Inductive Logic Programming and Its Application to the Temporal Expression Chunking Problem

Jordi Poveda Poveda
jpoveda@lsi.upc.edu

Jordi Turmo Borràs
turmo@lsi.upc.edu

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

Contents

1 ILP Overview

Inductive Logic Programming (ILP) comprises a group of machine learning techniques that falls under the broader category of *inductive concept learning*. ILP extends the capabilities of traditional attribute-value concept learners (e.g. decision trees) by bringing the expressiveness of first-order logic into the equation.

Concept learners based on attribute-value example descriptions generate concept definitions which make use of conditions on the value or ranges of values for individual attributes (in the form of rules or otherwise). In contrast, ILP methods are able to exploit the *relations* among the features that characterize examples —which are inherent to many instances of learning problem scenarios— and among examples themselves, and express them in the form of first-order logic predicates with variables. The form of hypotheses generated by ILP methods is that of logic programs, in which each concept to be learned is represented by a predicate consisting of one or more clauses.

1.1 Basic ILP terminology

In inductive concept learning, the aim is to learn a concept $\mathcal{C} \subseteq \mathcal{U}$ that defines a class of objects, examples, or individuals, where $\mathcal{U}$ represents the universe of discourse. As a result of learning, the learner generates an hypothesis $\mathcal{H} \subseteq \mathcal{U}$ which is, ideally, a good approximation of the original concept $\mathcal{C}$. Different learning algorithms use different representations for both individual objects $x \in \mathcal{U}$, and for concepts C and the generated hypotheses $\mathcal{H}$ (i.e. for representing sets of individual objects).

In inductive logic programming ([LAV94]), target concepts are represented in the form of first-order *predicates* $p_i(X_1, \ldots, X_n)$ of a certain arity n (in the context of ILP, predicates are also called *relations*). Hypotheses generated by the learning algorithm are definitions for each of the target predicates p_i, in the form of a set of logic clauses (rules) $p_i(X_1, \ldots, X_n) \leftarrow L_1, \ldots, L_m$, where each of the L_i are literals.

The learner is supplied with an extensional set of *training examples* $\mathcal{E}$ describing each target predicate (or target concept), from which the concept is learned. Each example e is represented as a ground fact of the corresponding target predicate, i.e. as a n-tuple of constants $< x_1, \ldots, x_n >$ that define an assignment of values for the predicate arguments. Examples are divided in positive examples ($\mathcal{E}^+ \subseteq \mathcal{E}$), which satisfy the target predicate p_i and, optionally, negative examples ($\mathcal{E}^- \subset \mathcal{E}$), which provide counterexamples of the target concept.

Moreover, the learner may be provided with additional non-target predicate or relation definitions q_i, which together constitute the existing *background knowledge* $\mathcal{B}$. These background knowledge predicates may be defined either intensionally (i.e. as a set of first-order logic rules defining the cases that satisfy the relation) or extensionally, through a set of positive and/or negative examples that are ground facts of the predicate in question. The background knowledge predicates both define a vocabulary for the learner, in terms of which the hypothesis for predicates p_i may be constructed (the may appear in the literals L_i on the body of the clauses), as well as place additional restrictions on the generated hypothesis $\mathcal{H}$ since the hypothesis must be consistent with both the training examples and the background knowledge.

Lastly, each particular ILP method assumes a certain language bias $\mathcal{L}$ —the language for representing hypothesis and the background knowledge—, which places certain restrictions on the type of clauses that may be used in the predicate definitions, for example: Horn clauses, function-free clauses (no complex terms must appear as arguments to predicates in the clause literals, only variables or constants), non-recursive predicates (the target predicate must not appear in the body of the clause), etc. This language bias $\mathcal{L}$ introduces an expressivity vs. complexity dilemma to ILP: the less complex the hypotheses that may be expressed in the language, the smaller the complexity of the search in the space of candidate hypotheses the learner has to perform; but, on the other side, there may be complex target concepts for which the learner is unable to find a suitable representation in the language.

Training examples		Background knowledge	
$daughter(mary, ann).$	$\oplus$	$parent(ann, mary).$	$female(ann).$
$daughter(eve, tom).$	$\oplus$	$parent(ann, tom).$	$female(mary).$
$daughter(tom, ann).$	$\ominus$	$parent(tom, eve).$	$female(eve).$
$daughter(eve, ann).$	$\ominus$	$parent(tom, ian).$	

Table 1: >From Lavrač, N. and Džeroski, S. (1994) [LAV94]

Let us illustrate the above concepts by means of an example: we want to learn the target concept *daughter* expressed as the relation $daughter(X, Y)$ which translates into "X is the daughter of Y". Table 1 gives the training examples $\mathcal{E}$ and background knowledge $\mathcal{B}$.

An ILP algorithm could learn the following definition for the target relation $daughter(X, Y)$, assuming the hypothesis language $\mathcal{L}$ of Horn clauses:

$$daughter(X, Y) \leftarrow female(X), parent(Y, X).$$

The above hypothesis is both complete and consistent with respect to the given examples $\mathcal{E}$ and background knowledge $\mathcal{B}$. A hypothesis $\mathcal{H}$ is said to be *complete* with respect to a set training of examples $\mathcal{E}$ and background knowledge $\mathcal{B}$ if every positive example in the training set can be derived from the hypothesis and the background knowledge (i.e. $\mathcal{B} \cup \mathcal{H} \models e, \forall e \in \mathcal{E}^+$), and it is said to be *consistent* if no negative example in the training set can be derived from the hypothesis and the background knowlegde (i.e $\mathcal{B} \cup \mathcal{H} \not\models e, \forall e \in \mathcal{E}^-$). In principle, it is desirable that the hypothesis returned by an ILP learner be both complete and consistent, but these requirement may need to be relaxed in a practical situation with imperfect data (missing values, errors in the training examples, insufficient number of examples, or a language $\mathcal{L}$ that cannot represent the target concept exactly), and as a measure to avoid producing too complex hypotheses that produce *overfitting* of the training examples.

1.2 Types of ILP systems

ILP systems in existence can be classified according to several criteria, among these:

- Whether they accept a single target concept or multiple target concepts.
- Whether they take all the training examples at once and produce a hypothesis (batch learners), or accept examples one by one (incremental learners).
- Whether they enquire the user about the goodness of the so-far learned hypothesis at different points through the learning process, and/or ask the user to classify new examples generated by the system as positive or negative (interactive learners), or not (non-interactive).

- Whether they learn a concept from scratch, or accept an initial partial definition of the target predicate which is then refined through the learning (theory revisors).
- The representation language $\mathcal{L}$ for hypotheses.

With regard to the above criteria, a first essential distinction is drawn between *empirical ILP systems* and *interactive ILP systems*. Empirical ILP systems have become the most common variety. Their goal is typically to learn a single concept (although some systems allow learning several target relations at the same time) from a large training set of examples. Also, empirical ILP systems are typically batch learners and non-interactive, and learn concepts from scratch. Systems that belong in this category are FOIL ([QUI90]), GOLEM ([MUG90]) and PROGOL ([MUG95]), among others.

On the contrary, interactive ILP systems are oriented towards learning multiple target predicates from a small amount of examples, incrementally. These systems compensate the sparsity of examples by resorting to an oracle (the user) to verify the validity of a partially constructed hypothesis and to classify newly generated examples. They may also accept an initial hypothesis that is used to guide the search. Examples of interactive ILP systems are MIS ([SHA83]), MARVIN ([SAM86]) and CIGOL ([MUG88]).

1.3 General concepts of ILP algorithms

All ILP systems employ some form of search in the space of hypothesis permitted by their representation language $\mathcal{L}$, which turns the problem of concept learning into a problem of search. Before briefly describing the basic techniques used in ILP, let us introduce a way of structuring the search space of program clauses $p(X_1, \ldots, X_n) \leftarrow L_1, \ldots, L_m$ based on the θ-subsumption relation.

A clause $c = T \leftarrow Q$ (where T is an atom $p(X_1, \ldots, X_n)$ and Q is a conjunction of literals $L_1, \ldots, L_m$) is said to θ-subsume clause c' if there exists a substitution $\theta = \{X_1/t_1, \ldots, X_k/t_k\}$ from the variables X_i in c to terms t_i, such that the clause c with the substitution θ is a proper subset of the clause c' ($c\theta \subseteq c'$). For example, clause $c = daughter(X, Y) \leftarrow parent(Y, X)$ expressed as a set (a disjunction of literals) becomes $\{daughter(X, Y), \overline{parent(Y, X)}\}$. Clause c θ-subsumes the clause

$$daughter(mary, ann) \leftarrow female(mary), parent(ann, mary) \equiv$$
$$\equiv \{daughter(mary, ann), \overline{female(mary)}, \overline{parent(ann, mary)}\}$$

under the substitution $\theta = \{X/mary, Y/ann\}$. The θ-subsumption relation corresponds with the notion of generality: if clause c θ-subsumes clause c', then clause c is *at least as general as* clause c' (expressed $c \leq c'$); and it holds that any training example that is explained by c' is also explained by c. In this manner, θ-subsumption introduces a lattice (partial ordering) in the set of program clauses, which can be exploited when tackling ILP as a search problem.

The search in the hypothesis space for a candidate concept definition $\mathcal{H}$ can be carried out top-down (from the general to the specific), bottom-up (starting from single examples and working up towards predicates that account for more of the training set), or in a combination of both ways. The most general hypothesis is the empty hypothesis ($p(X_1, ..., X_n) \leftarrow$), from which any ground fact about p can be derived. The most specific hypotheses are the ground facts expressed by individual training examples ($p(x_1, \ldots, x_n)$, with x_i constants), which cover only a single example. The objective of the search is to converge towards a description of the target predicate p as a set of clauses that covers (most of) the positive examples and (almost) none of the negative examples.

Generalization techniques (bottom-up search) are employed mainly by interactive (incremental) ILP systems. They use two basic generalization operations: applying an inverse substituion to a clause (i.e. a substitution that maps terms to variables instead of variables to terms), and removing a literal from the body of a clause. In particular, in order to minimize the risk of allowing negative examples to be covered by the hypothesis, these systems utilize the notion of *least general generalization* (lgg), which is defined recursively for terms and predicates, and is related to the θ-subsumption lattice on the space of clauses.

The two generalization operations described combine into the generalization technique of *inverse resolution*, which attempts to invert the procedure of SLD-resolution used in first-order logic deduction. The basic idea is to combine (or inversely resolve) individual examples e with facts from the background knowledge b_j into a clause $c = e \leftarrow b_1, \ldots, b_k$, while applying at each step an inverse substitution θ^{-1} that maps some of the constant terms in the clause to variables, whereby generalization is introduced. This idea is used in CIGOL ([MUG88]) and GOLEM ([MUG90]).

Specialization techniques (top-down search) are employed, in contrast, mainly by empirical ILP systems, since top-down search adapts better to the use of heuristics to prune a large search space. These use two basic specialization —also called refinement— operations, namely the opposite of generalization operations: applying a substitution to a clause (changing some variables to constant terms), and adding literals to the body of a clause. These two operations are the basis for the ILP specialization technique known as *top-down search of refinement graphs* (first introduced in the MIS system [SHA83]).

A refinement graph is a directed acyclic graph (DAG) where nodes correspond to the possible clauses for a target predicate, and arcs correspond to the application of one of the two basic refinement operations (substituting a variable with a term and adding a literal to a clause). The algorithm iteratively adds new clauses c_i to the final hypothesis $\mathcal{H}$ in order to cover an increasingly larger group of examples, and eliminates previously added clauses if they are found to cover some negative example, until the final hypothesis $\mathcal{H}$ is consistent and complete. For each new clause c_i, a refinement graph thus constructed is used to guide the search from the most general clause (the empty clause $c_i = T \leftarrow$) to a more specific clause $c_i' = T \leftarrow L_1, \ldots, L_m$, that covers a subset of the positive examples $\mathcal{E}^+$.

ILP algorithms also introduce some safeguard mechanisms in order to deal

with noisy data. These can take the form of limiting the description length of the generated hypothesis (by restricting the number of literals in the body of clauses), at the expense of accuracy of the clauses. Allowing for partial completeness and consistency with respect to the training examples $\mathcal{E}$ can also contribute to avoid overfitting, when the learned hypothesis is used to classify unseen examples.

2 FOIL

FOIL is an empirical ILP system developed by Quinlan ([QUI90], [QUI93]). Its antecedents are attribute-value based learning systems such as ID3 trees and AQ (a system that learnt concepts in the form of if-then propositional rules concerning the values of attributes). FOIL utilizes a form of the top-down search of refinement graphs algorithm introduced by the MIS ([SHA83]) ILP system.

The hypothesis language $\mathcal{L}$ of FOIL restricts clauses in the hypothesis to containing function-free literals (only constants and variables may appear as arguments in the predicates of the body literals, not complex terms). Recursive predicate definitions are supported, which means that the target predicate symbol p may appear in the body of a clause. Representation of the background knowledge $\mathcal{B}$ is restricted to extensional definitions of relations (predicates) in the form of a list of ground facts (i.e. background knowledge predicates cannot be defined using rules). Arguments of relations in FOIL are typed, which means they take values over a particular domain.

A typical input to FOIL consists of the following:

1. A set of type definitions, where each type specifies a set of values (a domain) that the arguments of predicates can take. A type definition consists of a type name followed by a set of constants (the possible values for that type). FOIL supports unordered, ordered (where a particular ordering of the constants exists) and possibly ordered types (where FOIL attempts to discover an ordering of the constants that is useful for recursive predicates). FOIL supports continuous types, which take values over the integer and the real numbers.

2. A set of extensional relation definitions, one or several of which correspond to target predicates and the rest to predicates of the background knowledge (a vocabulary of relations). Relation definitions consist of a header, indicating the name of the predicate and the number and types of its arguments, and one or two sets of examples. First, a set of positive examples for the predicate are given, and then, optionally, a set of negative examples. Each example is a comma-separated tuple of constants. Negative examples need not be provided, FOIL can generate them under the closed world assumption.

3. Optionally, a number of test examples for the target relations may be provided, tagged as positive or negative.

>From these elements, FOIL returns an hypothesis consisting of a set of function-free Horn clauses for each of the target predicates. The predicates that may appear in the literals of the body of the solution clauses are: predicates from the background knowledge, any of the target predicates, a predicate that binds a variable to a value ($X_i = X_j$), and a predicate that specifies an ordering relation between values ($X_i < X_j$). These predicates may also appear negated (in the form $not\, L_i$), where *not* denotes the closed-world interpretation of negation. The arguments of predicates in the literals may only be variables or a special type of constants that have been marked as "theory constants". At least one of the variables in a literal must be bound (i.e. must have appeared in the head of the clause or in a previous literal).

The FOIL algorithm consists of three steps: pre-processing of examples, hypothesis construction and post-processing of the solution. Post-processing of the solution is a form of pruning that eliminates irrelevant clauses.

Hypothesis construction consist of an inner process of *clause specialization* embedded into an outer *covering loop*. The covering loop starts out with an empty hypothesis $\mathcal{H} := \emptyset$ and iteratively searches for new clauses c that cover (i.e. explain) a subset of the positive examples $\mathcal{E}^+_{cur}$ that remain so far, adding the new clause to the hypothesis ($\mathcal{H} := \mathcal{H} \cup \{c\}$). This process stops when either all positive examples are covered ($\mathcal{E}^+_{cur} = \emptyset$) or restrictions about maximum description length of the hypothesis are violated.

The inner clause specialization loop starts out with an empty clause ($c := T \leftarrow$, with T being the target predicate), and iteratively refines the clause c by adding new literals L_i to its body until either the clause covers no negative examples in the training set, or the maximum length restriction for a clause is violated. FOIL uses an heuristic based on *information gain* to guide the selection of literals L_i that are added to the clause, among the possible choices.

3 Chunking of temporal expressions using ILP

In this section we first describe the problem of temporal expression chunking within the bigger context of information extraction (IE), and next present two alternative ways of modelling this problem (one propositional and one relational) for tackling it using ILP.

3.1 Problem description

Chunking refers to any problem or task in NLP (natural language processing) which involves segmenting (i.e. identifying the boundaries of) the occurrences in a text of a certain type of entities. This gives rise to different types of chunking problems, for instance:

- Syntantic chunking, where the aim is to identify syntactic phrases (e.g. noun phrases, verb phrases, prepositional phrases, ...).

- Named entity recognition, where mentions in text of certain types of concepts that have a proper name designation (hence "named entity") are sought, for example: persons, locations, currency, organizations...
- Clause splitting (i.e. a group of words consisting of a subject and a predicate).

A related task, which may accompany chunking, is that of *classification* of the segmented chunks into appropriate types (e.g. PER for person, LOC for location, ORG for organization, etc., in the case of named entities).

Chunking of *temporal expressions* is a part of the more complex task called *temporal expression recognition and normalization* (TERN), in which the aim is to identify the mentions in a text of expressions that denote time, and to capture their meaning by writing them in a canonical representation. A common way to represent these temporal denoting expressions in text —as well as other entities of interest in information extraction, like named entities, events or relations— is to employ XML tags with convenient attributes to further qualify the entity. For example:

```
But even on <TIMEX2 VAL="1999-07-22">Thursday</TIMEX2>, there
were signs of potential battles <TIMEX2 VAL="FUTURE_REF"
ANCHOR_DIR="AFTER" ANCHOR_VAL="1999-07-22">ahead</TIMEX2>.
```

The TIMEX standard sets out guidelines for annotating and normalizing mentions of time expressions. These and a full account of the complex bestiary of temporal expressions can be found in [FER03].

There are two separate problems in the TERN task, one of identifying the *extension* of the mentions of temporal expressions in text (i.e. chunking) and the considerably more difficult problem of normalizing the value of the temporal expression. Here we are only concerned with the first of these two problems. Contrary to what it may at first seem, temporal expressions come into a wide variety of forms ([WIL01]):

- Fully-specified time references: *16th June 2006*, *the twentieth century, Monday at 3pm*.
- Expressions that depend on a context: *last month*, *three days from today*, *February last year*.
- Anaphorical expressions and expressions relative to the time when the expression is written: *that day*, *yesterday*, *currently*, *then*.
- Durations or intervals: *a month*, *three days*, *some hours in the afternoon*.
- Frequencies or recurring times: *monthly*, *every other day, once a week*, *every first Sunday of a month*.

- Culturally dependent time denominations: *Easter*, *the month of Ramadan*, *St. Valentine*.
- Fuzzy or vaguely specified time references: *the future*, *some day*, *eventually*, *anytime you so desire*.

The chunking-related problems have traditionally been represented as the multiple classification problem of assigning *I/O/B tags* to each token in a sequence of text. The tags stand for Inside (I), Outside (O) and Begin (B), and are enough for delimiting non-overlapping, non-recursive chunks (i.e. chunks that neither appear inside the boundaries of the extension of another larger chunk, nor overlap with another chunk). If the task involves further discriminating the chunks into several classes, the I and B tags can be appended with a suffix to signify the type of the chunk (e.g B-PER, I-PER, B-LOC, I-LOC, B-ORG, I-ORG, O for the named entity recognition task). In the case of temporal expression chunking, the result may look something like this:

```
In/O 2006/B ,/O thirty/B years/I after/O Mao/B Tse/I Tung/I
's/I death/I ,/O hordes/O of/O devote/O Chinese/O arrive/O
still/O daily/B to/O his/O memorial/O site/O ./O
```

For our experiments, we use the corpus of the ACE 2005 competition (Automatic Content Extraction) for training and test. This corpus has the mentions of time expressions manually annotated using TIMEX tags. The composition of the corpus is 550 documents distributed in five categories (newswire, broadcast news, broadcast conversations, conversational telephone speech and weblogs), containing 257000 tokens and approx. 4650 time expression mentions.

Regarding evaluation, we are interested in three measures for quantifying the performance of the temporal expression recognizer:

Precision: The rate of returned temporal expressions that are correctly identified (i.e. correctly tagged divided by total tagged).

Recall: The rate of existing temporal expressions that are correctly identified (i.e. correctly tagged divided by those that should have been tagged).

F_1 Score: It is the harmonic mean of the two previous values, $F_1 = \frac{2 \times Precision \times Recall}{(Precision + Recall)}$.

Note that only temporal expression mentions whose boundaries are correctly identified count positively towards the precision and recall scores. Partial matches (a temporal expression that is identified partially, or misplaced) are not regarded as hits. We are not counting the rate of correct assignments of I/O/B tags at the token level: that one is a different measure and is termed *accuracy* —and normally, the attained score in this measure is also considerably higher—.

3.2 Alternative modellings of the problem using ILP

We propose two alternative modellings of the temporal expression chunking problem in order to evaluate the performance that can be achieved using inductive logic programming (specifically, FOIL) for the task: one *propositional*, which takes into account features of individual *tokens*; and one *relational*, which tries to classify *fragments* (i.e. sequences of tokens) as to whether they correspond to the mention of a temporal expression, and which takes into account as well relations between the tokens.

In the propositional approach, the text is first tokenized (segmented into individual tokens) and then a number of features are computed for each token. Tokens correspond normally with individual words, but also include punctuation marks —which are separate tokens— and special constructs like the saxon genitive "'s". Features that we use for tokens include:

- Lexical: The token form itself, the token in lowercase, the token that results from removing all letters from the token (e.g. 3 for "3pm"), the token that results from removing all letters and numbers (e.g. - - - for 1995-07-12).
- Morphological: The POS (Part Of Speech) tag (for instance, NN for noun, JJ for adjective, CD for cardinal number, MD for modal verb, ...).
- Syntactic: The tag of the syntactic chunk that the token belongs to (e.g. NP for noun phrase, VP for verb phrase, ...).
- Format features: These are flags that indicate if the token has certain format characteristics (*isAllCaps*, *isAllCapsOrDots*, *isAllDigits*, *isAllDigitsOrDots* and *initialCap*).
- Features that indicate if the token belongs to certain specific classes of words: *isNumber* (e.g. *one*, *two*, *ten*, ...), *isMultiplier* (e.g. *hundred*, *thousands*, ...), *isDay* (e.g. *monday*, *mon*, *saturday*, *sat*, ...) and *isMonth* (e.g. *january*, *jan*, *june*, *jun.*, ...).
- Contextual features: All of the previous features, but in reference to the tokens occurring in a certain window to the left and right of the current token.
- The target classification: The correct I/O/B tag for a window of tokens in the left context (we call these dynamic features).

In order to transform this attribute-value representation of tokens into logic predicates that can be used as input to FOIL, each individual token is assigned an identifier. These token identifiers function as constants. All logic predicates take a single argument, which is a token identifier (therefore we refer to this approach as *propositional*, because the predicates do not really express *relations*). Each possible feature of a token will be represented as a predicate of the

background knowledge, with positive examples for the predicate being the identifiers of the tokens that have that feature. For example, the following sequence of tokens with some of their features:

```
          FORM       POS   SYNTAX    I/O/B tag
tok100: the          DT    B-NP      O
tok101: current      JJ    I-NP      O
tok102: deficit      NN    I-NP      O
tok103: will         MD    B-VP      O
tok104: narrow       VB    I-NP      O
```

would produce, among others, the following ground facts of the background knowledge: *form_deficit*(tok102), *form_current*(tok101), *POS_DT*(tok100), *POS_NN*(tok102), *syn_B_VP*(tok103), *context_l1_form_current*(tok102), *context_r2_POS_VB* (tok102), ...

In this propositional approach, we will have three target predicates, that correspond to the three possible classifications of a token, namely: *begin_time_exp (X)*, *inside_time_exp(X)* and *outside_time_exp(X)*, which take a token identifier as argument. The purpose of trying out this propositional approach is to compare how well ILP performs for this task in comparison with a previous experiment that we did with the same corpus using support vector machines, and with the same set of features.

Two relational approaches at automatically learning rules for information extraction are described in [FRE98] and [TUR02]. Freitag (1998) [FRE98] presents the SVR system, a relational learning system for acquiring extraction patterns for IE, and compares its performance with other methods (Bayesian learning and grammatical inference). Turmo & Rodriguez (2002) [TUR02] describe EVIUS, a relational learner that employs ILP (FOIL in particular) to automatically extract rules for information extraction (IE rules) in multiple domains, based on the idea of an scenario of extraction, which specifies which concepts are relevant to the domain at hand. The performance of EVIUS is also compared to the same methods treated in [FRE98] as well as other methods. Our proposal of a relational approach for the problem of temporal expression chunking draws on some ideas from the former papers.

In the relational approach, the aim is not to focus on individual token's features, but to exploit the relations between successive tokens as they appear in the text. In this approach we look at *fragments* of text rather than single tokens, and try to classify fragments as being temporal expressions or not. A fragment is defined as a sequence of consecutive tokens from the text, of a bounded length. In order to limit the number of examples to be considered, we look at the minimum and the maximum length of a temporal expression mention in the training corpus. We will consider only fragments of a length between that minimum and maximum.

This time, an identifier for each fragment in the training data is generated, and also token identifiers for individual tokens are generated. These fragment and token identifiers are constants that can be used as arguments for relations

(predicates). The difference with respect to the propositional approach is that using relations gives us much more freedom in the type of predicates we can use as background knowledge.

Thus, we can have predicates that capture the relation between fragments and tokens, like:

- *fragment_span(Fragment: FragmentId, FirstTok: TokenId, LastTok: TokenId)*,
- *contains_token(Fragment: FragmentId, Tok:TokenId)*;

and predicates that capture relations among individual tokens, like

- *follows(Tok1: TokenId, Tok2: TokenId)* and
- *precedes(Tok1: TokenId, Tok2: TokenId).*

Also, unlike in the propositional approach, where the limitation that predicates could only take token identifiers as arguments obliged us to define a separate predicate for each possible value of a token's feature, we may now define new types for each feature we intend to use and have generic predicates that link individual tokens to their features, as in:

- *token_form(Tok: TokenId, Word: TokenForm)*,
- *token_POS(Tok: TokenId, Tag: POSTag);*

even though due to computational complexity reason, it may be more convenient to keep the arity of predicates to a minimum and define a different predicate for each possible value of a feature, as in:

- *token_form_Form(Tok: TokenId)*, where Form means each of the lexicon's token forms.

We may also have predicates that describe features of fragments, as in

- *length_less_than(Fragment: FragmentId, Length: Integer)*, or else, for the same reason as above,
- *length_less_than_X(Fragment: FragmentId)*, for $X = maxlen - minlen + 1 \ldots maxlen$,

or that specify whether a fragment contains one of a list of *trigger words* (days, weeks, hours, year, ...) that point towards the fragment being a time expression

- *contains_trigger(Fragment: FragmentId, Word: Trigger).*

In the relational approach, our target predicate would simply be *is_time_expr (Fragment: FragmentId).*

4 Results

The following discussion concerns the results obtained by using the approach termed *propositional* as described in section 3.2 above. The implementation of the *relational* approach has been dropped due to performance issues discussed next. Some additional experiments intended to reduce the model complexity and the results obtained will also be outlined.

4.1 Performance issues

We encountered that the amount of time that FOIL requires to train a model for a large dataset and number of features (as it is our case with the ACE corpus and the feature set described above) renders it impractical for real use.

Our training set consists of 192182 tokens, 3613 of which are tagged as B (start of temporal expression), 3187 as I (inside temporal expression) and 185364 as O (outside). In the propositional knowledge representation, the number of domain predicates other than the 3 target predicates (i.e. predicates corresponding to possible features of the individual tokens) is 159175, and all predicates have arity 1 (the only variable is the token identifier). Training has been conducted in a cluster of workstations with Pentium 4 3.20 GHz and 4 GBytes of RAM.

Training 3 classifiers in this manner with FOIL (one for each of the target predicates: B, I and O) took over three and a half weeks for each —the three of them were trained simultaneously—. FOIL's temporal complexity is of order $\mathcal{O}(\|\mathcal{B}\| \times \|\mathcal{E}^+ \cup \mathcal{E}^-\| \times A)$, where $\|\mathcal{B}\|$ is the number of background knowledge predicates, $\|\mathcal{E}^+ \cup \mathcal{E}^-\|$ is the number of examples and counterexamples for the target predicate and A is the maximum arity (number of arguments) of a predicate. As a matter of comparison, consider that training 3 one-vs-all SVM classifiers to perform the same task with the same dataset takes on the order of 8 hours. Obviously, this impairs FOIL's ability to be considered a viable alternative for our problem. This is one of the reasons why we did not even attempt running FOIL with a *relational* representation of the problem (see section 3.2), the other reason being that the classification results obtained in the testing were not up to the expectations. A relational representation of the problem, in spite of its purportedly greater expressive power, should have required a prohibitive amount of time to train, provided that the arity of the predicates increases as well as the number of examples to consider (fragments of tokens).

Nevertheless, we attempted to speed up the training of our model with FOIL, at the expense of filtering predicates from the background knowledge (thus reducing the number of available options for FOIL to construct literals for clauses) and reducing the number of counterexamples for the target predicates. The decrease in training time achieved in this way was only limited, and not without a considerable penalty in the classification results.

4.2 Reducing the model complexity

An attempt was made to decrease the time required for training by introducing two measures to reduce the model complexity:

- Eliminating domain predicates (token features) that had few positive examples.
- Eliminating negative examples for the target predicate that were not discriminant enough with respect to the set of positive examples (i.e. the information they provide can be deemed less relevant).

The rationale behind filtering out the less frequent domain predicates is that a great number of these predicates are generated by the token forms of words encountered in the training section of the corpus and other token features that depend directly on the the token form (e.g. lower case form, and the lexical features). This is increased by the use of contextual features, which multiplies the number of this type of predicates by the width of the context window. Many of the thus generated predicates may be irrelevant, because the ocurrence in the corpus of the words that motivate them may only be anecdotal. Therefore, we introduced the restriction that background knowledge predicates must have at least X positive examples in order to be included as input to FOIL.

As for the second simplification, the motivation behind filtering negative examples for the target predicates is that, in our problem, tokens pertaining to temporal expressions appear sparingly through the whole training set. The tokens that are negative examples for one of the target predicates, B (begin), I (inside) and O (outside), are those that are a positive examples for either one of the other two. This creates a situation where the B and the I predicates have few positive examples and a lot of negative examples, whereas the opposite is true for the O predicate. By eliminating the non-informative counterexamples, we reduce the burden placed on FOIL to ensure consistency with the set of examples.

We considered informative counterexamples to be those that are "similar enough" to one or more of the positive examples, so that taking this particular counterexample into account is important for the classifier to learn establish the frontier among positive and negative cases accurately. To this end, we defined an *ad hoc threshold* on the similarity of a negative example with respect to the set of positive examples. A particular negative example will only be included in the set of negative examples for the target predicate presented to FOIL if it shares at least M features with (i.e. satisfies at least M domain predicates that are also satisfied by) at least N tokens from the set of positive examples —note that the features the counterexample in question shares with each of the N positive examples may be a different M in each case—.

We experimented with different values of these thresholding parameters X, M and N, using more restrictive values each time until we achieved a reduction that we deemed reasonable in both the resulting number of domain predicates and the number of negative examples, while at the same time looking at not

compromising classification accuracy excessively. How much such a reduction would compromise correctness of classification was still a guess, as we could not afford to run a full FOIL training and testing for each choice of thresholding parameters' values. The values that were finally decided upon are $X = 10$, $M = 20$ and $N = 5$. Observe that the number of token features that we demand a negative example to share with some of the positive examples in order to be taken into consideration is large ($M = 20$). This is so because once two tokens have the same token form (i.e. they both are the same word), a lot of other features automatically follow suit in most cases: the lower case form, the lexical features, the POS tag, the syntactic chunk tag, the format and class features... This is again increased if some of the words in the left or right context of the two tokens in consideration coincide as well.

With the above values for threshold parameters ($X = 10$, $M = 20$ and $N = 5$), the number of domain predicates was reduced from 159175 to only 20105; and the number of negative examples for the target predicates was reduced from 188551 to 66351 for the B predicate, from 188997 to 35768 for the I predicate, and from 6800 to 5629 for the O predicate. The training time of FOIL was reduced proportionately: for example, the classifier for B needed 2 and a half days to complete, and the classifier for O needed around 3 days to complete. However, classification performance was also considerably impaired, with losses in the order of 8%-10% in both precission and recall with respect to using the full model (see section 4.4).

4.3 Testing

Once classifiers for each of the B, I and O target predicates had been trained with FOIL, we took advantage of the built-in inference engine of PROLOG for evaluation against the test set. We collected the set of clauses produced by FOIL for the three target predicates into one file of rules for PROLOG by using a PERL script. In a similar manner, we produced a file of PROLOG facts which contained the features of the tokens in the test partition expressed as a set of ground facts (i.e. predicates that take a constant as argument, the constant being a token identifier). Next, we ran a PROLOG program to output our classifier's guess (that is, a B, I or O tag) for each of the test set tokens in sequence, and place them alongside the correct tag for each token in a tabular file. And lastly, we used a PERL script that computes the precission, recall and F_1 measures to evaluate the results.

There are several details that need to be taken into consideration which preclude evaluation of the test set with PROLOG from being straightforward. The most obvious difficulty that arises is that we have trained three classifiers with FOIL independently of each other (one to decide if a token carries the tag B or not, another for the tag I and another for the O). Each of the three classifiers uses its own set of clauses (produced by FOIL) to provide an individual yes/no response for a token, but several of the three classifiers may give a 'yes' response (or none of them for that matter), whereas the correct classification for a token is one and only one of the three possible tags. Therefore, it becomes

necessary to establish a procedure to reach a consensus or agreement on the classification of a token, given the outputs of the three classifiers. Also, and not less importantly, the sequence of tags output by this *consensus of individual classifiers* has to comply with certain consistency constraints with respect to the sequence of tokens, such as that an I (inside) tag cannot follow an O (outside) tag, or that the first token in a sentence cannot carry an I tag.

Thus, we need to resort to some type of measure that gives us an indication of the amount of evidence supporting each possible decision (that is, whether to assign a B, I or O tag), in order to break a possible tie when several of the classifiers give a 'yes' response. One such evidence measure, which we have employed, is the *confidence* of a rule. Each of the three classifiers applies a series of rules in order (i.e. clauses produced by FOIL), which have the form $A \Leftarrow B$, to decide on a yes/no response for the token in question. In these FOIL or PROLOG rules, the left hand side A is one of the three target predicates (i.e. *begin_time_exp(X)*, *inside_time_exp(X)* or *outside_time_exp(X)*), and the right hand side B is a conjunction of literals or negated literals constructed from the domain predicates (that is, the possible features of a token as described in section 3.2, including the contextual features). Each classifier will output a 'yes' response for a token whenever it finds a clause whose right hand side B can be satisfied when the variable in the literals is unified with that token's identifer, or a 'no' otherwise. The satisfiability of the rules' right hand sides with respect to a token is computed by PROLOG, using the ground facts supplied in a facts file about the tokens in the test set. The confidence of each rule or clause is computed thus:

$conf(A \Leftarrow B) = \frac{\#(A \wedge B)}{\#B}$, where $\#(A \wedge B)$ refers to the number of tokens (cases) that satisfy both the antecedent (B) and the consequent (A) of the rule, and $\#B$ refers to the number of tokens that satisfy the antecedent.

This quantity is computed for each clause by taking into account the ground facts (i.e. predicate facts derived from the tokens' features) about tokens from the training test (in a way similar to how we consider the ground facts about tokens from the test set for the final evaluation). The confidence of each clause is stored as an additional constant argument to the target predicates in a modified PROLOG rule file. Thus, each rule from the rule file has a form similar to this:

```
learned_begin_time_exp(X, 0.986486) :- form_now(X),
context_l1_syn_B_VP(X).
```

And the PROLOG facts file contains a succession of ground facts expressing the individual tokens' features, such as:

```
form_July(tok1992).
pos_JJ(tok25983).
context_r2_syn_I_VP(tok199238).
ground_begin_time_exp(tok34).
```

Notice the subtle distinction above between the target predicate as a classification hypothesis (that is, the target predicate for which FOIL learns a set of

clauses), and the target predicate as in expressing the correct tag for a token in the file of ground facts. In order to distinguish between the two, we prepend the prefix "learned" in the first case to the name of the predicate, and the prefix "ground" in the second.

Evaluation is then performed by having PROLOG produce a classification decision of one and only one tag, for each token of the test set in sequence, based on all the information discussed above. The *consensus* among the decisions of the three classifiers (B, I and O) makes use of the confidences of the rules that support each of the individual decisions. Because a 'yes' response from a classifier could be based on several rules (that is, the token in question could satisfy the antecedents of one or several clauses), each one having a different confidence value, we have tried two different approaches to computing the confidence of each possible decision (the results of both are reported in section 4.4 below):

1. Considering the confidence of the best clause among those satisfied by the token.

2. Considering the sum of the confidences of all the clauses satisfied by the token.

The procedure for reaching the final agreement on the classification of a token is as follows:

- If the three individual classifiers give a 'no' response, the token is assigned the O (outside) tag, as it is by far the most common situation that a token does not belong to a time expression.

- If one of the possible classifications is I (inside), i.e. the I classifier gave a 'yes' response, and the token in question either appears at the start of a sentence, or the classification of the previous token was an O, the tag I is discarded from the options to consider —as it would create an inconsistent tag sequence— and the final classification is based on the remaining options alone. To this end, we add a special predicate "begins_sentence" to the file of ground facts, which tells whether a token occurs at the start of a sentence.

- In any other case, the token is assigned the tag corresponding to the classification decision with the highest confidence (among those for which the corresponding classifier gave a 'yes' response).

One final detail which is worth mentioning concerns the way how the dynamic contextual features of tokens are generated (that means, the ground fact predicates that tell the correct tag for the tokens within the context window of a given token). In the training stage, it is acceptable to take the correct classification tag for the contextual tokens in order to generate this particular token feature. However, this is not the case for the testing stage because now, as we are testing the learned classifier, the classification of the previous tokens would be an

CLASSIFIER	PREC	RECALL	F_1
FOIL (best)	77.58	**52.15**	**62.37**
FOIL (sum)	**81.32**	50.28	62.13
SVM	80.05 (-1.27)	73.71 (+21.56)	76.75 (+14.38)

Table 2: Performance of the propositional approach with FOIL vs. a SVM classifier

imperfect one. This means that in order to produce coherent testing results, we need to generate this dynamic contextual features "on-the-fly": the corresponding ground facts about the previous tokens are added to the PROLOG base of facts as the classification takes place.

4.4 Summary of results

Table 2 shows the precission, recall and F_1 values achieved by the propositional classifiers trained with FOIL using the full data (i.e. without any kind of reduction of complexity). We include the results obtained with both strategies for computing the confidence of a classification decision ('best' indicates the confidence of the best clause is taken, 'sum' indicates that the sum of confidences of all the satisfied clauses is taken). The precission, recall and F_1 results achieved with a statistical SVM classifier (one-vs-all) trained with the same data and using exactly the same set of features is included for the sake of comparison.

The quantities in bold correspond to the best result obtained by FOIL (with each of the two strategies for computing the confidence), and the numbers in parentheses are the score differences of the SVM classifier with respect to FOIL. It can be observed that the precission values are close in both cases, but there is consistently a drop in the recall value of FOIL with respect to that achieved by the SVM. This loss in recall damages the F_1 figures of FOIL considerably. We believe that this low recall of FOIL is due to an implicit bias of the learning method that makes it more susceptible to commiting overfitting than, for instance, an SVM, provided that the two classifiers were trained with exactly the same data and feature sets.

Table 3 compares the results after applying complexity reduction to the data in order to reduce the trainig time of FOIL (by filtering the less frequent domain predicates and the less informative counterexamples as described above), with those achieved by using the full data. It can be observed that both precission and recall suffer an important decrease of about 8%-9%.

5 Conclusions

First, we have presented general notions about ILP (inductive logic programming), including a basic vocabulary of ILP, a typology of ILP systems and a description of the main techniques in ILP. Then, we have turned our attention to one ILP system in particular, FOIL. And last, we have delimited the

	PREC	**RECALL**	$\mathbf{F_1}$
Full model (best)	77.58	52.15	62.37
Reduced model (best)	71.18	44.55	54.80
Full model (sum)	81.32	50.28	62.13
Reduced model (sum)	72.50	41.47	52.76

Table 3: Performance of the reduced model with FOIL vs. the full model

problem on which we intended to apply ILP, that of temporal expression recognition, and have proposed two alternative logic representations for the problem: propositional and relational.

We have evaluated the performance achieved by FOIL for the propositional approach only, and have observed that, while the precission scores lie within an acceptable range, the recall values are abnormally low, perhaps owing to an implicit bias in the learner. We saw also that the execution time of FOIL for training a full model under the assumptions that occupy us in this problem renders its usage impractical, and that simplifying the training data set in order to reduce execution time carries a non-negligible penalty associated in both precission and recall.

Bibliography

[FER03] Ferro, L. et al. (2003). *TIDES Standard for the Annotation of Temporal Expressions v1.3.* Technical Report, MITRE Corporation.

[FRE98] Freitag, D. (1998). *Machine Learning for Information Extractionin Informal Domains*, Ch. 5, pp. 73–117. PhD thesis, Carnegie Mellon University, 1998.

[LAV94] Lavrač, N. and Džeroksi, S. (1994). *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, New York.

[MUG88] Muggleton, S. and Buntine, W. (1988). Machine Invention of First-Order Predicates by Inverting Resolution. In *Proc. Fifth International Conference on Machine Learning*, pp. 339–352. Morgan Kaufmann, San Mateo, CA.

[MUG90] Muggleton, S. and Feng, C. (1990). Efficient Induction of Logic Programs. In *Proc. First Conference on Algorithmic Learning Theory*, pp. 368–381. Ohmsha, Tokyo.

[MUG95] Muggleton, S. (1995). Inverse Entailment and Progol. *New Generation Computing Journal*, Vol. 13, pp. 245–286

[QUI90] Quinlan, J. R. (1990). Learning Logical Definitions from Relations. *Machine Learning* 5, pp. 239–266.

[QUI93] Quinlan, J. R. and Cameron-Jones, R.M. (1993). FOIL: A Midterm Report. In *Proc. European Conference on Machine Learning*, pp. 3–20, Springer Verlag.

[SAM86] Sammut, C. and Banerji, R. (1986). Learning Concepts by Asking Questions. In Michalski, R., Carbonell, J., and Mitchell, T., eds., *Machine Learning: An Artificial Intelligence Approach*, Vol. 2, pp. 167–191. Morgan Kaufmann, San Mateo, CA.

[SHA83] Shapiro, E. (1983). *Algorithmic Program Debugging.* MIT Press, Cambridge, MA.

[TUR02] Turmo, J. and Rodriguez, H. (2002). Learning Rules for Information Extraction. *Natural Language Engineering* 8, pp. 167–191. Cambridge University Press.

[WIL01] Wilson, G., Mani, I., Sundheim, B., Ferro, L. (2001). A Multilingual Approach to Annotating and Extracting Temporal Information. In *Proc. Workshop on Temporal and Spatial Information Processing*, Vol. 13, pp. 1–7. ACM.