An Inductive Learning System for XML Documents

Xiaobing Jemma Wu

CSIRO ICT Centre, Australia

Abstract. This paper presents a complete inductive learning system that aims to produce comprehensible theories for XML document classifications. The knowledge representation method is based on a higher-order logic formalism which is particularly suitable for structured-data learning systems. A systematic way of generating predicates is also given. The learning algorithm of the system is a modified standard decision-tree learning algorithm driven by predicate/recall breakeven point. Experimental results on XML version of Reuters dataset show that this system is able to produce comprehensible theories with high precision/recall breaken point values.

Key words: knowledge representation, XML documents, precision/recall, decision-tree learning.

1 Introduction

XML (eXtensible Markup Language) documents are one of the most important source of semistructured data. Semistructured data is data that has some structure, but the structure may not be rigid, or complete and generally the data does not conform to a fixed schema. Higher-order logic is particularly suitable for structured-data learning systems, as it is able to represent individuals with complex structures, precisely describe the hypothesis languages, and test the hypotheses on individuals.

Decision-tree algorithms are well-studied learning algorithms. In contrast to many other learning algorithms, such as neural networks and support vector machines, decision trees could provide comprehensible theories to the users. A comprehensible theory could provide insight about the observations. Most existing decision-tree algorithms are based on accuracy heuristic. However, sometimes the accuracy is not a good criterion for classification. Precision and recall are two trade-off criteria which are traditional standards for text document classification problems. Most XML documents have plenty text contents, so precision and recall are more appropriate criteria than accuracy for XML document classifications.

This paper presents a novel inductive learning system for XML documents. Section 2 gives the representation method for XML documents using the higher-order logic formalism. Section 3 presents the decision-tree learning algorithm

driven by precision and recall. Section 4 shows the experimental results on XML version of Reuters dataset.

2 Knowledge representation for XML documents

We adopt a typed higher-order logic as the knowledge representation formalism [9] for XML documents, as it is particularly suitable for representing individuals with complex structures, precisely describe the hypothesis languages, and test the hypotheses on individuals. The higher-order logic is used in two essential ways here. First, individuals are represented as higher-order logic terms; second, predicates are constructed by composing transformations.

2.1 The structure of an XML document

This section is an overview of the general structure of XML documents. Further description appears in later sections as we discuss the representation over each part of it.

An XML document has a well-nested structure. Each entity is enclosed by a start tag and an end tag. This structure is best described by an example. A simple but complete XML document is given in Figure 1. The first line of the document is the XML version declaration. The second line is the declaration of its DTD (Document Type Definition) [1]. Next comes the root element *bib* which has two subelements *book* describing information about two books. The first *book* has four further subelements *title*, *author*, *publisher* and *price*, which give the basic information about this book. The first *book* also has an attribute *year* with a value of *2003*. The *author* further contains two subelements *last* and *first* which contains the last name and the first name of the author, respectively. The second *book* element is similar to the first one, but contains three element *author*.

2.2 Representation of individuals

A well-formed XML document is represented as a six tuple.

$$type\ XML = XMLDecl \times Misclist \times DTD \times Misclist \times Element \times Misclist$$

Here, *XMLDecl* represents the XML declaration; *Misclist* represents a list of miscellaneous items such as comments, processing instructions, and spaces; *DTD* represents the document type declaration; and *Element* represents the root element of the document. All these six components are non-atomic type values and could be further defined. As an example, we give the representation of *Element* below.

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!DOCTYPE bib SYSTEM‘‘books.dtd’’>

<!-- Two books are described here --!>
<bib>
  <book year=‘‘2003’’>
    <title>Logic for Learning</title>
    <author>
       <last>Lloyd</last>
       <first>John</first></author>
    <publisher>Springer</publisher>
    <price>49.95</price>
  </book>
  <book year=‘‘2000’’>
    <title>Data on the Web</title>
    <author>
      <last>Abiteboul</last>
      <first>Serge</first></author>
    <author>
      <last>Buneman</last>
      <first>Peter</first></author>
    <author>
      <last>Suciu</last>
      <first>Dan</first></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>39.95</price>
  </book>
</bib>
```

Fig. 1. An example XML document with an external DTD

The formal representation for an element is

$$data\ Element = Elem\ TagName\ Attributelist\ Contents$$
$$type\ TagName = String$$
$$type\ Attributelist = [Attribute]$$
$$type\ Contents = [Content]$$

Here, type *Elem* is defined as a data constructor which has three arguments representing the element name, the attributes and the element contents, respectively. *TagName* is a synonym of type *String*. *Attributelist* and *Contents* is a list of attributes and a list of content, respectively.

An attribute is composed of the attribute name and attribute value, and is represented using a tuple as follows.

$$type\ Attribute = AttName \times AttValue$$

where *AttName* and *AttValue* are both synonym of *String*, written as follows.

$$type\ AttName = String$$
$$type\ AttValue = String$$

It is the element content that makes the element and the XML document nested and hierarchical. The content of an element could be another element, a piece of text, a reference to some entities, a CDATA section, a processing instruction or a comment. The content is formally represented as follows.

$$data\ Content = El\ Element \mid Tx\ CharData \mid Ref\ Reference \mid CD\ CDSect \mid ContentPI\ PI \mid ContentCom\ Comment$$

Here, *El*, *Tx*, and so on, are constructors of type *Content*. Constructor *El* needs an argument of type *Element* to construct a data of type *Content*, and *Tx* needs a type *CharData* to construct a type *Content*, and so on. Type *Element* has been introduced above. *CharData* is a synonym of a list of characters. Type *PI* and *Comment* have already been defined in the previous section.

$$type\ CharData = [Char]$$

CDATA sections are used to tell the parser not to parse the content inside but display it directly. They are very handy when trying to display part of XML code.

A CDATA section is formally represented as follows.

$$data\ CDSect = CDSe\ CData$$
$$type\ CData = [Char]$$

Entity references are used to reference some pre-defined entities in XML documents to represent some reserved characters, Unicode characters, and entities that are defined in DTDs. An entity reference refers to the content of an named entity. Entities may be either parsed or unparsed. Entity references begin with the ampersand '&' and ends with a semicolon ';' and between them are the entity names or the character codes.

Type *Reference* is defined as follows.

$$data\ Reference = EnRef\ \ EntityRef \mid ChRef\ \ CharRef$$
$$type\ EntityRef = String$$
$$type\ CharRef = String$$

A complete knowledge representation for XML documents using this method can be found in [14].

2.3 Representation of features

Here, features refer to predicates on the type of individuals. These predicates are constructed incrementally by composition of *transformations* using *predicate rewrite systems* [9]. A transformation f is a function having a signature of the form $f : (\varrho_1 \to \Omega) \to \cdots \to (\varrho_k \to \Omega) \to \mu \to \sigma$, where $\varrho_1, \ldots, \varrho_k$, σ and μ are all types and $k \geq 0$.

Transformations on XML are classified into two categories: *generic transformations* and *data-specific transformations*. *Generic transformations* come straight from the XML document representation and are applicable to all well-formed XML documents. Some examples of generic transformations are given next.

$$projRootElement : XML \to Element$$
$$projTagName : Element \to TagName$$
$$projAttributes : Element \to Attributes$$
$$projContents : Element \to Contents$$

Here, *projRootElement* projects onto the root element of an XML document. *projTagName*, *projAttributes* and *projContents* project an element onto its tag name, attributes and contents, respectively.

Another example of generic transformations on element attributes are given next.

$$listToSet : Attributes \to \{Attribute\}$$
$$setExists_1 : (Attribute \to \Omega) \to \{Attribute\} \to \Omega$$

The above two transformations convert a list of attributes to a set of attributes and check whether the set has an attribute satisfying a predicate, respectively.

Data-specific transformations capture some specialized concepts that will be useful in a particular application. For example, a transformation (= "*book*") :

$TagName \rightarrow \Omega$ is a data-specific transformation which checks whether a $TagName$ is the string "book".

A predicate rewrite system is a set of predicate rewrites. A predicate rewrite is defined as $p \rightarrowtail q$ where p and q are both types and p is more general than q. Given a predicate rewrite system $\rightarrowtail$, to generate a predicate search space for an application, one starts with an initial predicate p_0, usually the weakest one top, and generate all the predicates via a predicate deviation step. Now a simple predicate rewrite system is given as an example to illustrate how to generate a predicate search space via the predicate rewrite system.

$$
\begin{aligned}
&top \rightarrowtail projRootElement \circ top\\
&top \rightarrowtail \wedge_2\ (projTagName \circ top\ projContents \circ top)\\
&top \rightarrowtail listToSet \circ top\\
&top \rightarrowtail setExists_1\ (\wedge_2\ (isElement\ projContentElement \circ top))\\
&top \rightarrowtail (= \text{“}bib\text{”})\\
&top \rightarrowtail (= \text{“}book\text{”})\\
&top \rightarrowtail (= \text{“}author\text{”})\\
&top \rightarrowtail (= \text{“}price\text{”})
\end{aligned}
$$

The following is a path in the search space.

$$
\begin{aligned}
&top\\
&projRootElement \circ top\\
&projRootElement \circ \wedge_2\ (projTagName \circ top\ projContents \circ top)\\
&projRootElement \circ \wedge_2\ (projTagName \circ (= \text{“}bib\text{”})\ projContents \circ top)\\
&projRootElement \circ \wedge_2\ (projTagName \circ (= \text{“}bib\text{”})\ projContents \circ\\
&\qquad listToSet \circ top)\\
&projRootElement \circ \wedge_2\ (projTagName \circ (= \text{“}bib\text{”})\ projContents \circ\\
&\qquad listToSet \circ setExists_1\ (\wedge_2 (isElement\\
&\qquad\qquad projContentElement \circ top)))\\
&projRootElement \circ \wedge_2\ (projTagName \circ (= \text{“}bib\text{”})\ projContents \circ\\
&\qquad listToSet \circ setExists_1\ (\wedge_2 (isElement\\
&\qquad\qquad projContentElement \circ \wedge_2\ (projTagName \circ\\
&\qquad\qquad\qquad (= \text{“}book\text{”})\ projContents \circ top))))\\
&\qquad\qquad \ldots
\end{aligned}
$$

3 Precision/Recall-driven decision-tree (PRDT) algorithm

Having represented a well-formed XML document as a typed higher-order logic term, in this section, we will present a decision-tree learning algorithm based on the precision and recall criterion for XML document classifications.

3.1 Precision and recall

Precision and recall were originally two statistical measures widely used in information retrieval. They have been borrowed to evaluate the performance of text classification [8, 12, 15] recently.

Precision Pr and recall Rc can be defined using TP, FP and FN as follows

$$Pr = \frac{TP}{TP + FP} \qquad Rc = \frac{TP}{TP + FN} \tag{1}$$

where TP is the number of documents correctly assigned to the positive class; FP, the number of documents incorrectly assigned to the positive class; FN, the number of documents incorrectly assigned to the negative class; and TN, the number of documents correctly assigned to the negative class.

Normally there is a trade-off between high precision and high recall. A good classifier should have both high precision and high recall. When precision and recall are equal (or very close), this point is called *precision/recall-breakeven point (BEP)* of the system. BEP is commonly used as a performance measure in text classification problems. By (1), we can see if FP and FN are equal (or very close), we get at BEP.

The *F-measure* was introduced by van Rijsbergen [13]. The most commonly used form of F-measure is when $\beta = 1$. In this case, the F-measure becomes F_1 measure which balances precision and recall. F_1 measure is defined as:

$$F_1 = \frac{2Pr \cdot Rc}{Pr + Rc} \tag{2}$$

From (2), we can see when precision and recall are equal (at BEP), $F_1 = Pr = Rc$. F_1 is maximized when precision and recall are equal or close (at BEP). Hence, if we maximise F_1, we can get the BEP value which is equal (or close) to the maximized F_1. By using F_1 measure, we turn the two measurements into a single measurement which is easier for the system to control.

3.2 Structured feature selection

Unlike unstructured text documents, the same word appearing in different elements of an XML document could have different meaning or influence on document classification. Different elements could contain irrelevant text content. Therefore, it is not suitable to gather all the text in a XML document and conduct feature selection. My method is to build n independent corpus of text by

analysing the DTD, where n is the number of elements in the DTD. The text in an element of an XML document is collected and formed a new text document with the same name of the XML document and stored in a corpus corresponding to this *element*. Thus, all the documents in the same corpus come from the same *element* and are ready to do feature selection.

In text learning, a *feature* is defined for each word in the training set. Feature selection is commonly used when learning on text, as text documents often have thousands of features. Some of them are not relevant or not beneficial on the performance of the text learning. Removing these features can highly improve the learning speed and the classification accuracy. *Feature selection* aims to select a minimal subset of features which still contain enough features for classification. The commonly used approach for feature selection in text learning is to use a fixed measure method to evaluate and score all the features and then sort them by score. Several scoring methods have been proposed, including information gain [7, 16], document frequency thresholding [6, 10, 16], χ^2 statistic [11, 16] and mutual information [8, 5, 16].

Mutual information is a commonly used criterion for feature selection in text categorization, and this method is adopted in this thesis. It measures the mutual information between a word and a class. In information theory, mutual information is defined as the reduction in the entropy of a random variable due to introducing another random variable.

Like in the traditional text learning, we introduce a weighting scheme into our text features. The occurrences of some features in a document provides a better indication of the content of the document than other features. The locality of a feature decides its weight in classification, i.e., the localised feature that occurs frequently in only a few documents is more informative, but the global feature that occurs frequently in most of the documents is not informative. We adopt the *tfidf* measure which is an important and commonly used weighting scheme. It combines term frequencies (TF) with inverse document frequencies (IDF). *Term Frequency* $TF(t, d)$ is defined as the number of times a term t occurs in a document d. *Document Frequency* $DF(t)$ is the number of documents in which term t occurs at least once. The *inverse document frequency* $IDF(t)$ can be calculated from $DF(t)$ by

$$IDF(t) = log(\frac{N}{DF(t)})$$

where N is the total number of documents.

$TFIDF(t, d)$ is defined as

$$TFIDF(t, d) = TF(t, d) \times IDF(t)$$

The intuitive idea behind the tfidf measure is that a term t is more important as a feature for document d if it appears more frequently in d and appears in fewer documents.

Under this new model, a text document d is represented as

$$d =< (t_1, tfidf(t_1)), \ldots, (t_n, tfidf(t_n)) >$$

where $t_1, \ldots, t_n$ is the IDs of the features in the feature subset which appear in document d.

3.3 Node selection

Starting from a single root node containing all the training examples, the decision-tree algorithm iteratively makes a binary splits at the selected node in the existing tree. In our PRDT algorithm, we use a novel node selection method to control the system to work towards reducing the difference between the precision and recall. Next, we give some theoretical analysis for our node selection method.

First, we define the TP, FP and FN values for the node associated with $\mathcal{E}$, where $\mathcal{E}$ is a (non-empty) set of examples.

Definition 1. *Let $\mathcal{E}$ be a set of examples. We define:*

1. $TP(\mathcal{E}) = n_+^{\mathcal{E}},\ FP(\mathcal{E}) = n_-^{\mathcal{E}},\ FN(\mathcal{E}) = 0 \qquad$ *if* $n_+^{\mathcal{E}} \geq n_-^{\mathcal{E}} \geq 0$
2. $TP(\mathcal{E}) = 0,\ FP(\mathcal{E}) = 0,\ FN(\mathcal{E}) = n_+^{\mathcal{E}} \qquad$ *if* $n_-^{\mathcal{E}} > n_+^{\mathcal{E}} \geq 0$.

In the above definition, $n_+^{\mathcal{E}}$ and $n_-^{\mathcal{E}}$ denote the number of positive examples and negative examples in $\mathcal{E}$, respectively. Notice in the above definition, $FP(\mathcal{E})$ and $FN(\mathcal{E})$ is one zero and one non-zero. A decision-tree node is called a FP-type node if $FP(\mathcal{E}) \geq 0$, otherwise a FN-type node. It is not hard to see that an FP-type set of examples is a positive majority set of examples and an FN-type set of examples is a negative majority set of examples.

For a decision tree T, to balance the $FP(T)$ and $FN(T)$, our node selection method is

1. select a FP-type leaf node of T, if $FP(T) > FN(T)$, or
2. select a FN-type leaf node of T, if $FP(T) < FN(T)$.

Next, we give a proposition to support our node selection method above. Let $\mathcal{P} = \{\mathcal{E}_1, \mathcal{E}_2\}$ be a partition of a set of examples $\mathcal{E}$. Proposition 1 shows that $FP(\mathcal{P})$ will decrease and $FN(\mathcal{P})$ will increase if $\mathcal{E}$ is FP-type. Similarly, $FN(\mathcal{P})$ will decrease and $FP(\mathcal{P})$ will increase if $\mathcal{E}$ is FN-type.

Proposition 1. *Let $\mathcal{E}$ be a set of examples and $\mathcal{P} = (\mathcal{E}_1, \mathcal{E}_2)$ a partition of $\mathcal{E}$.*

1. *If $\mathcal{E}$ is FP-type, then $FP(\mathcal{E}) \geq FP(\mathcal{P})$, $FN(\mathcal{E}) \leq FN(\mathcal{P})$.*
2. *If $\mathcal{E}$ is FN-type, then $FP(\mathcal{E}) \leq FP(\mathcal{P})$, $FN(\mathcal{E}) \geq FN(\mathcal{P})$.*

Proof. 1. $\mathcal{E}$ is a FP-type, so it is a set of positive majority examples. There are three cases to consider: $\mathcal{E}_1$ and $\mathcal{E}_2$ are both FP-type, both FN-type, or one FP-type and one FN-type.

(a) Suppose $\mathcal{E}_1$ and $\mathcal{E}_2$ are both FP-type. $FP(\mathcal{P}) = n_-^{\mathcal{E}_1} + n_-^{\mathcal{E}_2} = n_-(\mathcal{E})$. Thus, $FP(\mathcal{E}) = FP(\mathcal{P})$. $FN(\mathcal{E})$, $FN(\mathcal{E}_1)$ and $FN(\mathcal{E}_2)$ are all 0. Thus, $FN(\mathcal{E}) = FN(\mathcal{P})$. Now the first case has been proved.

(b) Suppose $\mathcal{E}_1$ and $\mathcal{E}_2$ are both FN-type. Partition $\mathcal{P}$ partitions a positive majority set of examples, $\mathcal{E}$, into two negative majority sets of examples $\mathcal{E}_1$ and $\mathcal{E}_2$. This case could not happen.
(c) Suppose $\mathcal{E}_1$ and $\mathcal{E}_2$ are one FP-type and one FN-type. Without loss of generality, we suppose $\mathcal{E}_1$ is FP-type and $\mathcal{E}_2$ is FN-type. Thus, $n_+^{\mathcal{E}_1} \geq 0$, $n_-^{\mathcal{E}_2} > 0$, $FP(\mathcal{E}_1) = n_-^{\mathcal{E}_1}$, and $FP(\mathcal{E}_2) = 0$. $FP(\mathcal{P}) = n_-^{\mathcal{E}_1} < n_-^{\mathcal{E}_1} + n_-^{\mathcal{E}_2} = FP(\mathcal{E})$. $FN(\mathcal{E}_2) = n_-^{\mathcal{E}_2} > 0$. $FN(\mathcal{P}) = FN(\mathcal{E}_1) + FN(\mathcal{E}_2) = n_-^{\mathcal{E}_2}$, while $FN(\mathcal{E}) = 0$. Thus, $FN(\mathcal{E}) > FN(\mathcal{P})$.

2. Similar to the proof of 1.

Proposition 1 states that the *FP* value will decrease or not change, and the *FN* value will increase or not change by partitioning a FP-type set of examples. Similarly, the *FP* value will increase or not change, and the *FN* value will decrease or not change by partitioning a FN-type set of examples. As noted in Case (c) of the Part 1 of the proof, the *FP* (*FN*) value will strictly decrease (increase) by splitting a FP-type set of examples into two different types of sets of examples. In the same way, the *FP* (*FN*) value will strictly increase (decrease) by splitting a FN-type set of examples into two different types of sets of examples.

3.4 The precision/recall-driven decision-tree algorithm

In this section, the Precision/Recall-driven Decision-Tree (PRDT) algorithm will be described. (See Figures 2 and 3.)

Figure 2 is the PRDT algorithm. The goal of this algorithm is to find a tree that has the best precision-recall breakeven point value. Starting from a single node which is composed of the training data, the algorithm works towards two goals at the same time: looking for the point where the global precision and recall are equal and improving the F_1 measure. The first goal is achieved by selecting the node which can most balance the two values, that is, the leaf with the largest *FP* when $FP(T) \geq FN(T)$ and the leaf with the largest *FN* otherwise. The second goal is achieved by finding a predicate to split this node which can best improve $F_1(T)$. If the partition made to the selected node fails to improve $F_1(T)$, the leaf node with the next largest *FP* or *FN* will be chosen, and this procedure will continue until all the leaf nodes with positive *FP* or *FN* are tested and $F_1(T)$ fails to improve.

The training examples and a predicate rewrite system are input into the PRDT algorithm. The tree T is initialised as a single node containing all the training examples. The algorithm then enters an iteration by which T is kept split. At the start of each iteration, i.e., when a new split is going to be made, three kinds of leaves are managed in three different sets: $leaves_{FP}$, $leaves_{FN}$ and $leaves_F$, which store those unseen leaves that have positive FP, positive FN and positive FP or FN, respectively. $Leaves(T)$ returns the set of leaves of tree T. The algorithm then enters an inner iteration which selects a leaf and splits it. The leaf to be selected for splitting should mostly balance the $FP(T)$ and $FN(T)$. If $FP(T) > FN(T)$, then the leaf with the biggest *FP* is selected; if $FP(T) < FN(T)$, then the leaf with the biggest *FN* is selected; otherwise if

FP and *FN* are already balanced but non-zero, then the leaf with the biggest *FP* or *FN* is selected. However, if $F_1(T)$ could not be improved by splitting the selected leaf, then this splitting should be given up and the next leaf that satisfies the corresponding conditions should be considered. This iteration continues until $F_1(T)$ improves by splitting the selected leaf or there is no leaf to select in the corresponding leaf set.

In Figure 2, $T(l, \mathcal{P})$ denotes the new tree obtained by splitting leaf l with partition $\mathcal{P}$.

function $PRDT(\mathcal{E}, \rightarrowtail)$; **returns**: a decision tree;
inputs: $\mathcal{E}$, a set of examples;
$\rightarrowtail$, a predicate rewrite system;

T := single node (with examples $\mathcal{E}$);
finished := false;
while not *finished* **do**
  $leaves_{FP} := \{l | FP(l) > 0 \wedge l \in Leaves(T)\}$;
  $leaves_{FN} := \{l | FN(l) > 0 \wedge l \in Leaves(T)\}$;
  $leaves_F := leaves_{FP} \cup leaves_{FN}$;
  while true **do**
    if $FP(T) \geq FN(T) \wedge leaves_{FP} \neq \phi$ **then**
      $l := argmax_{l \in leaves_{FP}} FP(l)$;
      $leaves_{FP} := leaves_{FP} \setminus \{l\}$;
    else if $FP(T) < FN(T) \wedge leaves_{FN} \neq \phi$ **then**
      $l := argmax_{l \in leaves_{FN}} FN(l)$;
      $leaves_{FN} := leaves_{FN} \setminus \{l\}$;
    else if $FP(T) = FN(T) \neq 0 \wedge leaves_F \neq \phi$
      $l := argmax_{l \in leaves_F}(FP(l) + FN(l))$;
      $leaves_F := leaves_F \setminus \{l\}$;
    else
      finished := true;
      break;
    $p := Predicate(TP(T), FP(T), FN(T), \mathcal{E}_l, \rightarrowtail)$;
    $\mathcal{P}$:= partition of $\mathcal{E}_l$ induced by p;
    if $F_1(T(l, \mathcal{P})) > F_1(T)$ **then**
      $T := T(l, \mathcal{P})$;
      break;
label each leaf node of T by its majority class;
return T;

Fig. 2. Decision-tree algorithm based on the precision/recall heuristic

The function *Predicate* in the PRDT algorithm is shown in Figure 3. The predicate output by this algorithm is the one that could best improve the global

F_1 measure of the tree. The TP, FP and FN of the current tree T, the set of training examples in the selected leaf and the predicate rewrite system are input into function *Predicate*. An openlist is used to keep all the candidate predicates. Variable *predicate* is used to keep the current best predicate, and *bestScore* is used to keep the F_1 of the current best predicate. Initially, the *openList* contains only the weakest predicate *top*, and the *bestScore* is set to be the F_1 of the current tree T. The algorithm then enters an iteration. In each iteration, the first candidate predicate is drawn from the openlist and a sub-search space is generated from this predicate. Each predicate q in this sub-search space is tested by creating a new partition $\mathcal{P}$ using it. The F_1 of the tree, after adding the new partition $\mathcal{P}$, is computed using the updated TP, FP and FN. If F_1 is higher than the current *bestScore*, the best predicate and the best score are set as q and its corresponding F_1, respectively. Predicate q is also inserted into the openlist as a candidate for further sub-search space. The iteration terminates when the openlist is empty. Finally the predicate that most improves the F_1 of the tree by splitting the selected leaf is returned.

Note that function *predicate* may not terminate if the search space defined by $\rightarrowtail$ is infinite. In this case, a non-negative parameter *cutout* can be set to stop the searching if the algorithm investigates *cutout* predicates without finding a better predicate than the current best predicate. Every time a new predicate is found, the *cutout* parameter is reset to the initial value.

4 Experiments

In this section, experiments on a real-world dataset, the XML version of Reuters dataset [2], is reported. Reuters dataset is a traditional test collection for text categorization.

4.1 The dataset

Reuters-21578[2] is a collection of newswire stories in Reuters newswire in 1987. It is a commonly used test collection for text classification [6, 16, 8, 3]. The original collection consists of 22 SGML data files. Each of the first 21 files contain 1000 articles, while the last contains 578 articles.

The 21578 documents are assigned to 135 categories according to their TOPICS attributes. The "ModApte" split, which leads to a corpus of 9603 training documents and 3299 test documents, is used in all experiments here. The number of documents in each category varies widely, ranging from "Earn" which contains 3964 documents to "Castor-oil" which contains only one document. However, the ten most frequent categories account for 75% of the training instances. These ten categories are often used in the experiments of text categorization.

In the preprocessing, each article in each SGML file was transformed to an XML document. In each XML document, there are altogether about 12 elements. The categories of an XML document are set by element `TOPICS` which are determined by the text content of the document instead of other factors. The text

```
function Predicate(TP, FP, FN, E, ↣) returns a predicate;
inputs: TP, TP of the current existing tree;
        FP, FP of the current existing tree;
        FN, FN of the current existing tree;
        E, a set of examples;
        ↣, a predicate rewrite system;

openList := [top];
predicate := top;
calculate Pr and Rc of the current existing tree;
bestScore := 2Pr×Rc / (Pr+Rc);
while openList ≠ []
    p := head(openList);
    openList := tail(openList);
    for each LR redex r via r ↣ b, for some b, in p do
        q := p[r/b];
        if q is regular then
            P := partition of E induced by q;
            update TP, FP, FN;
            update Pr, Rc;
            F1 := 2Pr×Rc / (Pr+Rc);
            if F1 > bestScore then
                predicate := q;
                bestScore := F1;
            openList := Insert(q, openList);
return predicate;
```

Fig. 3. Algorithm for finding a predicate to split a node

content of the article is stored in element *TEXT* which has four subelements: *TITLE*, *BODY*, *AUTHOR* and *DATELINE*. *TITLE* stores the title of the story and *BODY* stores the main text of the story. The background knowledge tells us that the hypothesis should be with the element *TEXT*, to be more specific, should be with element *TITLE* and *BODY*. This information helps us build the predicate rewrite system.

The text contents in the XML documents are preprocessed using structured feature selection method. Text contents in an element is collected and formed a new text document and stored in a corpus corresponding to this element. Feature selection is done independently for each corpus. The procedure of the feature selection includes stop words removal, stemming, feature selection and feature weighting.

4.2 Experimental results

Part of the predicate rewrite system used in our experiments are as follows.

$top \rightarrowtail projRootElement \circ top$
$top \rightarrowtail \wedge_2\ (projTagName \circ top\ projContents \circ top)$
$top \rightarrowtail listToSet \circ top$
$top \rightarrowtail setExists_1\ (\wedge_2\ (isElement\ projContentElement \circ top))$
$top \rightarrowtail setExists_1\ (\wedge_2\ (isFeature\ projContentFeature \circ top))$
$top \rightarrowtail setExists_1\ (\wedge_2\ (proFeatureId \circ top\ projFeatureWei \circ top))$
$top \rightarrowtail (= REUTERS)$
$top \rightarrowtail (= i)$
$top \rightarrowtail (\geq x)$
$top \rightarrowtail (\leq y)$

In the above predicate rewrite system, i is an integer representing the text feature number, and x $(0 \leq x \leq 1)$ and y $(0 \leq y \leq 1)$ represents the feature weights.

One binary decision tree was built for each of the 10 most frequent classes. Table 1 gives the precision/recall-breakeven point value on the ten most frequent categories and their micro-average for five learning algorithms. The results for Findsim, NBayes, BayesNets and LinearSVM are reported in [4]. No published

	Findsim	NBayes	BayesNets	PRDT	LinearSVM
earn	92.9%	95.9%	95.8%	96.4%	98%
acq	64.7%	87.8%	88.3%	85.8%	93.6%
money-fx	46.7%	56.6%	58.8%	66.5%	74.5%
grain	67.5%	78.8%	81.4%	93.9%	94.6%
crude	70.1%	79.5%	79.6%	84.9%	88.9%
trade	65.1%	63.9%	69.0%	71.2%	75.9%
interest	63.4%	64.9%	71.3%	74.2%	77.7%
ship	49.2%	85.4%	85.4%	76.6%	85.6%
wheat	68.9%	69.7%	82.7%	92.1%	90.3%
corn	48.2%	65.3%	76.4%	90.4%	90.3%
microave	64.6%	81.5 %	85.0 %	88.0 %	92.0 %

Table 1. Precision/recall-breakeven point on the ten most frequent Reuters categories and their micro-average.

results are available for the distance between the precision and recall on the breakeven point for Findsim, NBayes, BayesNets and LinearSVM. Table 2 summarises the precision and recall on the test set for each of the ten classes on the

test set for PRDT. The two values of most classes are very close (around 5%), and the average difference of the two values is 8.1%.

	earn	acq	money	grain	crude	trade	intst	ship	wht	corn	avr
Pr	95.5	89.6	72.2	91.1	86.7	67.2	79.6	84.7	88.0	84.4	89.2
Rc	97.3	81.9	60.9	96.6	83.1	75.2	68.7	68.5	96.2	96.4	86.8

Table 2. The precision and recall value on the ten most frequent categories of Reuters and their average

From Table 1 and Table 2, we can see that the PRDT algorthm performs well on producing hypothesis with high BEP values on all 10 classes. Another important point is that the PRDT algorithm provides comprehensible hypothesis while most of other algorithms in Table 1 does not. The hypothesis produced by PRDT algorithm contains information of the structure that is comprehensible to human beings. Though the text features appear in the hypothesis is not comprehensible at the first sight, it is easy to transform the features IDs to their original terms by searching the feature-ID table by either by a script program or a human being.

Next, an example is given to show the comprehensibility of the hypothesis produced by our system. The following one-split decision tree gives a binary classifier for category "trade".

$$
\begin{aligned}
&\mathit{trade}\ m = \\
&\quad \mathit{if}\ \mathit{projRootElement} \circ \wedge_2\ (\mathit{projTagName} \circ (= \mathit{REUTERS})\ \mathit{projContents} \circ \\
&\qquad \mathit{listToSet} \circ \mathit{setExists}_1\ (\wedge_2(\mathit{isElement}\ \mathit{projContentElement} \circ \\
&\qquad\quad \wedge_2(\mathit{projTagName} \circ (= \mathit{TEXT})\ \mathit{projContents} \circ \mathit{listToSet} \circ \\
&\qquad\qquad \mathit{setExists}_1(\wedge_2(\mathit{isElement}\ \mathit{projContentElement} \circ \\
&\qquad\qquad\quad \wedge_2(\mathit{projTagName} \circ (= \mathit{TITLE})\ \mathit{projContents} \circ \\
&\qquad\qquad \mathit{listToSet} \circ \mathit{setExists}_1(\mathit{isFeature}\ \mathit{projContentFeature} \circ \mathit{setExists}_1(\\
&\qquad\qquad\quad \wedge_2(\mathit{projFeatureId} \circ (= 89)\ \mathit{projFeatureWei} \circ \mathit{top}))))))))))\ m \\
&\quad \mathit{then}\ \top \\
&\quad \mathit{else}\ \bot
\end{aligned}
$$

This theory clearly states the structure of the XML documents belonging to "trade" class as follows.

"An XML document belongs to class trade iff (i) the root element of the document is "REUTERS" and (ii) it contains an element named "TEXT" and (iii) element "TEXT" contains an element named "TITLE" and (iv) element "TITLE" contains the feature No.89 which represents word "budget"."

5 Conclusion

This paper presents a novel inductive learning system that aims to produce comprehensible hypothesis for XML document classification. The knowledge representation method is based on a higher-order logic formalism which is suitable for representing individuals with complex structures. A systematic way for generating predicates is given by using predicate rewrite systems. The learning algorithm of our system is a decision-tree learning algorithm driven by precision and recall. The algorithm works towards two goals at the same time: decreasing the difference between the precision and recall and improve the F_1 value. Experimental results show that the PRDT algorithm performs very well on XML data classification, and its ability to proving comprehensible theories make it more distinctive.

References

1. Extensible markup language (XML) 1.1.
2. *Reuters-21578.* http://www.daviddlewis.com/resources/testcollections/reuters21578/.
3. I. Dagan, Y. Karov, and D. Roth. Mistake-driven learning in text categorization. In *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing*. AAAI Press, Menlo Park, US, 1997.
4. S. Dumais, J. Platt, D. Heckerman, and M. Sahami. Inductive learning algorithms and representations for text categorization. In *Proceedings of the Seventh International Conference on Information and Knowledge Management*, pages 148–155, 1998.
5. S.T. Dumais and H. Chen. Hierarchical classification of web content. In *Proceedings of ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 256–263, Athens, 2000.
6. T. Joachims. A probabilistic analysis of the Rocchio algorithm with TFIDF for text categorization. In *Proceedings of ICML-97, 14th International Conference on Machine Learning*, 1997.
7. T. Joachims. Text categorization with support vector machines: learning with many relevant features. In *Proceedings of ECML-98, 10th European Conference on Machine Learning*, pages 137–142. Springer Verlag, 1998.
8. D. Lewis and M. Ringuette. A comparison of two learning algorithms for text categorization. In *Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, 1994.
9. J. W. Lloyd. *Logic for Learning: Learning Comprehensible Theories from Structured Data.* Springer-Verlag, 2003.
10. M.E. Ruiz and P. Srinivasan. Hierarchical neural networks for text categorization. In *Proceedings of ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 281–282, Berkeley, CA, 1999.
11. H. Schutze, D.A. Hull, and J.O. Pedersen. A comparison of classifiers and document representations for the routing problem. In *Proceedings of ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 229–237, Seatle, WA, 1995.

12. F. Sebastiani. A tutorial on automated text categorisation. In *Proceedings of ASAI-99, First Argentinian Symposium on Artificial Intelligence*, pages 7–35, Buenos Aires, AR, 1999.
13. C.J. van Rijsbergen. *Information Retrieval*. Butterworths, London, 1979.
14. X. Wu. *Knowledge Representation and Learning For Semistructured Data*. PhD thesis, The Australian National University, 2006.
15. Y. Yang. An evaluation of statistical approaches to text categorization. *ACM Transactions on Information Systems*, 12(3):296–333, 1998.
16. Y. Yang and J.O. Pedersen. A comparative study on feature selection in text categorization. In *Proceedings of ICML-97, 14th International Conference on Machine Learning*, pages 412–420, Nashville, TX, 1997. D.H. Fisher.