Adam Przepiórkowski

NuTech Solutions and Polish Academy of Sciences

INFORMATION EXTRACTION AND LEARNING OF TEMPLATE TYPES

The aim of this paper is to describe a prototype state-of-the-art Information Extraction (IE) system for English. This system is unique in at least two respects:

- although various techniques employed by this system have mostly been described in the literature, the way they are integrated here is novel;
- the system has been designed and implemented in Poland, at NuTech Solutions, Poland, unlike the vast majority of such systems, developed in the U.S. or Great Britain — to the best of our knowledge, it is the first such an IE system developed in Poland.

The relevance of this paper to practical applications of corpus linguistics is two-fold:

- first of all, the shallow syntactic parser used for text pre-processing was implemented on top of the corpus-based Brill tagger;
- learning of extraction templates takes place on the basis of pre-classified corpora.

As a result, this system illustrates two rather different applications of corpus linguistic in the Natural Language Processing (NLP) domain.

The outline of this paper is as follows:

Section 1 briefly describes the Information Extraction task, mentions a possible application of IE techniques and presents a general architecture of IE systems. Section 2 characterizes the pre-processing stage of the IE system presented here. Section 3 outlines the design of the IE engine proper, while section 4 discusses the learning component of this IE system which may at least partially obliviate the need for manual construction of IE templates.

1 Introduction

What is Information Extraction? IE is the task of finding relevant information in natural language (i.e., unstructured) texts. It should be distinguished from the Information Retrieval (IR) task, i.e., finding those texts which are relevant. Simple examples of IR are known to almost anybody who has ever used the Internet: Google, Alta Vista, Infoseek and other search engines are particularly simple IR tools. IE, on the other hand, turned out to be a much more difficult task and it is rarely encountered outside of research laboratories.

A typical IE scenario could be finding, within a large set of job offers, the information about the degree (M.Sc., Ph.D., etc.) required for the position and, say, the knowledge of programming languages assumed for the position. A tool automatically extracting this kind of information from job offers written in a natural language could be employed by an employment agency, e.g., as a part of a larger system matching job offers and CVs.

IE systems typically consist of three components:

- pre-processing: the raw text in which certain information is sought must be linguistically pre-processed to facilitate the actual IE task;
- IE proper: finding relevant information in pre-processed texts; this is usually done on the basis of so-called templates, i.e., characterizations of lexical and syntactic structures which carry the relevant meanings (see below);
- learning: in early IE systems, templates had to be manually specified by experts and linguists; current systems try to at least partially automate the process by implementing various machine learning techniques for semi-automatic discovery of such templates.

These three components of the system developed at NuTech Solutions are described in turn below.

2 Text preparation for IE

Even if IE engines only took into account the lexical contents of the text, some basic pre-processing, such as word tokenization (splitting the text into words and punctuation marks) and lemmatization (finding dictionary forms, or lemmata, of particular words) would be necessary. Usually, IE engines take into consideration (albeit to various degrees) also syntactic information, so sentence tokenization (splitting the text into sentences) and some kind of syntactic parsing are necessary.

This section briefly presents the main stages of text pre-processing within the NuTech Solutions prototype IE system. We ignore here word tokenization, as relatively uninteresting and depending on particular part-of-speech (PoS) marking system (so-called tagger), and sentence tokenization, as it relies on a number of heuristics, none of which is fail-safe. The next two subsections describe the PoS tagger / lemmatizer (section 2.1) and the shallow parser (section 2.2) employed within the system.

2.1 PoS tagging

The part-of-speech tagger used in the IE system is basically the classic tagger of Eric Brill (Brill 1993, 1994). The tagger is a general learning system which can learn the rules of morphosyntactic annotation on the basis of manually pre-tagged corpora. The tagger is freely available and it is distributed with a system of tagging rules learned on the basis of the Wall Street Journal corpus and, hence, can be used as an off-shelf tagger for English.

The functionality of this tagger is rather limited: it finds, for any word occurring in the text, its most probable part of speech in this context, as any tagger would, but — unlike

many taggers — it does not provide the lemma, or basic form, of the word. For this reason, it was necessary to prop up the tagger with a separate lemmatizer. To this end, the lemmatizing functions of WordNet (Fellbaum 1998), a hierarchical dictionary of English, were used.

On the technical note, Brill's tagger was completely re-implemented in Java (the original code was written in C and Perl), disambiguation rules learned on the basis of Wall Street Journal were manually improved, and WordNet C libraries were called from Java via the native interface. The integrated tagger/lemmatizer was implemented so as to produce well-formed XML tags. An example output of the tagger for the text *technical designs of applications and tools* is shown below:

```
<JJ lemma="technical">technical</JJ>
<NNS lemma="design">designs</NNS>
<IN lemma="of">of</IN>
<NNS lemma="application">applications</NNS>
<CC lemma="and">and</CC>
<NNS lemma="tool">tools</NNS>
```

The set of morphological markers used by this tagger, i.e., the so-called tagset, is that of The University of Pennsylvania (Penn) Treebank (http://www.scs.leeds.ac.uk/amalgam/tagsets/upenn.html). In the example above, `JJ` indicates an adjective, `NNS` — a common plural noun, `IN` — a preposition, and `CC` — a coordinating conjunction.

2.2 Shallow parsing

Semantics is often informed by syntax, so having syntactic information available facilitates the task of find relevant semantic information. For this reason, text pre-processing for IE usually involves some kind of syntactic processing.

On the other hand, full syntactic parsing is both expensive and difficult: usually full-fledged grammars, e.g., comprehensive context-free grammars, are computationally expensive and produce many different parses for a given sentence.

The usual approach to syntactic parsing in IE tasks is to employ so-called shallow parsing techniques, where context-free grammars are replaced by finite-state parsers (so-called finite-state automata or FSAs). The latter typically do not produce the full syntactic structures for given sentences, but they find various kinds of phrases (nominal phrases, verbal groups, etc.) and other syntactic constructions, which is all that is needed for typical IE tasks. Apart from not introducing massive ambiguities and being faster, FSAs are also easier to develop and adapt, as argued by Vilain (1999).

The shallow parser implemented within the NuTech Solutions IE system employs a cascade of finite-state automata, where:

- the first FSA parses the original PoS-tagged text and inserts information about the simplest kinds of syntactic constructs (say, contiguous verb groups such as *might have done*, conjunction "phrases" such as *and/or*, etc.);
- the second FSA parses the output of the first FSA and finds less simple kinds of constructs;
- ...
- the last FSA parses the output of the penultimate FSA and finds complex kinds of syntactic constructs.

The following example, presenting a sequence of FSAs (in the form of rewrite rules), illustrates this.

Num → (at (least|most))? CD

20; *at least twenty two*

ConjNum → **Num** ((, **Num**)* CC (more|less))?

twenty; *at least 20, 30 or more*

NumP → **ConjNum** (TO **ConjNum**)?

at least 20 to 40 or more

Noun → NN | NNP | NNS

university; *John*; *windows*

NounMod → **Noun** (POS)?

university; *John's*; *windows'*

AdjP → RB* JJ

v*ery nice*

NomPremodP → **NounMod**? (**AdjP**|**Noun**|**NumP**)+ lookahead(**Noun**)

John's twenty very interesting (books)

The first FSA annotates any numbers (cf. CD) perhaps preceded as *at least* or *at most* as Num. The second FSA presupposes the existence of such **Num** annotations and marks certain kinds of coordinate constructions of numbers as **CojnNum**. The third FSA takes the output of the second FSA and annotates constructions of the kind
20 to 40 as **NumP**, etc.

As the example above shows, the constructions found by this shallow parser are actually both of semantic and of syntactic nature. For example, **AdjP** is a syntactic construction, a simple adjectival phrase, where the adjective may be modified by any number of adverbs, while **NumP** is a construction that can be characterized in semantic terms, as a number or range.

Technically speaking, the parser schematically presented above has been implemented in Java via a sequence of substitution rules based on Perl-like regular expressions (Wall, Christiansen and Orwant 2000).

2.3 Example

The example below presents the result of the pre-processing of the phrase *technical designs of applications and tools selections*. Note that the output is a well-formed XML, where annotations are in the form of opening and closing tags:

(1)

```
<s_fullNP>
    <s_smplNP>
        <s_premd>
            <JJ lemma="technical">technical</JJ>
        </s_premod>
        <NNS lemma="design">designs</NNS>
    </s_smplNP>
    <s_smplPP>
        <IN lemma="of">of</IN>
        <s_smplNP>
            <NNS lemma="application">applications</NNS>
            <CC lemma="and">and</CC>
            <s_premod>
                <NNS lemma="tool">tools</NNS>
            </s_premod>
            <NNS lemma="selection">selections</NNS>
        </s_smplNP>
    </s_smplPP>
</s_fullNP>
```

The tags `JJ`, `NNS`, `IN` and `CC` are the output of PoS tagging, as described in section 2.1. The other tags, starting with `s_`, are the result of shallow parsing, with more nested tags (e.g., `s_premod`, as opposed to `s_smplNP`, `s_smplPP` and `s_fullNP`) corresponding to FSAs applied earlier in the cascade.[1]

---

[1] This example uses different symbols than the shallow grammar illustrated in section 2.2.

3 IE: Regular expressions, templates and template schemata

The previous section outlined the text pre-processing module of the IE system developed at NuTech Solutions. How is all this morphological and syntactic information taken advantage of in the task of finding the appropriate information?

The usual approach to the IE task is to use so-called templates, defined in terms of morphological and syntactic annotations inserted into the text during the pre-processing stage. Templates are best explained through a couple of examples.

Let us consider the following template:

(2) skills:

```
<extract><s_premod></extract><NNS lemma="skill">
```

This template says that the text corresponding to the `<s_premod>` tag occurring right before the noun *skill* (in plural, cf. `NNS`) should be extracted as possibly representing information of type "skill". This template, applied to an appropriately pre-processed sentence *Demonstrates strong technical and analytical skills.*, will extract the "premodifier group" *strong technical and analytical*. Given example (1) above, such a template could be used to extract the premodifier *technical* (if the value of lemma were "design") or *tools* (if the value of lemma were "selection").

Another, more complicated template could be:

(3) seek:

```
<lemma="seek"><extract><VBN>?(<s_fullNP>|<s_smplNP>)
<s_smplPP>|<TO><s_fullBVP>)*(<s_relClause>)?</extract>
```

This template can be used to extract text following any form of the word *seek* if this text corresponds to the following sequence: an optional past participle (cf. <VBN>?; e.g., *devoted*) followed by either a full nominal phrase (cf. <s_fullNP>) or (cf. |) simple nominal phrase (cf. <s_smplNP>), followed by any number (including zero; cf. *) of prepositional phrases or *to*-phrases (cf. <s_smplPP>|<TO><s_fullBVP>), followed by an optional relative clause. This template, applied to the (appropriately pre-processed) sentence *Small pharmacy store operation is currently seeking a pharmacist to work in the retail pharmacy.*, would extract the sequence *a pharmacist to work in the retail pharmacy* (*a pharmacist* would match the <s_smplNP> specification, while *to work in the retail pharmacy* would match <TO><s_fullBVP>).

A more complex sequence extracted by this template would be: *devoted* (<VBN>) *professionals* (<s_smplNP>) *with lots of ideas* (<s_smplPP>) *who are not afraid of challenges* (<s_relClause>). Of course, such extracted text would have to be further processed in order to find more atomic information about the professionals sought for this position.

The syntax of templates as used in the IE system described here is basically that of simple regular expressions, where round brackets ( ) are used to group expressions, the question mark ? marks the optionality of the preceding expression, the Kleene star * marks any number of repetitions of the expression, and vertical bar | marks alternative. Further, symbols in angle brackets <> match sequences marked with corresponding tags (morphological, e.g., <VBN>, <TO> or <lemma="...">, or syntactic, e.g., <s_fullNP>). The symbols <extract> and </extract> have the special meaning of delimiting the part of the text which contains the desired information.

The following subsection, 3.1, briefly describes how templates are translated into real regular expressions which do the actual job of finding the relevant passages in documents. This subsection may be skipped without harm for understanding the rest of this paper. Subsection 3.2, on the other hand, describes a way of abstracting over

templates; section 4, on automatic learning of templates, presupposes the material of subsection 3.2.

3.1 Templates and RegExes

Templates such as the two templates mentioned above are automatically translated into real regular expressions (actually, into Perl 5 regular expressions). This translation proceeds in such a way that simple regular expression markers like *, ? or | are preserved. On the other hand, morphological and syntactic tags are replaced by Perl regular expressions matching fragments of pre-processed text corresponding to these tags; for example, <NNS lemma="skill"> is translated to (?:(?:<[^/\s>]+(?:\s+[^>]+)?>\s*)*<NNS\s+ lemma="skill">[^<>]*</NNS>(?:</[^\s>]+ (?:\s+[^>]+)?>\s*)*), which is Perl's way of saying (roughly) "any number of closing or opening tags (but no real text; cf. (?:(?:<[^/\s>]+(?:\s+[^>]+)?>\s*)*), the opening <NNS lemma="skill"> tag (cf. <NNS\s+ lemma="skill">), anything that is not a tag (cf. [^<>]*), the closing <NNS> tag, and again any number of closing or opening tags".

These regular expressions involve various special Perl conventions, e.g., round brackets starting with ?:, i.e., (?:...) group expressions without copying the matching text to memory, while the usual round brackets (), absent in the example above, group expressions and copy the matching text to memory; also, \s means any white character and [^abc] means anything that is not a, nor b, nor c.

The mechanism of copying to memory the string matched by an expression within round brackets is crucial for extracting parts of regular expressions as those bits of text which represent the desired information. For example the templates in (4) will be translated into Perl regular expression (5), where parts corresponding to extracted texts are in bold face.

(4) <extract>**<s_premod>**</extract><NNS lemma="skill">

(5)

```
(?:(?:<[^/\s>]+(?:\s+[^>]+)?>\s*)*<s_premod
(?:\s+lemma="[^">]+")*>(?:(?!</s_premod>).)*
</s_premod>(?:</[^\s>]+(?:\s+[^>]+)?>\s*)*))(?:(?:<[^/
\s>]+(?:\s+[^>]+)?>\s*)*<NNS\s+lemma="skill">[^<>]*</N
NS>(?:</[^\s>]+(?:\s+[^>]+)?>\s*)*)
```

Of course, the bits of text extracted using this mechanism are fragments of the *pre-processed* text, so they may look like this: `</IN> <s_smplNP> <NNS lemma="application">`**applications**`</NNS> <CC lemma="and">`**and**`</CC> <s_premod> <NNS lemma="tool">`**tools**`</NNS> </s_premod> <NNS lemma="selection">`**selections**`</NNS> </s_smplNP> </s_smplPP> </s_fullNP>`. In order to get raw text, *applications and tools selections* in this case, it is necessary and usually sufficient to remove all tags.

3.2 Abstracting templates

Templates as described above correspond to particular verbatim textual realizations of semantic information. For example, in order to extract information about who hired whom, it is necessary to define a number of templates corresponding to different ways in which this information may be represented in texts. In particular, templates corresponding to the following textual realizations should be constructed:

<company> hired <person>;
<person> was hired by <company>;
<company>, which hired <person>;
<person>, hired by <company>, etc.

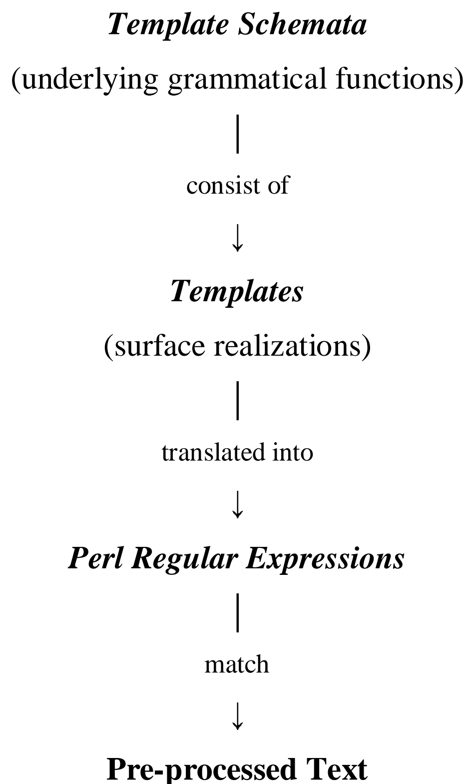
But just as templates can be regarded as abstractions over the nitty-gritty of actual texts, it is possible to define lexical semantic abstractions over particular templates (see Grishman

1997). We will call such abstractions at the level of deep grammatical functions *template schemata*. For example, the four templates above correspond to the following template schema:

deep subject = <*company*>
verb = *hire*
deep object = <*person*>

Thus, a template schema can be understood as a set (or an equivalence class) of templates anchored by forms of the same lexeme (*hire* in the example above) and involving the same underlying grammatical functions (deep subject and deep object above).

The three abstraction levels assumed by the IE system are schematically depicted below:



4 IE: Learning templates

The quality of the approach sketched above crucially depends on the quality and number of templates. Constructing such templates manually is a task which requires some linguistic (and domain) expertise and is usually time consuming; according to various reports in Pazienza 1997, 1999, it is the most time consuming task in many IE projects. Also, such a manual template construction is error-prone and suffers from adaptability problems — given an IE system with templates developed for one domain (e.g., the domain of job offers), it is necessary to develop a new set of templates in order to apply the same IE engine to a different domain (e.g., corporate moves). It is clear that construction of appropriate templates is a serious bottleneck in IE projects and some automation of this process is required.

A common solution to this problem has been to adopt machine learning techniques well-known from corpus linguistics: to manually annotate a corpus of relevant texts with template-like information and then to automatically learn templates from such corpora. The problems that this solution faces resemble the difficulties encountered in machine learning of PoS taggers: the quality of inferred rules crucially depends on the size of the corpus, and manual annotation is expensive, even more so in the case of template-annotation than in the case of PoS-annotation. A more robust solution is necessary.

4.1 Learning by pre-classification

One such robust solution was originally proposed by Riloff 1996 and it is based on the idea of learning from pre-classified texts.

The crux of this method is to classify a set of texts into two subcorpora: texts relevant to the intended domain of IE (e.g., job offers) and all other texts.[2] Then, assuming that the space of possible templates can be constrained (more on this below), occurrences of template matches for each possible template are counted for both sets of texts and those templates which are frequent in relevant texts but infrequent in the other set of texts are considered potentially useful — they can be used to extract information specific to the given domain.

The following subsection describes the instantiation of this general idea that has been implemented within the NuTech Solutions IE system. What is novel in this approach is the fruitful combination of Riloff's learning by classification with Grishman's idea of abstracting over templates.

4.2 Discovering templates by finding template schemata

The approach sketched in the previous subsection crucially depends on there being a method of defining a finite space of possible templates; once the set of possible templates is finite (and not too large), matches of particular templates in the two sets of texts can be counted and the counts compared.

In order to define such a space of possible templates, we will first introduce the notions of template *types* and schema *types*. The relation of templates and schemata to template types and schema types is illustrated in the table below:

	specific	**schema**
template	require(s)? NP NP (is)? required	require + Obj

[2] Riloff reports that best results are achieved when those other texts are similar to the domain texts, but she does not specify in what way they should be similar.

	...	
type	<anchor>(-s)? NP NP (is)? <anchor>-ed ...	<anchor> + Obj

We will use the notions *template* and *specific template* interchangeably, and we will call *template types specific types*.

As can be seen in the table above, template types are simply templates with the lexical content (in this case, *require*) replaced by <anchor>. By substituting different lexical elements (different verbs in the case at hand) for <anchor> in the template type <anchor>(-s)? NP, we get different specific templates, e.g., require(s)? NP, want(s)? NP, adore(s)? NP, etc. Thus, mathematically speaking, template types are functions from lexical elements to templates. Similarly, schema types are functions from lexemes to template schemata.

Now, it should be clear that a set of schema types defines a space of specific templates: by substituting, in each schema type, particular lexemes for <anchor>s, we get a set of specific template schemata, each of which corresponds to a set of templates. So, in order to construct a space of possible templates, it is sufficient to construct possible schema types, i.e., essentially, possible ways of expressing relations in a natural language, such as (deep) Subj + <anchor> + (deep) Obj, <anchor> + (deep) Obj, etc.

Given the notions introduced the, above algorithm for learning of templates relevant to a given domain is following:

0. Define possible schema types and, for each schema type, define the set of specific types which textually realize this schema type. This is a manual and time-consuming

process, but it is task-independent, i.e., it is performed once for all uses of the IE system in different domains. This is how the adaptability problem mentioned above is evaded.

1. For a given IE task (for a given domain), pre-classify a corpus of texts into two subcorpora: texts relevant to the IE task at hand and texts irrelevant to this task. This is a manual task, but it is not time-consuming.

2. For each schema type T defined in step 0,

    for each lexeme L that can fill the `<anchor>` role in T,

        set two counters (corresponding to occurrences in relevant and irrelevant subcoropra), <T,L,R> and <T,L,IR> to 0.

3. For each subcorpus,

    for each specific type ST defined in step 0,

        find all matches of ST in the subcorpus, i.e.,

        for each lexeme L than can fill the `<anchor>` role in ST,

            find the number of occurrences of the ST(L) template in the corpus. (Note that we understand here specific types as functions from lexemes to templates, as mentioned above.)

4. If the template ST(L) occurs n times in the corpus and the specific type ST is an instantiation of the schema type T, then increase the appropriate counter for T and L (i.e., <T,L,R> or <T,L,IR>, depending on the subcorpus) by n.

5. For each schema type T and for each lexeme L,

    compare the number of matches of the template schema T(L) in both subcorpora and, on this basis, statistically infer how relevant the template schema T(L) is for the IE task at hand.

6. The most relevant templates are candidates for useful templates and should be manually validated as such (not time-consuming). This algorithm can be implemented relatively efficiently.

4.3 Examples

The learning algorithm presented above was tested on a corpus containing 20K words, with equal-sized (10K words each) subcorpora of relevant texts (job offers) and irrelevant texts (news).

Some of the best template schemata discovered are listed below, together with examples of bits of texts matching these template schemata:

- `<premodifiers>` + *skills*:
    - *good interpersonal and communication skills*
    - *demonstrated well-developed interpersonal, communication and Teaming Skills*
    - *strong inter-personnel and communication skills*
    - *excellent project management, organizational and analytical skills*
    - *strong mathematical skills*
- *require* + `<object>`:
    - *various C-130 AMP system PDRs / CDRs will be required*
    - *requires employees to work*
    - *California licensure is required*
    - *a minimum of 4 years engineering experience is required*
    - *require ingenuity and innovation*
    - *broad latitude in determining technical solutions is required*
    - *MBA required*
- *look for* + `<object>`:
    - *looking for a quick learner, talented individual*
    - *looking for qualified Retail Pharmacists for short and long-term assignments in the Toledo, Ohio area*

    - *looking for someone who has strong communication skills and the ability to work with others and is detail oriented*
- *manage* + `<object>`:
    - *managing expectations and conflict resolution*
    - *manage specific marketing projects and coaches*
    - *manage positive customer relationships*

4.4 Preliminary evaluation

A preliminary prototype of the approach outlined above has been implemented and tested on a small corpus of 20K words. It is too early to fully evaluate it, but preliminary tests suggest the following advantages and possible disadvantages.

Advantages:

- Very quick learning of common patterns in new areas; this deals with the difficult problem of adaptability of IE systems;
- so, a prototype IE system can be developed rapidly for a new domain.

Possible disadvantages:

- the quality of this approach crucially depends on the quality of a robust tagger and a reliable shallow parser; this, however, can be said about most IE systems;
- more ephemeral and idiosyncratic constructions (`<degree>` in..., e.g., *B.S. in Engineering*) cannot easily be learned; they must be added manually;
- so fine-tuning of discovered patterns and developing new patterns manually still necessary for high-quality IE systems.

It seems that the approach to IE and machine learning of IE templates described above is especially valuable when a general purpose IE engine is developed intended to be applied to many different domains. In such cases, the combination of Grishman's idea of abstraction over specific templates with Riloff's approach to learning by pre-

classification, supported by the power of Perl regular expressions, leads to a particularly robust IE system.

REFERENCES

Brill, E. (1993). A Corpus-Based Approach to Language Learning. Unpublished Ph.D. dissertation, University of Pennsylvania.

Brill, E. (1994). "Some Advances in Transformation-Based Part of Speech Tagging." In: Proceedings of AAAI-94.

Fellbaum, Ch. (ed.). (1998). WordNet. An Electronic Lexical Database. Cambridge, MA: MIT Press.

Grishman, R. (1997). "Information Extraction: Techniques and Challenges." In Pazienza 1997. 10–27.

Pazienza, M.T. (ed.). (1997). Information Extraction: A Multidisciplinary Approach to an Emerging Information Technology. Berlin: Springer-Verlag.

Pazienza, M.T. (ed.). (1999). Information Extraction: Towards Scalable, Adaptable Systems. Berlin: Springer-Verlag.

Riloff, E. (1996). "Automatically Generating Extraction Patterns from Untagged Text." In: Proceedings of AAAI-96. 1044—1049.

Vilain, M. (1999). "Inferential Information Extraction." In: Pazienza 1999. 95–119.

Wall, L., Christiansen, T. and J. Orwant. (2000). Programming Perl. Sebastopol: O'Reilly.

KEYWORDS: **Information Extraction**, **corpora**, **NLP**, **machine learning**, **shallow parsing**, **PoS tagging**.