

Ex and IET tutorial

Martin Labský, Marek Nekvasil, Vojtěch Svátek

{labsky, nekvasim, svatek}@vse.cz

University of Economics Prague

October 2007 - September 2008

Created within the MedIEQ EU project (www.medieq.org)

This document is intended to provide a brief walkthrough on how to operate Ex, the information extraction tool, how to create extraction ontologies for it, and how to run it using the Information Extraction Toolkit (IET).

Table of contents

Table of contents.....	2
1.What are Ex and IET?.....	3
2.Requirements.....	4
3.Running an extraction task.....	5
4.The extraction ontology structure.....	8
5.Classes and attributes.....	9
6.Extraction evidence types and confidence scoring.....	11
7.Extraction patterns.....	13
8.Axioms, formatting patterns and other evidence.....	19
9.Using training data.....	21
10.Evaluator.....	26
11.Extraction task modes and cross-validation.....	28
12.Methodology and conclusion.....	30
References.....	31

1. What are Ex and IET?

Ex¹ is an IE system based on extraction ontologies which aims to extract standalone named entities (standalone attributes) and instances (groups of attributes which "belong together"). The advantage of this technology is that it can utilize multiple sources of extraction knowledge which should lower the requirement for training data. Ex can be used for extraction from heavily structured (e.g. tabular) documents, semi-structured documents and also from free-text documents.

For a domain of interest, the user writes an extraction ontology. An extraction ontology is essentially a conventional domain ontology extended with extraction knowledge that can be used to identify the described objects in text. An extraction ontology can be viewed as a set of attribute definitions, class definitions, and extraction knowledge. Extraction knowledge can be expressed in various forms:

- manually authored patterns,
- axioms that are required to hold in the extracted data,
- training data that can be used to train automatic classifiers.

This tutorial is aimed at explaining the basics of working with this tool as well as of writing the extraction ontologies. This will be demonstrated using a simple extraction task from the weather forecast domain, and using snippets from more complex contact extraction ontology.

IET² is a container for information extraction engines such as Ex. IET can:

- define and run *extraction tasks* over specified documents using chosen IE engines in sequence,
- measure extraction accuracy in terms of precision, recall and F-measure using its Evaluator component,
- provide statistics useful to adapt extraction models of the underlying IE engines,
- run extraction tasks in cross-validation mode which includes support for feature induction using held-out folds,
- transform extracted information as it flows through the extraction task,
- be accessed using a command-line interface. A very simple GUI is part of IET but so far it has only served demonstration purposes.

¹ Ex has been developed since 2004 at the Department of Knowledge Engineering, Prague University of Economics and its development continued within the MedIEQ project (www.medieq.org).

² IET has been developed since 2006 within the MedIEQ project.

2. Requirements

To run the information extraction system, we will need the following:

- **HW**

1 GB of RAM, 128 MB of storage or more, depending on task, to store intermediate documents, gazetteer lists, trained models etc.

- **Supported OS**

Ex has been tested so far on Windows XP and Linux.

- **Java virtual machine**

You can get the latest version of Java Runtime Environment (JRE version 1.4.2 or higher) at <http://www.java.com/getjava>.

- **Compiled binaries** of the **Ex** IE engine and the **IET** (Information Extraction Toolkit)

You can get the latest distribution at <http://eso.vse.cz/~labsky/ex/>

If you need to build your distribution from sources, you will need the Java Development Kit (JDK) which you can get at the same source as the JVM, and Apache Ant (<http://ant.apache.org/>). Ant needs to be on your path. Start the build process by running one of the `buildall.cmd` or `buildall.sh` scripts in the root directory. This will create distribution jar files in the `dist` directories of individual subfolders.

- **Document/s** from which you would like to extract information

- **Extraction ontology** suited for the domain of information that is going to be extracted

- **Machine learning SW** to use trainable classifiers integrated in *Ex*, you will need to install either the Weka toolkit, and add it to classpath, or the CRF++ package.

- **Other SW**

To run some auxiliary helper scripts (not necessary), you will need a Perl interpreter.

In the next sections we show how to create an extraction ontology and how to run extraction tasks. First we will show how to run the extraction if such ontology already exists and then guide you step-by-step through the process of creating a new extraction ontology.

3. Running an extraction task

First step is to define an *extraction task*. The definition is stored in a XML format; for our purposes we will store it in the `tasks/weather.task` file. Each extraction task specifies a set of documents to be processed and a sequence (pipeline) of procedures to be carried out for the documents. Our example task definition can look like this:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
  <task name="weather tutorial">
    <mode> instances </mode>
    <datamodel> weather </datamodel>

    <pipeline mode="doc">
      <proc engine="ex.api.Ex" >
        <param name="cfg"> ../ex/config.cfg </param>
        <param name="model"> ../ex/data/weather/weather.xml </param>
        <param name="parser_nbest"> 1 </param>
      </proc>
    </pipeline>

    <set>
      http://weather.msn.com/
      c:/temp/weather/MSN.html
    </set>
  </task>
```

In the example above, we use the `mode` element to say we wish to extract instances (i.e. groups of attributes as opposed to standalone independent attributes). Provide arbitrary string that describes your domain as the name of the `datamodel`. The pipeline contains a single procedure for which the IE engine is *Ex* (`ex.api.Ex` is the name of the interface class name). All parameters inside procedures are engine-specific. For *Ex*, we need to specify a configuration file (containing settings affecting e.g. logging, named entity extraction, instance parsing etc.). The `model` parameter specifies that we want to extract using our weather extraction ontology. Finally, we can use a number of other parameters that override default settings or settings previously loaded from a configuration file.

The last part of the task is a document set definition, which may contain both URLs and local filenames. The set element also understands the following attributes:

- `basedir` - can point to a directory which will be used as the root directory to resolve relative document paths inside the document set
- `type` - can specify the content type of all documents in the set. Known content types include:
 - `text`: plain text documents, without annotations
 - `html`: all flavors of HTML code, without annotations
 - `atf`: Annotation Tool Format or Ellogon format, with annotations
 - `seminar`: Seminar announcement task format (SGML tags like `<speaker> Charlie </speaker>`)
 - `bike`: Bike task format (SGML tags like `<b_name/>` and `<e_name/>`)

Ex and IET tutorial

- `encoding` - specifies suggested encoding to use for documents that do not contain encoding information
- `forceenc` - if set to 1, the value of the `encoding` attribute is forced for all documents even though the documents might specify encoding
- `class` - specifies the classification of the documents in the set. This can be used to supply additional information about the semantic type of the processed documents that can be utilized in the extraction ontology. For example, the value `class="contact:70, about:10"` states that each document in the set is with 70% confidence classified as a "contact page", and with 10% as an "about page".

As an example document we will use the Microsoft Network Weather Forecast (available at <http://weather.msn.com>) for Prague. To demonstrate access to local and remote documents, we will use both the online version and a previously saved snapshot of the page. To run the extraction task, use the script named `runtask.cmd` (on Linux, use `runtask.sh`) in the IET folder. This script takes a single parameter that is the task definition file. In our case the command will look like this:

```
runtask.cmd tasks/weather.task
```

The task outputs several types of information. On *standard output*, the extracted items will gradually appear in XML format. For each document, you will typically see a list of instances followed by a list of standalone attributes, which may look like this:

```
<instance id="1" p="0,9411">
  <lab att="Forecast.day" ins="1" p="0,9448">Today</lab>
  <lab att="Forecast.temperature" ins="1" p="0,9993">19 °</lab>
  <lab att="Forecast.temperature" ins="1" p="0,9993">15 °</lab>
  <lab att="Forecast.condition" ins="1" p="0,9500">Partly Cloudy</lab>
  <lab att="Forecast.temperature" ins="1" p="0,8157">19 °</lab>
  <lab att="Forecast.condition" ins="1" p="0,9500">Mostly Cloudy</lab>
</instance>
```

```
<lab att="Forecast.day" p="0,9448">Tomorrow</lab>
```



Second, on *standard error*, you will see several logging messages which are logged at one of the *user*, *error*, or *warning* log levels. More detailed logging messages can be found in a log file (`iet.log` by default). In case you want to report problems, always include the log file with your message. The task output depends on which procedures the task uses, e.g. when using the Evaluator (see chapter 10), you will also see accuracy tables on output.

Third, the extraction task creates two additional files for each document it processes. These may appear in the folder where the original document resides, or in the temporary folder (based on configuration, `iet/temp` by default).

The first file will have the `.atf` extension and it will contain the extracted annotations (i.e. labeled instances and attributes) in the format suitable for Ellogon-based tools (can be viewed by the NCSR annotation tool).

The second file will have the `.lab.html` (i.e. labeled HTML) extension which includes annotations labeled in color with floating balloons containing annotation names. This is done by embedding extra HTML formatting and JavaScript into the original document. This format enables you to view the

annotated documents in common web browsers (tested with Mozilla Firefox). At its beginning, each labeled document will have a table inserted that shows a summary of the extracted attributes and instances. The labeled document created by our first task should look similar to:

Detailed weather forecast			°F °C
Day	Forecast	Description	Precip chance
Today Sep 02 Details	 Partly Cloudy Hi: 19° Lo: 15° temperature p=0,9993	Day: Partly cloudy skies. High 66F, humidity 55%. Winds W at 5 to 10 mph. Night: Mostly cloudy skies. Low 59F. Winds SW at 5 to 10 mph.	5% 10%
Tomorrow Sep 03 Details	 Showers Hi: 16° Lo: 9°	Day: Rain. High 60F, humidity 65%. Winds WSW at 10 to 15 mph.	95%

Apart from using the `runtask` command line script, there are other ways to access the IET and Ex functionality. To get a very simple GUI that enables the user to run tasks, run the script `rungui.cmd` (`rungui.sh` on Linux). Both Ex and IET also have documented APIs enabling their usage in 3rd party systems.

In the next steps we will focus on how to construct new extraction ontologies.

4. The extraction ontology structure

Before we delve into the details of creating extractable classes and attributes, we mention a few general remarks about the extraction ontology language and its structure. The concept of extraction ontologies was originally introduced in . Our recent research in this area is described in , and .

Extraction ontologies for the *Ex* engine are stored as XML files using the Extraction Ontology Language (EOL). A DTD file that defines the language comes with the distribution under `./data/eol.dtd`. Every extraction ontology should contain common XML header as well as a reference to the EOL DTD to ensure syntax validity. Inside the ontology there should be exactly one root element called `<model>` which contains everything else. This element should have a `name` attribute set for identification purposes. At the very beginning, our weather forecast ontology could look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE model SYSTEM "../eol.dtd">
<model name="weather">
  <!-- everything else will be written here -->
</model>
```

Before we move onto the classes, there is one useful thing we can do on the model level, and that is importing some other model. This is particularly useful in case you would like to separate patterns that do not explicitly belong to attributes of the given class for future use (or in case you would like to re-use someone another's work). The importing is done with `<import>` element which has a mandatory attribute `model` that specifies the file with the other model we would like to import. This can be used to import some generics or reusable parts of complex models.

There is yet another powerful element - `<base>`. Without its use all relative paths, used in the model as references to resources, are resolved relative to the location of the model file. By including this element and specifying a base path in the `url` attribute we can change this default behavior.

We can use these two features in our model:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE model SYSTEM "../eol.dtd">
<model name="weather">
  <base url=".././data"/>
  <import model="../datatypes.xml" />
</model>
```

With the first one we changed the default location of resources and with the second we included some pattern definitions into our model, this other model's path was given relative to the newly set base url.

With such a bare model one cannot do much, so next we will look into how to write an extractable class definition.

5. Classes and attributes

The main reason why the extraction model is considered an ontology is that it consists of class definitions and of definitions of their attributes. In the EOL notation, every attribute is attached to a single class. Both classes and attributes are denoted by XML elements in the body of the model definition, member attributes being embedded in their containing class. The model may contain several extractable classes.

An extractable class is defined using the `<class>` element within the `<model>` element. Its XML attributes include (required attributes in bold):

- **id** - denotes the name of the class, must be unique model-wide.
- **counts** - constrains how many instances of this class should be extracted from single document. The value can be a number such as "1" or a range like "0-3". This forces the system to search for and output only such paths of extractable objects that fulfill this criterion.
- **prune** - prune threshold ranging from 0..1, allowing for speeding up the parsing stage of extraction by pruning all class instance candidates with lower probabilities than this threshold.
- **log** - 0 or 1. If set to 1, additional user-level logging will be done describing the creation of instance candidates for this class.
- **enabled** - 0 or 1. If set to 0, the whole class including its content will be ignored as if it was commented out. This is to offset the nasty property of XML comments that cannot be nested.

The extractable attributes are defined using the `<attribute>` element. This element can either be nested in the `<class>` element it belongs to, or it can reside under the `<model>` element in case it should always be extracted standalone. The XML attributes of the `<attribute>` element include (required attributes in bold):

- **id** - denotes the name of the attribute, must be unique class-wide.
- **type** - specifies the "data type" of the possible attribute values. The default value is name, which can be used for arbitrary short texts. Further types, not all of which are supported, include: int, float, time, date, datetime and text (stands for longer texts such as whole paragraphs).
- **card** - determines the possible cardinality of the given attribute wrt. the containing class. It should be in the form of a number or a range of numbers; e.g. "1" or "0-1".
- **extends** - allows for the creation of attribute hierarchies. If the current attribute is a specialization of some more generic attribute (like seminar start time vs. a generic time), this refers to the parent attribute using its id.
- **eng** - this is an estimate of probability with which the attribute values appear as part of instances of the containing class, as opposed to standalone occurrences (e.g. a product name can appear as part of a sale offer including its price, but it can also appear standalone in text). The value must be in 0..1 range.

Ex and IET tutorial

- `prior` - this is an estimate of the prior probability with which we expect to observe this attribute in texts of the processed documents. By default, this is set to 0.01, which means we expect to see the values of this attribute approximately once in every 100 words.
- `units` - for numeric data types, units of measure can be specified that link to unit definitions contained in the model (units are defined using the `<unit>` elements at `<model>` level). The system of units can also have limited conversion capabilities. Unit details are currently not covered by this tutorial as they were little tested to date.
- `prune` - prune threshold ranging from 0..1, allowing for speeding up both the named entity recognition and parsing stages of extraction. This parameter prunes all attribute value candidates with lower probabilities than this threshold.
- `log` - 0 or 1. If set to 1, additional user-level logging will be done describing the creation of candidate values for this attribute.
- `enabled` - 0 or 1. If set to 0, the whole attribute is ignored.

Typically only a small subset of the available XML attributes is used to define classes and attributes. In the case of our weather forecast extraction ontology, we can define a single extractable class that consists of up to four attributes. The ontology, yet without extraction evidence, could look this way:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE model SYSTEM "../eol.dtd">
<model name="weather">
  <import model="../datatypes.xml" />
  <script src="helpers.js"/>
  <base url="../../data"/>

  <class id="Forecast">
    <attribute card="1-2" id="temperature" type="float" eng="0.95">
    </attribute>

    <attribute card="1" id="day" type="text" eng="0.9">
    </attribute>

    <attribute card="0-5" id="condition" type="text" eng="0.75">
    </attribute>

    <attribute id="city" type="name" card="0-1" eng="0.50">
    </attribute>
  </class>
</model>
```

Now we have written a class and its attributes, however we still need to supply some extraction evidence that the system can use to identify the attribute values in text and to group them into Forecast instances. Such extraction knowledge can come from two sources – from a human designer, or from annotated training documents. The next chapter and the following chapters outline the types of manually provided extraction evidence and show how to author the extraction evidence. Chapter 9 then describes how training data can be used as extraction evidence.

6. Extraction evidence types and confidence scoring

The following types of extraction evidence can be provided manually:

- **Attribute value patterns** match possible values of an extractable *attribute*.
- **Attribute context patterns** can identify an attribute value by matching a context the attribute often appears in. There are right, left and bidirectional context patterns.
- **Class patterns** are used to match specific attribute orderings inside class instances and they can also match the “glue” used between member attributes or some typical context around the potential class instance.
- **Axioms** are constraints imposed on the possible values of an extractable attribute or class. When applied to attributes, axioms can be used to do complex filtering of the possible value candidates. When defined for classes, axioms can effectively enforce complex relations to hold among member attributes of extracted instances. Axioms are written in the JavaScript scripting language.
- **Other constraints** include value length constraints for text attributes and numeric value constraints for numeric attributes.

Each piece of the above types of extraction evidence can be equipped with two probability estimates: **precision** and **coverage**. Precision expresses the certainty with which the presence of the evidence predicts the presence of the attribute value or instance. Coverage, on the other hand, expresses the degree with which the piece of evidence is required for the attribute value or instance to get extracted.

For example, let’s assume a *person age* attribute and two pieces of evidence: a left context pattern “age:” and an axiom stating that the person age in years is a number between 1 and 100. Let’s say the context pattern can be assumed to indicate a following person’s age in 75% of cases, so we can set its precision to 0.75. It however does not have a wide coverage since there will be many age mentions in other contexts, so we will set its coverage only to 5%. The other evidence, axiom, will have a very low precision since conventional text contains a lot of numbers between 1 and 100 that do not express age. Let’s set it to 2%. Its coverage will be however 0.9 as we expect at least 90% of extracted age mentions to fit in that range. Our attribute definition can look as follows.

```
<class id="Person">
  <attribute id="age" card="1" type="int" eng="1">
    <value>
      <pattern type="script" p="0.02" cover="0.90">
        inRange($, 1, 100)
      </pattern>
    </value>
    <context>
      <pattern ignore="case" p="0.75" cover="0.05"> age: $ </pattern>
    </context>
  </attribute>
</class>
```

In the example above, the axiom is encoded using a pattern of `script` type which invokes a JavaScript function. The `inRange()` function is assumed to have been defined by a script included earlier in the ontology. The `$` placeholder represents the attribute value.

All extracted attribute values and instances are assigned **confidence scores** by *Ex*. In fact, *Ex* only extracts items that reach at least 50% confidence. The score for a particular extracted item is computed based on which pieces of evidence were observed for it and which were not. When there is just one piece of evidence defined for an attribute value, and that evidence is observed, the confidence of the extracted attribute value will equal the precision of that single evidence. In most cases, however, attribute and class definitions use many pieces of evidence, some of which are observed for a particular item and some are not. Both observed and unobserved pieces of evidence are then taken into account to compute a combined score using probabilistic independence assumptions.

It is important to realize the following **rules of thumb** when assigning evidence precision and coverage: assigning high precision to evidence makes it more decisive in extracting the item in question. Assigning close-to-zero precision (less than the prior probability of the predicted item) makes the evidence *negative*, i.e. observing it will prevent the predicted item from getting extracted. Regarding coverage, high values will make the evidence *mandatory* for the item to get extracted, i.e. not observing evidence with 100% coverage means that the item in question cannot get extracted. Low coverage values, such as 2%, are a safe choice for pieces of evidence that have marginal coverage over the predicted items.

In the next section, we show how to author the extraction patterns for attributes and classes.

7. Extraction patterns

In general, patterns are rules that match pieces of the processed text. Patterns can be nested, i.e. their body may contain references to other patterns. Pattern semantics depends on where we define the pattern in the extraction ontology. **Within attribute definitions**, there are three locations where one can define patterns:

```
<class id="Forecast">
  <attribute ...>
1. <pattern id="pat1"> ... </pattern>
   <value>
2.   <pattern p="0.1" cover="0.9">...<pattern ref="pat1"/>...</pattern>
   </value>
   <context>
3.   <pattern p="0.7" cover="0.2">...</pattern>
   </context>
  </attribute>
</class>
```

The first location is where **generic, reusable patterns** are defined. These patterns do not have a meaning alone (they do not constitute extraction evidence); but they can be used from other patterns defined within the two following sections. Second, the `<value>` section defines **value patterns** which try to match the textual content of the attribute. Last, the `<context>` section contains **context patterns** which match the left, right or surrounding context that is indicative of the attribute value presence. Each value and context pattern constitutes a single piece of evidence and must be equipped with precision and coverage estimates.

Extraction patterns thus form a system of nested regular patterns. The basic building blocks of extraction patterns are:

- Word literals (tokens).
- Regular grammar operators, allowing us to define regular patterns at word level. All of them can be escaped using `\` (backslash) should they be matched literally:
 - `+` (the preceding constituent may repeat 1 or more times)
 - `*` (the preceding constituent may repeat 0 or more times)
 - `?` (the preceding constituent is optional)
 - `{n, m}` (the preceding constituent may repeat at least n and at most m times)
 - `(a|b|c)` (or operator: one of a, b or c)
 - `^` (at class pattern beginning, this matches the start of the instance candidate)
 - `$` (at class pattern end, this matches the end of the instance candidate)

- Word (token) wildcards: the `<tok/>` element represents a single word whose properties are constrained using XML attributes of the `tok` element and also by its content. If no attributes and no content are specified, `<tok/>` matches arbitrary single word. If token content is specified, it is interpreted as a character-level regular expression in the syntax used by Java (ignore case flag can be specified using the `ignore` attribute). The `<tok/>` will then only match words in which this regular expression can be *found*. To require the regular expression to match the *whole word*, use regular expressions like `<tok>^[0-9]+$</tok>`. The following attributes can constrain the set of words matched by the `<tok/>`:
 - `type`: constrains the superficial word type assigned by tokenizer. The token types assigned by the default tokenizer are: `alpha` (alphabetic chars only), `apos` (alphabetic chars with apostrophe), `acronym`, `p` (punctuation), `alphanum` (alphanumeric starting with a letter), `numalpha` (alphanumeric starting with a digit), `int` (integer), `float` (floating point number), `numdots` (e.g. an IP address), `email`, `host`, `url`, `sign` (dollar and euro). You can also edit the tokenizer's grammar under `ex/src/ex/reader/tokenizer/GrmTokenizer.jj` to customize the tokenizer to your needs. The whole tokenizer can also be switched for a different one simply by changing the `tokenizer` config parameter which contains the tokenizer class name. The `type` attribute accepts multiple values like `<tok type="alpha|apos|alphanum"/>`
 - `case`: constrains the case of the word to one of `UC` (uppercase), `CA` (capital), `LC` (lowercase), or `NA` (not available). Multiple values are supported as well, e.g. `<tok case="UC|CA"/>`
 - `ignore`: relevant when `<tok/>` contains a regular expression. Can contain one or both of `case`, `accent`. `ignore case` sets the corresponding regular expression flag. `ignore accent` switches on a feature of Ex which defines character equivalence classes for matching the regular expression. Use this if you want e.g. `<tok>^Jose $</tok>` to match `Jose` as well as `José`. Equivalence classes also include mappings between the Latin and Greek alphabets. Mappings can be adapted by editing `ex/res/unaccent.txt`
- Label bodies and label boundaries: the `<lab/>` element may refer to labels spanning one or more words and having arbitrary meanings. Standard labels that are provided by Ex by default include:
 - Labels corresponding to HTML tags and HTML tag types. Each HTML tag can be matched by either its tag name in *capital letters*, e.g. `"DIV"`, or by the name of the class it belongs to, e.g. `"BLOCK"`. For a list of known HTML tag names and their classes, see `ex/res/unaccent.txt` where you can also edit their mapping.
 - Labels added by preprocessing components of Ex. The sentence splitter produces one label per detected sentence named `"S"`. A part-of-speech tagger is currently not available in the distribution but this is the place where to add it.

- Labels created from pre-existing annotations present in the analyzed document. These can have arbitrary meanings such as extractions from previous tools or output of natural language preprocessors.

There are two attributes of the `<lab/>` element (required in bold):

- **name**: the name of the label(s) to match. The attribute accepts multiple values, such as `<lab name="P|TD|DIV"/>`, which would match the bodies of all paragraphs, table cells and divisions in the document. To only match label boundaries, add the `^` symbol at the beginning of the name attribute to match the label start only, or add the `$` symbol at the end of the name attribute to match the label end. E.g. the pattern `<lab name="^P|TD|DIV"/> <tok/>` matches the initial word of all paragraphs, cells and divisions in input documents.
- **neg**: 0 or 1. If set to 1, it inverts the match of the element; that is, it won't match if it originally matched, and it will match if it originally did not.
- References to other patterns: the element `<pattern ref="pat1"/>` requires the pattern with the given id to match at this position.
- References to other attribute value candidates created during the extraction process. Only candidates that reach a certain confidence threshold are taken into account. For example, a right context pattern of a person name, referring to organization and department attributes, could look like:

```
<pattern> $ from ($organization|$department) </pattern>
```

Having enumerated the building blocks of extraction patterns, we can list options that can be set for each pattern using the XML attributes of `<pattern>`:

- **id**: pattern identifier, used to embed this pattern in other patterns, and also for logging. Id, if provided, should be unique within the attribute definition. It is possible to refer to patterns defined within other attributes' definitions by prepending the pattern id with the path of dot-delimited class and attribute ids, e.g. "weather.temperature.patId".
- **ref**: this is just a reference to another pattern; any other attributes are discarded.
- **p**: encodes evidence precision as described in Chapter 6. Mandatory when the pattern is defined inside value or context sections, ignored otherwise. Precision can also be set to -1 if *observing* the evidence is not indicative of the attribute presence.
- **cover**: encodes evidence coverage as described in Chapter 6. Mandatory when the pattern is defined inside value or context sections, ignored otherwise. Recall can also be set to -1 if *not observing* the evidence is not indicative of the attribute presence.
- **ignore**: possible values are `case`, `lemma` and `accent`. This flag controls whether the word literals *directly contained* within the pattern body should be matched case-insensitive, inflection-insensitive in case lemmatizer is available, or accent-insensitive. As in case of the `<tok/>` element, this turns on using character equivalence classes as defined by the mapping

at `ex/res/unaccent.txt`. Use `ignore accent` when you want words like `Jose` to also match `José`. Equivalence classes also include mappings between Latin and Greek alphabets, so that e.g. `Pythagoras` will also match `Πυθαγόρας`. The file `ex/res/unaccent.txt` can be edited to update the mappings.

- `case`: requires all tokens in the text matched by the pattern to be of particular case. Multiple values are supported, e.g. `"UC|CA"` will only match phrases where all tokens are either uppercase or capital (have the first letter capitalized). Further choices are `LC` and `NA` for lowercase and not available. Prepending the case attribute with `^` will require only the first word of the pattern match to comply.
- `src`: allows storing the pattern source in a separate file, which is convenient for large patterns and necessary for large gazetteer lists which can hold hundreds of thousands of items.
- `encoding`: specifies encoding of the external file referred to by the `src` attribute. The default is `utf-8`.
- `type`: there are four supported types of patterns:
 - `pattern` (the default), which corresponds to the syntax described so far.
 - `list` is only capable of accepting lists of phrases to be matched, one per line. The `pattern` type is preferred over the `list` type.
 - `format` patterns: there are special types of patterns which would be hard or impossible to encode using conventional regular patterns, such as telling if the value fits exactly inside its HTML parent element. See Chapter 8 for formatting pattern details.
 - `script`: the `<pattern>` element is also (ab)used to hold attribute axioms, which contain JavaScript expressions to be evaluated. See Chapter 8 for axioms.
- `cond`: condition required for the pattern to be taken into account as evidence. This only applies to patterns defined at class level and the condition aims to select only a subset of class instance candidates that this pattern should be applied to. Possible values are: `none`, `any`, `all`. `None` is the default and it means that there is no condition, so the pattern will apply to all class instance candidates. `Any` specifies that the pattern will only apply to class instance candidates that contain at least one of the attributes referred to by the pattern. `All` means that the pattern will only apply to instance candidates that contain values for all attributes referred to by the pattern.
- `log`: 0 or 1. If 1, each match of the pattern is logged at user level.




Let's now try to write down a very simple pattern for identifying a day of the week (which can also be useful in the domain of weather forecasts):

```
<attribute card="1" id="day" type="text" eng="0.9">
  <value>
    <pattern cover="0.95" p="0.95">
      Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
    | Today | Tomorrow </pattern>
    </value>
  </attribute>
```

This pattern contains just a simple list of possible values; however it already enables us to use our ontology for extraction. When run on the MSN weather forecast site, we get the results shown below. Also notice the XML attributes of the `<pattern>` and `<attribute>` elements. The precision (`p`) and coverage (`cover`) are used to compute the confidence of the extracted value. As seen from the screenshot, the confidence for “Tomorrow” is the same as the precision of the single evidence we have, as described earlier in Section 6. In real cases, multiple pieces of evidence would combine to produce confidence scores.

Let’s further add to our ontology a context pattern, which uses `$` (dollar) in the pattern body to denote the position of the predicted attribute:

```
<attribute card="1" id="day"
  type="text" eng="0.9">
  <context>
    <pattern cover="0.1" p="0.7">
      day :? $
    </pattern>
  </context>
</attribute>
```

Tomorrow Sep 03 Details	 Showers Hi: 16° Lo: 9°	Day: Rain. High 60F, humidity 65%. Winds WSW at 10 to 15 mph. Night: Partly cloudy skies. Low 48F. Winds NW at 5 to 10 mph.	95% 90%
Tuesday Sep 04	 Showers Hi: 10° Lo: 5°	Mostly cloudy with showers. High 51F and low 42F. Winds NW at 10 to 15 mph.	90%
Wednesday Sep 05	 AM Rain Hi: 10° Lo: 9°	Mostly cloudy skies. High 51F and low 48F. Winds N at 10 to 15 mph.	65%

Note that the extraction patterns interpret new lines as implicit OR symbols. This behavior is subject to change. Currently, the following patterns are equivalent:

```
<pattern>
  Monday | Tuesday | Wednesday
</pattern>
```

```
<pattern>
  Monday
  Tuesday
  Wednesday
</pattern>
```

Next, let’s create a more real pattern for matching city names which will utilize large lists of known cities. At the same time the sample shows how patterns can refer to each other. There are many uses for embedded patterns; in our case we just combine multiple lists into one:

```
<attribute id="city" type="name" card="0-1" eng="0.50">
  <pattern id="eu_big_cities" src="eu_cities.txt" encoding="utf-8"/>
  <pattern id="us_big_cities" src="us_cities.txt" encoding="iso-8859-1" />
  <pattern id="all_cities">
    <pattern ref="europe_big_cities"/> | <pattern ref="us_big_cities"/>
  </pattern>
  <pattern id="city_suffix"> City | Village | Town </pattern>
  <value>
    <pattern p="0.55" cover="0.05">
      <tok case="CA|UC"/>{1,2} <pattern ref="city_suffix"/>
    </pattern>
    <pattern p="0.80" cover="0.50" ignore="case" case="^CA|UC">
      <tok case="all_cities"/> <pattern ref="city_suffix"/>?
    </pattern>
  </value>
```

```
</attribute>
```

Finally, we will illustrate specifics of **patterns defined at class level**. The following pattern states that 30% of instances (of a containing Person class) start with either the name of the person, immediately followed by his/her name, or with the person's name followed by the degree (with an optional dot or comma in between). For class-level patterns, the ^ symbol matches the start of the instance candidate. (similarly, the \$ symbol would match its end). A possible modification would be to add the XML attribute cond="all" that would cause the pattern to only apply to instances that contain both the name and the degree.

```
<pattern id="title_name_first" cover="0.3" p="0.6">  
  ^ ( ($degree .? $name) | ($name ,? $degree) )  
</pattern>
```

The following pattern simply says that in 75% of Person instances, the address follows after the person's name, and that these are not more than 20 words apart. We are not sure about precision so we do not set it (the evidence will only be used when the pattern does not hold, to suppress scores of instances that violate this axiom).

```
<pattern id="address_follows_name" cover="0.75" p="-1">  
  $name <tok/>{0,20} $address  
</pattern>
```

In the next chapter we will focus on using axioms and formatting patterns.

8. Axioms, formatting patterns and other evidence

This chapter discusses types of manually encoded evidence other than regular patterns. These are esp. axioms that enforce certain attribute properties, or enforce relations between attribute values of a single class instance. We will follow with formatting patterns and numeric constraints.

Axioms. The `<pattern>` element is overloaded to also encode axioms definitions, when its `type="script"`. The body of the element should then contain a JavaScript expression instead of a regular pattern. When declared in **attribute definitions**, this expression takes the candidate attribute value as an input parameter, denoted by `$` (dollar), and evaluates to `true` if the axiom holds, and to `false` otherwise. For example, you can ensure that the extracted phrase contains an even number in this way:

```
<pattern type="script" cover="1.0" p="-1"> isEven($) </pattern>
```

Note the use of `-1` as precision; this has the effect of the evidence being ignored when observed and only taking effect when it is not observed (denying extraction through its 100% recall). The example calls a JavaScript function `isEven()` in the axiom body that we assume was defined in a separate JavaScript file that was loaded earlier by the extraction ontology. A script can be loaded at `<model>` scope by using e.g.:

```
<script src="helpers.js" />
```

When axioms are declared inside **class definitions**, they may refer to all member attributes of the candidate class instance. The following pattern applies to all instance candidates that have both their name and email specified. It improves the instance's score if it detects a similarity between the name and the email (such as James Brown and brownj@company.com), and suppresses it otherwise.

```
<pattern id="name_email" type="script" cover="0.8" p="0.7" cond="all">
  nameMatchesEmail($name, $email) > 0
</pattern>
```

Another typical axiom usage is borrowed from the product sale area, where it requires the price with tax to be greater than the price without tax.

```
<pattern id="price_relation" type="script" cover="1" p="-1" cond="all">
  $price_without_tax > $price_with_tax
</pattern>
```

Formatting patterns. Further type of pattern is `type="format"`. In this case, the pattern content is limited to one of the following pre-defined keywords. Patterns of this type refer to the way an attribute value or class instance is embedded in the surrounding layout and formatting of the containing web page. They make it easy to express constraints that are hard or impossible to specify using regular patterns. Possible values are:

- `no_crossed_inline_tags` matches when the value does not cross any HTML *inline* tag
- `no_crossed_block_tags` matches when the value does not cross any HTML *block* tags

Ex and IET tutorial

- `has_one_parent` the pattern is matched when each word of the candidate text has the same parent HTML element
- `fits_in_parent` matches when the text exactly fits in its parent HTML element

These patterns are typically only used as negative evidence which applies when the pattern does not hold. When it does hold, it will not affect the to-be-extracted object as its precision is set to -1. This is illustrated in the example below, where we define an attribute to extract weather conditions. Including all four formatting patterns is generally a good idea since it promotes candidate texts that “fit well” inside the formatting blocks in the analyzed pages.

```
<attribute card="0-5" id="condition" type="text" eng="0.75">
  <value>
    <pattern cover="0.75" p="0.75" ignore="case">
      ( mostly | partly | almost | light | heavy | fine )? (clear | cloudy |
      clouds | fair | scattered | overcast | obscured | fog | foggy | sprinkles |
      rain | tempest | drizzle | moist | showers | spray | hail | rain | rainfall
      | snow | snowfall | downfall sleet | spray | storm )
    </pattern>
    <pattern cover="1.0" p="-1" type="format">has_one_parent </pattern>
    <pattern cover="0.3" p="-1" type="format">fits_in_parent </pattern>
    <pattern cover="1.0" p="-1" type="format">no_crossed_inline_tags</pattern>
    <pattern cover="1.0" p="-1" type="format">no_crossed_block_tags</pattern>
  </value>
</attribute>
```

Length constraints. To constrain the length in words of attribute values, the `<length>` element can be used inside the attribute’s `<value>` section to set the minimal and maximal attribute lengths. In future it may be possible to provide more sophisticated length distributions:

```
<value>
  <length>
    <distribution min="1" max="5" />
  </length>
</value>
```

Numeric value constraints.

In case of numeric attributes, i.e. for the `int` and `float` attribute data types, one can constrain their minimum and maximum values. This is done by adding the `<distribution>` element directly to the `<value>` section of the numeric attribute. In future, Ex might support more sophisticated value distributions.

```
<value>
  <distribution min="1" max="100" />
</value>
```

9. Using training data

If annotated training documents exist, these can be used for several purposes:

- to train machine-learning classifiers,
- to dump statistics that might help improve the current extraction ontology by hand,
- annotated documents can be used as test data to measure extraction accuracy, with the help of the Evaluator component of IET (see the next chapter).

In this section we only focus on the first item above. Within IET and Ex, there are two ways of incorporating trainable classifiers into the extraction process:

- using IET: add a new standalone trainable IE engine as a further step of the extraction task,
- using Ex: couple the extraction ontology with one of the trainable classifiers supported by Ex.

Using IET extraction tasks, you can chain multiple procedures in one extraction task. For example, you can use your training data to train an arbitrary trainable IE engine, wrap it inside an extraction task procedure, and follow up with a procedure that utilizes an extraction ontology of the *Ex* engine. Definition of such extraction task can look as follows.

```
<pipeline mode="doc">
  <proc engine="elgwrp.api.EllogonWrapper" >
    <param name="elgDir"> c:/IE_app_NCSR/IE_App_Contact_Other </param>
    <param name="model"> Medieq_FI_80.model </param>
    <param name="author"> elg </param>
    <param name="prefix"> elg. </param> <!-- prefix produced labels -->
  </proc>

  <proc engine="ex.api.Ex" >
    <param name="cfg"> ../ex/config.cfg </param>
    <param name="model"> ../ex/data/med/contact_elg.xml </param>
    <param name="parser_nbest"> 1 </param>
  </proc>
</pipeline>
```

Inside the extraction ontology used in the second procedure, you can utilize the information extracted by preceding engines to extract your attributes and classes as described in Section , using extraction pattern elements like `<lab name="elg.label_name"/>`.

An alternative approach would be to run an extraction ontology as the first procedure, and then use its extractions as features for training the trainable classifier which would follow (not tested so far).

Using Ex alone, you can couple an extraction ontology with trainable classifier(s) that have been integrated in Ex. One advantage of this approach is that the classifiers will have access to detailed features generated by the Ex engine. These include pattern matches, axiom matches and also matches of word n-gram features that can be induced by Ex from training data. Currently integrated classifiers are:

- Weka toolkit of machine learning algorithms (includes classifiers like SVM, decision trees, neural networks). Experiments were carried out so far with the SMO and JRip classifiers.
- CRF++ toolkit for labeling sequential data.

Further classifiers can be added by implementing the interface `ex.train.SampleClassifier`.

When using a trained classifier inside an extraction ontology, one needs to decide first which *document representation* is the best to use for the classifier at hand. By document representation, we mean the way how a processed document is transformed into a sequence of data samples to be given to the classifier (both for training and testing). Currently there are two document representations supported:

- Word-level. Here, a document of N words is transformed into a sequence of N samples. Each sample describes one word and can have many features, including:
 - simple word features: the word itself, word superficial type, case, word length,
 - pattern matches beginning or continuing at that word, pattern matches before or after the word,
 - matches of induced n-grams at various positions wrt. the word.
- Phrase-level. This is a “variable length sliding window” method. A document is transformed into a sequence of word n-grams (phrases) of some minimal and maximal length (determined by length constraints of the shortest and longest extractable attributes). As in case of the word-level representation, phrase-level samples can have many features including:
 - exact pattern matches of the phrase,
 - pattern matches in other position types wrt. the phrase sample: patterns ending just before or starting just after the phrase, matches of the prefix or suffix of the given phrase, matches that contain the phrase, matches that are contained inside the phrase, matches that overlap the left or right phrase boundary.
 - Matches of induced n-grams at certain positions wrt. the phrase sample. For example, suppose an n-gram “starts at” was induced from held-out data, where it indicated a following time entry. The feature corresponding to this n-gram will then be set to 1 whenever the n-gram is found just before the phrase sample.

Document representations and classifiers are not strictly tied to each other, but it is true that some representations are more suitable for a given classifier than others. Further document representations can be created by simply implementing the `ex.train.DataSource` interface of Ex, and then using its class name in the classifier definition inside an extraction ontology.

Below we show an extraction ontology snippet demonstrating the usage of a CRF++ classifier attached to word-level document representation, and an SMO classifier, implemented by Weka, attached to phrase-level representation.

```
<class id="Seminar">
  <classifier id="cls1" classtype="attribute"
    name="ex.train.crf.CRFConnector" datasource="ex.train.WordSource"
    elements="speaker location stime etime" model="train/crf.bin">
    <feature type="model" />
    <param name="template"> c:/keg/medieq/ex/res/template.crf </param>
```

Ex and IET tutorial

```
<param name="crf_dir"> c:/projekty/crf_pp </param>
<param name="tmp_dir"> tmp_crf </param>
<param name="encoding"> utf-8 </param>
</classifier>

<classifier id="cls2" classtype="attribute"
  name="ex.train.weka.WekaConnector" datasource="ex.train.PhraseSource"
  elements="speaker location stime etime" model="train/smo_slse.bin">
  <param name="algorithm" value="weka.classifiers.functions.SMO" />
  <feature type="model" />
  <feature type="ngram" ignore="case" length="1-2" minocc="2"
    maxcnt="150" mi="0.1, 10"
    position="before, after, equals, prefix, suffix, contained"
    book="train/slse.fgram" />
</classifier>

<attribute id="speaker" type="name" card="0-1" eng="1">
  <value>
    <pattern id="cpat" p="0.92" cover="0.6" feature="no">
      <lab name="cls1.speaker" />
    </pattern>
    <pattern id="cpat" p="0.92" cover="0.2" feature="no">
      <lab name="cls2.speaker" />
    </pattern>
    <!-- more value patterns here -->
  </value>
  <context>
    <!-- more contextual patterns here -->
  </context>
</attribute>

<!-- ... more attributes here -->
</class>
```

The classifier definitions can appear inside the `class` and `attribute` elements. When defined for a `class`, the classifier is expected to predict a subset of the attributes defined for the class, selected using the `elements` attribute. When defined for an attribute, the classifier is assumed to be a binary predictor of the attribute it is contained in (the `elements` attribute is ignored). Mandatory attributes of the classifier definition are:

- `id` - denotes the ID of the classifier, should be unique ontology-wide. The ID is used to prefix the names of labels generated by the classifier. E.g. when the classifier extracts “speaker”, the label name will become “ID.speaker”. Also, classifier ID is often used by classifier implementations to generate intermediate file names.
- `classtype` - currently the only supported type of class is “attribute”.
- `name` - the name of the Java class that implements this classifier. This is typically a thin wrapper class written to communicate with the real classifier implementation. Two implementations provided with the distribution are `ex.train.weka.WekaConnector` and `ex.train.crf.CRFConnector`.
- `datasource` - specifies how document is transformed into a sequence of samples suitable for the chosen classifier. Two data source implementations are provided: `ex.train.WordSource` and `ex.train.PhraseSource`.

- `element` - list of possible class values to be output by this classifier. Only used for classifiers defined at class level, this attribute selects the attributes of the class definition to be handled by the classifier.
- `model` - name of the trained model to use. Meaning of the model name is classifier-dependent. For both of the provided classifier implementations, model name specifies the file name where a trained model should be stored (in training mode) or from which it should be loaded (in test mode). In cross-validation mode, the model name is used as the base name for storing models for individual folds.

Each classifier definition may contain a set of parameters which are name/value string pairs with classifier-specific meanings. E.g. the **Weka classifier** has a mandatory “algorithm” parameter that specifies which classification algorithm to use from the Weka toolkit (its class name). The “options” parameter can be used to communicate a set command-line-style options to the chosen Weka classifier.

The **CRF classifier** needs at least the “template” and “crf_dir” parameters to point to a *template* of the CRF feature template file; and to the directory where the CRF++ package resides. A feature template tells the CRF++ package which features at which positions to use. *Ex* generates such feature templates from a *template* of the feature template. A sample template for the CRF feature template is at `ex/res/template.crf`. Feature placeholders in this (meta) template are expanded based on the current extraction ontology content. For example, the line

```
U23:%x[0,$value.pattern]
```

will turn into as many lines as there are value patterns in the current ontology, with `$value.pattern` getting replaced by feature indices corresponding to each value pattern. The first number encodes the relative position of the feature wrt. the current word. Assuming we are using the `WordSource` document representation, this will generate three binary features for each value pattern, first of which will become 1 when a match of that pattern starts at the current word, another when the pattern match continues through the current word, and the last one when there is no match present. For description of the CRF++ feature template syntax, see CRF++ documentation. The expanded feature template file is finally fed to the training and/or testing modules of the CRF++ package, along with data files to be classified or used for training.

In addition, each classifier definition may define types of features to be generated for the classifier samples. Inclusion of (only) the declared feature types is handled by the selected `datasource` implementation. Both provided data sources allow the inclusion of the following feature types:

- *default features* are always included by the data source. These include at least the n-ary class feature consisting of possible attribute names to classify into. The `WordSource` implementation also includes by default several word-level features: the word string, lower-cased string, lemmatized string (if lemmatizer is on), word type and case.
- *model features* are derived from the extraction ontology and are based on: all value and context pattern matches, attribute length constraints, axiom constraints.

- *ngram features* are based on matches of known word sequences that are assumed to be indicative of presence of some of the predicted attributes. In the example presented above, these ngrams and associated features are loaded from a “feature ngram book” `train/slse.fgram` when in test model. In training mode, these features are automatically *induced* from training data based on the further arguments of the `feature` definition element and saved as the ngram book:
 - `length="1-2"`: ngram size range (induce unigrams and bigrams)
 - `minocc="2"`: minimal occurrence count of an ngram required to become a part of feature
 - `maxcnt="150"`: maximum number of n-gram features to induce
 - `mi="0.1, 10"`: range of point-wise mutual information required between the ngram and one of the predicted class values
 - `position="before, after, equals, prefix, suffix, contained"`: enumerates which ngram positions to take into account wrt. the predicted class. E.g., “before” means that ngrams appearing just before each predicted attribute value will be examined whether they are good enough predictors of the attribute.

Using classifier decisions. The example above outlines a simple way of using classifier decisions as value patterns. The precision and recall of such pattern should in general resemble the precision and recall achieved by the classifier on test data. Note the `feature="no"` attribute that prevents these patterns from getting used as further features.

10.Evaluator

This section shows the usage of the Evaluator component that is part of the IET. It serves to measure precision, recall and F-measure by comparing pre-existent gold-standard annotations in documents with their automatically produced versions. To use Evaluator, simply enqueue it at the end of the extraction task's pipeline of procedures, as if it was another IE engine:

```
<pipeline mode="doc">
  <proc engine="ex.api.Ex" >
    <param name="cfg"> ../ex/config.cfg </param>
    <param name="model"> ../ex/data/med/contact_elg.xml </param>
    <param name="parser_nbest"> 1 </param>
  </proc>

  <proc engine="medieq.iet.components.EvaluatorImpl" >
    <param name="eval_mode"> occurrence loose </param>
    <param name="homo_spec"> 1 </param>
  </proc>
</pipeline>
```

The processed documents are expected to contain gold-standard annotations. The Evaluator parameters include:

- `eval_mode`: this controls how the evaluation statistics are measured. There are two binary modes of evaluation that can be combined: `occurrence` vs. `presence`, and `strict` vs. `loose`. In `occurrence` mode, each occurrence of an attribute value is counted independently from the others. In `presence` mode, it is considered a success when a single mention of the gold-standard attribute value is extracted per document. Second, the `strict` mode only considers exact extractions as successful, while the `loose` mode also gives partial credit to partial or overflowed extractions, based on their relative character-based overlap with the gold-standard version. Default setting is "occurrence loose".
- `homo_spec`: a binary flag that controls whether attributes with a common prefix should be treated alike for evaluation. If enabled, extracting e.g. "name" or "name_responsible" instead of "name_webmaster" will count as correct. This can be used e.g. to discard differences between attribute specializations in cases the specializations are not labeled in the annotated data.

The following is a sample output from the Evaluator on the seminar announcement task.

	P	R	F	GOLD	AUTO	AMAT	GMAT
etime-strict	98.46	88.89	93.43	216	195	192	192
etime-loose	99.49	89.12	94.02	216	195	2	0.5
location-strict	58.78	74.15	65.58	325	410	241	241
location-loose	79.72	85.61	82.56	325	410	85.84	37.23
speaker-strict	71.11	68.73	69.90	371	360	256	255
speaker-loose	76.58	73.83	75.18	371	360	19.69	18.9
stime-strict	95.75	88.07	91.75	486	447	428	428
stime-loose	95.75	88.07	91.75	486	447	0	0
avg-strict	79.11	79.83	79.47	1398	1412	1117	1116
avg-loose	86.72	83.88	85.28	1398	1412	107.5	56.6

Ex and IET tutorial

The odd table rows list precision, recall and F-measure for the strict mode of evaluation; the even rows show the corresponding loose values. In addition, occurrence counts of gold-standard annotations and of automatic annotations are shown in the GOLD and AUTO columns. The last two columns show match counts: how many of the GOLD resp. AUTO annotations have corresponding AUTO vs. GOLD annotations. The match counts are used to compute precision resp. recall. The match counts for the loose rows only contain the sums of the fractional scores produced by partial matches (e.g. 0.5 is added for an item of which we only extracted 50%) and do not include the counts of full matches.

In addition, the Evaluator can present useful summary statistics regarding frequent annotation errors. These can aid in developing the extraction ontology using a set of annotated sample “training” pages. The sample output below shows that start time (`stime`) was mistaken 4 times for background (i.e. was missed), and also lists which values were missed, in which documents, and how many times in each document. The latter lines show that `speaker` was mistaken for a different value of `speaker` also 4 times. The left-hand side always shows the gold-standard annotations, the right-hand-side shows the actual automatic annotations.

```
stime->bg(4)
```

```
10 -> seminars/src/cmu.cs.robotics-1236_0
12:00 noon -> (2) seminars/src/cmu.cs.proj.cimds-468_0
12:00noon -> seminars/src/cmu.cs.robotics-299_0
```

```
speaker->speaker(4)
```

```
Charles R. Price -> Charles R. Price - NASA(2) cmu.cs.robotics-648_0
Dan R. Olsen Jr. -> Dan R. Olsen cmu.andrew.org.cmu-hci-7_0
Dr. Birnbaum -> Dr. Birnbaum's cmu.cs.scs-2926_0
```

11.Extraction task modes and cross-validation

So far we have run our extraction tasks only in `test` mode, which is the default mode of operation where IE engines try to extract data using their trained or hand-crafted extraction models. In Chapter 9, we have mentioned that trainable classifiers can also be trained using IET's extraction tasks when they run in modes other than the test mode. This chapter outlines possible modes of task operation.

The following is a list of valid values for the `<mode> ... </mode>` parameter, which is placed inside the root `<task>` element of task definition. For all modes except for cross-validation, all documents in the task's document set(s) are processed exactly once and in document order.

- `test` or `instance`: engines in the task's pipeline are invoked to extract attribute values and to group them in instances.
- `attributes`: engines are used to extract attribute values only.
- `dump`: this mode is useful for the development of trained models. The IE engines are asked to dump the processed documents in format suitable for trainable classifiers (data formats are IE engine dependent).
- `train`: in this mode, all documents are fed once to the training procedure of each IE engine. At the end, each engine is asked to train an extraction model based on the documents observed. The assumption is that all the documents contain gold-standard annotations that can be read by IET or by the IE engines.
- `cv-K`: stands for K -fold document-level cross-validation. The folds are created by evenly distributing documents in K folds in document order. E.g., when there are 95 documents and a 10-fold cross-validation was ordered, the 1st ten documents (as ordered in task file) will end up in the 1st fold and so on until the 6th fold, which will only have 9 documents. The setup is intentionally not random in order to be able to replicate results. Perl scripts for mixing up documents randomly are provided (`ex/src/tools/shuffle.pl`). At each of the K cross-validation steps, a model is trained using $K-1$ folds and then tested using the remaining fold.
- `cv-K fi-M`: stands for K -fold document-level cross-validation with feature induction done using M folds as held-out data at a time. The procedure is similar to cross-validation but it involves a bit more processing. At each turn of cross-validation, only $(K-M-1)$ folds are used for training. Before training occurs, the M held-out folds, chosen from the $(K-1)$ folds that would normally be used for training, are used to induce new features for the classifiers. The nature of the to-be-induced features is IE engine dependent; the task of IET is only to automate the cross-validation procedure with held-out data. In the Ex engine, the induced features are word n -grams that are detected using extreme values of point-wise mutual information. After having induced new features using the held-out set, documents in the

training set, already with updated feature sets, are fed to the training procedures of the IE engine. The remaining test fold is then used to do extraction.

Not all IE engines need to support training or feature induction. The only required running mode is the "attributes" mode. The Evaluator, if enqueued at the end of the task's pipeline, is aware of the current mode and will dump results when running in test modes (instances or attributes). It will do nothing when in dump and train modes. For the two cross-validation modes, it will dump overall statistics at the end.

12. Methodology and conclusion

In this tutorial we have shown how to use the Ex IE engine and how to run extraction tasks using the Information Extraction Toolkit. We have given a quick introduction to authoring extraction ontologies for the Ex engine. The reader should be able to create own extraction ontologies for domains of choice. What follows is an attempt to suggest a methodology how extraction ontologies should be used in practical IE application scenarios.

In absence of training data, the suggested methodology is to start with a simple proof-of-concept ontology that is gradually grown by adding more extraction evidence. Soon it may become necessary to annotate several (10-20) documents to obtain some gold standard against which it is possible to benchmark the developed ontology. This is important e.g. to track down whether certain changes to the ontology actually improved the results on this “training data” or made them worse.

To get here, the work required should be typically in the range of one person week. At this stage, a definition of the desired IE application functionality will be more mature as there already is a functional prototype that can be evaluated for its potential usefulness. A decision should be made now whether to continue or stop the development. If the decision is to continue, then some more (10-20) documents can be collected, annotated and tested to get an estimate of what the results will be on unseen test data. Based on accuracies observed for individual attributes and classes, we should decide for which we need to collect “full” training data (typically at least 100s of documents). Using this training data, trainable classifiers such as CRF or Weka algorithms can be trained to improve performance for problematic fields.

On the other hand, **when training data exists** for some of the to-be-extracted attributes, that data should be used first to train appropriate classifiers. For the remaining attributes, we can try adding extraction evidence manually. To improve accuracy of trained classifiers when used within extraction ontologies, one can add new extraction evidence (such as regular patterns, axioms, formatting patterns; patterns defined using NLP labels) that the classifiers can utilize as further features. Alternatively, feature induction can be done over the training data in order to enlarge the feature set or to substitute low-level features by the induced features which can be clustered.

Along with this tutorial there are several versions of the weather forecast extraction ontology included. These versions follow the progress of this tutorial and can be used as a starting point for doing experiments. We suggest that you first try to run an extraction with different versions of our model to see the difference and then try to edit the extraction evidence and later also the structure of the model. Then you can try to forge your own extraction model for some other domain. Also see the more complex *contact extraction* ontologies for examples.

We wish you good luck in authoring extraction ontologies and we are happy if you are interested in Ex and IET.

References

- [1] Embley, D. W., Tao, C., Liddle, D. W.: Automatically extracting ontologically specified data from HTML tables of unknown structure. In Proc. ER 2002, pp. 322–337, London, UK, 2002.
- [2] Karkaletsis V., Karampiperis P., Stamatakis K., Labský M., Ružicka M., Svátek V., Mayer M.A., Leis A., Villarroel D.: Automating Accreditation of Medical Web Content. Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008).
<http://www.medieq.org/system/files/MedIEQ%20Pais%202008%20FINAL.pdf>
- [3] Kudo T.: CRF++: Yet Another CRF toolkit. <http://crfpp.sourceforge.net>
- [4] Labský, M., Svátek, V. Combining Multiple Sources of Evidence in Web Information Extraction. Toronto 2008. In: Foundations of Intelligent Systems. Springer-Verlag, 2008, pp. 471–476. ISBN 978-3-540-68122-9. ISSN 0302-9743.
http://www.medieq.org/system/files/ismis08_short2.pdf
- [5] Labský M., Svátek V., Nekvasil M., Rak D.: The Ex Project: Web Information Extraction using Extraction Ontologies. In: Proc. PriCKL'07, ECML/PKDD Workshop on Prior Conceptual Knowledge in Machine Learning and Knowledge Discovery. Warsaw, Poland, 2007.
<http://nb.vse.cz/~svatek/prickl07.pdf>
- [6] Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques (2nd Ed). Morgan Kaufmann 2005. ISBN 0-12-088407-0.
<http://www.cs.waikato.ac.nz/ml/weka>