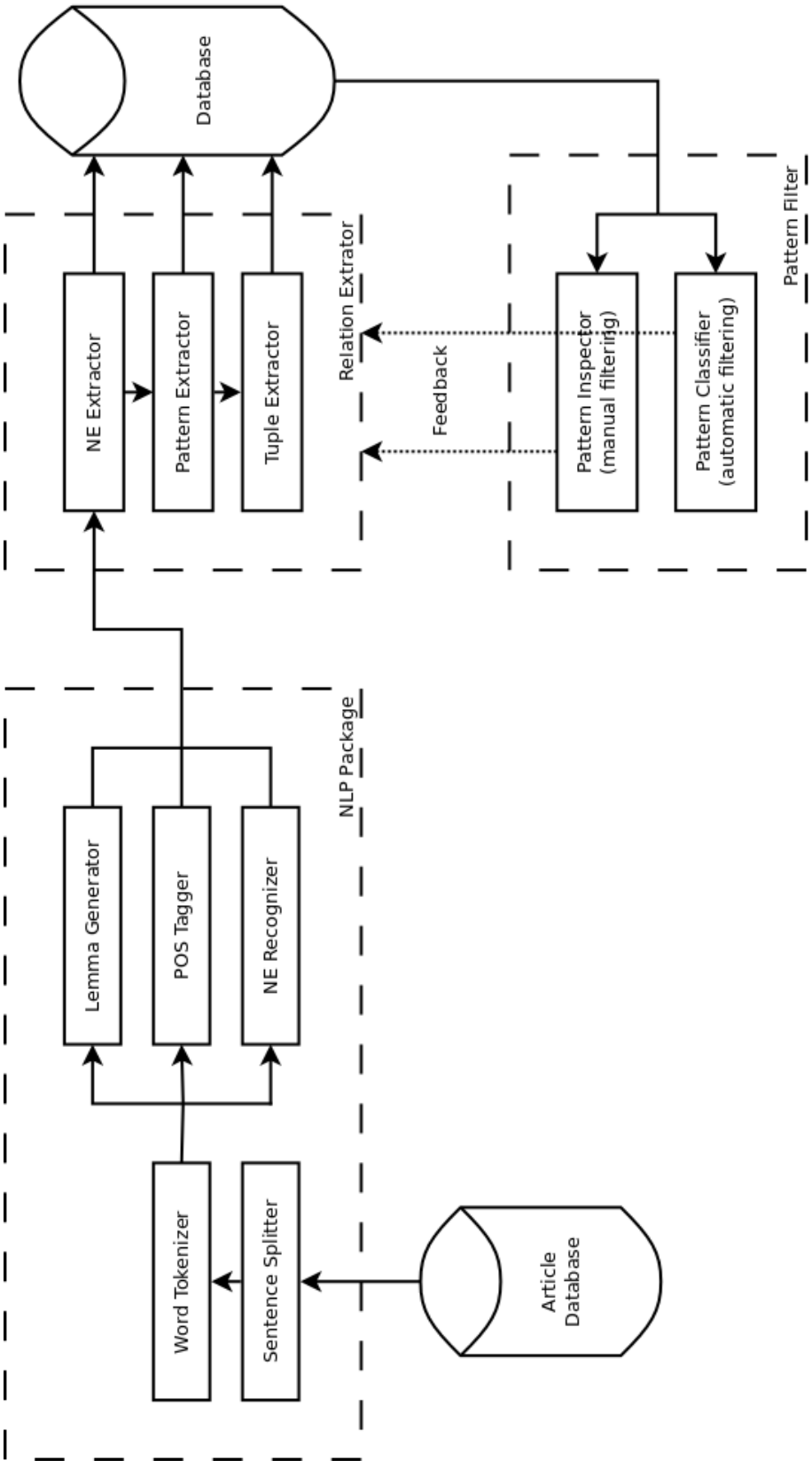


Figure 3.1: Model of our relation extraction system

training data, it is possible to train our own model based on the Maximum Entropy tagger.

- **Named Entity Recognizer (NER)** [16] *"provides a general (arbitrary order) implementation of linear chain Conditional Random Field (CRF) sequence models, coupled with well-engineered feature extractors" [12].* The default NER extractor in the Stanford package uses a combination of three models trained on a mixture of CoNLL[1], MUC-6[2], MUC-7 and ACE[3] named entity corpora for English to detect the named entities PERSON, ORGANIZATION, LOCATION and MISC as well as numerical entities DATE, TIME, MONEY and NUMBER. All unknown entities are labeled as OTHER. With this combination of training sets, the developers created models which should be very robust across domains. Therefore, we do not limit our test to any specific domain, but are only interested in relations containing the named entities PERSON, ORGANIZATION and LOCATION.

  Additionally, the Stanford package contains a so called *REGEXNER* which stands for Regular Expression NER. This component puts the user in the position to supply a pattern and a entity class, which will override the entity recognition. Providing the pair *(Mubarak, PERSON)* to the component would override any wrong recognition (as happened during our tests) like *(Mubarak, ORGANIZATION)* by the Regular Expression NER without learning a new model which would require a lot of human work for tagging a training a corpus manually. Even though we do not take advantage of them to reduce the human interaction to a minimum, the developers provided additional patterns to recognize ideologies (IDEOLOGY), nationalities (NATIONALITY), religions (RELIGION) and titles (TITLE).

- Finally, a **Syntactic Parser** and **Co-reference Resolution** is available too. However, work by Taghunathan et al. [17] contains a detailed description of the implementation of co-reference resolution. While a co-reference resolution system would improve our recall (e.g. it would identify pronouns that could be replaced by entities and form a new relation <entity>...<entity>), it turned out, that the implementation is not as stable as we would expect it. While we provided 6 GB memory to the JVM[4], we got *OutOfMemoryException*s on a regularly base. So we decided to disable this feature.

The usage of these components is explained in Section 6.2.2.

[1] Conference on Computational Natural Language Learning
[2] Message Understanding Conference
[3] Automatic Content Extraction Program
[4] Java Virtual Machine

### 3.2.2 Relation Extractor

While the NLP components are ready to use, the Relation Extractor connects them to process the input step-by-step. The first component is the *NE Extractor* (see Figure 3.1). It is responsible for the extraction and storage of all detected entities found by the NLP Package. The entities will be stored in the format <*key, value*> where key contains one of <LOCATION, ORGANIZATION, PERSON> and value the concrete word in the article tagged as entity. The entities would not be required for our pattern extractor, but we will compare them with the entities found by OpenCalais. While we are interested in relations between entities of type LOCATION, ORGANIZATION or PERSON, sentences containing two or more entities are passed on to the *Pattern Extractor* which extracts the words between and around the entity pair. If the number of words between the entities is smaller than or equal a defined threshold (default is six words) we assume a relationship and extract a pattern in the following form: *{*<*left*> <*NE1*> <*middle*> <*NE2*> <*right*>*}* (see [2] [9]). The size of <*left*>, <*middle*> and <*right*> parts can be configured and contain the POS tags of the words represented in the article whereas <*NE1*> and <*NE2*> contains the type of the corresponding entity (see example of a pattern below).

We suppose, that <*NE1*> <*middle*> <*NE2*> forms the relation whereas the <*left*> and <*right*> parts provide the context of the relation. The context is stored with the relation to help the *Relation Filter* to identify bad patterns.

The *Tuple Extractor* uses the same layout *{*<*left*> <*NE1*> <*middle*> <*NE2*> <*right*>*}* but stores the concrete values from the article text into the *Pattern & Tuple Database*. Each tuple is connected with the corresponding pattern and the article containing the relation.

This is an example sentence from an Reuters article:

> *"He told a gold seminar in Tokyo that gold could rise if President Bill Clinton wins the U.S. presidential election in November and then raises U.S. interest rates – something that would push down the U.S. stock market."*

A selected pattern looks like:

> WDT NN MD VB IN NNP **PERSON** VBZ DT **LOCATION** JJ NN IN NNP CC RB

The corresponding, stored tuple is:

> that gold could rise if President **Bill Clinton** wins the **U.S.** presidential election in November and then

### 3.2.3 Database Design

Before implementing the *Relation Extractor* (Figure 3.1) which is tied to the database we developed a data-model for persisting the entities, patterns and tuples. Therefor additional tables were required (Figure 3.2) to store the information. The additional tables called *NE* and *CALAIS* are required for evaluation purposes only. The tables *TUPLE* and *PATTERN* contain the extraction results of the Tuple/Pattern Extractor.

As Figure 3.2 shows, each article can have any number of extracted tuples identified by the primary key *tuple_id*. The column *article_id* establishes the relation between article and tuple whereas *pattern_id* connects each pattern with any number of tuples. The columns *left, ne1, middle, ne2, right* contained in the tables pattern and tuple are explained in Section 3.2. Finally, the *iteration* column contains a number which is incremented at each start of the program, while *enabled* stores the status of a pattern: *true* for good patterns, <*false*> for bad ones. The column *modifier* stores the modifying user: <*m*> for manual change by using *Pattern Inspector* (Section 3.2.4) or <*a*> for automatic change by *Pattern Classifier* (Section 3.2.5).

Section 6.2.1 explains the steps required to establish the data-model in a PostgreSQL database system in detail.

### 3.2.4 Pattern Inspector

After extraction of entities and relations, a tool for further investigation of the results (Figure 3.1) was developed. It allows filtering and sorting of patterns, tuples and articles and presents a good overview to identify incorrect relational patterns (bad patterns). These patterns can be disabled by simply unchecking a check-box next to the malicious patterns. The remaining patterns build the input for the *Pattern Classifier* (Section 3.2.5). To recover from bad patterns, manual control of the generated patterns is necessary especially in Open IE systems. Because we are not limited to a domain, some periodic occurring articles (e.g. sport results, stock rate tickers, etc.) will push specific bad patterns. Furthermore, the manual pattern filtering GUI[5] should be able to process a lot of tuples and provide a good overview for finding and selecting bad patterns (based on the contributions of [18]). Therefor we implemented a GUI to accomplish this steps. It is called *Pattern Inspector* and is entirely written in JAVA (Figure 3.3).

The area on the left side contains the filter options and a table to display the found patterns. Both filter options will be passed to the select statement triggered with the *Search* button. Resulting rows can be sorted by clicking on the table column headers. The section in the middle lists all tuples. Filtering is carried out on the client by regular expressions (Java RowFilter) to improve filter speed by reducing network traffic (in case your database is not local). In contrast the radio-buttons act on database side. If the radio-button *enabled*

[5] Graphical User Interface

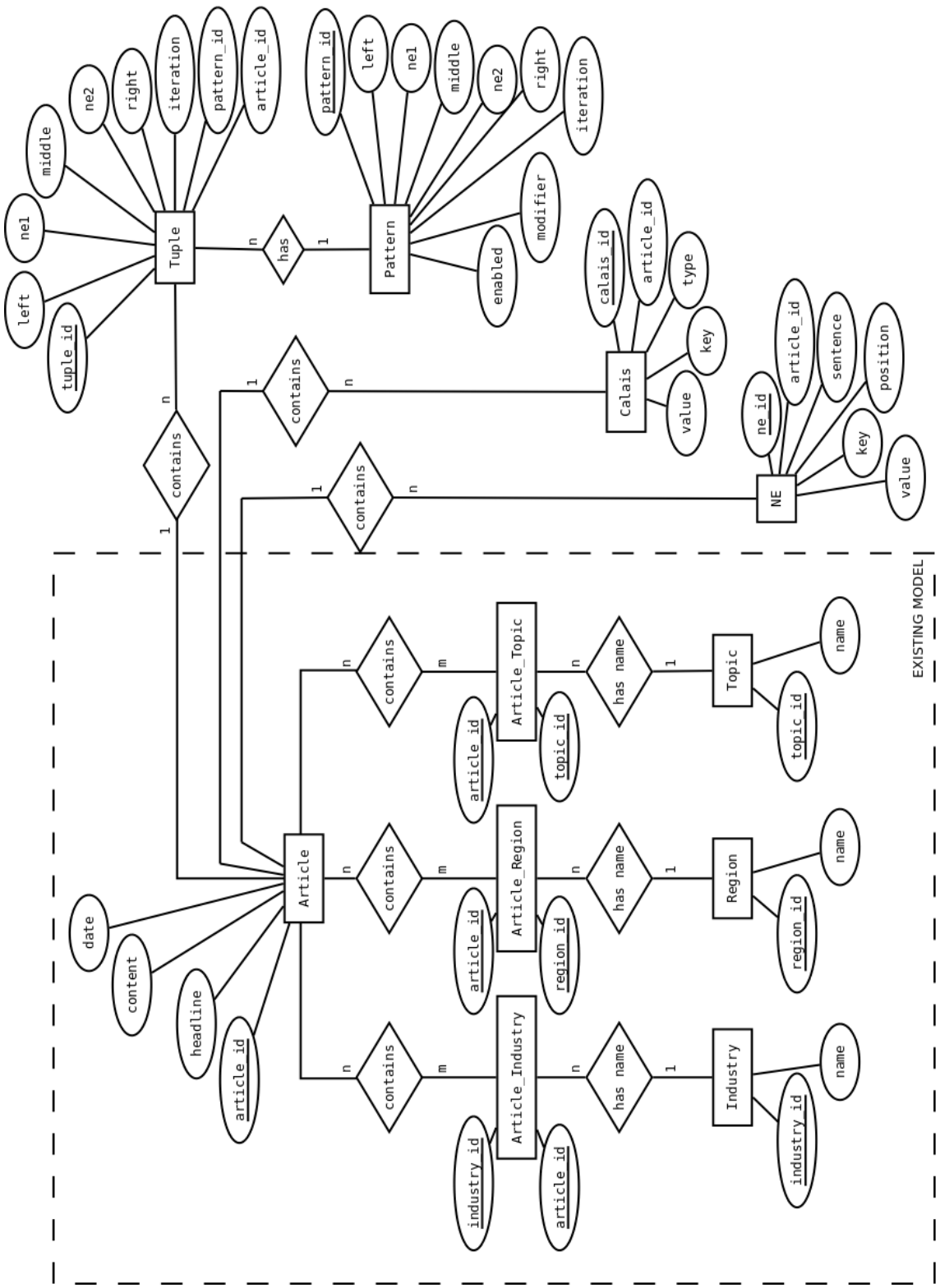


Figure 3.2: ER diagram for the prototype

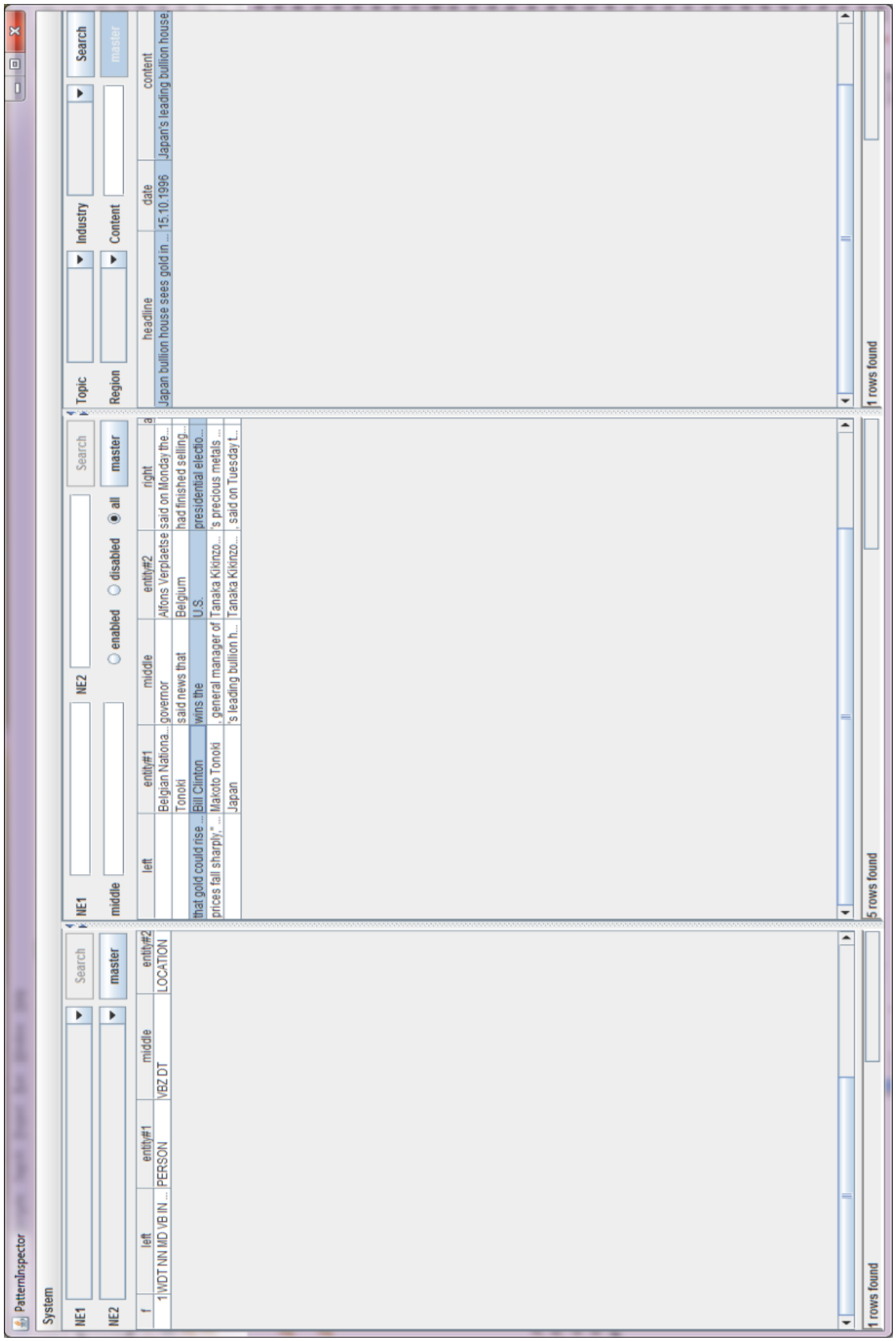


Figure 3.3: Java GUI of the Pattern Inspector

is selected, all tuples with an enabled pattern will be queried. Finally, articles will be displayed on the right side. Industry, region and topic are filtered at the database, the content can be filtered at the client. Furthermore, each area includes a *master* toggle-button. If this button is selected, all other areas depend on the master area. If pattern is defined as master area, selecting a row in the table would trigger the tuple search with the selected patterns and present the results in the middle section. Efficiently finding concrete tuples to the patterns is therefore facilitated. The other way round, if the tuple area is assigned to be master, a list selection triggers a pattern search. The same principle applies to the article area. Although the content of an article does not fit in the table, a dialog will display the currently selected article after double clicking on it.

Implementation details are provided in Section 6.2.3.

### 3.2.5 Pattern Classifier

The pattern classifier is a linear classifier used to divide extracted patterns automatically into good and bad patterns. Input for the classifier are hand labeled patterns from the *Pattern Inspector* (Section 3.2.4). While labeling all patterns would take a long time, we limited our labeling effort to the first 2638 articles (article_id < 5000) containing entities of type ORGANIZATION, LOCATION or PERSON.

The resulting 4699 patterns were split into good and bad patterns by manual interaction with the *Pattern Inspector* (Section 3.2.4). Only with this work, the classifier has data to learn how good patterns look like. For learning the classifier we used 90% (=4228) of the labeled patterns, keeping 10% (=471) back for testing it. After training the classifier, the test set was labeled using the classifier observing the following result (confusion matrix Table 3.4):

| | T | F |
|---|---|---|
| T' | TP=258 | FP=57 |
| F' | FN=4 | TN=152 |

Figure 3.4: Confusion matrix of the classifier

The resulting measures are:

- Precision = 0.8190
- Recall = 0.9847
- F1-score = 0.8943

### 3.2.6 OpenCalais Client

As our reference extraction system we use Thompson-Reuter's service called OpenCalais[6]. Its free version is limited to 4 queries per second or 50,000 per day whereas the commercial version has no limitations. We build a simple client around the provided Java API[7] which uses the articles from our database limited to topics *Economics (ECAT)* and *Government/Social (GCAT)* to prevent our prototype from learning soccer results or stock quotations which are also part of the corpus.

Our first step required to build a OpenCalais client to automatically annotate our selected articles. After registration at Thompson-Reuters, we got an API key which authorized us to use the service. We limited the analysis of the articles to entities and relations because our Stanford NLP prototype is not capable of classifying topics or social tags provided by OpenCalais. The results of the extraction with OpenCalais are stored in table *CALAIS*, which differentiates relations and entities by the *type* field containing <ENTITY> or <RELATION> (see ER diagram in Section 3.2). The field *key* describes the content of field *value* in more detail and contains one of the following item:

| | | |
|---|---|---|
| Anniversary | City | Company |
| Continent | Country | Currency |
| EmailAddress | EntertainmentAwardEvent | Facility |
| FaxNumber | Holiday | IndustryTerm |
| MarketIndex | MedicalCondition | MedicalTreatment |
| Movie | MusicAlbum | MusicGroup |
| NaturalFeature | OperatingSystem | Organization |
| Person | PhoneNumber | PoliticalEvent |
| Position | Product | ProgrammingLanguage |
| ProvinceOrState | PublishedMedium | RadioProgram |
| RadioStation | Region | SportsEvent |
| SportsGame | SportsLeague | Technology |
| TVShow | TVStation | URL |

Figure 3.5: 39 different types of OpenCalais entities

[6] `http://www.opencalais.com/`
[7] Application Programming Interface

# 4 Evaluation

This chapter describes the used hardware equipment and presents the results achieved with our prototype.

## 4.1 Test setup

For out tests, we used the software components described in Section 6.1. The hardware for our tests was a Intel Core2Duo with 2 x 2,8 GHz and 8 GB DDR3 RAM. The extractor requires a minimum of 3 GB RAM to run, so the Java Virtual Machine is generously set to -Xmx5000m.

We started three different extraction tasks, the first with 10.000 articles, followed by 50.000 and finally 100.000 articles. The results are presented in Table 4.1:

| | 10,000 Articles | 50,000 Articles | 100,000 Articles |
|---|---|---|---|
| Duration [min:sec.msec] | 20:11.039 | 122:21.518 | 264:47.644 |
| Speed - Articles/Second | 8.264 | 6.849 | 6.329 |
| Articles containing Relations | 6,588 | 33,624 | 67,331 |
| Number of Extracted Patterns | 17,334 | 85,111 | 161,811 |
| Number of Extracted Tuples | 26,616 | 140,696 | 282,610 |

Figure 4.1: Extraction details

The extraction details show that the speed is decreasing with the number of articles queried. The explanation is, that the number of database request increases because the whole article set is split into packages of 5,000 (*fetchsize*). This setting limits the memory usage and can be increased depending on the available hardware setup.

## 4.2 Extract Entities and Relations

First, to get a large amount of reference relations, OpenCalais worked roughly one day and processed 59,562 articles receiving 1,521,348 entities and relations. The extraction speed of the free client is limited by Thompson-Reuters. They provide commercial access, which does not have these limitations.

In a second step our prototype extracted named entities and relations from the same articles processed by OpenCalais. Stanford extracted entities matching ten different tags (Table 4.2):

| DATE | LOCATION | MISC |
|---|---|---|
| MONEY | NUMBER | ORDINAL |
| ORGANIZATION | PERCENT | PERSON |
| TIME | | |

Figure 4.2: Ten different types of Stanford entities

While both systems use different keys (OpenCalais goes into more detail) and we are only interested in ORGANIZATION, LOCATION and PERSON, we implemented database table containing a mapping between these keys as shown in Table 4.3:

| Stanford | OpenCalais |
|---|---|
| LOCATION | City |
| LOCATION | Contintent |
| LOCATION | Country |
| LOCATION | ProvinceOrState |
| LOCATION | Region |
| ORGANIZATION | Company |
| ORGANIZATION | Organization |
| PERSON | Person |

Figure 4.3: Entity type matching between Stanford NLP and OpenCalais

With this mapping stored in the database, it was possible to compare found entities from OpenCalais with the matching ones from Stanford. But, as we found out after a detailed analysis of the values extracted by Stanford and OpenCalais, OpenCalais entity types would map to multiple Stanford entities which would lead into problems with our queries calculating precision and recall. Therefore, the mapping was rejected, and the queries compare the values of the three Stanford entities with any occurrence on Calais side.

Before we calculate recall and precision over all tagged articles, we look at one article in more detail. As you can see from the SQL statements, the items found by Stanford are not unique. So an article containing the location term *Brazil* twice will result in two database entries. OpenCalais handles this problem by listing these entities only once. Since we are not interested in the position of the found entity, but only in the fact, that both systems found it at all, we reject duplicate entries. So the results are unified which is important for the calculation of precision and recall.

Query the Stanford entities:

```
select distinct ne.article_id, ne.key, ne.value
from ne, mapping m
where ne.key in ('ORGANIZATION', 'LOCATION', 'PERSON')
and ne.article_id = 2286
order by ne.value;
```

The result of the Stanford query (Table 4.4):

| Article | Key | Value |
|---|---|---|
| 2286 | LOCATION | Argentina |
| 2286 | PERSON | Boni |
| 2286 | LOCATION | Brazil |
| 2286 | ORGANIZATION | Federal Reserve |
| 2286 | PERSON | Felix Boni |
| 2286 | PERSON | James Capel |
| 2286 | PERSON | Lars Schonander |
| 2286 | ORGANIZATION | Lehman Brothers |
| 2286 | PERSON | Matthew Hickman |
| 2286 | LOCATION | Mexico |
| 2286 | LOCATION | Mexico City |
| 2286 | LOCATION | New York |
| 2286 | LOCATION | Santander |
| 2286 | LOCATION | South American |
| 2286 | ORGANIZATION | Treasury |
| 2286 | LOCATION | U.S. |

Figure 4.4: 16 entities found by Stanford in article 2286

Query the OpenCalais entities:

```
select distinct c.article_id, c.key, c.value
from calais c, mapping m
where c.type = 'ENTITY'
and c.article_id = 2286
order by c.value;
```

The result of the OpenCalais query (Table 4.5):

As you can see from the results below, our prototype and OpenCalais find different entities which will have a negative influence on precision and recall.

| Article | Key | Value |
|---|---|---|
| 2286 | Country | Argentina |
| 2286 | Country | Brazil |
| 2286 | Person | Felix Boni |
| 2286 | IndustryTerm | gross domestic product |
| 2286 | Position | head of research |
| 2286 | Position | head of researcher |
| 2286 | Person | Lars Schonander |
| 2286 | Company | Lehman Brothers |
| 2286 | Person | Matthew Hickman |
| 2286 | Country | Mexico |
| 2286 | City | Mexico City |
| 2286 | City | New York |
| 2286 | Country | United States |
| 2286 | Organization | US Federal Reserve |

Figure 4.5: 14 entities found by OpenCalais in article 2286

## 4.3 Precision and Recall of Entities

**Precision** is the fraction of the documents retrieved that are relevant to the user's information need[1].

**Recall** is the fraction of the documents that are relevant to the query which have been successfully retrieved[2].

We not only present a single number for the entity precision and recall of all tagged articles, but a distribution of the precision for all processed articles (58,148) (precision Table 4.6, recall Table 4.6):

While checking these numbers, some problems became evident:

- Entities containing the '&' symbol are not stored correctly by Stanford NLP. E.g. the company *A.G. Edwards & Sons, Inc.* is stored as *A.G. Edwards& Sons, Inc.* with a missing space before '&', which would reduce precision. There are 10626 such problems in our database, which were corrected by an SQL statement:

```
update ne
set value = regexp\_replace(value, '&␣', '␣&␣', 'g')
where ne.value ~ '.*\\w+\\S\&␣\\w+.*';
> Query OK, 10626 rows affected (00:01:33)
```

[1] http://en.wikipedia.org/wiki/Information_retrieval#Precision
[2] http://en.wikipedia.org/wiki/Information_retrieval#Recall

| | absolute | relative distribution |
|---|---|---|
| precision = 1 | 2154 | 3,7043 |
| 1 > precision >= 0.9 | 371 | 0,6380 |
| 0.9 > precision >= 0.8 | 3,709 | 6,3785 |
| 0.8 > precision >= 0.7 | 8,317 | 14,3031 |
| 0.7 > precision >= 0.6 | 15,041 | 25,8667 |
| 0.6 > precision >= 0.5 | 15,350 | 26,3981 |
| 0.5 > precision >= 0.4 | 5,907 | 10,1585 |
| 0.4 > precision >= 0.3 | 3,543 | 6,0930 |
| 0.3 > precision >= 0.2 | 2,313 | 3,9777 |
| 0.2 > precision >= 0.1 | 1,098 | 1,8882 |
| 0.1 > precision >= 0 | 345 | 0,5933 |

Figure 4.6: Precision distribution of discovered entities by our prototype

| | absolute | relative distribution |
|---|---|---|
| recall = 1 | 1,651 | 2,8393 |
| 1 > recall >= 0.9 | 339 | 0,5829 |
| 0.9 > recall >= 0.8 | 2,252 | 3,8728 |
| 0.8 > recall >= 0.7 | 4,521 | 7,7749 |
| 0.7 > recall >= 0.6 | 10,529 | 18,1072 |
| 0.6 > recall >= 0.5 | 16,481 | 28,3431 |
| 0.5 > recall >= 0.4 | 10,734 | 18,4597 |
| 0.4 > recall >= 0.3 | 6,648 | 11,4328 |
| 0.3 > recall >= 0.2 | 3,493 | 6,0070 |
| 0.2 > recall >= 0.1 | 1,288 | 2,2150 |
| 0.1 > recall >= 0 | 212 | 0,3645 |

Figure 4.7: Recall distribution of discovered entities by our prototype

- Some abbreviations are converted by OpenCalais to their long version. E.g. the abbreviation *U.S.* is automatically converted and stored as *United States*. Stanford NLP do not convert these items. Therefore we updated some of them by running the following SQL Statements:

```
BEGIN;

update ne set value = 'United States'
where value = 'U.S.';
update calais set value = 'United States'
where value = 'U.S.' and type = 'ENTITY';

update ne set value = 'European Union'
where value = 'EU';
update calais set value = 'European Union'
where value = 'EU' and type = 'ENTITY';

update ne set value = 'United Nations'
where value = 'U.N.';
update calais set value = 'United Nations'
where value = 'U.N.' and type = 'ENTITY';

update ne set value = 'North Atlantic Treaty Organization'
where value = 'NATO';
update calais set value =
        'North Atlantic Treaty Organization'
where value = 'NATO' and type = 'ENTITY';

update ne set value = 'International Monetary Fund'
where value = 'IMF';
update calais set value = 'International Monetary Fund'
where value = 'IMF' and type = 'ENTITY';

update ne set value = 'European Monetary Union'
where value = 'EMU';
update calais set value = 'European Monetary Union'
where value = 'EMU' and type = 'ENTITY';

update ne set value = 'Standard & Poor''s'
where value = 'S&P';
update calais set value = 'Standard & Poor''s'
where value = 'S&P' and type = 'ENTITY';

update ne set value = 'World Trade Organisation'
```

```
40 where value = 'WTO';
41 update calais set value = 'World Trade Organisation'
42 where value = 'TWO' and type = 'ENTITY';
43
44 update ne set value = 'Israel'
45 where value = 'Israeli' and key = 'LOCATION';
46 update ne set value = 'Iraq'
47 where value = 'Iraqi' and key = 'LOCATION';
48 update ne set value = 'Russia'
49 where value = 'Russian' and key = 'LOCATION';
50
51 COMMIT;
```

With these simple improvements, the distribution of our precision and recall improved considerable (precision Table 4.8 & recall Table 4.9):

| | absolute | relative distribution |
|---|---|---|
| precision = 1 | 2,509 | 4,3106 |
| 1 > precision >= 0.9 | 580 | 0,9964 |
| 0.9 > precision >= 0.8 | 4,951 | 8,5061 |
| 0.8 > precision >= 0.7 | 9,655 | 16,5879 |
| 0.7 > precision >= 0.6 | 15,264 | 26,2245 |
| 0.6 > precision >= 0.5 | 13,628 | 23,4137 |
| 0.5 > precision >= 0.4 | 4,868 | 8,3635 |
| 0.4 > precision >= 0.3 | 3,148 | 5,4084 |
| 0.3 > precision >= 0.2 | 2,206 | 3,7900 |
| 0.2 > precision >= 0.1 | 1,073 | 1,8434 |
| 0.1 > precision >= 0 | 323 | 0,5549 |

Figure 4.8: Precision distribution of entities after applying the updates

## 4.4 Precision and Recall of Relations

After comparing the entities of Stanford and OpenCalais to see how precise our prototype works, we finally want to calculate the precision and recall of the relations extracted. To improve our results we take advantage of a classifier, which is able to disable bad patterns to improve our results.

Before we classified the patterns, the distribution of the results looks like (precision Table 4.10, recall Table 4.11):

| | absolute | relative distribution |
|---|---|---|
| recall = 1 | 1,720 | 2,9550 |
| 1 > recall >= 0.9 | 396 | 0,6803 |
| 0.9 > recall >= 0.8 | 2,457 | 4,2212 |
| 0.8 > recall >= 0.7 | 5,055 | 8,6848 |
| 0.7 > recall >= 0.6 | 11,244 | 19,3179 |
| 0.6 > recall >= 0.5 | 16,529 | 28,3979 |
| 0.5 > recall >= 0.4 | 10,065 | 17,2923 |
| 0.4 > recall >= 0.3 | 6,032 | 10,3633 |
| 0.3 > recall >= 0.2 | 3,272 | 5,6215 |
| 0.2 > recall >= 0.1 | 1,248 | 2,1441 |
| 0.1 > recall >= 0 | 187 | 0,3212 |

Figure 4.9: Recall distribution of entities after applying the updates

| | absolute | relative distribution |
|---|---|---|
| precision = 1 | 5,694 | 15.0119 |
| 1 > precision >= 0.9 | 150 | 0.3955 |
| 0.9 > precision >= 0.8 | 1,820 | 4.7983 |
| 0.8 > precision >= 0.7 | 2,471 | 6.5146 |
| 0.7 > precision >= 0.6 | 4,787 | 12.6206 |
| 0.6 > precision >= 0.5 | 7,065 | 18.6264 |
| 0.5 > precision >= 0.4 | 3,559 | 9.3831 |
| 0.4 > precision >= 0.3 | 4,659 | 12.2832 |
| 0.3 > precision >= 0.2 | 4,587 | 12.0933 |
| 0.2 > precision >= 0.1 | 2,457 | 6.4777 |
| 0.1 > precision >= 0 | 681 | 1.7954 |

Figure 4.10: Precision distribution of discovered relations

| | absolute | relative distribution |
|---|---|---|
| recall = 1 | 1,216 | 3.2059 |
| 1 > recall >= 0.9 | 21 | 0.0554 |
| 0.9 > recall >= 0.8 | 378 | 0.9966 |
| 0.8 > recall >= 0.7 | 738 | 1.9457 |
| 0.7 > recall >= 0.6 | 2,091 | 5.5128 |
| 0.6 > recall >= 0.5 | 4,509 | 11.8877 |
| 0.5 > recall >= 0.4 | 4,675 | 12.3253 |
| 0.4 > recall >= 0.3 | 7,095 | 18.7055 |
| 0.3 > recall >= 0.2 | 8,491 | 22.386 |
| 0.2 > recall >= 0.1 | 6,573 | 17.3293 |
| 0.1 > recall >= 0 | 2,143 | 5.6499 |

Figure 4.11: Recall distribution of discovered relations

| F1-score | relative distribution |
|---|---|
| 1 | 6,1754 |
| 0.9 | 0.1135 |
| 0.8 | 1.9290 |
| 0.7 | 3.5023 |
| 0.6 | 8.9691 |
| 0.5 | 16.9630 |
| 0.4 | 12.4536 |
| 0.3 | 17.3322 |
| 0.2 | 18.3544 |
| 0.1 | 11.0224 |
| 0 | 3.1849 |

Figure 4.12: F1-score distribution of discovered relations

Then the classifier was trained with the hand tagged patterns of the Pattern Inspector to disable unqualified patterns. Finally precision and recall were recalculated without these patterns (precision Table 4.13, recall Table 4.14):

| | absolute | relative distribution | difference |
|---|---|---|---|
| precision = 1 | 9,107 | 25.4499 | 59.9403 |
| 1 > precision >= 0.9 | 109 | 0.3046 | -27.3333 |
| 0.9 > precision >= 0.8 | 2,021 | 5.6478 | 11.044 |
| 0.8 > precision >= 0.7 | 2,687 | 7.5089 | 8.7414 |
| 0.7 > precision >= 0.6 | 4,875 | 13.6234 | 1.8383 |
| 0.6 > precision >= 0.5 | 6,933 | 19.3746 | -1.8684 |
| 0.5 > precision >= 0.4 | 2,515 | 7.0283 | -29.3341 |
| 0.4 > precision >= 0.3 | 3,486 | 9.7418 | -25.1771 |
| 0.3 > precision >= 0.2 | 2,858 | 7.9868 | -37.6935 |
| 0.2 > precision >= 0.1 | 1,090 | 3.0461 | -55.637 |
| 0.1 > precision >= 0 | 103 | 0.2878 | -84.8752 |

Figure 4.13: Precision distribution of relations after applying the classifier

| | absolute | relative distribution | difference |
|---|---|---|---|
| recall = 1 | 790 | 2.2077 | -35.0329 |
| 1 > recall >= 0.9 | 16 | 0.0447 | -23.8095 |
| 0.9 > recall >= 0.8 | 275 | 0.7685 | -27.2487 |
| 0.8 > recall >= 0.7 | 558 | 1.5594 | -24.3902 |
| 0.7 > recall >= 0.6 | 1,595 | 4.4573 | -23.7207 |
| 0.6 > recall >= 0.5 | 3,595 | 10.0464 | -20.2706 |
| 0.5 > recall >= 0.4 | 3,998 | 11.1726 | -14.4813 |
| 0.4 > recall >= 0.3 | 6,564 | 18.3434 | -7.4841 |
| 0.3 > recall >= 0.2 | 8,601 | 24.0359 | 1.2955 |
| 0.2 > recall >= 0.1 | 7,190 | 20.0928 | 9.3869 |
| 0.1 > recall >= 0 | 2,602 | 7.2714 | 21.4186 |

Figure 4.14: Recall distribution of relations after applying the classifier

As you can see from the result tables, after applying the classifier and removing irrelevant patterns, the number of totally matching patterns/article jumped up by 59%. But at the same time, recall dropped by 35% in the high precision section. This reduces the number of relations with the highest F1-score by 2.11%. Nevertheless, as Table 4.15 shows, there is a overall movement from bad F1-scores (0 - 0.2) to better F1-scores (0.3 and above). A grafical comparison between the F1-scores is available in Figure 4.16. Based on this

| F1-score | relative distribution | difference |
|---|---|---|
| 1 | 6.0449 | -2.1135 |
| 0.9 | 0.116 | 2.1925 |
| 0.8 | 2.0129 | 4.3490 |
| 0.7 | 3.8422 | 9.7045 |
| 0.6 | 9.9936 | 11.4224 |
| 0.5 | 19.6864 | 16.0545 |
| 0.4 | 12.8378 | 3.0856 |
| 0.3 | 18.9331 | 9.2364 |
| 0.2 | 17.8384 | -2.8115 |
| 0.1 | 7.8708 | -28.5930 |
| 0 | 0.8239 | -74.1316 |

Figure 4.15: F1-score distribution of relations after applying the classifier

results, we affirm, that our method of designing a generic relation extraction system is suitable.

Finally, we contrast precision with recall (see Table 4.17) to get the following PR-curve (Figure 4.18). It characterizes the quality of the classifier which is limited by the overall low recall in our case:

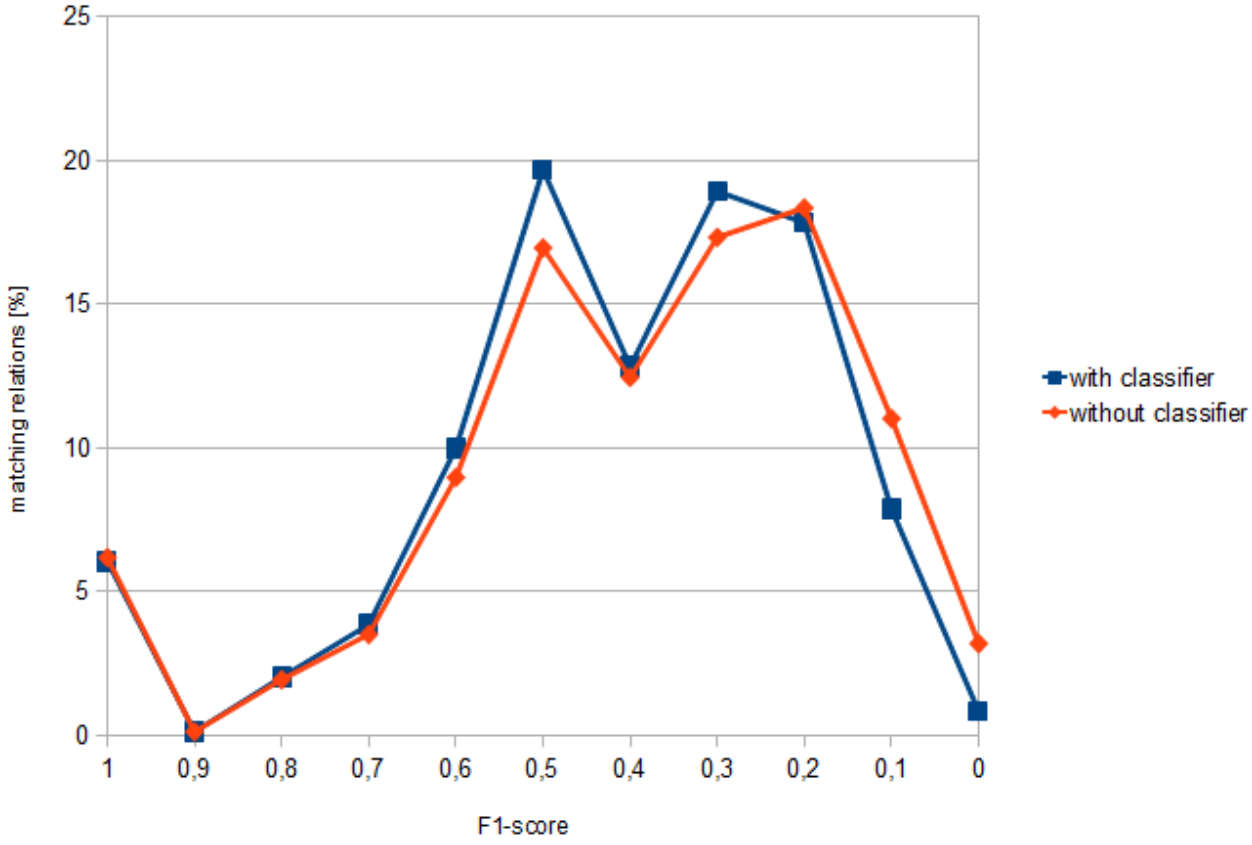


Figure 4.16: Comparison of F1-scores with and without using a classifier

| pattern probability | recall | precision |
|---|---|---|
| 0.8 < x < 0.9 | 0.1321 | 0.9769 |
| 0.5 < x < 0.6 | 0.1357 | 0.9060 |
| x < 0.1 | 0.1596 | 0.6190 |
| x < 0.08 | 0.1592 | 0.3921 |
| x = 0 | 0.3532 | 0 |

Figure 4.17: PR-curve calculations

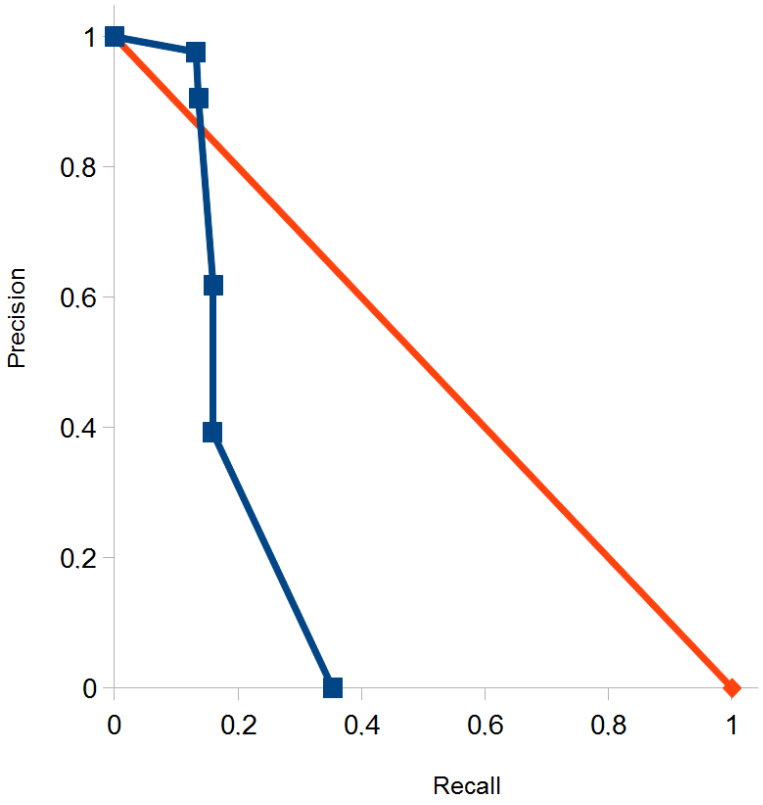


Figure 4.18: PR-curve of the classified patterns

# 5 Conclusions and Future Work

In previous work, different algorithms have been applied to pattern learning. From frequency based (e.g. DIPRE, Section 2.1) over weighting the surrounding context (e.g. SNOWBALL, Section 2.4) to the usage of sophisticated mathematical models like Markov models or CRF (e.g. O-CRF Section 2.3). Furthermore, DIPRE does not use NLP like POS or NER, others take advantage of them. Nevertheless, most systems have the same problems when it comes to recall and precision. While this systems should work at Web scale it is hard to estimate their effectiveness which depends on many parameters. The supplied corpus, the relation definition or the quality of the used NLP tools influences the extracting patterns and tuples. Nevertheless, we provided a way to extract entities and their relations in a fast way by using POS tagging and NER using an existing and actively managed NLP package. With the further usage of our relation extraction tool, qualified patterns will divergent against a maximum number of possible patterns because of the limited number of Penn Treebank tags and the number of tokens used in our representation of relations. Therefore, a set of good, domain-independent patterns will be defined. The usage of the classifier reduces manual work - especially at Web scale - but it also reduces the recall. So the classifier should receive more attention in comparable systems.

As we can see from the PR-curve the recall is limited to 15% whereas the precision is above 90% for patterns marked positive with a probability of more than 50%. After investigating the results, we have to say, that our system is limited to relations between entities by design, OpenCalais is not. Thompson-Reuters service finds relations between entities, entities and co-references as well as statements with only one entity. Co-reference resolution is implemented in the StanfordNLP package, but crashes every one a while so, we are waiting for a new version of the API to get it tested again. Furthermore, OpenCalais does not retrieve all relations existing in the articles, which reduces the recall too. Hand tagged relations or another, more complete extraction system would improve the recall further.

Nevertheless, our prototype was capable of processing 100.000 articles in roughly 5 hours on a dual core CPU with a memory footprint of 5 GB RAM (see Section 4.1). The software included is licensed under GNU GPL v2 and is allowed to use for research purposes, free software projects, .... Although Sergei Brin said in 1998 that 20% of recall would be acceptable whereas the precision less than 90% would not, we are confident that improvements are possible after implementation of the points mentioned.

# 6 Appendix

## 6.1 Setup of the Software Environment

This section describes the installation and deployment of our prototype. We will only discuss the most important steps here, additional information can be directly taken from the source code of the prototype.

### 6.1.1 Introduction

By selecting Java as our developing language and Stanford CoreNLP as NLP package, we explicitly take advantage of the following main software components for our development platform (development was split into Microsoft Windows 7 x86 64 Bit and Ubuntu 10.10 64 Bit platform):

- Oracle Java Development Kit[1] version 1.6.0-b23 **(required)**
- PostgreSQL[2] Database version 9.0.3-1 **(required)**
- Stanford CoreNLP[3] version 1.0.3 **(required)**
- Stanford Classifier[4] version 2.1.1 **(required)**
- PostgreSQL JDBC4 Driver[5] version 8.4 build 701-2 **(required)**
- Eclipse[6] Helios (3.6.1) Integrated Development Environment (IDE) **(additional)**
- Git[7] version control system **(additional)**
- EGit[8] a Eclipse integration of Git **(additional)**

Some of the software components come with installer packages, which makes it easy to install these on your preferred platform. Especially for the installation of the PostgreSQL

[1] `http://www.oracle.com/technetwork/java/index.html`
[2] `http://www.postgresql.org/`
[3] `http://nlp.stanford.edu/software/corenlp.shtml`
[4] `http://nlp.stanford.edu/software/classifier.shtml`
[5] `http://jdbc.postgresql.org/`
[6] `http://www.eclipse.org/`
[7] `http://git-scm.com/`
[8] `http://www.eclipse.org/egit/`

database and the Java Development Kit (version 6 build 23) (JDK) are excellent guides available on-line.

The setup routine for the *PostgreSQL JDBC4 Driver* is topic of Section 6.1.2 followed by the initialization of the article database (Section 6.1.3). Finally, the installation of the selected NLP Package is covered in Section 6.1.4.

### 6.1.2 Java Database Connectivity (JDBC) driver

Start the PostgreSQL installer and follow the instructions. After finishing the installation routine, it starts a wizard for additional plug-in selection. In this dialog look for the PostgreSQL JDBC driver and select it (see Figure 6.1).

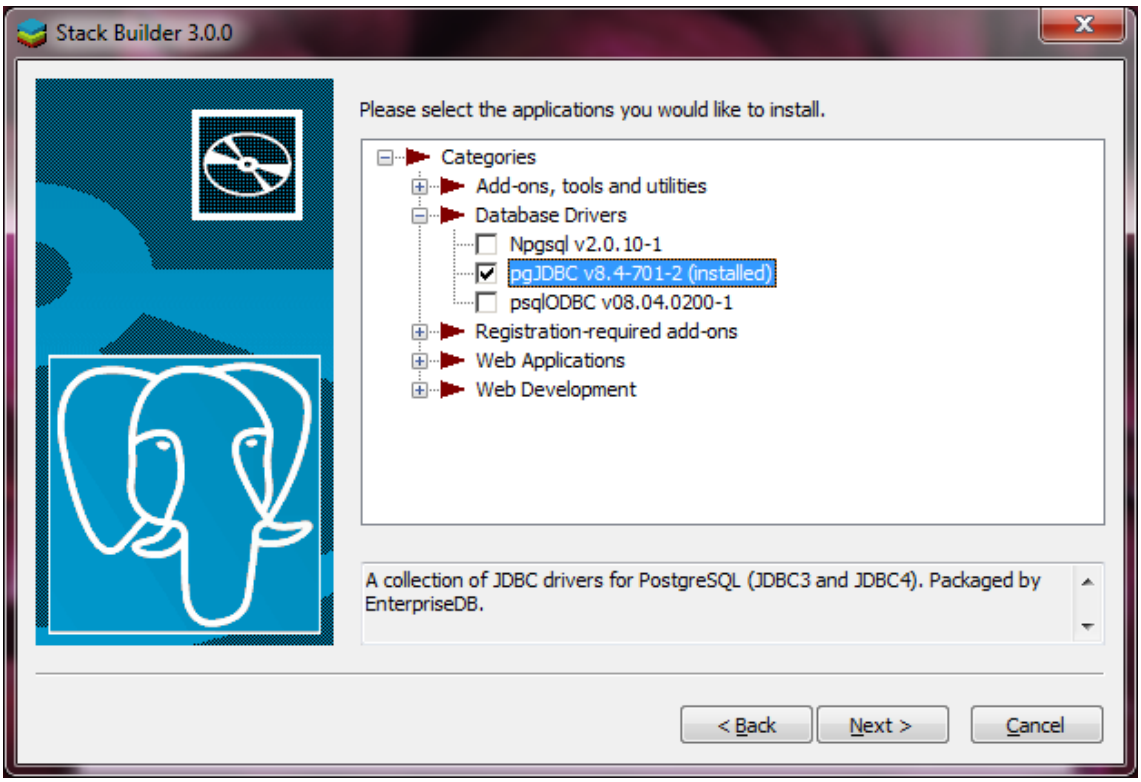


Figure 6.1: PostgreSQL Application stack builder - install JDBC driver

The JDBC driver will now be located in the selected directory. In case you already have PostgreSQL installed, the JDBC driver is also available for separate download[9].

### 6.1.3 Create User and Database

To work with PostgreSQL create a role and a database to work with. You can also recycle an existing one. First, create the log-in role:

[9] `http://jdbc.postgresql.org/download.html`

```
CREATE ROLE j0125536 LOGIN PASSWORD 'j0125536'
  SUPERUSER CREATEDB CREATEROLE VALID UNTIL 'infinity';
```

Now the database can be created. The just created role will be the owner of the new database:

```
CREATE DATABASE "j0125536"
  WITH ENCODING='UTF8'
       OWNER=j0125536;
```

The Reuters articles are already available as an SQL script, so the import should start with running the following statement on a command line:

```
>psql -d j0125536 -U j0125536 -f reuters.sql
```

The parameters are:

- -d j0125536 selects the database to insert the articles in
- -U j0125536 selects the user with which the statements should be executed
- -f reuters.sql the path to the SQL script containing the articles

After successfully loading the articles, the database should look like:

```
                List of relations
 Schema |           Name           |   Type   | Owner
--------+--------------------------+----------+---------
 public | article                  | table    | j0125536
 public | article_article_id_seq   | sequence | j0125536
 public | article_industry         | table    | j0125536
 public | article_region           | table    | j0125536
 public | article_topic            | table    | j0125536
 public | industry                 | table    | j0125536
 public | region                   | table    | j0125536
 public | topic                    | table    | j0125536
 public | vw_article_region        | view     | j0125536
(9 rows)
```

Run the following statement and compare the results:

```
psql -d j0125536 -U j0125536 -c "select count(1) from article;"
```

```
 count
--------
 806791
(1 row)
```

### 6.1.4 Natural Language Processing (NLP) Package

Implementation of this prototype would be possible with different software packages, capable of processing text in the desired manner (sentence splitting, tokenization, lemma generation, POS and NER tagging). Three considered packages are available under open licenses (usage and distribution is at least free for research purposes) and are written in Java:

- **Stanford CoreNLP**[10] is a suite of NLP tools, containing tools to handle English text. It is available in Version 1.0.3 under full GPL and *"is designed to be highly flexible and extensible" [12].*
- **LingPipe 4.0.1**[11] is available for free under Royalty Free License as well as professional licenses for enterprises. *"LingPipe's architecture is designed to be efficient, scalable, reusable, and robust" [19].* It provides models trained on different corpora for download and a comprehensive documentation.
- **MALLET**[12] stands for *MAchine Learning for LanguagE Toolkit* written by Andrew McCallum at the University of Massachusetts and is available under Common Public License. Java API documentation and some tutorials are available, but documentation in general seems kind of fragmented. Nevertheless, MALLET includes *statistical natural language processing, document classification, clustering, topic modeling, information extraction, and other machine learning applications" [20].*

Download the required libraries from the provided website and add the location path to your Java class-path.

## 6.2 Implementation

The topic of this section is the implementation of the design parts of Figure 3.1. We pick some of the key parts from code and explain these. Please look at the documented source code for deeper investigation.

We start with implementation details of the Database (Section 6.2.1) followed by the design parts of the NLP Package containing Relation Extractor (Section 6.2.2), Pattern Inspector (Section 6.2.3) and finally Pattern Classifier (Section 6.2.4).

### 6.2.1 Database

To run the prototype, the following tables are required in the database.

[10] `http://nlp.stanford.edu/software/corenlp.shtml`
[11] `http://alias-i.com/lingpipe/index.html`
[12] `http://mallet.cs.umass.edu/`

Create now the tables and views for storing the extracted entities, patterns and tuples. Start with table *pattern* first because the primary key *pattern_id* is used in a foreign key constraint of table *tuple*:

```
1  CREATE TABLE "public"."pattern" (
2    "pattern_id" SERIAL,
3    "left" VARCHAR(255),
4    "ne1" VARCHAR(255),
5    "middle" VARCHAR(255),
6    "ne2" VARCHAR(255),
7    "right" VARCHAR(255),
8    "iteration" INTEGER NOT NULL,
9    "enabled" BOOLEAN DEFAULT true NOT NULL,
10   "modifier" CHAR(1),
11   CONSTRAINT "pattern_pkey" PRIMARY KEY("pattern_id"),
12   CONSTRAINT "pattern_unique_pattern_idx"
13         UNIQUE("left", "ne1", "middle", "ne2", "right")
14 ) WITHOUT OIDS;

16 CREATE INDEX "pattern_enabled_idx" ON "public"."pattern"
17   USING btree ("enabled");

19 CREATE INDEX "pattern_modifier_idx" ON "public"."pattern"
20   USING btree ("modifier");

22 CREATE INDEX "pattern_ne1_idx" ON "public"."pattern"
23   USING btree ("ne1");

25 CREATE INDEX "pattern_ne2_idx" ON "public"."pattern"
26   USING btree ("ne2");
```

This table contains the part-of-speech representation of discovered tuples. Therefor it is similar to table tuple below.

Important for this table is the creation of a unique key constraint (line 12-13) because it protects us from inserting identical rows in table *pattern*. We anticipate that many identical patterns will be discovered, depending on the number and the domain of the articles. So each insert statement during the extraction process has to recover from a *unique key constraint violation* (see Section 6.2.2).

Continue with table *tuple*:

```
1  CREATE TABLE "public"."tuple" (
2    "tuple_id" SERIAL,
3    "article_id" INTEGER NOT NULL,
4    "pattern_id" INTEGER NOT NULL,
```

```
    "left" VARCHAR(255),
    "ne1" VARCHAR(255),
    "middle" VARCHAR(255),
    "ne2" VARCHAR(255),
    "right" VARCHAR(255),
    "iteration" INTEGER NOT NULL,
    CONSTRAINT "tuple_pkey" PRIMARY KEY("tuple_id"),
    CONSTRAINT "tuple_article_id_fkey" FOREIGN KEY ("article_id")
        REFERENCES "public"."article"("article_id")
        ON DELETE NO ACTION
        ON UPDATE NO ACTION
        NOT DEFERRABLE,
    CONSTRAINT "tuple_pattern_id_fkey" FOREIGN KEY ("pattern_id")
        REFERENCES "public"."pattern"("pattern_id")
        ON DELETE NO ACTION
        ON UPDATE NO ACTION
        NOT DEFERRABLE
) WITHOUT OIDS;

CREATE INDEX "tuple_article_id_idx" ON "public"."tuple"
    USING btree ("article_id");

CREATE INDEX "tuple_ne1_idx" ON "public"."tuple"
    USING btree ("ne1");

CREATE INDEX "tuple_ne2_idx" ON "public"."tuple"
    USING btree ("ne2");

CREATE INDEX "tuple_pattern_id_idx" ON "public"."tuple"
    USING btree ("pattern_id");
```

This table will store all concrete relations discovered by the extractor. Each row is tied to a pattern and a article in which the relation occurs. The additional indices are by reason of performance.

For a more intuitive handling of the extracted patterns, we packaged some logic into views. A view in SQL is a predefined select statement that is executed each time the view is called.

The view *v_pattern_tuple_frequency* selects all patterns and counts how many tuples are generated by each pattern (frequency). The result is order by frequency descending:

```
CREATE VIEW "public"."v_pattern_tuple_frequency" (
    frequency,
    pattern_id,
```

```
    "left",
    ne1,
    middle,
    ne2,
    "right",
    enabled,
    modifier)
AS
SELECT count(tuple.tuple_id) AS frequency,
        tuple.pattern_id, pattern."left",
    pattern.ne1, pattern.middle, pattern.ne2,
    pattern."right", pattern.enabled,
    pattern.modifier
FROM tuple, pattern
WHERE tuple.pattern_id = pattern.pattern_id
AND (pattern.ne1 in ('ORGANIZATION', 'PERSON', 'LOCATION')
AND (pattern.ne2 in ('ORGANIZATION', 'PERSON', 'LOCATION')
GROUP BY tuple.pattern_id, pattern."left", pattern.ne1,
        pattern.middle, pattern.ne2, pattern."right",
        pattern.enabled, pattern.modifier
ORDER BY count(tuple.tuple_id) DESC;
```

For storage of reference entities and relations of OpenCalais Viewer, the following table is required:

```
CREATE TABLE "public"."calais" (
  "calais_id" SERIAL,
  "article_id" INTEGER NOT NULL,
  "key" VARCHAR(255) NOT NULL,
  "value" VARCHAR(255) NOT NULL,
  "type" VARCHAR(20) NOT NULL,
  CONSTRAINT "calais_pkey_idx" PRIMARY KEY("calais_id"),
  CONSTRAINT "calais_article_fk" FOREIGN KEY ("article_id")
    REFERENCES "public"."article"("article_id")
    ON DELETE NO ACTION
    ON UPDATE NO ACTION
    NOT DEFERRABLE
) WITHOUT OIDS;

ALTER TABLE "public"."calais"
  ALTER COLUMN "calais_id" SET STATISTICS 0;

ALTER TABLE "public"."calais"
  ALTER COLUMN "article_id" SET STATISTICS 0;
```

```
CREATE INDEX "calais_article_id_idx" ON "public"."calais"
  USING btree ("article_id");

CREATE INDEX "calais_key_idx" ON "public"."calais"
  USING btree ("key");

CREATE INDEX "calais_value_idx" ON "public"."calais"
  USING btree ("value");
```

The last table required will contain the entities extracted by the Stanford NLP Package:

```
CREATE TABLE "public"."ne" (
  "ne_id" SERIAL,
  "article_id" INTEGER NOT NULL,
  "sentence" INTEGER NOT NULL,
  "position" INTEGER NOT NULL,
  "key" VARCHAR(255) NOT NULL,
  "value" VARCHAR(255) NOT NULL,
  CONSTRAINT "ne_pk" PRIMARY KEY("ne_id"),
  CONSTRAINT "ne_article_fk" FOREIGN KEY ("article_id")
    REFERENCES "public"."article"("article_id")
    ON DELETE NO ACTION
    ON UPDATE NO ACTION
    NOT DEFERRABLE
) WITHOUT OIDS;

CREATE INDEX "ne_article_id_idx" ON "public"."ne"
  USING btree ("article_id");

CREATE INDEX "ne_key_idx" ON "public"."ne"
  USING btree ("key");

CREATE INDEX "ne_value_idx" ON "public"."ne"
  USING btree ("value");
```

Finally, after running all SQL statements on your database, your current schema for should look like this:

```
                List of relations
 Schema |           Name                |    Type     | Owner
--------+-------------------------------+-------------+-----------
 public | article                       | table       | j0125536
 public | article_article_id_seq        | sequence    | j0125536
 public | article_industry              | table       | j0125536
```

```
 public | article_region              | table     | j0125536
 public | article_topic               | table     | j0125536
 public | calais                      | table     | j0125536
 public | calais_calais_id_seq        | sequence  | j0125536
 public | industry                    | table     | j0125536
 public | ne                          | table     | j0125536
 public | ne_ne_id_seq                | sequence  | j0125536
 public | pattern                     | table     | j0125536
 public | pattern_pattern_id_seq      | sequence  | j0125536
 public | region                      | table     | j0125536
 public | topic                       | table     | j0125536
 public | tuple                       | table     | j0125536
 public | tuple_tuple_id_seq          | sequence  | j0125536
 public | v_pattern_tuple_frequency | View      | j0125536
 public | vw_article_region           | View      | j0125536
(18 Zeilen)
```

### 6.2.2 Relation Extractor

In this subsection we present some key parts of the implementation of the relation extractor.

```
1 private StanfordCoreNLP initStanfordNLP()
2 {
3         Properties props = new Properties();
4         props.put("annotators", "tokenize, ssplit, pos, lemma,
              ner, regexner");
5         props.put("regexner.ignorecase", "true");
6         StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
7         setStartTime(System.currentTimeMillis());
8         return pipeline;
9 }
```

This function sets the required environment for the execution of the NLP task like tokenization, sentence splitting, POS tagging, lemma generation, NER and regnexer (Section 3.2.1). The properties defined in line 3 includes all settings required for the StanfordCoreNLP class.

```
1 Connection conn = extractor.connectDB();
2 TreeMap<Long, String> input = extractor.readArticles(conn,
      extractor.getFirstArticleID());
```

The connection to the database is established by calling the method *connectDB()* in line 1. Reading the articles is executed in line 2. All found articles within the fetch-size are

returned to the TreeMap for further iteration.

```
1 for (CoreLabel token : sentence.get(TokensAnnotation.class))
2 {
3         this.getFeatureWORD().add(token.get(TextAnnotation.class
              ));
4         this.getFeaturePOS().add(token.get(
              PartOfSpeechAnnotation.class));
5         this.getFeatureNER().add(token.get(
              NamedEntityTagAnnotation.class));
6         this.getFeatureLEMMA().add(token.getString(
              LemmaAnnotation.class));
7 }
8 this.extractRelations(conn, article_id, this.getFeatureNER(),
      this.getFeaturePOS(), this.getFeatureWORD());
```

Inside the article loop, the method *analyzeArticle()* is called. This is a part of this method, analyzing each token of a sentence in lines 1-7. Finally, the relation candidates are extracted for each sentence in line 8.

```
1  public void extractRelations(Connection conn, int article_id,
       Vector<String> featureNER, Vector<String> featurePOS, Vector<
       String> featureWORD)
2  {
3          Vector<NamedEntity> entities = this.findNamedEntities();
4          if (entities.size() > 1)
5          {
6                  Relation r = null;
7                  for (int i = 0; i < entities.size() - 1; i++)
8                  {
9                          r = Relation.createRelation(featureNER,
                               entities.get(i), entities.get(i + 1))
                               ;
10                         if (r != null)
11                                 storeRelationToDB(conn, r,
                                       article_id, featurePOS,
                                       featureWORD);
12                 }
13         }
14 }
```

For each sentence the method *extractRelations()* is called with the POS tags, NER annotation and the plain words of the article. While the named entities can consist of more than one word, the function *findNamedEntities()* in line 3 concatenates these words and returns them. If there is more than one entity in the sentence, a relation candidate can be

extracted in line 9. In the case of successfully extracting a relation, it will be stored in line 11. Iterating through the loop in the line 7-12 the method will find all possible relation between each pair of entity: 1-2, 2-3, 3-4 ....

```
1  ResultSet rs = null;
2  try
3  {
4   pstmtPattern.setString(1, r.getLeft().toString(featurePOS));
5   pstmtPattern.setString(2, ((NamedEntity) r.getNe1()).getNeTyp()
       );
6   pstmtPattern.setString(3, r.getMiddle().toString(featurePOS));
7   pstmtPattern.setString(4, ((NamedEntity) r.getNe2()).getNeTyp()
       );
8   pstmtPattern.setString(5, r.getRight().toString(featurePOS));
9   pstmtPattern.setInt(6, getIteration());
10  result = pstmtPattern.executeUpdate();
11  if (result == 1)
12  {
13          rs = pstmtPattern.getGeneratedKeys();
14          if (rs != null && rs.next())
15                  pattern_id = rs.getInt(1);
16  }
17 }
18 catch (SQLException e)
19 {
20  if (e.getSQLState().equals("23505"))
21  {
22     c.rollback();
23          pstmtSelect.setString(1, r.getLeft().toString(featurePOS
               ));
24          pstmtSelect.setString(2, ((NamedEntity) r.getNe1()).
               getNeTyp());
25          pstmtSelect.setString(3, r.getMiddle().toString(
               featurePOS));
26          pstmtSelect.setString(4, ((NamedEntity) r.getNe2()).
               getNeTyp());
27          pstmtSelect.setString(5, r.getRight().toString(
               featurePOS));
28          rs = pstmtSelect.executeQuery();
29          if (rs != null && rs.next())
30                  pattern_id = rs.getInt(1);
31          }
32          else
33          {
```

```
34                      e.printStackTrace();
35              }
36      }
```

Finally, the patterns become stored inside the method *storeRelationToDB*. We are using prepared statements (because of performance benefits) to insert the pattern (lines 4-10). If the insert is successful the result will become 1 and the serial number generated for the primary key *pattern_id* will be stored in line 15. In case the insert fails, the exception will be caught in line 18. If it fails because of an unique key constraint violation (line 19) the pattern is selected from the database and the corresponding *pattern_id* is stored in line 30. Should the select statement fail too, the exception is print to the command-line and this pattern and its tuple is skipped and not inserted into the database.

### 6.2.3 Pattern Inspector

The GUI of the Pattern Inspector consists of three filterable tables that depend on each other. So the application is relatively simple while there is only one database update: Disable or enable a pattern. Filtering of the tables is achieved by using the class *RowFilter* which is already part of Java 1.5. The usage of the application is explained in Section 6.3.

### 6.2.4 Pattern Classifier

The Linear Stanford classifier is used to mark all extracted patterns as positive or negative depending on a hand tagged set of patterns. To do so, we have to train the classifier with the training data (hand tagged patterns) generated by the Pattern Inspector (Section 6.3). Each training pattern is encapsulated into a *BasicDatum* object consisting of multiple features and a label:

```
1  features = new ArrayList<String>();
2  features.add(rs.getString(1)); // pattern.left
3  features.add(rs.getString(2)); // pattern.ne1
4  features.add(rs.getString(3)); // pattern.middle
5  features.add(rs.getString(4)); // pattern.ne2
6  features.add(rs.getString(5)); // pattern.right
7  trainingSet.add(new BasicDatum(features, Boolean.valueOf(rs.
       getBoolean(6)))); // true or false
```

After learning we apply the classifier to the remaining, not marked patterns by create a *BasicDatum* and read the most likely class it would assign:

```
1  enabled = classifier.classOf(new BasicDatum(features, null));
```

The returning value is stored into the database to the specific pattern to enable or disable it.

## 6.3 Using the Pattern Inspector

After finishing the extraction tasks, the Pattern Inspector displays the discovered patterns and tuples. In the pattern view, we sorted by tuple frequency and found out, that some bogus tuples have raised the frequency of bad patterns. Understandably, not all extracted tuples are qualified relations but there were some relation between the extracted entities, we did not expect. The corresponding patterns will therefor disabled in the PatternInspector GUI in Figure 6.2.

Because of this result, an automatic pattern learner based on the frequency would distract further extractions. Because of this fact, there is no implementation of an automatic learner in the application, but within some minutes, the number of bogus tuples was manually reduced by roughly 180,000 (Figure 6.3). While disabling of the bad patterns was really fast, the patterns will not be compared during the extraction process. Each further SQL transaction would reduce the extraction speed. For querying the results, join the pattern table with the tuples and remove the disabled patterns. This action can be adjusted in the Pattern Inspector which was highly helpful to disable the bogus patterns.

As Figure 6.3 shows, before disabling bogus patterns, the row count of the tuples was 449,992 rows whereas after disabling them only 274,464 returned.

To compare our results, we used the Reuters CalaisViewer[13] to compare our precision (Figure 6.4:

The first part (1) of Figure 6.4 shows the result OpenCalais return for tagging the supplied article. The recognized entities are underlined. OpenCalais also resolved co-references for Makoto Tonoki twice. The second part (2) shows the article in the Pattern Inspector whereas (3) displays the extracted tuples of the article. The phrase *"Tokyo that gold could rise if President Bill Clinton"* does not constitute a relation because our defined maximum distance is six words. These entities are seven words apart. Nevertheless, all other relations depending on our definition (distance <= six words) were identified.

## 6.4 Penn Treebank Annotation Tags

Alphabetical list of part-of-speech tags used in the Penn Treebank Project[14] (Table 6.5:

[13] `http://viewer.opencalais.com/`

[14] `http://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`

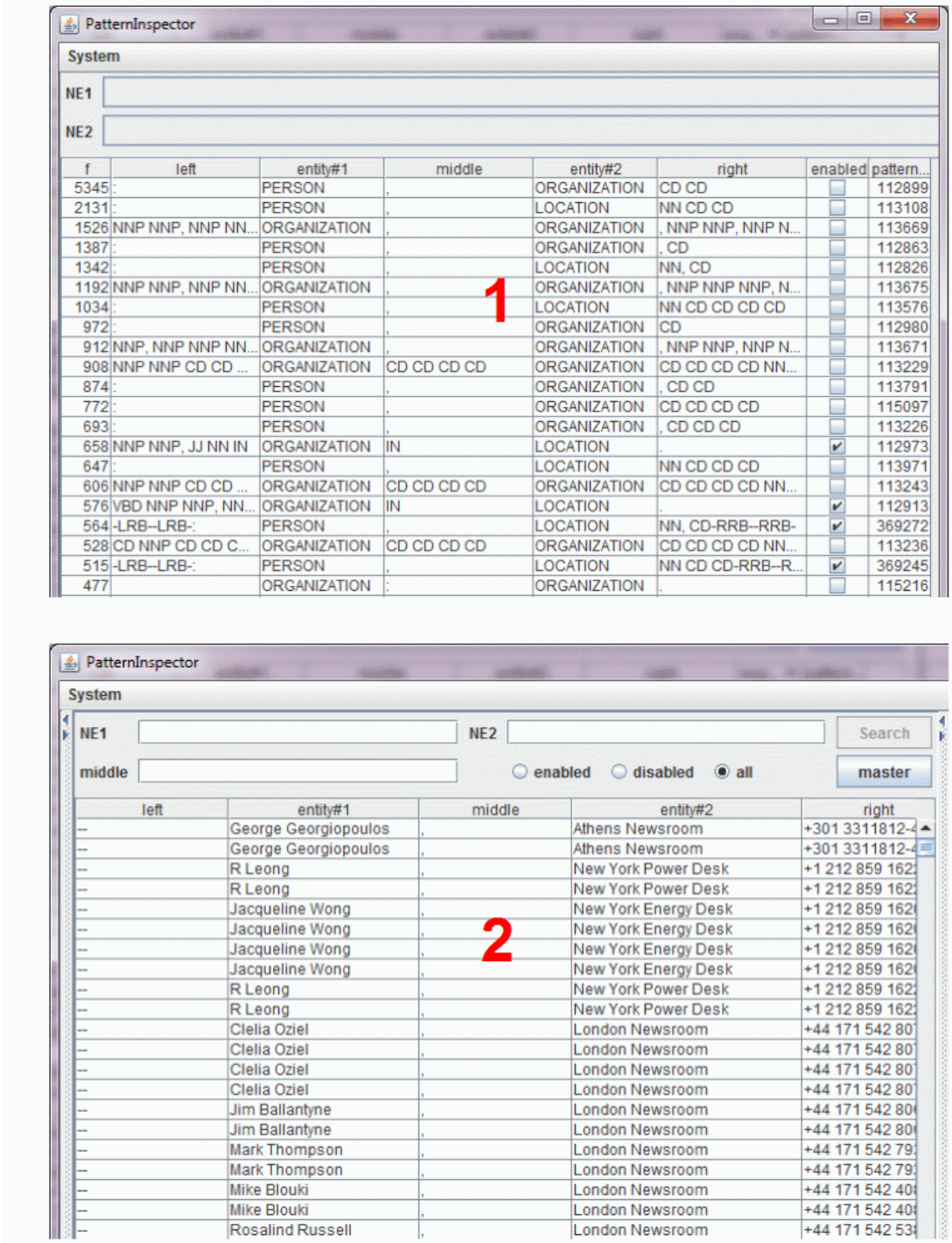


Figure 6.2: Pattern Inspector displaying patterns (1) and corresponding tuples (2); the signature of the Reuters articles pushing the patterns (e.g. *"– David Chance, London Newsroom +44 171 542 5887"*)

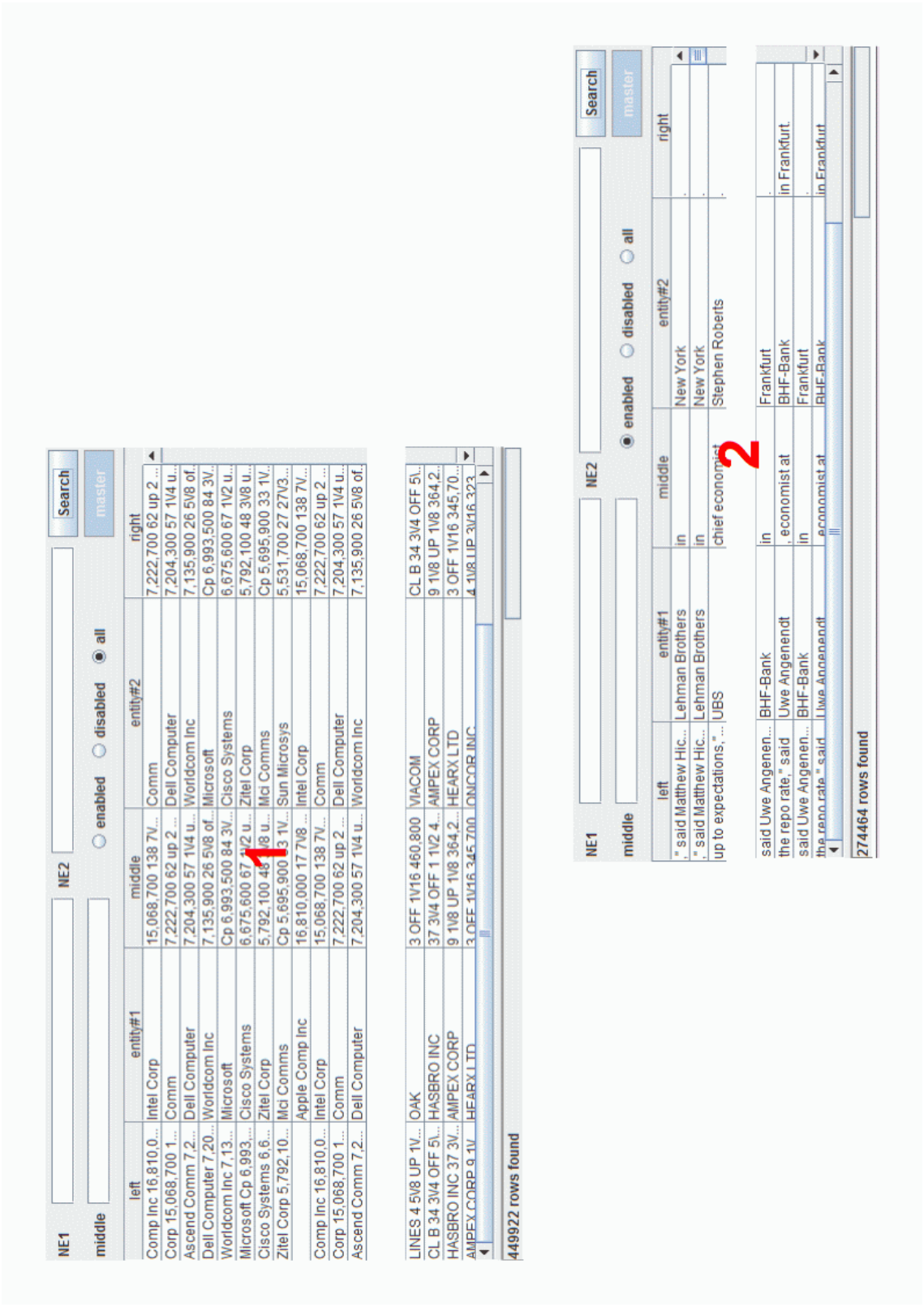


Figure 6.3: Pattern Inspector showing the difference between all (1) and disabled patterns (2)

Figure 6.4: CalaisViewer (1) in comparison to the Pattern Inspector: article (2), tuple (3) and pattern (4) view

| Number | Tag | Description |
|---|---|---|
| 1. | CC | Coordinating conjunction |
| 2. | CD | Cardinal number |
| 3. | DT | Determiner |
| 4. | EX | Existential there |
| 5. | FW | Foreign word |
| 6. | IN | Preposition or subordinating conjunction |
| 7. | JJ | Adjective |
| 8. | JJR | Adjective, comparative |
| 9. | JJS | Adjective, superlative |
| 10. | LS | List item marker |
| 11. | MD | Modal |
| 12. | NN | Noun, singular or mass |
| 13. | NNS | Noun, plural |
| 14. | NNP | Proper noun, singular |
| 15. | NNPS | Proper noun, plural |
| 16. | PDT | Predeterminer |
| 17. | POS | Possessive ending |
| 18. | PRP | Personal pronoun |
| 19. | PRP$ | Possessive pronoun |
| 20. | RB | Adverb |
| 21. | RBR | Adverb, comparative |
| 22. | RBS | Adverb, superlative |
| 23. | RP | Particle |
| 24. | SYM | Symbol |
| 25. | TO | to |
| 26. | UH | Interjection |
| 27. | VB | Verb, base form |
| 28. | VBD | Verb, past tense |
| 29. | VBG | Verb, gerund or present participle |
| 30. | VBN | Verb, past participle |
| 31. | VBP | Verb, non-3rd person singular present |
| 32. | VBZ | Verb, 3rd person singular present |
| 33. | WDT | Wh-determiner |
| 34. | WP | Wh-pronoun |
| 35. | WP$ | Possessive wh-pronoun |
| 36. | WRB | Wh-adverb |

Figure 6.5: Penn Treebank Annotation Tags

# List of Figures

# Bibliography


[1] We knew the web was big... `http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html`, 2008. last visited 2011-01-23.

[2] Sergey Brin. Extracting patterns and relations from the world wide web. In *Proceedings of the First International Workshop on the Web and Databases, WebDB 1998*, pages 172–183, March 1998.

[3] Michael J. Cafarella, Jayant Madhavan, and Alon Halevy. Web-scale extraction of structured data. *SIGMOD Rec.*, 37(4):55–61, 2008.

[4] Michael J. Cafarella, Alon Y. Halevy, Daisy Zhe Wang, Eugene Wu 0002, and Yang Zhang. Webtables: exploring the power of tables on the web. *Publication of the Very Large Database Endowment (PVLDB)*, 1:538–549, 2008.

[5] Michele Banko, Michael J. Cafarella, Stephen Soderl, Matt Broadhead, and Oren Etzioni. Open information extraction from the web. In *Proceedings of the International Joint Conferences on Artificial Intelligence 2007 (IJCAI-2007)*, pages 2670–2676, 2007.

[6] Doug Downey, Oren Etzioni, and Stephen Soderland. A probabilistic model of redundancy in information extraction. In *Proceedings of the International Joint Conference on Artificial Intelligence 2005 (IJCAI-2005)*, page 1034. IJCAI-05, 2005.

[7] Michele Banko and Oren Etzioni. The tradeoffs between open and traditional relation extraction. In *Proceedings of the Association for Computational Linguistics 2008 (ACL-2008)*, pages 28–36, Columbus, Ohio, June 2008. Association for Computational Linguistics.

[8] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proceedings of the Association for Computational Linguistics Conference 2003 (ACL-2003)*, pages 423–430, 2003.

[9] Eugene Agichtein and Luis Gravano. Snowball: Extracting relations from large plain-text collections. In *Proceedings of the 5th ACM International Conference on Digital Libraries*, pages 85–94, 2000.

[10] Jun Zhu, Zaiqing Nie, Xiaojiang Liu, Bo Zhang, and Ji-Rong Wen. Statsnowball: a statistical approach to extracting entity relationships. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 101–110, New York, NY, USA, 2009. ACM.

[11] Michael J. Cafarella, Doug Downey, Stephen Soderland, and Oren Etzioni. KnowItNow: Fast, Scalable Information Extraction from the Web. In *Proceedings of the Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing 2005*, pages 563–570, Vancouver, British Columbia, Canada, October 2005. Association for Computational Linguistics.

[12] Stanford core nlp: A suite of core nlp tools. `http://nlp.stanford.edu/software/corenlp.shtml`. last visited 2011-02-10.

[13] Kristina Toutanova and Christopher D. Manning. Enriching the knowledge sources used in a maximum entropy part–of–speech tagger. In *Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-2000)*, pages 63–70, 2000.

[14] Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology (NAACL-2003)*, pages 173–180, Morristown, NJ, USA, 2003. Association for Computational Linguistics.

[15] The penn treebank project. `http://www.cis.upenn.edu/~treebank/`. last visited 2011-02-10.

[16] Jenny Rose Finkel, Trond Grenager, and Christopher D. Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In The Association for Computer Linguistics, editor, *Proceedings of the 43nd Annual Meeting of the Association for Computational Linguistics (ACL-2005)*, pages 363–370, 2005.

[17] Karthik Raghunathan, Heeyoung Lee, Sudarshan Rangarajan, Nate Chambers, Mihai Surdeanu, Dan Jurafsky, and Christopher Manning. A multi-pass sieve for coreference resolution. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing (EMNLP-2010)*, pages 492–501, Cambridge, MA, October 2010. Association for Computational Linguistics.

[18] Anish Das Sarma, Alpa Jain, and Divesh Srivastava. I4e: interactive investigation of iterative information extraction. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the Special Interest Group on Management of Data Conference 2010 (SIGMOD-2010)*, pages 795–806. ACM, 2010.

[19] Lingpipe: A tool kit for processing text using computational linguistics. `http://alias-i.com/lingpipe/index.html`. last visited 2011-02-10.

[20] Andrew Kachites McCallum. Mallet: A machine learning for language toolkit. `http://mallet.cs.umass.edu/`, 2002. last visited 2011-02-10.