XCRF User Manual
Version 1.0

March 12, 2008

Contents

1 About XCRF

XCRF is a framework for building conditional models for labeling XML documents. XCRF is a Java implementation of Conditional Random Fields (CRF [LMP01]) for trees.

XCRFs are very flexible and can also label sequences. Using XCRFs usually assumes

- *a training phase* that consists in building the model using a sample of annotated data; and
- *an inference phase* that consists in using the model to annotate data.

CRF definition relies on a set of feature functions. In this XCRF implementation features can be easily designed by end users. We also provide perl scripts that automatically generate features from a given sample of XML trees.

An example of application of XCRF is the RSS feed generator available at http://r2s2.futurs.inria.fr.

This implementation follows the paper [JGTT06]. XCRF is developed by Hanh-Missi Tran as part of the MOSTRARE INRIA project[1]. XCRF is open source and released under the Gnu Public Licence. Last version of XCRF can be downloaded from http://treecrf.gforge.inria.fr.

2 Getting Started

2.1 Requirements

XCRF is developed in Java 1.6 on Amd64 architecture on Linux machines. Java 6.0 last release can be downloaded from http://java.sun.com/javase/downloads/index.jsp. XCRF is known to work with Java JRE 1.5 or higher.

The XCRF system may need a lot of memory. Use the -Xms and -Xmx options of the Java interpreter to respectively set the initial heap size and the maximum heap size. In the examples provided in the XCRF release, both values are set to 1GB.

2.2 Installation

Download the last version from http://treecrf.gforge.inria.fr. XCRF is tarred and compressed using GNU tar.

```
$> tar xzf xcrf-<release>.tar.gz
```

2.3 Building

You can fetch sources from the INRIA gforge platform. Use Apache Ant to build the XCRF project.

```
$>tar xzf xcrf-src-<release>.tar.gz
$>cd xcrf-src-<release>
$>ant clean
$>ant jar
```

[1] http://mostrare.futurs.inria.fr

2.4 First Annotation

There are two examples in the release that may help you to learn how to use XCRF. You can check your installation with the provided shell scripts:

```
$> cd xcrf-<release>
$> ./exampleLearnTree.sh
$> ./exampleAnnotateTrees.sh
```

The first script trains a given XCRF on a given dataset. The script calls the main XCRF executable with 4 arguments: the way the XML files are read (annotation reader), the location of the given XCRF, the location of the corpus, the location where to store the trained XCRF. Ouputs of this command recall these parameters and log some informations:

```
Apply on:
        .Annotation reader specified by the file at xcrf/examples/tree/annotationReader.xml
        .CRF specified by the file at xcrf/examples/tree/crfL0_0.xml
        .Corpus specified by the directory at xcrf/examples/tree/L0-0
        .Result stored in the file at xcrf/examples/tree/crfResult.xml
xcrf/examples/tree/L0-0/right.png was not taken into account. See log file for further information.
xcrf/examples/tree/L0-0/plus.png was not taken into account. See log file for further information.
xcrf/examples/tree/L0-0/soccer.css was not taken into account. See log file for further information.
xcrf/examples/tree/L0-0/valid-xhtml10.png was not taken into account. See log file for further
 information.
xcrf/examples/tree/L0-0/vcss.png was not taken into account. See log file for further information.
xcrf/examples/tree/L0-0/left.png was not taken into account. See log file for further information.
Features number: 169
Step 0 : log-likelihood = -1042.9554566639504
Step 1 : log-likelihood = -1010.065095643635
Step 2 : log-likelihood = -889.5786477936114
Step 3 : log-likelihood = -714.3432154612206
Step 4 : log-likelihood = -714.3076709756092
Step 5 : log-likelihood = -714.181690449722
Step 6 : log-likelihood = -713.8406801412021
Step 7 : log-likelihood = -713.72957530278
Step 8 : log-likelihood = -713.6635863166109
Step 9 : log-likelihood = -713.6326852066597
Step 10 : log-likelihood = -713.6168350671788
Step 11 : log-likelihood = -713.6089847731439
Step 12 : log-likelihood = -713.6050431327774
Step 13 : log-likelihood = -713.6030743016719
Step 14 : log-likelihood = -713.6020888749797
Step 15 : log-likelihood = -713.6015960522686
Step 16 : log-likelihood = -713.6013495280447
Step 17 : log-likelihood = -713.6012262423114
Step 18 : log-likelihood = -713.6011646042857
Step 19 : log-likelihood = -713.6011338084849
Step 20 : log-likelihood = -713.601118442908
Step 21 : log-likelihood = -713.601110798083
Step 22 : log-likelihood = -713.6011070168237
Step 23 : log-likelihood = -713.6011051697308
Learning time:0.45705 min
```

Messages indicate that some parts of the data (images files in png format, and stylesheets which are not XML files) are not considered. Training is based on a gradient based method which maximizes likelihood of the data set. Intermediate results are also reported on the standard output.

The second scripts (`./exampleAnnotateTrees.sh`) annotates a label free data set with predicted labels. The script take 4 arguments: the way the predicted labels are written into the `XML` files (annotation writer), the location of the `XCRF` (in this case, the `XCRF` obtained previously), the location `XML` file to annotate. The result is stored in the location given by the fourth argument.

```
Apply on:
        .Annotation writer specified by the file at xcrf/examples/tree/annotationWriter.xml
        .CRF specified by the file at xcrf/examples/tree/crfResult.xml
        .File to annotate specified by the directory at xcrf/examples/tree/testDelta1.xml
        .Result stored in the file at xcrf/examples/tree/annotated.xml
```

3 `XCRF` definition

Basically, an `XCRF` annotates with labels element nodes (internal nodes), text nodes (leaves) such as `PCDATA` or attribute nodes of `XML` data. An `XCRF` essentially annotates trees but can also be used on sequences. This section only describes `XCRF` for trees. In the following, an `XML` file without any annotation will be called an *observation*.

The way `XML` files are annotated is very flexible. It can be parameterized by annotation processors. You can write your own annotation processor or use the default one. We assume in this section that the default annotation processor is used. Section 5 covers the details about annotation processors.

An `XCRF` is represented by an `XML` file whose schema is available in `xml/crf.xsd`. An `XCRF` as a root element `Crf` and four kinds of first level nodes.

`Labels` **(element-required)** a set of annotation labels ;

`SetsOfLabels` **(element-optional)** subsets of labels, which can be useful for feature functions and constraints definition;

`Constraints` **(element-optional)** a list of constraints;

`Features` **(element-required)** a set of feature functions.

The list of constraints and the list of subsets of labels are not required. The set of labels and the set of features cannot be non empty: at least one of each is required.

The root element `Crf` has an attribute `type` which is either

- `tree` for `XML` trees labeling, that is node labelling;
- `sequence` for sequence labeling;
- `mixt` for labeling nodes and sequences in text nodes of `XML` trees.

A skeleton of an `XCRF` definition is:

```
<Crf xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="crf.xsd"
     type="tree">

  <Labels>
```

```
    ...
  </Labels>

  <!-- Optional element -->
  <SetsOfLabels>
    ...
  </SetsOfLabels>

  <!-- Optional element -->
  <Constraints>
    ...
  </Constraints>

  <Features>
    ...
  </Features>
</Crf>
```

3.1 Labels

An annotation label is described by an identifier and by its value. Identifiers are used in the definition of subsets of labels and the definition of features. The identifier must be of the form `label_someword` where `someword` is written with alphanumeric symbols and `_` `\` or `-`[2]. Note that there cannot be two annotation labels with the same value.

It may happen that you don't want to annotate each node of a tree (for example, you want to annotate the text leaves only). This absence of label is specified using the attribute `emptyAnnotation` of the element `Labels`.

Figure 1 shows how you can describe a set of labels. This set holds four annotation labels: `NOANN`, `club`, `date` and `league`. In the element `Labels`, we associate the label `NOANN` with the "empty annotation label".

Figure 1: Example of a set of labels specification

```
<Labels emptyAnnotation="label_NOANN">
  <Label value="NOANN" name="label_NOANN"/>
  <Label value="club" name="label_club"/>
  <Label value="league" name="label_league"/>
  <Label value="date" name="label_date"/>
</Labels>
```

[2]The identifier must match the regular expression `label_[a-zA-Z0-9_\-]+` and the value must match the regular expression `[a-zA-Z0-9_\-]+`.

3.2 Subsets of labels

A subset of labels is composed of a list of one or more labels. It can be referred to by an identifier[3]. Figure 2 provides an example of a subset of labels definition that contains only one element. Its identifier is `set_SomeSet` and it contains two labels: `club` and `league`. Identifiers of labels are written in the attribute `labels` of the element `SetOfLabels`.

Figure 2: Example of a set of label subsets

```
<SetsOfLabels>
  <SetOfLabels labels="label_club label_league" name="set_SomeSet"/>
</SetsOfLabels>
```

3.3 Features

A feature is a $\mathbb{R}$-valued function. All features are evaluated on all nodes and their results are weighted. (Therefore, we usually use 0-1-valued features.)

3.3.1 Features for `XML` Data

Basically, given a current node, features evaluate some tests on the observation and on the annotation to compute their outcomes. Tests on the observation are unrestricted. Tests on the annotation are limited to a neighborhood of the current node. In the `XCRF` implementation, we consider four types of features:

1. features that can make test on the annotation of the current node;
2. features that can make test on the annotation of the current node and its parent;
3. features that can make test on the annotation of the current node and its next-sibling;
4. features that can make test on the annotation of the current node, its parent and its next-sibling.

For a given set of one, two, or three nodes, a feature function evaluates to 1 (or more generally to some custom value) if the observation tests are true and the node labels meet the feature definition and evaluates to 0 otherwise.

The set of features is defined by an element named `Features`, and each feature is defined by an element `Feature` with:

`name` **(attribute-required)** The identifier of the feature.

`weight` **(attribute-optional)** The weight of the feature. Its default value is 0.

`xsi:type` **(attribute-required)** The type of feature, a value in the set {`CurrentNode`, `Current-Sibling`, `Current-Parent`,`Current-Parent-Sibling`}.

[3]The identifier must match the regular expression `set_[a-zA-Z0-9_\-]+`.

`FeatureValue` **(element-optional)** By default a feature is a 1-0-valued function, but the feature may be a x-0 function where x is the result of the computation of the expression contained in this element.

`Ycur, Ypar, Ysib` Elements that store the label of the current node, its parent, its next-sibling.

- Element `Ycur` is required,
- Element `Ypar` is required only for features of type: `Current-Parent`, `Current-Parent-Sibling`;
- Element `Ysib` is required only for features of type: `Current-Parent-Sibling`, `Current-Parent-Sibling`;

Elements `Ycur`, `Ypar`, `Ysib` have a required attribute `xsi:type` whose value can be `Label` or `SetOfLabels`.

`testX` **(element-optional)** a set (which can be empty) of tests on observations. An observation test is specified as an `XPATH` expression which is evaluated in the context of the current node.

3.3.2 Examples

Figure 3 describes a feature that checks if the annotation of the current node is the label `club`. Note the type of the feature and the type of the annotation test.

The feature will be applied to any node and will be 1 valued for a given node if it is labeled by `club` and 0 otherwise.

```
<Feature xsi:type="CurrentNode" name="Simple feature">
  <Ycur xsi:type="Label" value="label_club"/>
</Feature>
```

Figure 3: A simple feature without any test on the observation.

Figure 3 describes a feature that checks if the annotation of the current node is the label `club` and its sibling has an annotation in the set `set_SomeSet`.

```
<Feature xsi:type="Current-Sibling" name="Simple feature2">
  <Ycur xsi:type="Label" value="label_club"/>
  <Ysib xsi:type="SetOfLabels" value="set_SomeSet"/>
</Feature>
```

Figure 4: A feature, without any test on the observation, that uses a set of labels.

Figure 5 adds two observation tests to the feature of the figure 3. Tests on the observation are `XPATH` expressions. The first one says that the current node has a parent `td` and a grand-parent `tr`. The second one says that the parent of the current node is the first `td` node.

Figure 6 depicts a complete example of feature. Its name is `f_client` and its weight is equal to `7.46`. The annotation tests describe that:

```
<Feature xsi:type="CurrentNode" name="Simple feature">
  <Ycur xsi:type="Label" value="label_club"/>
  <TestX value="./parent::*[name()='td']/parent::*[name()='tr']"/>
  <TestX value="count(./parent::*[name()='td']/preceding-sibling::*[name()='td'])=0"/>
</Feature>
```

Figure 5: A feature with tests on the observation.

- the annotation label of the current node is the same as the label identified by `label_club`;
- the annotation label of the parent node is equal to the label identified by `label_name`;
- the annotation label of the sibling node is equal to the label identified by `label_city`.

```
<Feature name="f_client" weight="7.46" xsi:type="Current-Parent-Sibling">
  <Ycur xsi:type="Label" value="label_club"/>
  <Ypar xsi:type="Label" value="label_name"/>
  <Ysib xsi:type="Label" value="label_city"/>
  <TestX value="name() = 'tr'"/>
  <TestX value="ancestor::*/@id"/>
  <TestX value="name(following-sibling::*[1]) = 'td'"/>
</Feature>
```

Figure 6: A complete example of feature on trees.

3.3.3 Features for Sequences

Given a current position in the sequence, features evaluate some tests on the observation and on the annotation to compute their outcomes. There exists two kinds of features for sequences:

1. Features that consider only the annotation on the current token (`CurrentToken`).
2. Features that consider the annotation on the current token and the annotation on the preceding one `Token-PrevToken`.

In both cases, the test on the observable is specified by exactly two tests (`TestX`). The first one is a regular expression applied to the sequence up to the current token. The second one is a regular expression applied to the sequence from the current token to the end of the sequence. Admitted regular expressions are described in the java api documentation.

Element `Ycur` (resp. `Yprev`) store the label of the current node (resp. its predecessor).

3.3.4 Example

In the following example, the feature `f_club` is evaluated to 1 if and only if the current token is labeled by the value of the label `label_club` and the observed string before the current token matches `Club: {1,3}$` and the observed string after the current token matches `results`.

So the observed string before the current token must end with `Club:` followed by 1, 2 or 3 spaces and the observed string after the token must contain the substring `results`.

```
<Feature name="f_club" xsi:type="CurrentToken">
  <Ycur xsi:type="Label" value="label_club"/>
  <TestX value="Club: {1,3}$"/>
  <TestX value="results"/>
</Feature>
```

3.4 Constraints

Constraints work currently with `CRF` for trees only.

Constraints are used to restrict the possible label assignments. Types of constraints are similar to types of features. Only local tests can be done on the labeling through constraints. Accepted constraints are `CurrentNode-Constraint`, `Current-Parent-Constraint`, `Current-Sibling-Constraint` and `Current-Parent-Sibling-Constraint`. All contraints are evaluated at each node, denoted by the current node. The `Current-Sibling-Constraint` is not implemented. You can use the `Current-Parent-Sibling-Constraint` constraint to get the equivalent effects.

For example, we want to forbid all annotations where a node and its parents are both labeled by `club`. In the `XML` specification shown in the figure 7, we describe that the constraint is associated with every couple of nodes formed by a node and its parent (hence the use of the `Current-Parent-Constraint` type). Then we specify the couple of annotations we forbid for all these node couples. The element `Ycur` refers to the current label of the forbidden annotation pair and the element `Ypar` refers to the parent label of the forbidden annotation pair.

```
<Constraint xsi:type="Current-Parent-Constraint">
  <Ycur xsi:type="AnnotationLabel" value="label_club"/>
  <Ypar xsi:type="AnnotationLabel" value="label_club"/>
</Constraint>
```

Figure 7: A constraint example: any couple of the form a node and its parent cannot be labeled by (`club`, `club`).

The use of sets of labels provides shortcuts to define more complex constraints. For instance, if you want to forbid the annotation pair (a,b) where a is either the label `club` or the label `league` and where b is the label `club`, there are two solutions. The first one consists in writing a constraint that forbids (`club`,`club`) and another constraint that forbids (`league`,`club`). The second solution allows the user to write only one constraint in this case. It relies on the use of labels subset. Instead of referring to a single label, the element of an annotation pair may refer to a subset of labels as shown in the figure 8.

```
<Constraint xsi:type="Current-Parent-Constraint">
  <Ycur xsi:type="AnnotationSeqLabel" value="set_SomeSet"/>
  <Ypar xsi:type="AnnotationLabel" value="label_club"/>
</Constraint>
```

Figure 8: A more complex constraint example. Here `setSomeSet` is the set of labels containing the values `club` and `league`.

It is possible to add predicates on the observation to build constraints that behave differently depending on the position of the current node and the observed data. In an `XCRF`, the predicate is an `XPATH` expression whose context node is the current node.

For example, if we want to forbid the annotation pair shown in the figure 7 only for node couples where the parent node is an element `p` and the child node is an element `strong`, we add a predicate under the element `ConstraintPredicate` with the `XPATH` test `.[name() = 'strong']/parent::p`.

```
<Constraint xsi:type="Current-Parent-Constraint">
  <ConstraintPredicate value=".[name() = 'strong']/parent::p"/>
  <Ycur xsi:type="AnnotationLabel" value="label_club"/>
  <Ypar xsi:type="AnnotationLabel" value="label_club"/>
</Constraint>
```

Figure 9: Local constraint example

4 XCRF operations

Shell scripts are provided to help you start with `XCRF` operations.

4.1 Training

Training an `XCRF` is launched by calling the `xcrf.LearnTree` class. Examine the script `learnTree.sh` as a first example.

Basically, training requires to give an `XCRF` and a corpus as an argument and produces a new `XCRF`. The command synopsis is:

```
$>java -jar xcrf.jar train [options] crf_in corpus crf_out
```

- `train` is the command
- `crf_in` and `crf_out` are `XML` files that describe `XCRF`. See Section 3 for the format of this file.
- `corpus` is the path to the training corpus. The `XCRF` system takes only into account well-formed and well-annotated documents.

4.1.1 Options

`-v` : print log-likelihood during the training phase.

`-c file.xml` : specify an annotation reader in `file.xml`

`-d attributeNameForInternalNodes attributeNameForTextLeaves attributeNameForAttributes`: use the default annotation reader by specifying the attribute names used to hold the annotation label respectively for internal nodes, for text leaves and for attributes.

The last two options are mutually exclusive and described in section 5.

4.1.2 Example

The operation explained here uses the example in `examples/tree/L0-0`. The goal of this operation is to learn to annotate pages about soccer (provided by the corpus of the example). The set of labels that are used to annotate pages is described in the `XCRF` file (`crfL0_0.xml`). In this file, we have written only features specific to the example but you can of course specify as many features as you want.

The script `exampleLearnTree.sh` of the XCRF release shows how to perform a corpus training:

```
./learnTree.sh -v -c examples/tree/L0-0/annotationReader.xml\
    examples/tree/L0-0/crfL0_0.xml examples/tree/L0-0/trainingCorpus\
    examples/tree/L0-0/crfResult.xml
```

In this example, the variation of the log-likelihood is printed (option `-v`). We use a custom-built annotation reader using (option `-c`) located at `examples/tree/L0-0/annotationReader.xml`. At last three paths are given:

- the path to the CRF (`examples/tree/L0-0/crfL0_0.xml`);
- the path to the training corpus (`examples/tree/L0-0/trainingCorpus`);
- the path to the training result (`examples/tree/L0-0/crfResult.xml`).

4.2 Corpus Annotation

The corpus annotation is performed by calling the `xcrf.AnnotateTrees` class. Examine the script `annotateTrees.sh` as a first example.

Basically, corpus annotation requires to give an `XCRF` and a corpus as an argument and produces a new corpus. The command synopsis is:

```
$>java -jar xcrf.jar annotation [options] crf corpus_in corpus_out
```

- `annotate` is the command
- `crf` is the `XML` file that describe `XCRF`. See Section 3 for the format of this file.
- `corpus_in` and `corpus_out` are the paths to corpus.

4.2.1 Options

`-cMargProb file.xml` : specify a marginal probabilities setter in `file.xml`

`-dMargProb attributeNameForInternalNodes attributeNameForTextLeaves attributeNameForAttributes`: use the default marginal probabilities setter by specifying the attribute names used to hold the marginal probabilities respectively for internal nodes, for text leaves and for attributes.

`-cAnnWriter file.xml` : specify an annotation writer in `file.xml`

`-dAnnWriter attributeNameForInternalNodes attributeNameForTextLeaves attributeNameForAttributes`: use the default annotation writer by specifying the attribute names used to hold the annotation label respectively for internal nodes, for text leaves and for attributes.

The first two options are mutually exclusive. The last two options are mutually exclusive.

4.2.2 Example

At the end of the training step, the weights of the features were computed according to the provided annotated corpus. We want now to use the updated `XCRF` to annotate a raw corpus.

The script `exampleAnnotateTrees.sh` of the `XCRF` release shows how to perform a corpus training:

```
./annotateTrees.sh -cAnnWriter examples/tree/L0-0/annotationWriter.xml\
                    examples/tree/L0-0/crfResult.xml\
                    examples/tree/L0-0/rawCorpus\
                    examples/tree/L0-0/annotatedCorpus
```

We use a custom-built annotation writer using `-cAnnWriter` option. The writer is located at `examples/tree/L0-0/annotationWriter.xml`. Then three paths are given in this example:

- `examples/tree/L0-0/crfResult.xml`: the path to the `XCRF`;
- `examples/tree/L0-0/rawCorpus`: the path to the corpus to be annotated;
- `examples/tree/L0-0/annotatedCorpus`: the path to the directory where to save the annotated documents.

5 Annotation Processor Format

An annotation processor is a tool that reads or writes annotation labels. Readers are called in the training phase and writers are called in the labeling phase. Readers and Writers can be customized to fulfill your requirements. A default annotation processor is given and can be slightly configured on the command line.

5.1 Default Annotation Processor

The default annotation processor writes and reads labels at the node level into new attribute nodes. Names of these new attributes depend on the type of node that is labeled:

- `slot` is the attribute name for the label of an **element** node;
- `slot_attr` is the attribute name for the label of an **attribute** node named `attr`;
- `slotL` is the attribute name for the sequence of labels of **text** nodes. Such attributes are located in the parent of text nodes. For instance, when an element node has two text nodes, it has an attribute `slotL="label1 label2"` which means that the first text node is labeled with `label1` and the second text node is labeled with `label2`.

Let us consider the following example:

```
<div slot="lab1">
  <p slotL="lab3 lab4" slot="lab5">a piece of
  <a href="other.html" slot_href="lab2">text</a>
  in a node.
  </p>
</div>
```

In this example,

- element node `div` is labeled by `lab1`
- element node `p` is labeled by `lab5`
- attribute node `href` of node `a` is labeled by `lab2`
- text nodes '`a piece of`' and '`in a node`' of node `p` are respectively labeled by `lab3` and `lab4`.

The names of the attributes used to hold the annotation labels can be chosen on the command line when the training or the labeling is launched (the `-d` option see 4.1.1).

For elements, attribute nodes and text nodes, the empty annotation label (see section 3.1 for more details) is assigned if no label is specified.

5.2 Advanced Annotation Processors

This section is dedicated to the design of specific annotation writers and readers. Let us start with a simple example.

5.2.1 Simple Example

Suppose we are faced an annotation tasks that concerns only elements nodes of some `XML` files. Therefore, we want to get rid off any prediction about attributes and text nodes. This will speed up the learning and inference tasks because the number of labels to predict will be reduced. To do that, we define an annotation processor that reads annotation labels only for internal nodes. The annotation processor can be designed by a simple declaration of some java class together with its parameters. This declaration follows an `XML` syntax:

```
<AnnotationProcessor
  class="mostrare.tree.annotation.AttributeForInternalNodeAnnotationReader">
  <Argument>slotE</Argument>
</AnnotationProcessor>
```

We use the annotation reader that fits our needs by referring to the Java class:
`mostrare.tree.annotation.AttributeForInternalNodeAnnotationReader` .
Details about this reader can be obtained from the XCRF Java doc. The unique constructor is:

```
AttributeForInternalNodeAnnotationReader(String attributeName, CRF crf)
```

Each constructor of an annotation processor should have a `XCRF` as its last argument. That is why it is omitted in the `XML` specification. The only parameter to provide is the argument `attributeName`. In the example, we have set its value to `slotE` in the element `Argument`.

5.2.2 More Complex Example

How to specify an annotation processor that reads the annotation labels of internal nodes and text leaves ? The first solution lies in creating an implementation that perform both readings. A smarter solution consists in gathering an annotation reader for internal nodes and an annotation reader for text leaves as described in the figure 10.

The `XCRF` API provides an annotation reader for internal nodes and an annotation reader for text leaves by means of two classes:

- `mostrare.tree.annotation.AttributesForTextLeavesAnnotationReader`
- `mostrare.tree.annotation.AttributesForInternalNodeAnnotationReader`

We put them together by using the annotation reader implemented by the class:
`mostrare.tree.annotation.AnnotationReaderLooseWrapper`
This class acts as a list node. It wraps an annotation reader and points to another annotation reader. When it is called to read annotation labels, it will first use the annotation reader it wraps and then the other annotation reader. In our example, the annotation reader first reads annotation labels for internal nodes and then reads annotation labels for text leaves.

5.3 `XML` Description

The `XML` specification of an annotation processor is based on the Java implementation. You have to refer to the constructors of the Java class that implement the annotation processor you want to use. As illustrated in previous examples, arguments of an annotation processor constructor are either annotation processors or strings.

So in `XML` an annotation processor is specified by an element named `AnnotationProcessor` and its Java implementation is put in the attribute named `class`. The children of the annotation processor element are either annotation processor elements or elements named `Argument` that contain text values.

```
<AnnotationProcessor
  class="mostrare.tree.annotation.AnnotationReaderLooseWrapper">
  <AnnotationProcessor
    class="mostrare.tree.annotation.AttributeForInternalNodeAnnotationReader">
    <Argument>slotE</Argument>
  </AnnotationProcessor>
  <AnnotationProcessor
    class="mostrare.tree.annotation.AttributeForTextLeavesAnnotationReader">
    <Argument>slotL</Argument>
  </AnnotationProcessor>
</AnnotationProcessor>
```

Figure 10: Annotation reader for internal nodes and text leaves

5.4 Changing Default Annotation Processor

You can find the default annotation reader at `xml/annotationReaderDefault.xml` in the release. This annotation reader gets the annotations of the attributes, the text leaves and the internal nodes of a tree. It also names the attributes (`slot`, `slot_attr`, `slotL`) to handle labels.

When you invoke the `XCRF` jar with the `-c` option (see 4.1.1), this `XML` file is no more considered. Corresponding Readers and Writers are directly synthetized and parametrized with the corresponding classes. This is also the case with the `-d` option.

6 Configuration

An `XML` file `config.xml` in the main directory handles the configuration of the `XCRF` program. Currently ten properties can be specified:

- `crfXSD`: the path to the `XCRF` XML-Schema;
- `annotationProcessorXSD`: the path to the annotation processor XML-Schema;
- `search`: the implementation of the algorithm that maximizes the log-likelihood;
- `iterMax`: the maximum number of iterations for the previously selected algorithm;
- `XPath`: the `XPATH` implementation;
- `penalty`: the penalty applied to the log-likelihood;
- `attributeAsNode`: the addition of the attributes in the tree model or not;
- `inputEncoding`: the encoding of the input documents;
- `globalConstraintIterator`: the kind of iterator over annotations used for an `XCRF` with constraints.
- `prefixPreprocAttr`: the start of the name of the attributes that shouldn't be taken into account in the `XCRF` training and in the annotation.

6.1 Algorithms to Maximize the log-likelihood

The algorithm used in XCRF API is a Quasi-Newton algorithm (LBFGS[LN89]). Two implementations are available:

- a translation from Fortran to Java[4];
- a modified version of the Mallet implementation[5].

The value of `search` property is either `Riso` or `Mallet`.

6.2 XPATH Version

You can use two XPATH implementations:

- Jaxen [6]: it is the XPATH implementation on which the Dom4j API [7] lies. It provides XPATH1.0;
- Saxon [8]: this API allows to use XPATH2.0.

The value of the XPATH property is either `Jaxen` or `Saxon`.

6.3 Penalty Applied to the log-likelihood

In order to avoid overfitting when learning a new XCRF, the criterion for parameter estimation is the penalized log-likelihood. For more information, see [Gup].

7 Example of an XCRF Based Application

An RSS feed generator (available at http://r2s2.futurs.inria.fr) has been built on top of XCRF. The main idea consists in using XCRF to annotate web pages with labels relative to an RSS feed (title of an article, its description, ...) and then in applying heuristics to transform annotated documents into an RSS feed.

8 Known Bugs

8.1 Namespace problems

1. XPATH expressions used in the definition of the features do not work with documents which have a default namespace.
2. When a document has two attributes of the same name in different namespaces (e.g. `ns1:attr` and `ns2:attr`), the labeling does not work correctly. Indeed, it creates two new attributes, both called `slot_attr`. One of them is thus overwritten.

[4] http://riso.sourceforge.net
[5] http://mallet.cs.umass.edu
[6] http://jaxen.org
[7] http://dom4j.org
[8] http://www.saxonica.com/

References

[Gup] Rahul Gupta. Conditional random fields. Report.

[JGTT06] Florent Jousse, Rémi Gilleron, Isabelle Tellier, and Marc Tommasi. Conditional random fields for xml trees. In *ECML Workshop on Mining and Learning in Graphs*, 2006.

[LMP01] John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. 18th International Conf. on Machine Learning*, pages 282–289. Morgan Kaufmann, San Francisco, CA, 2001.

[LN89] Dong C. Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Math. Programming*, 45(3, (Ser. B)):503–528, 1989.