TCS-TR-A-05-02

TCS Technical Report

Reputation Extraction Using Both Structural and Content Information

by

H. Hasegawa, M. Kudo and A. Nakamura

Division of Computer Science

Report Series A

February 17, 2005

Hokkaido University
Graduate School of
Information Science and Technology

Email: atsu@ist.hokudai.ac.jp

Phone: +81-011-706-6806
Fax: +81-011-706-7832

Reputation Extraction Using Both Structural and Content Information

Hiroyuki Hasegawa
Graduate School of Information Science and Technology
Hokkaido University
Sapporo, 060-0814 Japan
hase@main.ist.hokudai.ac.jp

Mineichi Kudo
Graduate School of Information Science and Technology
Hokkaido University
Sapporo, 060-0814 Japan
mine@main.ist.hokudai.ac.jp

Atsuyoshi Nakamura
Graduate School of Information Science and Technology
Hokkaido University
Sapporo, 060-0814 Japan
atsu@main.ist.hokudai.ac.jp

Abstract

We propose a new method of extracting texts related to a given keyword from Web pages collected by a search engine. By combining structural pattern matching and text classification, texts related to a given keyword such as reputations of a given restaurant can be extracted automatically from Web pages in unfixed sites, which is impossible by conventional wrappers. According to our cross validation results on extracting reputations of a given *Ramen* shop from Web pages collected by a search engine, our method achieved 79.3% precision and 56.6% recall by allowing acceptable errors.

1. INTRODUCTION

What do you use to get information when you want to find a nice restaurant that serves a certain kind of dishes? Today, many people will answer that they use WWW (World Wide Web). Actually, you may find a restaurant easily through WWW, but it is difficult to judge whether the found restaurant really suits your taste. Maybe, most people try to judge based on its reputations, which are also found in WWW. Though there are restaurant-evaluation sites in which you can find a lot of reputations on restaurants, you cannot obtain enough reputations from one such site for minor restaurants. Then, you have to collect their reputations from several sites until enough information is obtained. But this is not an easy task because not all pages that contain a restaurant name contain its reputation, and even such pages that contain a reputation of a target restaurant may be very large pages with a lot of irrelevant information including information of other restaurants.

.

The problem we address in this paper is extraction of texts that have a certain relation with a given keyword from all Web pages that contain the keyword. Given a restaurant name, extracting texts that are its reputations from all Web pages retrieved by a search engine is one example of the problem. This problem would be easy if the Web pages from which reputations are extracted were those in a number of fixed sites that use fixed formats. In such case, simple wrappers such as LR wrapper [8] and TreeWrapper [9] can extract such pairs of a keyword and its related text by pattern matching around them. But by fixing sites from which reputations are extracted, sometimes enough information cannot be obtained though there is a lot of information in other sites. Besides, new information that is obtainable from new sites can never be extracted by doing so. However, extraction from unfixed sites is difficult, for example, LR wrapper and TreeWrapper cannot be constructed because there are no common patterns around the parts to be extracted.

Extraction from unfixed sites can be done to some extent by constructing a wrapper that uses a set of *consistent* patterns (rules), namely, each of which correctly extract information from some pages but does *not* wrongly extract information from any pages [4]. But there may be no such set of patterns. For example, consider a case that a restaurant name and its reputation are written in the first and the third columns in one table while a restaurant name and how to access to it are written in those columns in another table. If all the patterns around the columns are the same in the two tables, it is impossible to create a consistent pattern unless conditions on the contents of the third column are also in-

corporated into the created pattern. Content-based pattern matching is easy if contents always appear in the same format like email addresses and telephone numbers, but it is not so trivial when contents are given by free texts.

To overcome this difficulty, we propose a method that combines pattern matching and text classification. In our method, given a keyword, candidates of related texts are extracted by pattern matching using a set of patterns that relates a keyword to a target text, then predicted target texts are selected from the candidates by text classification. We also propose a learning method of a set of patterns and a text classifier from a training data.

In our method, each pattern is represented by a simple *DOM-tree* that consists of a root node and two paths leading to the leaves, one is a node for a keyword and the other is a node for a target text. This pattern DOM-tree is similar to that used by TreeWrapper [9] and L_{tagpath} [4], but our pattern matching allows elastic matching as considered by Zaki [12] in the context of frequent-tree mining, that is, two nodes of parent-child relation in a pattern tree can also match such two nodes as the relation between them is not parent-child but ancestor-descendant. By allowing this matching, we can construct more general patterns from training data. Using general patterns is important because only a few target texts are extracted by using patterns that are too specific and non-target texts extracted by pattern matching can be filtered out by text classification.

We propose a method of learning a set of above patterns, by which common pattern DOM-trees are obtained for each subset of positive training DOM-trees that have the same tag for the node containing target texts and also have the same tag for the least common ancestor of a node containing a given keyword and the node containing target texts. Common pattern DOM-trees can be obtained by using the sequencial mining algorithm AprioriALL [1] several times.

In the text classification phase, any classifier [2] can be used. Negative training data for learning a classifier are automatically generated from training data by using a set of patterns created in the previous phase. we adopted support vector machine (SVM) [11] as a classifier in our experiments because its good performance was reported [6].

According to our cross validation results on extracting reputations of a given *Ramen* shop from Web pages collected by a search engine, our method achieved 79.3% precision and 56.6% recall by allowing acceptable errors.

1.1 Other Related Work

As a wrapper using SVM, there is a method proposed by Kashima and Koyanagi [7]. Their use of SVM is not text classification but tree classification, and they use a special kernel for tree structure. Their method make use of tree structure only, so it seems to be difficult to extract target information from unfixed sites.

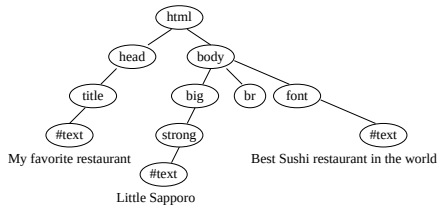
The method proposed by Ikeda, Yamada and Hirokawa [5] is one method that uses not only tree structure but also text information in a target text node. But they used text information in order to extract a part of a text in a text node by detecting template strings like LR wrapper does.

Tateishi, Ishiguro and Fukushima [10] developed a reputation search engine by using natural language processing technique. Their engine extracts reputation sentences from documents and does not use any special features of HTML documents.

2. PROBLEM STATEMENT

A *rooted* tree $T = (V, B)$ is a connected acyclic graph with a set V of vertices and a set B of directed edges $(N_{T,i}, N_{T,j}) \in V \times V$ that represent *parent-child* relation, which satisfies the condition that every vertex but just one vertex (*root*) has just one parent vertex. For a tree, a vertex and an edge are called a *node* and a *branch*, respectively. An *ordered* tree $T = (V, B, \preceq)$ is a rooted tree (V, B) with partial order '$\preceq$' on V representing a sibling relation, where the order is defined just for all the pair of *siblings*, children having the same parent. A (simplified) *DOM-tree*, a representation of an HTML document in *document object model*, is an ordered tree T in which each node N_T has two attribute *tag* and *text*. We assume tha a value of a tag attribute is an element in a label set L and a value of a text attribute is a string that is a sequence of characters in a character set Σ. We also assume that a text attribute of N_T is not *null* only when a tag attribute of N_T is "#text"[1].

[1]In HTML, the tag "#text" does not exist, but here we assume that all text nodes in a DOM-tree have this tag attribute value.




```
<html>
 <head><title> My favorite restaurant </title></head>
 <body bgcolor="yellow">
  <big><strong> Little Sapporo </strong></big><br>
  <font color="blue">Best Sushi restaurant in the world</font>
 </body>
</html>
```

Figure 1: Example of a DOM-tree and the HTML document it represents

An example of a DOM-tree and the HTML document it represents are shown in Figure 1. Note that nodes in our DOM-tree do not have other attributes like the attribute "bgcolor" for the node with the tag "body"[2].

An *id* of a node N_T is its position in the depth-first traversal of the tree T. In this paper, we let the depth-first traversal mean the unique one in which siblings are passed in the partial order '$\preceq$'.

The parent-child relation, which is defined by a set B of branches, induces another partial order '$\leq$', an *ancestor-descendant* relation, by extending the relation so as to satisfy reflexivity and transitivity. If $N_{T,i}$ and $N_{T,j}$ are not comparable in order '$\leq$', $N_{T,j}$ is said to be a *left-collateral* of $N_{T,i}$ when the id of $N_{T,i}$ is larger than the id of $N_{T,j}$ and a *right-collateral* of $N_{T,i}$ when the id of $N_{T,i}$ is smaller than the id of $N_{T,j}$

A *part-tree* S of a tree T is a connected subgraph of T. A node id of a part-tree S of a tree T is not its position in the depth-first traversal of the tree S, but the same as the id of its correspondence in T. Each tree T is assumed to have an unique *id*. In addition to such an unique id, each part-tree S of a tree T has a *tree id* that is the id of T.

We use notations shown in Table 1 throughout the paper.

In this paper, we address the following problem.

PROBLEM 2.1. *Let a training data set $\mathcal{D}$ be a set of triples (W, T, N_T^*), where W is a keyword, T is a DOM-tree that has text nodes containing W, and N_T^* is a node in T that has a target relation with W or N_T^* is* null *when there are no such nodes in T. For a given $\mathcal{D}$, learn a function f of a keyword W and a DOM-tree T that outputs a node in T that has a target relation with a keyword W.*

Relation between a restaurant and its reputation is an example of a target relation. In this paper, we set a node N_T^* of a training data (W, T, N_T^*) to the least common ancestor of all the text nodes that have a target relation with a keyword W.

[2]TreeMiner [9] uses those attributes.

T.root	root of T
T.id	id of T
S.treeid	tree id of S
N_T.tag	tag of N_T
N_T.text	text of N_T
N_T.parent	parent of N_T
N_T.firstchild	first child of N_T
N_T.nextsibling	next sibling of N_T
N_T.prevsibling	previous sibling of N_T
N_T.id	id of N_T
contain(s_0, s_1)	boolean function that is *true* if and only if s_1 appears in s_0
duplicate(N_T)	copied node of N_T
s.first	first element of s
x.value	value of x
x.next	next element of x
append(s, u)	append a list element with value u to s

Table 1: Notations used in this paper for a tree T, a part-tree S, a node N_T of T, strings s_1 and s_2, a list s and its element x.

3. METHOD

3.1 Prediction Function

A function f to be learned makes a prediction by using three functions f_{pat}, f_{con} and f_{dec}. The function f_{pat} is a function of a keyword W and a DOM-tree T that outputs a candidate node set $\{N_{T,1}, N_{T,2}, ..., N_{T,k}\}$ that may have a target relation between W. We call f_{pat} a *pattern-matching function* because the function winnows the candidates by pattern matching. The function f_{con} is a function of a node N_T that outputs a label 1 or -1 based on the attributes of the node. Here as attributes, we use the ones extracted from the texts that are contained in the subtree rooted by the node. So, we call f_{con} a *content-based function*. The function f_{dec} is a function of a set $\{(N_{T,1}, l_1), (N_{T,2}, l_2), ..., (N_{T,k}, l_k)\}$, where each pair is composed of a node $N_{T,j}$ and a label l_j, and out-

puts one node $N_{T,j}$ for $j \in \{1, 2, ..., k\}$ or null. We call f_{dec} a *decision function*. In this paper, we fix a function f_{dec} to the function that outputs the node of the smallest id among those having the positive label.

Then, a function f is defined as follows.

$$f(W, T) = f_{\mathrm{dec}}(\{(N_T, f_{\mathrm{con}}(N_T)) : N_T \in f_{\mathrm{pat}}(W, T)\})$$

3.2 Pattern-Matching Function

A function f_{pat} selects all the nodes in a DOM-tree T of which structural relation to a keyword W matches one of patterns in a set $\mathcal{P}$. A pattern is a pair of a *pattern tree* P and its *leaf distance* r. A pattern tree P is a simple DOM-tree that consists of a root node and two paths leading to the leaf nodes, a *keyword node* K_P whose tag is "#text" and a *target node* C_P. Here[3], we restrict pattern trees to the ones in which K_P appears before C_P, that is, $K_P.\mathrm{id} < C_P.\mathrm{id}$. The structural relation of a node N_T to a keyword W *matches* a pattern (P, r) if and only if there is a one-to-one mapping ϕ from the set of nodes in P to the set of nodes in T that satisfies the following six conditions for all nodes $N_{P,1}$ and $N_{P,2}$ in P.

1. **target matching**: $N_T = \phi(C_P)$
2. **Label preserving**: $N_{P,1}.\mathrm{tag} = \phi(N_{P,1}).\mathrm{tag}$
3. **Keyword matching**: $\mathrm{contain}(\phi(K_P).\mathrm{text}, W)$
4. **Ancestor-descendant relation preserving**: If node $N_{P,1}$ is a child of node $N_{P,2}$, node $\phi(N_{P,1})$ is a descendant of node $\phi(N_{P,2})$.
5. **Sibling relation preserving**: $\phi(K_P).\mathrm{id} < \phi(C_P).\mathrm{id}$.
6. **Acceptable distance**: $|\phi(K_P).\mathrm{id} - \phi(C_P).\mathrm{id}| \leq r$

An example of a pattern tree P and a matching function ϕ are shown in Figure 2. Note that the matching satisfying the above 6 conditions is elastic, and the nodes with tag "big" and tag "font" can be jumped over in the matching.

3.2.1 Implementation

An implementation of f_{pat} is shown in Figure 3. In the implementation, text nodes that contain the keyword

[3]Extension for allowing the reverse order is easy, but a related text appears after a keyword in most cases, we restricted the pattern trees.

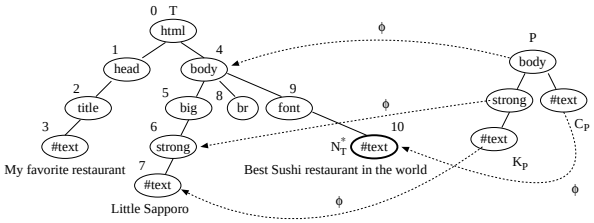


Figure 2: Example of a pattern tree P. The structural relation of the node N_T^* to keyword "Little Sapporo" matches the pattern $(P, 3)$.

W are searched first in the procedure *find-candidate*. For each found node and for each pattern $(P, r) \in \mathcal{P}$, its ancestors to which some mapping ϕ satisfying the above 6 conditions may map from the root node of P are searched in the procedure *matching*. Finally, for each such ancestor, its descendants whose structural relation to W matches the pattern (P, r) are searched in the procedure *find-node*. The worst-case computational time of this implementation is $O((k_{W,T} n_{\mathcal{P}} + 1) n_T + n_W + n_t)$, where $k_{W,T}$ is the number of nodes in T that contains the keyword W, $n_{\mathcal{P}}$ is the number of patterns in $\mathcal{P}$, n_T is the number of nodes in T, n_W is the length of W and n_t is the total length of all texts contained in T.

3.2.2 Learning

A pattern matching function f_{pat} needs a pattern set $\mathcal{P}$, which is learned from a training data $\mathcal{D}$. A proposed learning function *Learn-f_{pat}* is shown in Figure 4. The training data $\mathcal{D}$ is composed of triples (W, T, N_T^*), where W is a keyword, T is a DOM-tree and N_T^* is a node in T that has a target relation with W or N_T^* is null when there are no such nodes in T. The function Learn-f_{pat} uses *positive* data only, that is, triples (W, T, N_T^*) with $N_T^*!$ =null.

For each positive data (W, T, N_T^*), Learn-f_{pat} calls the function *extract-parttree* shown in Figure 5, which extracts one part-tree for each text node $N_{T,W}$ that is a left-collateral of N_T^* and of which text contains W. Each extracted part-tree consists of two nodes N_T^* and $N_{T,W}$, the least common ancestor $N_{T,0}$ of them, all the nodes between $N_{T,0}$ and N_T^* and all the nodes between $N_{T,0}$ and $N_{T,W}$. Search for $N_{T,W}$ is done by checking, for each ancestor node of N_T^*, all the descendant leaf text node that is a left-collateral of N_T^*, which

```
Function f_pat(W, T)
W : keyword, T : DOM-tree
begin
  return find-candidate(W, T.root)
end
Function find-candidate(W, N_T)
W : keyword, N_T : node of T
begin
  C = ∅
  if N_T.tag=="#text" and
        contain(N_T.text, W)==true then
    for each (P, r) ∈ P do
      matching(K_P, N_T, N_T.id+r, C)
    enddo
  endif
  M = N_T.firstchild
  while M! =null do
    C = C∪find-candidate(W, M)
    M = M.nextsibling
  enddo
  return C
end
Procedure matching(N_P, N_T, b, C)
N_P : node of P, N_T : node of T
b : maximum allowable node id
C : candidate set (updated when returned)
begin
  while N_T.parent! =null do
    if N_P.parent == null and
                          N_T.parent.tag == N_P.tag then
      M_P = N_P.firstchild.nextsibling
      M_T = N_T.nextsibling
      while M_T! =null do
        find-node(M_T, M_P, b, C)
        M_T = M_T.nextsibling
      enddo
    else if N_P.parent.tag==N_T.parent.tag then
      N_P = N_P.parent
      if N_P.parent ==null then continue
    endif
    N_T = N_T.parent
  enddo
end
Procedure find-node(N_T, N_P, b, C)
N_T : node of T, N_P : node of P
b : maximum allowable node id
C : candidate set (updated when returned)
begin
  if N_T.id > b then return
  if N_P.tag == N_T.tag then
    if N_P.firstchild == null then C = C ∪ {N_T}
                              else N_P = N_P.firstchild
  endif
  M = N_T.firstchild
  while M! =null do
    find-node(M, N_P, b, C)
    M = M.nextsibling
  enddo
end
```

Figure 3: Implementation of the function $f_{\mathbf{pat}}$

```
Function Learn-f_pat(D)
D : training data
begin
  D_pat = ∅
  for each (W, T, N*_T) ∈ D do
    if N*_T! =null then
      extract-parttree(D_pat, W, N*_T)
    endif
  enddo
  P = ∅
  for each E_{l,t} ∈ D_pat do
    P = P∪maximal-common-pattern(E_{l,t}, n)
  enddo
  return P
end
```

Figure 4: Function of learning a pattern set $\mathcal{P}$

is done by the function *keynode*. In the keynode function, the set of the extracted part-trees S are classified by two tags, a tag l of the root node of S and a tag t of the leaf node corresponding to N_T^*. After the executions of the function extract-parttree for all positive data, $\mathcal{D}_{\rm pat}$, a set of classes $\mathcal{E}_{l,t}$ of part-trees, is constructed. Note that function *get_partition*, *create* and *add* used in the procedure keynode are defined as follows. The function get_partition($\mathcal{D}_{\rm pat}, l, t$) returns the class $\mathcal{E}_{l,t}$ of part-trees for the pair (l, t) of tags if exists and null otherwise. This function is efficiently implemented by using a hash table. The function create($\mathcal{D}_{\rm pat}, l, t$) creates an empty class $\mathcal{E}_{l,t}$ for the pair (l, t) of tags and returns it. The function add($\mathcal{E}_{l,t}, S$) adds a part-tree S to the class $\mathcal{E}_{l,t}$.

For each class $\mathcal{E}_{l,t}$, generalized patterns are obtained by the function *maximal-common-pattern* shown in Figure 6, which is called by the function Learn-$f_{\rm pat}$. The function maximal-common-pattern outputs the maximal common patterns (P, r) of part-trees in a given set $\mathcal{E}_{l,t}$, namely, maximal patterns (P, r) that are matched by the structural relation of the node N_T^* to a keyword W for all $(W, T, N_T^*) \in \mathcal{D}$ satisfying that for T there is an $S \in \mathcal{E}_{l,t}$ with S.treeid $= T$.id. The acceptable distance r is calculated by the function max-leaf-distance (Figure 6), which outputs the maximum difference between two leaf node ids of S in $\mathcal{E}_{l,t}$.

The maximal common patterns for a given set $\mathcal{E}_{l,t}$ of part-trees can be obtained by using the algorithm *AprioriAll* several times with minimum support 1.0, where AprioriALL is an algorithm for mining sequential patterns developed by Agrawal and Srikant [1]. The first execution of AprioriAll

Procedure extract-parttree($\mathcal{D}_{\rm pat}, W, N_T$)
$\mathcal{D}_{\rm pat}$: transformed data (updated when returned)
W : keyword, N_T : node in T
begin
  S.root=null
  $t = N_T$.tag
  insert_child(S, duplicate(N_T))
  while N_T.parent! =null **do**
    insert_child(S, duplicate(N_T.parent))
    $l = N_T$.parent.tag
    $M_T = N_T$.prevsibling
    while M_T! =null **do**
      keynode($\mathcal{D}_{\rm pat}, W, M_T, S$.root,$l$,$t$)
      $M_T = M_T$.prevsibling
    enddo
    $N_T = N_T$.parent
  enddo
end
Procedure keynode($\mathcal{D}_{\rm pat}, W, N_T, N_S, l, t$)
$\mathcal{D}_{\rm pat}$: transformed data (updated when returned)
W : keyword, N_S : node in S, N_T : node in T
l : tag of the least common ancestor node
t : tag of the target node
begin
  add_child(N_S, duplicate(N_T))
  if N_T.tag=="#text" **and**
        contain(N_T.text, W)==true **then**
    $\mathcal{E}_{l,t}$ =get_partition($\mathcal{D}_{\rm pat}, l, t$)
    if $\mathcal{E}_{l,t}$ ==null **then** $\mathcal{E}_{l,t}$ =create($\mathcal{D}_{\rm pat}, l, t$)
    add($\mathcal{E}_{l,t}, S$)
  else
    $N_T = N_T$.firstchild
    while N_T! =null **do**
      keynode($\mathcal{D}_{\rm pat}, W, N_T, N_S$.firstchild,$l$,$t$)
      $N_T = N_T$.nextsibling
    enddo
  endif
  remove_child(N_S)
end
Procedre insert_child(S, N_S)
S : DOM-tree, N_S : copied node in T
begin
  N_S.firstchild=S.root
  N_S.nextsibling= N_S.prevsibling=null
  S.root= N_S
end
Procedre add_child(N_S, M_S)
N_S : node in S, M_S : copied node in T
begin
  M_S.nextsibling=N_S.firstchild
  M_S.firstchild= M_S.prevsibling=null
  N_S.firstchild= M_S
end
Procedre remove_child(N_S)
N_S : node in S
begin
  N_S.firstchild= N_S.firstchild.nextsibling
  if N_S.firstchild! =null **then** N_S.firstchild.prevsibling=null
end

Figure 5: Function of extracting a part-tree

Function maximal-common-pattern($\mathcal{E}_{l,t}$)
$\mathcal{E}_{l,t}$: set of part-trees
begin
  $\mathcal{L} = \{(S.\text{treeid,tag-seq}(S.\text{root.firstchild}))_{S.\text{id}} : S \in \mathcal{E}_{l,t}\}$
  $n = |\{i : (i, s) \in \mathcal{L}\}|$
  $\mathcal{S}_{\mathcal{E}_{l,t}}$ =AprioriAll($\mathcal{L}, 1.0 \times n$)
  $\mathcal{P} = \emptyset$
  for each $(s_1, \mathcal{I}_1) \in \mathcal{S}_{\mathcal{E}_{l,t}}$ **do**
    $\mathcal{L} = \{(S.\text{treeid,tag-seq}(S.\text{root.firstchild.nextsibling}))_{S.\text{id}} :$
      $S \in \mathcal{E}_{l,t}, S.\text{id} \in \mathcal{I}_1\}$
    $\mathcal{S}_{s_1}$ =AprioriAll($\mathcal{L}, 1.0 \times n$)
    for each $(s_2, \mathcal{I}_2) \in \mathcal{S}_{s_1}$ **do**
      P = make-pattern-tree(l, t, s_1, s_2)
      r = max-leaf-distance($\mathcal{E}_{l,t}, \mathcal{I}_2$)
      $\mathcal{P} = \mathcal{P} \cup \{(P, r)\}$
    enddo
  enddo
  return $\mathcal{P}$
end
Function AprioriAll($\mathcal{L}, \sigma$)
$\mathcal{L}$: set of pairs (tree id,tag sequence)
σ : (minimum suport)$\times$(number of training data)
begin
  $L_1 = \{(c, \mathcal{I}) : \mathcal{I} = \{i : (j, X)_i \in \mathcal{L}, \text{contain}(X, c) == \text{true}\},$
      c is a length-1 tag sequence with
      $|\{j : (j, X)_i \in \mathcal{L}, i \in \mathcal{I}\}| \geq \sigma\}$
  $k = 2$
  while $L_{k-1} \neq \emptyset$ **do**
    $C_k = \{(c_1, \mathcal{I}_1) \otimes (c_2, \mathcal{I}_2) : (c_1, \mathcal{I}_1), (c_2, \mathcal{I}_2) \in L_{k-1},$
      $\text{prefix}(c_1, k-2) == \text{prefix}(c_2, k-2)\}$
    $L_k = \{(c, \mathcal{I}) : (c, \mathcal{I}) \in C_k, |\{j : (j, X)_i \in \mathcal{L}, i \in \mathcal{I}\}| \geq \sigma\}$
    $k = k + 1$
  enddo
  return $\{(c, \mathcal{I}) : (c, \mathcal{I}) \in \bigcup_{h=1}^{k-1} L_h, c \text{ is maximal}\}$
end
Function tag-seq(N_S)
N_S : node in S
begin
  s =null
  while N_S.firstchild! =null **do**
    append(s, N_S.tag)
    $N_S = N_S$.firstchild
  enddo
  return s
end

Figure 6: Function of calculating the maximal common patterns (1)

is done for finding the maximal common tag subsequences of all the paths from the root node to the *first* leaf node of the part-trees in $\mathcal{E}_{l,t}$. Note that support is calculated based on the number of distinct *tree ids* of part-trees, namely, the number of trees from which part-trees are extracted. Thus, even for support 1.0, some of the tag sequences given to the function AprioriAll may not contain the obtained tag subsequences. In our implementation of AprioriAll (Figure 6), in addition to the maximal common tag subsequences s_1,

```
Function make-pattern-tree(l, t, s1, s2)
l, t : tag, s1, s2 : tag sequence
begin
  N.tag = l
  N.firstchild = N.nextsibling = N.prevsibling =null
  P.root = duplicate(N)
  make-path(P.root, s2, t)
  make-path(P.root, s1,"#text")
  return P
end
Procedure make-path(NP, s, u)
NP : node of P, s : tag sequence, u : tag
begin
  x = s.first
  while x! =null do
    N.tag = x.value
    add_child(NP, duplicate(N))
    NP = NP.firstchild
    x = x.next
  enddo
  N.tag = u
  add_child(NP, duplicate(N))
end
```

Figure 7: Function of calculating the maximal common patterns (2)

the set $\mathcal{I}_1$ of ids of part-trees that contains s_1 is returned for each s_1. In the function AprioriAll, a length-k candidate c for frequent sequences is generated from two length-$(k-1)$ frequent sequences c_1, c_2 with the same prefix of length $k-2$ (prefix$(c_1, k-2)$ = prefix$(c_2, k-2)$) by appending the last tag of c_2 to c_1. By a *join* operation of $(c_1, \mathcal{I}_1)$ and $(c_2, \mathcal{I}_2)$ denoted by $(c_1, \mathcal{I}_1) \otimes (c_2, \mathcal{I}_2)$, $(c, \mathcal{I})$ is created, where $\mathcal{I} = \{i : i \in \mathcal{I}_1 \cap \mathcal{I}_2, \text{contain}(X, c) = \text{true for } (j, X)_i \in \mathcal{L}\}$.

For each $(s_1, \mathcal{I}_1)$ obtained by the first execution of AprioriAll, AprioriAll is executed for finding the maximal common tag subsequences of all the paths from the root node to the *second* leaf node of the part-trees in $\{S : S \in \mathcal{E}_{l,t}, S.\text{id} \in \mathcal{I}_1\}$. Note that all part-trees in $\mathcal{E}_{l,t}$ have the tag l of the root node, the tag "#text" of the first leaf node and the tag t of the second leaf node. The tag sequences given to AprioriAll do not contain the tags of those nodes. In the function make-pattern-tree (Figure 7), which is called by the function maximal-common-pattern, a pattern tree P is constructed finally from the tags of the root node and two leaf nodes, and the two common tag subsequences obtained by AprioriAll.

The worst-case computational time of the function extract-parttree is $O((k_{W,T,N_T^*}+1)n_T + n_W + n_t)$, where k_{W,T,N_T^*} is the number of nodes in T that is a left-collateral of N_T^* and of which text contains W, n_T is the number of nodes in T, n_W is the length of W and n_t is the total length of all texts in T. Note that searching nodes with text that contains W takes $O(n_T)$, copying found part-trees takes $O(k_{W,T,N_T^*} n_T)$, and string pattern matching takes $O(n_W + n_t)$. Therefore, the worst-case computational time of the first loop in the function Learn-f_{pat} is $O((k_{\mathcal{D}}+1)n_{\mathcal{D}} + n_{\mathcal{W}} + n_d)$, where $n_{\mathcal{D}}$ is the sum of the number of nodes in T belonging to $\mathcal{D}$, $k_{\mathcal{D}} = \max_{(W,T,N_T^*)\in\mathcal{D}} k_{W,T,N_T^*}$, and $n_{\mathcal{W}}$ and n_d are the sum of n_W and n_t, respectively, for all training data $(W, T, N_T^*) \in \mathcal{D}$. In all the executions of AprioriAll, totally at most $2^{l_{\mathcal{P}}+1} n_{\mathcal{P}} |L|$ candidates are generated, where $l_{\mathcal{P}}$ is the maximum number of nodes between a root node and a leaf node of pattern trees in $\mathcal{P}$, $n_{\mathcal{P}}$ is the number of the maximal common patterns, and $|L|$ is the number of tags. For each candidate pattern, computational time $O(k_{\mathcal{D}} n_{\mathcal{D}})$ is enough for its support calculation, so the worst-case computational time of function maximal-common-pattern is $O(2^{l_{\mathcal{P}}+1} n_{\mathcal{P}} |L| k_{\mathcal{D}} n_{\mathcal{D}})$. Thus, the worst-case computational time of the function Learn-$f_{\mathcal{P}}$ is $O(2^{l_{\mathcal{P}}+1} n_{\mathcal{P}} |L|(k_{\mathcal{D}}+1)n_{\mathcal{D}} + n_{\mathcal{W}} + n_d)$.

3.3 Content-based Function

The function f_{con} classifies a given node N_T into $+1$ or -1 by its attribute values. Since a target information is composed of texts in our problem setting, any text classification technique can be used to design the function f_{con}. As an attribute vector, we use an *index term vector* [2] that is created from all the texts of the descendant nodes of the given node N_T. An index term vector is a collection of weights associated with each representative keyword called *index term*. Its weights can be binary, normalized term frequencies or *tf-idf*s. A classifier used in the function f_{con} can be any one that inputs a real valued vector.

3.3.1 Training data generation for a classifier

A training data for a classifier is generated from the original training data $\mathcal{D}$ by the function *Data-generation* shown in Figure 8. For each $(W, T, N_T^*) \in \mathcal{D}$, the pattern matching function f_{pat} is executed. A positive training data instance is generated for each N_T^*. A negative training data instance is generated for each N_T in a set $\mathcal{C}$ output by the function f_{pat} except for $N_T.id \geq N_T^*.id$.

```
Function Data-generation(D)
D : training data
begin
  D_con = ∅
  for each (W, T, N*_T) ∈ D do
    C = f_pat(W, T)
    for each N_T ∈ C do
      l = −1
      if N*_T! =null then
        if N_T.id > N*_T.id then continue
        if N_T.id == N*_T.id then l = 1
      endif
      v⃗ =index-term-vector(N_T)
      D_con = D_con ∪ {(v⃗, l)}
    enddo
  enddo
  return D_con
end
```

Figure 8: Function of generating training data for a classifier

Shop No.	#HTML	#POS	#ONE	#MANY
1	34	26	8	18
2	19	9	2	7
3	73	46	15	31
4	24	14	4	10
5	26	12	4	8
6	23	17	7	10
7	21	11	6	5
8	33	21	6	15
9	20	13	5	8
10	28	20	5	15
total	301	189	62	127

Table 2: Experimental data (#ONE: number of pages containing reputations of the target shop only, #MANY: number of pages also containing reputations of other shops)

4. EXPERIMENTS

4.1 Methodology

The HTML documents we used in our experiment are Web pages containing the information about a given Ramen (lamian, Chinese noodles in Soup) Shop. Among the most popular 100 Ramen-shops introduced in a popular local town information magazine[4], we selected the most popular 10 Ramen shops with more than 15 Web pages retrieved by keyword search[5] using a shop name and its telephone number. Totally 301 pages were retrieved and 189 pages of them contained the reputation about a target shop. (See Table 2.)

We constructed the experimental data (W, T, N_T^*) from the collected HTML pages and shop names used to retrieve them. Each non-null target node N_T^* of a DOM-tree T was set to the least common ancestor node[6] of the text nodes whose text contains the target reputation.

We conducted cross validation by using the pages retrieved for each one of 10 shops as test data.

Index terms we used in our experiments are basically standard forms of nouns, verbs, adjectives and adverbs. We dealt with Japanese text, and Chasen[7], one of Japanese morphological analysis systems, is used to extract index terms. As a term weight, we used a term frequency normalized with respect to Euclidean norm. A support vector machine (SVM) was used as a classifier in the experiments. We used SVMTorch II[8] with polynomial kernel $K(x, y) = (xy + 1)^3$ and other default parameters.

We compared our method with other two simpler methods, a method using patterns only and a method using simple patterns. A method using patterns only is the one that uses$f_{\rm con} \equiv 1$ in our proposed prediction function. Simple patterns used in the second method are pairs (t, r) of a tag t for a target node and a distance r from a keyword node. For these patterns, all the nodes that have tag t and id i which satisfies $k < i \leq k+r$ are matched, where k is the id of a node whose text contains a given keyword. A method using simple patterns is the one that uses this pattern matching function as the function $f_{\rm pat}$ in our proposed prediction function. A set $\mathcal{P}$ of simple patterns (t, r) learn from training data $\mathcal{D}$ for tests and training data generation for a classifier is

$$\{(t, r) : t \in L, r = \max_{(W,T,N_T^*)\in\mathcal{D}_t} (N_T^*.\mathrm{id} - \mathrm{min\_id}(W, T))\},$$

where

$$\begin{aligned} L &= \{N_T^*.\mathrm{tag} : (W, T, N_T^*) \in \mathcal{D}\}, \\ \mathcal{D}_t &= \{(W, T, N_T^*) \in \mathcal{D} : N_T^*.\mathrm{tag} = t\} \text{ and} \\ \mathrm{min\_id}(W, T) &= \min\{N_T.\mathrm{id} : \mathrm{contain}(N_T.\mathrm{text}, W)\}. \end{aligned}$$

4.2 Results

The results are shown in Table 3. Note that in the columns

[4]Hokkaido Walker 2002 NO.3.
[5]Google(www.google.co.jp) was used in the search.
[6]If the subtree rooted by the least common ancestor node contains reputations of other restaurants, the most informative one text node is selected as a representative instead of the least common ancestor node.
[7]ChaSen 2.3.3. (URL:chasen.aist-nara.ac.jp/hiki/ChaSen)
[8]URL:www.idiap.ch/ bengio/projects/SVMTorch.html

for f_{pat}, f_{con} and f_{dec}, the number of correctly extracted nodes included in the outputs of the functions for N_T^*! =null is written. A *precision* is a rate of the number of correctly extracted nodes among all the extracted nodes, and a *recall* is a rate of the number of correctly extracted nodes among all the nodes that should be extracted.

The method using patterns only suffers low precision and recall even for training data. Compared to this method, the other two methods, which also use contents information, obtain rather higher precision and recall for both data. The proposed method outperformed the method using simple patterns by $5 \sim 7$ percentages. The numbers in the parentheses are precisions and recalls that were calculated by including extracted *acceptable nodes* in the set of correctly extracted nodes. Here, acceptable nodes are defined as the ones which include a part of reputations about a target shop and exclude any information about other shops. In the proposed method, 79.3% of extracted nodes were acceptable and acceptable nodes are extracted for 56.6% of the nodes that should be extracted.

4.3 Discussion

Figure 9 shows all patterns output by Learn-f_{pat} for the whole experimental data. #NUM is the number of data that matches each pattern. Each pattern tree is represented by its *string encoding*, a tag sequence of nodes in their passing order in a depth-first traversal, where the additional tag "−1" is added when backtracking from a child. Nodes corresponding to the tags with suffixes "(1)" and "(2)" are a keyword node and a target node, respectively. A several patterns of table structure in the found pattern set, such as "table tr td #text(1) -1 -1 -1 tr td(2) -1 -1", look useful to winnow candidates. Some patterns such as "body #text(1) -1 #text(2) -1" are too general to winnow candidates, and pruning or further specification of such patterns may increase precision. Though this seems to decrease recall, but note that this can also increase recall if quality of negative training data for learning a classifier improves by using different set of patterns.

The reputations extracted for Shop 2 by using patterns and an SVM learned from data of other shops are shown in Table 4. In the table, "◯" indicates that a reputation is correctly extracted, "△" indicates that the corresponding extracted node is acceptable, and "×" indicates that the text contains no reputation about a target shop. All the extracted texts looks reputations of some shops, so text classification by SVM works to some extent. For the first "×"-labeled text, the target node was included in the output of a pattern-matching function, but was labeled −1 by a content-based function. For the second "×"-labeled text, the target node was not output by a pattern-matching function.

△ A super-popular shop selected as the best Ramen shop!!
◯ In the shop, there was a card saying that the taste of our Ramen is the result of pursuing delicious Ramen. The taste is mild and fairly heavy but its aftertaste is clear. Fat back floating on the soup is made by stewing it several hours in a separate cooking pot. The soup has become whitish by stewing pig bones and pig legs more than 12 hours. The noodle, which is thickish and curly, is made by noodle company K. Besides this soy sauce Ramen, ZYANSYANMEN is also popular.
× Shop S. Sapporo. Tram station named NISHI 9 ZYO. You can enjoy pork-based salt, soy sauce and miso Ramens. The soup is made from HIDAKA konbu and dried SANRIKU saury. The Ramen contains three thick slices of roast pork, in most cases, two slices of pork belly and one slice of pork loin. I felt nice to see that the noodle was well drained by a flat bamboo sieve. Each taste of the three kinds of Ramens (700 yen each) is good. Besides, a chicken-based limited salt Ramen, which price is 800 yen, tastes better. Chicken fat is floating on all the surface of the soup. The soup also contains saury taste and the harmony of the taste is good. It is also nice that they use rock salt. My recommendation is this limited salt Ramen. However, the soup of the other Ramens is a little tepid, especially, miso Ramen is quite tepid, so it is not recommended. The fact that miso Ramen is tepid is the character of noodle company K? 2003.04.26. eating of the limited salt Ramen 2003.04.27. I have eaten miso Ramen. limited salt ⋆ Sapporo city ··· (telephone number) Closed on Monday. Open 11:30-20:30. Parking space for 3 cars. 15 seats.
◯ I had been hesitating to introduce my favorite Ramen shops because everyone has a different taste. (It was OK for me to introduce curry Ramens last time because it is a special kind of Ramens.) Some people might disagree with me, but the shop I am introducing here is the best among the Ramens I have tasted for the last 5 or 6 years. The noodle of this Ramen is thick and curly, and the soup is made by stewing pig bones. You can enjoy three tastes, miso, salt and soy sauce. The noodle is a little hard, so you can get normal noodle if you say that you like softish noodle. The soup is a little thick because of using thick noodle, but I felt that it was not so strong. The shop was introduced in Hokkaido Walker in Febuary, 2002, as the second popular Ramen shop. The most popular one introduced in the magazine was Shop G. (Evaluations appeared in this page are the ones I have done personally after eating the Ramens actually.) Q: Is Sapporo near Muroran? You said that you introduce about shops around Muroran. A: Yes! (I am sure.)
◯ Papa A: I have heard that the shop is crowded, and it was true. People might have been waiting for the opening of this famous shop at this convenient place. Hardness of noodle of Ramens we ordered, salt, miso and soy sauce was different, which cause might be crowdedness. They did not frequently change the water used for boiling the noodle. Are those acceptable? The miso and soy sauce Ramen still tasted good. Did the salt Ramen taste like this before? The noodle is made by Company K. ——— ⋆ ⋆ ⋆
× Pork that was like roast pork and stewed until it became soft is extremely delicious.
◯ Since there was a long waiting line at Sunday noon, I went there a little after 11 o'clock on a weekday. People came in the shop after by after, and the shop became crowded soon. The soup of the Ramens is based on pig bones. The miso soup is rich and a little spicy. The noodle is fairly hard and curly. I want to eat a popular soy sauce Ramen next time. The shop is clean. There is parking space.
△ Fat back poured finally makes the soup rich taste!

Table 4: Reputations extracted for Shop 2 (translated from Japanese)

used information	training data		test data						
	ave. pre.	ave. rec.	f_{pat}	f_{con}	f_{dec}	precision		recall	
patterns only	0.113	0.169	166	166	30	0.106		0.159	
simple patterns + contents	0.907	0.897	183	86	74	0.561	(0.727)	0.391	(0.508)
patterns + contents (proposed)	0.943	0.938	166	95	85	0.630	(0.793)	0.450	(0.566)

Table 3: Precisions and recalls for three methods

#NUM	r	pattern tree	simple pattern (t,r)
30	245	body #text(1) -1 #text(2) -1	(#text,496),#NUM=139
20	452	table tr #text(1) -1 -1 tr td #text(2) -1 -1 -1	
19	80	tr td #text(1) -1 -1 td #text(2) -1 -1	
19	256	td #text(1) -1 #text(2) -1	
16	177	html head title #text(1) -1 -1 -1 body #text(2) -1 -1	
10	8	p #text(1) -1 #text(2) -1	
8	496	tbody tr td #text(1) -1 -1 -1 tr td #text(2) -1 -1 -1	
7	38	li #text(1) -1 #text(2) -1	
5	270	div #text(1) -1 #text(2) -1	
1	12	blockquote strong #text(1) -1 -1 #text(2) -1	
1	28	ol li #text(1) -1 -1 li #text(2) -1 -1	
1	22	h3 #text(1) -1 #text(2) -1	
1	5	dl dt #text(1) -1 -1 dd #text(2) -1 -1	
1	2	span strong #text(1) -1 -1 #text(2) -1	
18	51	tr td #text(1) -1 -1 td(2) -1	(td,265),#NUM=52
13	56	table tr td #text(1) -1 -1 -1 tr td(2) -1 -1	
11	173	tbody tr td #text(1) -1 -1 -1 tr td(2) -1 -1	
4	265	html head title #text(1) -1 -1 -1 body table tr td(2) -1 -1 -1 -1	
4	64	body #text(1) -1 table tr td(2) -1 -1 -1	
2	47	td strong #text(1) -1 -1 table tr td(2) -1 -1 -1	
7	166	body #text(1) -1 p(2) -1	(p,166),#NUM=13
2	151	html head title #text(1) -1 -1 -1 body p(2) -1 -1	
2	33	tr td h2 #text(1) -1 -1 -1 td p(2) -1 -1	
2	9	td #text(1) -1 p(2) -1	
3	107	html head title #text(1) -1 -1 -1 body table(2) -1 -1	(table,107),#NUM=6
3	7	body table tr #text(1) -1 -1 -1 table(2) -1	
2	135	html head title #text(1) -1 -1 -1 body table tbody tr(2) -1 -1 -1 -1	(tr,135),#NUM=5
2	120	tbody tr td #text(1) -1 -1 -1 tr(2) -1	
1	66	body table tbody tr td strong #text(1) -1 -1 -1 -1 -1 -1 table tbody tr(2) -1 -1 -1	
4	1	html head title #text(1) -1 -1 -1 body(2) -1	(body,1),#NUM=4
4	12	tbody tr td #text(1) -1 -1 -1 tr td span(2) -1 -1 -1	(span,12),#NUM=4
2	17	html head title #text(1) -1 -1 -1 body blockquote(2) -1 -1	(blockquote,17),#NUM=4
2	2	body p strong #text(1) -1 -1 -1 blockquote(2) -1	
2	23	body strong #text(1) -1 -1 p strong(2) -1 -1	(strong,23),#NUM=3
1	17	html head title #text(1) -1 -1 -1 body p strong(2) -1 -1 -1	

Figure 9: Patterns output by $f_{\mathbf{pat}}$ for the set of all the data

According to Table 3, 12% of the target nodes were not extracted by a pattern-matching function, and 43% of the other target nodes were not labeled 1 by a content-based function. Considering high precision and recall for training data, the reason why so much target nodes were not labeled 1 by the content-based function may be that the number of training data for the function was too small to learn the function. This problem may be solved to some extent by reducing the number of index terms through more careful selection of them.

5. CONCLUSIONS

We proposed a combined method of structural pattern matching and text classification that can extract texts related to a given keyword from Web pages in unfixed sites. In our experiments on reputation extraction, our method achieved about 79.3% precision and 56.6% recall by allowing acceptable errors. The precision looks rather low compared to that of conventional wrappers, but our method can extract more information because sites from which texts are extracted are not fixed, and even a set of extracted texts that contains a small percentage of wrong ones can be used to assist people's decision if original Web pages can be accessed by clicking the extracted texts. In order to improve precision while keeping recall, it seems also necessary to detect repetition structures considered in [3].

Acknowledgments

We would like to thank Prof. Hiroki Arimura for helpful comments.

6. REFERENCES

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. 11th Int'l Conf. on Data Eng.*, pages 3–14, 1995.

[2] R. Baeza-Yates and B. Ribriro-Neto. *Modern Information Retrieval*. ACM Press, New York, NY, 1999.

[3] C.-H. Chang and S.-C.Lui. Iepad: Information extraction based on pattern discovery. In *Proc. of 10th Int'l World Wide Web Conf.*, pages 4–15, 2001.

[4] W. W. Cohen, M. Hurst, and L. S. Jensen. A flexible learning system for wrapping tables and lists in html documents. In *Proc. of 11th Int'l World Wide Web Conf.*, pages 232–241, 2002.

[5] D. Ikeda, Y. Yamada, and S. Hirokawa. Expressive power of tree and string based wrappers. In *Proc. of IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03)*, pages 21–26, 2003.

[6] T. Joachims. Text categorization with support vector machines: Learning with many relevant features. In *Proc. of 10th European Conference on Machine Learning*, pages 137–142, 1998.

[7] H. Kashima and T. Koyanagi. Kernels for semi-structured data. In *Proc. of 19th International Conference on Machine Learning (ICML 2002)*, pages 291–298, 2002.

[8] N. Kushmerick. Wrapper induction:efficiency and expressiveness. *Artificial Intelligence*, 118:15–68, 2000.

[9] Y. Murakami, H. Sakamoto, H. Arimura, and S. Arikawa. Extracting text data from html documents. *The Information Processing Society of Japan (IPSJ) Transactions on Mathematical Modeling and its Applications (TOM)*, 42(SIG 14(TOM 5)):39–49, 2001. In Japanese.

[10] K. Tateishi, Y. Ishiguro, and T. Fukushima. A reputation search engine that collects people's opinions by information extraction technology. *The Information Processing Society of Japan (IPSJ) Transactions on Databases (TOD)*, 45(SIG 07), 2004. In Japanese.

[11] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, NY, 1995.

[12] M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proc. SIGKDD'02*, pages 71–80, 2002.