A Bootstrapping Architecture for Time Expression Recognition in Unlabelled Corpora via Syntactic-Semantic Patterns

Jordi Poveda
jpoveda@lsi.upc.edu

Mihai Surdeanu
surdeanu@lsi.upc.edu

Jordi Turmo
turmo@lsi.upc.edu

LSI Department Technical Report
Ph.D. Programme on Artificial Intelligence (UPC)
June 2007

Abstract

In this paper we describe a semi-supervised approach to the extraction of time expression mentions in large unlabelled corpora based on bootstrapping. Bootstrapping techniques rely on a relatively small amount of initial human-supplied examples (termed “seeds”) of the type of entity or concept to be learned, in order to capture an initial set of patterns or rules from the unlabelled text that extract the supplied data. In turn, the learned patterns are employed to find new potential examples, and the process is repeated to grow the set of patterns and (optionally) the set of examples. In order to prevent the learned pattern set from producing spurious results, it becomes essential to implement a ranking and selection procedure to filter out “bad” patterns and, depending on the case, new candidate examples. Therefore, the type of patterns employed (knowledge representation) as well as the ranking and selection procedure are paramount to the quality of the results. We present a complete bootstrapping algorithm for recognition of time expressions, with a special emphasis on the type of patterns used (a combination of semantic and morpho- syntantic elements) and the ranking and selection criteria. Bootstrapping techniques have been previously employed with limited success for several NLP problems, both of recognition and classification, but their application to time expression recognition is, to the best of our knowledge, novel. As of this writing, the described architecture is in the final stages of implementation, with experimention and evalution being already underway.

Contents

1 Introduction

The problem of time expression recognition refers to the identification in free-format natural language text of the occurrences of expressions that denote time (e.g. "*Yesterday*'s polls unconclusive to foretell a winner to the presidential election to be held *next summer*." or "Sources from the company have confirmed that the merger will finally take place at *an undisclosed date*."). This task is closely related with event and relation recognition (e.g. "*German giant E.ON*'s board of directors announces plans for *takeover* of *Spanish ENDESA* for *$20 million* just *after* receiving *former CEO Bernotat's resignation* notice." would yield one TAKEOVER event (acquirer(E.ON), target(ENDESA), amount($20 million)), one RESIGNATION event (company(E.ON), person(Bernotat), position(CEO)) and one PRECEDES temporal relation between the former events). Several NLP subtasks benefit from the ability to extract such structured information from text as time expressions, events and relations; Information Extraction (IE), Question Answering (QA) and Autommatic Summarisation (AS) being only the most prominent ones. Moreover, time expression recognition —with subsequent "understanding" of the denoted date, time or period and representation in a normalized format— can be used as additional clues for event and relation extraction, and to provide a temporal ordering of the events in text.

Time-denoting expressions appear in a great diversity of forms, beyond the most obvious absolute time or date references (e.g. *11pm*, *February 14th, 2005*): time references that anchor on another time (*three hours after midnight*, *two* *weeks before Christmas*), expressions denoting durations (*a few months*), expressions denoting recurring times (*every third month*, *twice in the hour*), context-dependent times (*today*, *last year*), vague references (*somewhere in the middle of June*, *the near future*) or times that are indicated by an event (*the day G. Bush was reelected*). It is beyond the scope of this paper to introduce time expression recognition in more depth: refer to the Introduction in [4] for a more thorough presentation of the problem.

We present here a bootstrapping framework and a full algorithm for extraction of IE patterns that allow for the identification of time expressions (as in the above examples) in untagged corpora. In our context, bootstrapping refers to a semi-supervised machine learning approach that makes uses of the slightly supervised knowledge provided by a small initial set of hand-supplied examples (called "seeds") in order to *bootstrap* (i.e. initialize) a set of patterns or rules. These patterns can be used to identify or classify further examples in unlabelled data, which are then used to generate further rules that cover the new examples, and so on. This iterative approach is used to grow the set of patterns in which would otherwise be a fully unsupervised process, were it not for the small amount of initially assumed-correct examples (from a few units to the several hundreds, depending on the difficulty of the task and algorithm construction). Such a learning setup incurs in a number of typical machine learning tradeoffs:

- **Pattern representation**: The types of patterns used limit the algorithm's ability to generalize from the token-level information in the examples. However, allowing otherwise too-general patterns would lead to the pattern set producing spurious examples early in the learning process. This problem is equivalent to gauging the complexity of the hypothesis representation language in classical machine learning.

- **Ranking and selection**: Just as fully unsupervised learning needs a notion of "similarity" or distance to produce relevant clusters, this bootstrapping framework needs ranking (i.e. assigning scores) and selection procedures to guide the learning process. Again, a too stringent selection would cause relevant examples to be lost, whereas a too relaxed one would lead to spurious examples being accepted early.

- **Initial seed set**: The initial set of seed examples is the only bit of information supplied to the learning algorithm about the target concept. This, together with pattern representation, totally determines the quality and range of examples that the bootstrapping will be able to capture. Also, it provides a mechanism to act on the desired level of "supervision".

In our particular case, the seeds are a number of sample time expressions that are assumed to be correct positive examples. Patterns take the form a combination of morphological, semantic and syntactic generalizations that "cover" the examples from which they are generated as well as other related instances, already seen or otherwise. Patterns are generated from features of the individual tokens in the (alleged) time expression, or from features of the basic syntactic chunks of which it is made up.

The use of a bootstrapping approach for time expression recognition is justified, in part, by the limited availability of corpora annotated with time information from which to train supervised learners for this task. Learned IE patterns can be used standalone to annotate new unlabelled text, or they can be used in combination with other machine larning methods.

The rest of this document is structured as follows. Section 2 briefly discusses previous references in the literature of bootstrapping approaches being used to tackle problems in NLP. Section 3 provides a general overview of the bootstrapping framework employed and the workings of the bootstrapping algorithm, while the following sections deal with specific parts of the algorithm in detail: the representation of patterns and types of generalizations used (Section 4), the ranking and selection procedure for new candidate examples (Section 5) and the ranking and selection procedure for candidate patterns (Section 6).

2 Literature Review

The use of bootstrapping approaches for NLP-related tasks is not something new. On the contrary, there exists a rich body of literature covering the use of bootstrapping techniques for such diverse tasks as: word sense disambiguation ([10]), named entity classification ([1]), IE pattern acquisition ([5], [9], [8], [6]), text document classification ([7]) and fact extraction from the web ([3]).

The general idea of bootstrapping is a constant, but particular implementations differ in the details of the bootstrapping algorithm (sometimes introducing variations such as the simultaneous *co-training* of two classifiers), the nature of the rules or patterns that the algorithm learns, and the exact basis on which candidate patterns are selected. The features used for constructing rules or patterns and their representation are largely conditioned by the task which they aim to solve, and this representation affects —to some extent— how selection is performed. To the best of our knowledge, bootstrapping has not been used in the past for time expression recognition (although it could arguably be considered a particular case of IE pattern acquisition), and some elements of pattern representation and pattern ranking and selection will also introduce novelties from previous instances.

Yarowsky (1995) [10] used bootstrapping to train decision list classifiers to disambiguate between two senses of a word, achieving impressive classification accuracy. In his approach, contextual information about the words surrounding a target polysemic word up to a certain distance (i.e. collocations) is used as patterns for disambiguation. The seeds are given by a few selected collocations among those found in an untagged corpus, which are tagged with the correct word sense to give the algorithm an initial rule list. This paper also briefly discusses the effect of the number and adequacy of the seeds on performance.

Collins and Singer (1999) [1] devised a framework for named entity (NE) classification, in which set of rules for a decision list classifier is bootstrapped from a few handcrafted seed rules. Rules use either lexicographic (*spelling*) features from the named entity (e.g. whether it contains a given token, or patterns of capitalization and/or hyphenation) or contextual features, and assign one of the possible entity-type tags (e.g. PERSON, ORGANIZATION and PLACE). New candidate rules are ranked by their confidence, whose calculation takes into account previously classified NEs. Their main innovation with respect to Yarowsky's algorithm is that training is split in two stages that take place in alternation: during one step, rules that use only contextual features are sought; during the second step, the new contextual rules are used to tag further examples and these are used to grow the rule set with spelling rules only, and so on. It is important to note that Collins and Singer's architecture performs only classification of NE, not recognition: the place of occurrence of NEs in the text is already given to the algorithm. In this respect, the task is different from ours of time expression recognition. In the same paper, Collins and Singer suggest a variation of the bootstrapping algorithm that uses a co-training based on boosting ideas.

Surdeanu et al. (2006) [7] present a bootstrapping framework for document classification which also combines the idea of co-training (training simultaneously two different classifiers on two different views of the same data) with bootstrapping, as a means of increasing final precision without damaging recall. Their approach consists in training, in successive iterations of the bootstrapping algorithm: a decision list learner, which uses the presence of syntactic-semantic patterns in the text to decide on a document class label; and an Expectation Maximization classifier which uses the document words as features. The paper goes on to compare the performance of different criteria for ranking and selecting patterns in the decision list model.

Bootstrapping approaches are employed in Riloff (1996) [5], Yangarber et al. (2000) [9], Yangarber (2003) [8], and Stevenson and Greenwood (2005) [6] in order to find IE patterns for domain-specific event extraction. Here, patterns take the form of syntactic tuples representing common grammatical occurrences (such as <Subject>-<Verb>-<Object>, <Subject>-<passive verb>-<Agent> or <Noun>-be-<Adjective>), in which one or more of the general syntactic markers have been lexicalized, that is, substituted by specific words which are observed to indicate the presence in text of an event in the target domain. Both Riloff's AutoSlog [5] and Yangarber's ExDISCO [9] frameworks exploit extensively the idea of *redundancy*: extraction patterns (expressions) that are relevant to the target domain will appear frequently in documents that belong to that domain, and infrequently in documents that are irrelevant to the domain. This idea is taken to the point in Yangarber's framework that documents are partitioned in relevant or irrelevant at each bootstrapping iteration, according to their containing patterns that have been "accepted" as relevant, and the relevant documents are in turn searched for new candidate patterns. Both frameworks use some measure of confidence to rank new candidate patterns, which includes a part related to *precision* (accuracy in predicting the presence of a relevance event), and a part related to *coverage* (actual usefulness of the pattern). In Riloff (1996) [5] the algorithm is seeded by providing some sample passages of text which contain examples of the target events, whereas in Yangarber et al. (2000) [9] the seeds are in the form of a few sample correct patterns which extract the intended event.

Stevenson and Greenwood (2005) [6] departs from a document-centric approach to evaluating the relevance of patterns, and introduces *semantic similarity* with respect to the set of "accepted" patterns as a measure of the confidence for new candidates. The idea of *redundancy* continues to be present here, only under a different hood: patterns which are relevant for a domain will tend to contain semantically similar words. Similarity is evaluated with respect to an external ontology.

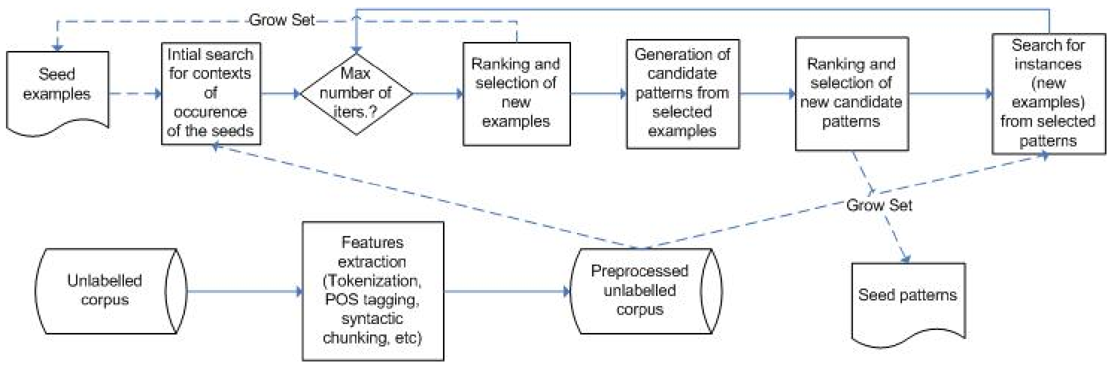
A somehow similar stance is taken in Paşca et al. (2006) [3], where the bootstrapping process is used to extract general facts from the Web. Facts are viewed as two-term relationships: for instance, the pair (Donald Knuth, 1938) could be an example of a Person-born in-Year fact. Patterns are taken to be the sequence of tokens that occur between the left and right terms of a fact in the documents (e.g. "a renowned computer scientist born in" in the former example). Semantic similarities —as provided by a corpus of pre-computed pairwise distributional similarity scores among words— are present in two respects: firstly, extracted patterns are made more general by "promoting" some of their words to their *semantic class* (i.e. an element that matches any of a set of distributionally similar words); secondly, new candidate examples are ranked primarily on the basis of their semantic similarity with the set of "accepted" examples (*seeds*). Their algorithm uses ranking and selection both of new candidate examples (through a combination of semantic similarity and several other factors) and of candidate patterns (by the frequency of their components). The seeding consists of a few initially given examples of the target fact, in the form of pairs. A point is made of following an eager strategy for growing the set of patterns, by allowing a large amount of patterns in each iteration and thus reducing the need for computationally- expensive passes through a very large corpus (the Web). This is in constrast with traditional "cautious" strategies, which advocate growing the pattern set by only a few new patterns each iteration.

3 General Pattern Acquisition Framework

This section describes the general blocks of our bootstrapping algorithm and how the different parts fit together. More in-depth details about the workings of individual parts are given in the next sections. Figure 1 illustrates the building blocks of the algorithm and their interactions, along with input and output data.

3.1 A Sample Iteration of the Bootstrapping Algorithm

Our bootstrapping algorithm works with two alternative views of the same target data (time expressions), that is: *patterns* and *examples* (the latter we will also refer to as an *instance of a pattern*). A *pattern* is a generalized view of data in the training corpus (sequences of tokens) such that, when matched against



Grow Set
Seed examples
Intial search for contexts of occurence of the seeds
Max number of iters.?
Ranking and selection of new examples
Generation of candidate patterns from selected examples
Ranking and selection of new candidate patterns
Search for instances (new examples) from selected patterns
Grow Set
Unlabelled corpus
Features extraction (Tokenization, POS tagging, syntactic chunking, etc)
Preprocessed unlabelled corpus
Seed patterns

Figure 1: Block diagram of bootstrapping algorithm

the text in the corpus, we obtain the exact sequences of tokens where a possible time expression is found —that is, the *extension* of a candidate time expression mention—. An *example* is an actual candidate occurrence of a time expression. Patterns are generated from examples found in the corpus (see Section 4 for more on this topic) and, in its turn, new examples are found by searching for matches of new patterns in the corpus. Patterns and examples are by-products of each iteration of the algorithm.

Both patterns and examples carry contextual information, that is, they include the extension of a possible time expression (whether the actual tokens or a generalization that yields a time expression when instantiated) plus a window of tokens in the left and right context of the time expression. The length of this context window is a configurable parameter of the algorithm. Context information is included because it allows for more specific patterns, where this additional "constraintness" may make it possible for a pattern to further separate cases where a certain sequence of tokens (or token features) constitutes a time expression from cases where it does not —by means of the additional information provided by the context—. Examples are also stored with the context where the candidate time expression occurs, but the context information in the case of examples serves no specific purpose other than facilitating the generation of patterns with that added contextual information.

"Seed examples" and "seed patterns" are the outputs of the bootstrapping process. Both the set of "seed examples" and the set of "seed patterns" are grown (i.e. increased) with each new iteration, by adding the new candidate examples (respectively, patterns) that have been "accepted" during the last iteration (i.e. those that have passed the ranking and selection step). A distinction is in order here to clarify the difference in the use of the term *example* within the domain of the bootstrapping algorithm, and the *seed examples* that are provided as input to the algorithm and obtained as an output in addition to the initial set. An *example* as is used internally by the bootstrapping algorithm encapsulates left and right context tokens together with all the relevant token features (i.e. token form, POS tag, syntactic chunk and head) of the candidate time expression, because all this information is put to use in the task of pattern generation; conversely, when the algorithm adds new *seed examples* to the growing seed example set (as part of the output of an iteration), we are only concerned with the actual tokens that integrate the time expression proper (i.e. neither the context nor the token features).

Initially, a single pass through the corpus is performed in order to find occurrences of the seeds in the text. Thus, we bootstrap an initial set of *examples*, one for each appearance of a seed example in the text with its accompanying context of occurrence. This single pass can be considered as *iteration zero*. From then on, the bootstrapping process consists of a succession of iterations with the following steps (Figure 1):

1. **Ranking and selection of examples**: Each example produced during any of the previous iterations, 0 to $i-1$, is assigned a score (ranking), which takes into account several factors regarding both the candidate time expression itself (the *extension* of the mention) and its context tokens. The top n examples are selected to grow the set of seed examples (selection) and will be used for the next step. Ranking and selection of examples is explained in detail in Section 5.

2. **Generation of candidate patterns**: A pool of candidate patterns for the current iteration is generated from the selected examples of the previous step. The generation of patterns uses the token features from the candidate time expression contained in the pattern, and its context. One example produces (at most) t patterns (t being the number of different types of patterns used), so the cardinality of the mapping from examples to patterns is $1 \rightarrow t$. Pattern generation is further developed in Section 4.

3. **Ranking and selection of candidate patterns**: Similarly to examples, each pattern from the *current* iteration is assigned a score and the top m patterns are selected to grow the set of seed patterns and to be used in the next step. Ranking and selection of patterns is explained in greater detail in Section 6. Actually, the ranking and selection of candidate patterns can be performed in a variety of ways, which greatly affects final results of the bootstrapping algorithm. We have made this procedure modular and experimented with several different ranking and selection procedures.

4. **Search for instances of the selected patterns**: The training corpus is iterated over, in order to search for instances (matches) of the selected patterns, which will form the set of cantidate examples for iteration $i+1$. We iterate over every token in the corpus and, for each position, attempt to match each pattern to a sequence of tokens starting at the current token. Every successful match becomes a candidate time expression occurrence (i.e. a example with context information) for the next iteration.

The initial search for candidate examples of *iteration zero* has worst-case temporal cost of $\mathcal{O}(NM)$, with N being the length of the corpus in tokens and M the number of seeds in the initial seed set. Ranking and selection of examples has cost $\mathcal{O}(N \log(N))$, with N the number of candidate examples. Generation of candidate patterns has cost linear with respect to the number of selected examples. The search for instances of selected patterns, has a worst-case complexity of $\mathcal{O}(NMl)$, with N being the length of the corpus in tokens, M the number of selected patterns to test, and l the average length (number of elements) of a pattern. But in practice, due to the use of a hierarchical data structure to aggregate patterns with common elements and to the indexing of patterns according to the token features they can match, the complexity of searching for pattern instances has been reduced to cost nearly $\mathcal{O}(N)$. Lastly, ranking and selection of patterns has cost $\mathcal{O}(M \log(M))$, with M the number of candidate patterns, but has worst-case cost as large as that of a search for pattern instances if a precision score for patterns is used as part of the ranking, because calculating the precision of patterns makes use of searching for pattern instances in (a subset of) the training corpus.

Notice that examples from the last iteration are accumulated to the examples from all of the previous ones, not discarded from one iteration to the next. This is done in order to prevent the quality of the set of examples (and consequently, of patterns) from degenerating excessively with each iteration, by allowing older examples (which will normally be of higher precision, since they are more closely related to the initial set of seeds) to compete with the new ones. Thus, we are giving older examples the chance to be re-selected and stay in the active set if they score better than the newer ones. This is in constrast to patterns, which are not conserved from one iteration to the next (save for the selected ones which are stored as output into the "seed patterns" file).

Also, two additional considerations are taken into account in order to relax the matching of patterns to corpus tokens and of token forms among them. One is doing the matching of token forms case-insensitive, and the second one is generalizing all the digits in a token to a generic digit marker (so, for instance the token "12-23-2006" is internally rewritten as "@@-@@-@@@@", and it would match a comparison against the token "03-13-1995", but would not match the token "12-1-2006"). Alternatively, any amount of consecutive digits can be made to match any other sequence of only digits (so that "12-01-2006" would match "12-1-06", for instance). These are implemented as configurable execution parameters to the bootstrapping algorithm. They affect not only the matching of patterns to the text, but also the ranking of examples and patterns when they are compared against the set of "accepted" examples and/or patterns, or their frequency is computed. These relaxed assumptions allow for increased generalization in the patterns produced.

3.2 Training Corpus

As unsupervised data for our bootstrapping experiments, we use LDC's ACE 2005 Unsupervised Data Pool, a 511 Mbytes untagged corpus consisting of 233K documents from seven different text categories. Among these, and for reasons of ensuring, to the degree possible, homogeneity in the distribution of the training data, we have chosen to work only with the largest of these seven categories, NW (Newswire) news feeds. It alone contains 456 Mbytes of data in 204K documents, comprising a total of over 82 million tokens. The percentage of tokens belonging to time expressions (our recognition target) has not been calculated, as they are not labelled in the corpus. For reference, this percentage accounts for 3.42% in the much smaller ACE supervised data set — time expressions are visibly sparse in the whole of the text.

The corpus has been preprocessed in order to tokenize the texts and extract a series of features at the token level, which are later employed during the learning process for pattern construction. The preprocessed corpus totals 1.9 Gbytes of data (only the NW section), and is arranged in a tabular format of one token (with all its extractd features) per line. These features are:

1. The token form (i.e the token itself).
2. The POS (Part-Of-Speech) tag, computed with the TnT tagger[1].
3. The lemma (e.g. families $\rightarrow$ family, grew $\rightarrow$ grow, ...), computed using the WordNet lemmatizer.
4. Basic syntantic chunk (non-recursive, non-overlapping) to which the token belongs (e.g. NP for noun phrase, PP for prepositional phrase, etc). These are computed as a sequence of BIO tags qualified with the chunk type using the YamCha tool[2] (a general-purpose chunker based on SVMs).
5. The lemma form of the "semantic head" of the syntactic chunks, computed by means of a script that uses ad-hoc heuristics (e.g. the head of a NP is its last token). The "semantic head" distinction is made because, in some occasions, the word that carries the meaning of the syntactic phrase is different from its syntactic head (e.g. have agreed to *overturn*).

3.3 Initial Set of Seed Examples

The second input to the bootstrapping algorithm (besides the trainig corpus) is a set of *seed examples*, consisting of a series of strings representing correct time expressions. The seeds are supplied with token-level information only (i.e. without the features outlined above), and without contextual information — even when the algorithm itself uses contextualized patterns, in order to harness the additional "constraintness" permitsted by taking into account the context of appearance—.

Seed patterns are selected semi-automatically from among a ranked list of the time expressions that appear in the ACE supervised corpus (this corpus contains a total of 2085 distinct time expressions, 4607 including repeated occurrences). Initially, we are working with a seed examples file of about 150 time expressions.

[1]http://www.coli.uni-saarland.de/~thorsten/tnt/
[2]http://chasen.org/~taku/software/yamcha/

The criterion for choosing the seeds is to rank the time expressions according to their precision or according to their frequency, and then to select the top n expressions while:

- avoiding having repeated instances of expressions that are quite similar semantically and/or syntactically (e.g. *last February* and *next January*), in order to produce a seed set as representative of the whole data distribution as possible while using fewer seeds,
- manually including some instances of the "odd" cases (e.g. *just one day after being charged with a felony*), to make sure the bootstrapping process has some chance of ever capturing that type of expressions.

A time expression has better precision the more times any occurrence of the sequence of tokens of that time expression in the supervised ACE corpus is actually tagged as being a time expression. Initially choosing the seeds with high precision aims for results with higher precision at the expense of some loss in recall, whereas choosing a seed set with high recall trades off better recall for worse precision. Provided that the overall precision of the pattern set tends to naturally decrease with each new bootstrapping iteration —with each iteration, it becomes increasingly likely that new patterns and examples will be accepted that drift from the correctness of the initial seed set, due to the unsupervised working of the algorithm—, it is generally a safe bet to start with a high-precision seed set.

4 Pattern Representation

Patterns capture a generalized view from a sequence of tokens in the text, and are constructed from a combination of morphological, semantic and syntactic features of these tokens. A pattern will match the sequence of tokens from which it was generated, plus an additional undefined number of different sequences (related to the generality of the pattern).

Same as the examples with which the algorithm works internally, patterns capture both the sequence of tokens that integrate a potential time expression (i.e. the *extension* of a time expression mention), and contextual information from the left and right context where a potential time expression occurs (up to a bounded length). Let us call *prefix* the part of the pattern that represents the left context, *infix* the part that represents a potential time expression mention and *postfix* the part that represents the right context. This terminology is suggested in [3], although their pattern representation only uses the infix part.

A example of a EBNF grammar that encodes our pattern representation is given in Figure 2. Patterns are composed of multiple *pattern elements*. The prefix and the postfix have zero or more pattern elements, whereas the infix has at least one pattern element. A pattern element is the minimal unit that is matched against the tokens in the text, and a single pattern element can match to one or several tokens, depending on the pattern element type. A

```
pattern ::= prefix FIELD-SEP infix FIELD-SEP postfix
prefix ::= (UNDEF-MARKER)* (pattern-elem)*
infix ::= (pattern-elem)+
postfix ::= (pattern-elem)* (UNDEF-MARKER)*
pattern-elem ::= FORM-MARKER token-form FORM-MARKER |
    SEMCLASS-MARKER token-form SEMCLASS-MARKER |
    POS-MARKER pos-tag POS-MARKER |
    SYNTAX-MARKER syn-chunk-type "(" head ")" SYNTAX-MARKER |
    SYNTAX-SEMC-MARKER syn-chunk-type "(" head ")" SYNTAX-SEMC-MARKER
```

Figure 2: A EBNF Grammar for Patterns

pattern is considered to match a sequence of tokens in the text when: first, all the pattern elements from the infix are matched (this gives the potential time expression mention) and, second, all the pattern elements from the prefix and the postfix are matched (this gives the left and right context information for the new candidate example, respectively).

There are two basic types of pattern element and, apart from these, several other subtypes exist that classify into either of these two types. We have defined an initial set of pattern element types, but this list could be expanded in the future with new types. The basic division of pattern elements is among:

- **Token-level generalizations**: These are pattern elements that have been generated from the features of a single token. These pattern elements will normally match a single token as well, but depending on the case, they may match a sequence of multiple tokens.
- **Chunk-level generalizations**: These are pattern elements that have been generated and group a sequence of several tokens (a chunk). In our implementation, these correspond to basic syntactic chunks, but pattern elements to represent other types of chunk are also possible. For instance, Named Entities (NE) could be generalized into a pattern element that matches any NE of the same type (e.g. PERSON). These pattern elements match one or several tokens in the text.

There exists a special type of pattern element that does not fit into any of the above two groups, because it is not generated from any token: the *undefined* pattern element. This pattern element can only appear in the prefix (before every other non-undefined pattern element) or in the postfix (after every other non-undefined pattern element). It does not match any token from the text, but it is used to represent explicitly the fact that there is no further left or right context (because the complete pattern matches a time expression mention that occurs near the beginning or the end of the corpus) or, if the continuity of chunks is broken at encountering an end-of-setence (i.e. the target entities, time expressions in our case, cannot span over the boundaries of a sentence), the fact that there is no further left or right context due to having reached the beginning or the end of a sentence, respectively. Notice how this is diferent from an empty prefix/postfix or one with fewer pattern elements: a prefix or postfix with fewer elements does not impose any restriction regarding what comes before/after its first/last pattern element; an *undefined* pattern element is more restrictive, since it only matches in the cases we just mentioned.

The *semantic similarity class* of a word is defined as the word itself plus a group of other semantically similar words. For computing semantic similarity classes, we employ Lin's corpus of pairwise similarities among words [2]. This corpus contains a list of pairwise similarity scores of a word to other related words, for nouns, adjectives and verbs. The Lin corpus assigns a similarity score in the range $(0, 1)$ to each similarity pair, but the real measure in which the given score is indicative of the actual similarity among the pair is greatly variable. For this reason, in order to increase the reliability of the similarity classes, we filter similarity pairs using an absolute threshold (only the top n similarities of a word are considered) and a relative threshold (only pairs whose similarity value differs by less than a percentage from the top similarity for a given word are considered). In some cases, similarity classes may contain a multiword similarity (a sequence of more than one token). For cases in which no similarity pair for a certain word is defined in the Lin corpus, a word belongs to its own similarity class with a value of 1.0 by definition.

The pattern element types that have been implemented so far are the following (these are represented by the different MARKER terminal symbols in the EBNF grammar of Figure 2):

1. **Token form pattern elements**: Token-level pattern elements that only match a token with the same token form as that of the token from which they have been generated. It is the most restrictive type of pattern element.

2. **Semantic class pattern elements**: Token-level pattern elements that match any token (or a sequence of several tokens, in case of a multiword similarity) which represents a word that belongs in the semantic similarity class of the token from which they have been generated.

3. **POS tag pattern elements**: Token-level pattern elements that match any token which has the same Part-Of-Speech as that of the token from which they have been generated.

4. **Syntactic chunk pattern elements**: Chunk-level pattern elements that match a full syntactic chunk of the same type (e.g. NP, VP, PP, etc) and which has as headword a word with the same lemma as the chunk type and headword of the basic syntactic chunk from which the pattern element has been generated.

5. **Generalized syntactic chunk pattern elements**: Same as the previous ones, but these match any full syntactic chunk of the same type and with any headword whose lemma belongs in the semantic similarity class

of the headword of the chunk from which the pattern element has been generated.

We use a *dynamic window* for the amount of left and right context that is encoded into a pattern. This means that from a given example, we generate all the possible patterns with the same infix and with a prefix and postfix of size between 0 (empty prefix/postfix) and the maximum amount of left and right context, respectively, permitted by the length of the context window (configurable parameters of the algorithm). Patterns with a wider context are more restrictive, because all of the pattern elements in the prefix and the postfix have to be matched (on top of the infix) for the pattern to yield a match. Therefore, we simply generate patterns for all possible context lengths and let the ranking and selection procedure decide which are best. Every different combination of left and right context sizes is generated. That is, if the maximum length of the context window is 2 tokens left and 2 tokens right, we will generate patterns with: empty prefix and empty postfix, empty prefix and 1-element postfix, empty prefix and 2-element postfix, 1-element prefix and empty postfix, and so on... The maximum length of the context refers to tokens for the case of patterns composed of token-level pattern elements, and to full chunks in the case of patterns composed of chunk-level pattern elements.

At the present, the following types of pattern are generated from each selected example found in the previous iteration:

1. **Semantic class patterns**: A pattern that is constructed by generalizing each token in the example (including left and right context) to a semantic class pattern element.

2. **Token form mixed with semantic class patterns**: A pattern that is constructed by turning some of the tokens in the example into the corresponding token form pattern element, and generalizing the remaining tokens into semantic class pattern elements.

3. **POS tag patterns**: Patterns constructed by generalizing each token in the example into a POS tag pattern element.

4. **Token form mixed with POS tag patterns**: Patterns constructed by turning some of the tokens into token form pattern elements, and generalizing the rest into POS tag pattern elements.

5. **Syntactic patterns**: A pattern that is constructed by generalizing every basic syntactic chunk found in the example (including chunks in the left and right context) to a corresponding syntactic chunk pattern element of the same type and headword.

6. **Syntactic patterns with generalized headwords**: A pattern constructed in the same way as a syntactic pattern, but generalizing the chunks in the example to "generalized syntactic chunk" pattern elements instead.

The two latter types of pattern (syntactic), can only be created when the "infix" of the example (the part that corresponds to the potential time expression) coincides with the boundary of a syntactic chunk at both its ends. Otherwise, the leftmost or rightmost pattern element of the infix would overlap, respectively, with the rightmost element of the prefix or the lefmost element of the postfix.

Of among the types of patterns described above, those that are a combination of two different types of pattern elements (token form with semantic class or POS tag) pose the problem of choosing exactly which tokens from the example are generalized and which are kept as a token form pattern element. Choosing all possible combinations of k different indexes (i.e. positions of the tokens in the example), with k ranging from 0 to the length l of the example, and generalizing the tokens at these positions is one possible solution, but it yields an exponential number of patterns from a single example ($\sum_{k=0}^{l} \binom{l}{k} = 2^l$).

Therefore, in order to limit the amount of patterns that get generated as a result of mixed types, we may impose a lower and upper bound in the number of tokens that get generalized into the corresponding more-general pattern element (either semantic class or POS tag ones), with the minimum and the maximum values of k being anything between 0 and the length l of the example. Also, these limits may be imposed on the prefix, the infix and the postfix separately, as opposed to considering the possible combinations of the tokens in the example as a whole. Another option is not to generalize at all the tokens of either the prefix (i.e. left context), the infix and/or the postfix (i.e. right context). And lastly, we contemplate the option of generalizing only those tokens whose relative frequency in the training corpus lies within a certain range (as in band-pass filtering). This is useful to generalize only those tokens that may act as a trigger signaling the presence of a time expression (e.g. "days", "month", "time", "Monday", "11pm", "23-03-2005"), after having studied in what range of relative frequencies this type of tokens lie, approximately. All of the above options are implemented as configurable parameters of the algorithm.

When the instances of a pattern in the corpus are being searched, the pattern is checked against the sequence of tokens starting at a specific position in the corpus. First, we attempt to match the pattern's infix, attempting to match each pattern element in order and stopping at the first pattern element unmatched. Then, we attempt to match the prefix by checking its pattern elements in reverse order, and moving backwards from the token at the current position. And last, we attempt to match the postfix by checking its pattern elements moving forward from the token at one position past the last token consumed for the infix. Only after the infix, prefix and postfix have been successfully matched, a new candidate example is created from the matched tokens.

5 Ranking and Selection of Candidate Examples

For the remaining of this discussion, the *infix of an example* is defined as the tokens of the example that correspond to a potential time expression, leaving aside the additional left and right context information. Thus, an example is a placeholder for a certain "context of occurrence" in the corpus of the supposed time expression represented by its infix.

Ranking and selection of candidate examples takes place in two stages. In the first stage, only individual distinct infixes are given a score. Only in the second stage, the full examples (infix plus context) receive a score that is a combination of the score for its infix and a score that evaluates that particular context of occurrence.

Our scoring system for the infix of examples is based on the ranking procedure for examples used in Paşca et al. (2006) [3], notwithstanding the necessary differences forced by the fact that their representation of examples is different from ours (their examples consist of two separate terms of a relation, and do not encode context information). Each distinct infix receives three partial scores, all of which are also used in Paşca et al.'s framework. The final score for the infix is a linear combination of these three:

1. A similarity-based score (sim_sc(ex)), which measures the semantic similarity of the infix with respect to set of "accepted" seed examples from all previous iterations.

2. A phrase-completeness score (pc_sc(ex)), which measures the likelihood that the infix is a complete time expression and not merely a part of one.

3. A context-based score (ctxt_sc(ex)), which measures the frequency of the words in that infix's contexts of occurrence over the words in the contexts of occurrence of all the infixes.

We explicitly exclude tokens in a list of stopwords from all example's score calculations, whether they involve a similarity value or counting frequencies. This is done to prevent stopwords, which have abnormally high frequencies of occurrence, from distorting the respective terms of the an example's score.

The similarity-based score of an infix measures the semantic similarity of the infix with the seed examples (as a set). For computing the similarity among the words of a given infix and the words of a seed example, we use the similarity values provided by Lin's corpus [2]. If $w_1, \ldots, w_n$ are the tokens in the infix (excluding stopwords); $s_{j,1}, \ldots, s_{j,m_j}$ are the tokens in the j-th example of the set of seed examples; $|S|$ is the number of seed examples; and $\mathrm{simv}(x, y)$ is the similarity value given in the Lin corpus for the pair of words (x,y) (defining that $\mathrm{simv}(x, x) = 1.0 \; \forall x$), the similarity $\mathrm{Sim}(w_i)$ of the i-th word of the infix with respect to the seeds is given by:

$$\mathrm{Sim}(w_i) = \sum_{j=1}^{|S|} \max(\mathrm{simv}(w_i, s_{j,1}), \ldots, \mathrm{simv}(w_i, s_{j,m_j})),$$

and the similarity-based score of an infix containing n words (excluding stopwords) with respect to the seed set is given by:

$$\text{sim\_sc(ex)} = \begin{cases} C_1 + \frac{\sum_{i=1}^{n} \log(1+\text{Sim}(w_i))}{n} & \text{, if Sim}(w_i) > 0 \ \forall i \in [1,n] \\ C_2 & \text{, otherwise} \end{cases}$$

C_1 and C_2 are normalization constants such that $C_2 \ll C_1$. The similarity-based score aggregates individual similarities $\text{Sim}(w_i)$ of each word in the infix with respect the seeds, and scales the resulting sum according to the length n of the infix (so as not to introduce a bias that favours longer infixes). The individual similarities of a word with respect to the seeds are higher if: the word has high similarity with the words in more than one seed; and the word has high similarity with any of the words in each of the seeds (the maximum similarity is taken with respect to each of the seeds).

The phrase-completeness score measures the likelihood that the infix is complete and not part of longer incomplete time expression. If we call INFIX the sequence of tokens that form the infix being considered, this score is computed as the proportion of times that INFIX appears exactly as the infix of an example, over the times that INFIX is included as a substring in the infix of an example. Using a regexp-like notation, where * is a wildcard that substitutes zero or more tokens, this score term is given by:

$$\text{pc\_sc(ex)} = \frac{\text{count(INFIX)}}{\text{count}(*\text{INFIX}*)},$$

evaluated over the entire set of candidate examples.

The context-based score of an infix is a measure of the relevance (in the sense of coverage) of that infix, over the set of all candidate examples found in the current iteration. This is computed by considering all the contexts of occurrence of that infix in the corpus, up to a certain context window length. For each context of occurrence, a partial score for that particular context is computed as the relative frequency of the word with maximum relative frequency (excluding stopwords), over the set of all the words (excluding stopwords) that appear in all the contexts of occurrence of the infix. Finally, the overall context-based score for the infix ctxt_sc(ex) is given by the sum of the partial scores for all its contexts of occurrence, scaled by the relative frequency of the infix over the set of candidate examples (so as to mitigate the advantage obtained by the more frequent infixes). The window length for this context-based score is a configurable execution parameter of the algorithm, different from the amount of left and right context that is stored in the examples regarding a time expression occurrence.

Finally, the score of an infix is given as a linear combination of the three former terms, with the λ_i being parameters:

$$\text{score(ex)} = \lambda_1\text{sim\_sc(ex)} + \lambda_2\text{pc\_sc(ex)} + \lambda_3\text{ctxt\_sc(ex)}$$

Apart from the score associated with the infix of the example, each example receives two additional scores: one frequency-based score for the tokens of the left context stored in the example, and one for the tokens of the right context. These two scores evaluate the particular context of occurrence of the infix represented by that example. The frequency-based score for the left (respectively, right) context of the example is given by the relative frequency of the token with maximum relative frequency (excluding stopwords) of the left (respectively, right) context, computed over all the tokens (excluding stopwords) that appear in the left (respectively, right) contexts of all the examples for the current iteration.

Selection takes place in two stages: first, the top n infixes are selected; then, for each selected infix, the m top-scoring contexts of occurrence are selected ($m \ll n$). The m contexts of occurrence (examples) for a given selected infix are selected on the basis of the sum of their frequency-based scores for the left and right context (computed as explained above). This yields a total of $n \times m$ examples selected (at most) per iteration.

6 Ranking and Selection of Candidate Patterns

We have implemented the ranking and selection of patterns part of our bootstrapping system as a replaceable module, which allows experimenting with different strategies for pattern selection. Different ranking and selection strategies may use different types of terms for the score of a pattern, but in general they tend to combine some sort of measure of the *precision* of the pattern with some measure of its *coverage*.

In our current implementation, each candidate pattern pat is assigned two partial scores, one for each of these two types of measure:

1. A frequency-based score freq_sc(pat) that measures the coverage of the pattern in the unsupervised corpus.

2. A precision score prec_sc(pat) that evaluates the precision of the pattern over a (section of) of the unsupervised corpus, measured against a control set of "allegedly correct" time expressions.

The frequency-based score of a pattern captures the degree to which its integrating pattern elements are represented in the unsupervised corpus (or, to be more precise, the token features which its pattern elements can match). In order to calculate this score, we compute first the relative frequency in the corpus of each possible token feature used to generate pattern elements. These are, so far:

- The frequency of each distinct token form in the corpus.

- The frequency of each distinct POS tag in the corpus.

- The frequency of each distinct combination in the corpus of a syntactic chunk type with a particular headword.

We define the *frequency of a pattern element* in a way which depends on the type of pattern element, and which is representative of the relative frequency of tokens in the corpus that the pattern element can match:

- For **token-form** pattern elements, the frequency equals the relative frequency of that token form, multiplied by a scaling factor for token forms.
- For **semantic class** pattern elements, the frequency equals the sum of relative frequencies of each token belonging to the semantic similarity class referenced in the pattern element, multiplied by a scaling factor for token forms.
- For **POS-tag** pattern elements, the frequency equals the relative frequency of that POS tag, multiplied by a scaling factor for POS tags.
- For **syntactic chunk** pattern elements, the frequency equals the relative frequency in the corpus of the combination of chunk type and headword indicated in the pattern element, multiplied by a scaling factor for combinations of chunk type with a particular headword.
- For **generalized syntactic chunk** pattern elements, the frequency equals the sum of the relative frequencies in the corpus of the combinations of the chunk type indicated in the pattern element with each of the tokens in the semantic similarity class of the indicated headword used as headword of the chunk, multiplied by a scaling factor for combinations of chunk type with a particular headword.

For each of the possible types of token feature (token form, POS tag, or a combination of a syntactic chunk type with a particular headword), this *scaling factor* is the number of possible distinct labels for that feature type, divided by the number of possible distinct labels for all three feature types together. So, for instance, the scaling factor for a token-form pattern element is equal to the number of distinct token forms found in the corpus, divided by the sum of the number of distinct token forms, the number of distinct POS tags and the number of distinct combinations of a chunk type with a headword. The purpose of applying a scaling factor to the frequency of each pattern element is to "equalize" the range in the relative frequencies of the elements of sets with different cardinality (for instance, the number of possible token forms is much higher than that of possible POS tags, therefore the range of values for relative frequencies of a particular token form will be much lower than for POS tags).

The frequency-based score for a pattern is determined by the pattern elements with highest frequency in the prefix, the infix and the postfix, adjusting in the prefix and postfix for the distance with respect to start and end of infix, respectively. α is an adjusting constant to soften the ratio with which the contribution of a pattern element in the prefix or postfix decreases with its distance with respect to the infix (we set the parameter initially at $\alpha = 0.5$):

$$\mathrm{freq\_sc(pat)} = \frac{\mathrm{arg\ max_{p \in prefix} freq(p)}}{\mathrm{dist(p,infix)}^{\alpha}} + \mathrm{arg\ max_{p \in infix} freq(p)} + \frac{\mathrm{arg\ max_{p \in postfix} freq(p)}}{\mathrm{dist(p,infix)}^{\alpha}}$$

Undefined-type pattern elements, token-form and semantic class pattern elements where the referenced token is a stopword, as well as syntactic chunk pattern elements where the headword is a stopword are excluded from contributing towards the frequency-based score of a pattern (the frequency of these pattern elements is by definition 0). Also, patterns with an empty prefix and/or an empty postfix are assigned a symbolic contribution for the prefix and/or postfix part of the frequency-based score, equal to middle value in the range of frequencies received by pattern elements in the prefix/postfix of candidate patterns of the current iteration (this is introduced so as to compensate a negative bias in the formula of frequency-based score against patterns with empty prefix or postfix, which would make the corresponding term in the formula 0).

The precision score of a pattern measures the precision achieved by the pattern in a fraction of the training corpus, evaluated against a "control set" of examples considered correct. In order to evaluate this precision, instances of the pattern are searched in a random section of the corpus of size a fraction of the original (e.g. a 10% of the corpus). What fraction is used is a configurable execution parameter.

The "control set" is formed by all the selected examples (including left and right contextual information of the context of occurrence) over the previous iterations. Each instance of the pattern, found in the section of the corpus, is considered either a *partial hit*, if only the infix part of the instance coincides with the infix of some example in the "control set"; or a *full hit*, if the full instance (infix plus left and right context tokens) coincides with some example in the "control set". Thus, the precision score for the pattern is given by:

$$\text{prec\_sc}(\text{pat}) = \frac{0.5 \times \text{partial hits} + 1.0 \times \text{full hits}}{\text{number of instances found}}$$

Given the frequency-based and the precision scores for a pattern, as defined above, two ranking and selection strategies for patterns have been implemented:

Multiplicative combination

In this strategy, the final score of a pattern is given by a multiplicative combination of the frequency-based score and the precision score, weighed by parameters λ_1 and λ_2:

$$\text{score}(\text{pat}) = \lambda_1 \log(\epsilon_1 + \text{freq\_sc}(\text{pat})) + \lambda_2 \log(\epsilon_2 + \text{prec\_sc}(\text{pat}))$$

The epsilons are introduced to prevent any of the two score terms from being exactly 0. Selection takes place in strict order of ranking. The n patterns with the top scores are selected.

Collins' strategy

This heuristic for pattern selection was proposed in Collins and Singer (1999) [1]. Patterns are first filtered by imposing a threshold on their precision score (for instance, prec_sc(pat) > 0.95). Only for those patterns that pass this first filter, their final score is considered to be their frequency-based score.

$$\text{score(pat)} = \begin{cases} \text{freq\_sc(pat)} & \text{, if prec\_sc(pat)} > \text{threshold} \\ 0 & \text{, otherwise} \end{cases}$$

Of those patterns that pass the precision filtering, the n (at most) with the top frequency score are selected. This selection strategy is more aggresive than the former, but is known to produce patterns with good precision.

Bibliography

[1] M. Collins and Y. Singer. Unsupervised models for named entity classification. In *Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, pages 100–110, College Park, MD, 1999. ACL.

[2] D. Lin. Automatic retrieval and clustering of similar words. In *Proceedings of the 17th International Conference on Computational Linguistics and the 36th Annual Meeting of the Association for Computational Linguistics (COLING-ACL-98)*, pages 768–774, Montreal, Quebec, 1998. ACL.

[3] M. Paşca, D. Lin, J. Bigham, A. Lifchits, and A. Jain. Names and similarities on the web: Fact extraction in the fast lane. In *Proceedings of the 21th International Conference on Computational Linguistics and 44th Annual Meeting of the ACL*, pages 809–816. ACL, 2006.

[4] J. Poveda and M. Surdeanu. SVMs for the temporal expression chunking problem. Technical Report LSI-06-37-R, Polytechnic University of Catalonia (UPC), Software Department (LSI), 2006.

[5] E. Riloff. Automatically generating extraction patterns from untagged text. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1044–1049. AAAI/MIT Press, 1996.

[6] M. Stevenson and M. Greenwood. A semantic approach to ie pattern induction. In *Proceedings of the 43rd Meeting of the Association for Computational Linguistics*, pages 379–386. ACL, 2005.

[7] M. Surdeanu, J. Turmo, and A. Ageno. A hybrid approach for the acquisition of information extraction patterns. In *Proceedings of the EACL 2006 Workshop on Adaptive Text Extraction and Mining (ATEM 2006)*. ACL, 2006.

[8] R. Yangarber. Counter-training in discovery of semantic patterns. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*. ACL, 2003.

[9] R. Yangarber, R. Grishman, P. Tapanainen, and S. Hutunen. Automatic acquisition of domain knowledge for information extraction. In *Proceedings*

of the 18th International Conference of Computational Linguistics, pages 940–946, 2000.

[10] D. Yarowsky. Unsupervised word sense disambiguation rivaling supervised methods. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*, pages 189–196, Cambridge, MA, 1995. ACL.