# XML Version Detection

Deise de Brum Saccol
Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500
Porto Alegre, RS, Brazil
55 51 37376345
deise@inf.ufrgs.br

Nina Edelweiss
Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500
Porto Alegre, RS, Brazil
55 51 33166808
nina@inf.ufrgs.br

Renata de Matos Galante
Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500
Porto Alegre, RS, Brazil
55 51 33 08 77 46
galante@inf.ufrgs.br

Carlo Zaniolo
University of California
405 Hilgard Avenue
Los Angeles, CA, United States
1 310 8258137
zaniolo@cs.ucla.edu

## ABSTRACT

The problem of version detection is critical in many important application scenarios, including software clone identification, Web page ranking, plagiarism detection, and peer-to-peer searching. A natural and commonly used approach to version detection relies on analyzing the similarity between files. Most of the techniques proposed so far rely on the use of hard thresholds for similarity measures. However, defining a threshold value is problematic for several reasons: in particular (i) the threshold value is not the same when considering different similarity functions, and (ii) it is not semantically meaningful for the user. To overcome this problem, our work proposes a version detection mechanism for XML documents based on *Naïve Bayesian* classifiers. Thus, our approach turns the detection problem into a classification problem. In this paper, we present the results of various experiments on synthetic data that show that our approach produces very good results, both in terms of recall and precision measures.

## Categories and Subject Descriptors

I.7.1 [**Document and Text Processing**]: Document and Text Editing – *version control, document management*.

## General Terms

Management, Measurement, Experimentation

## Keywords

XML, versioning, similarity functions, classification

*DocEng'07*, August 28–31, 2007, Winnipeg, Manitoba, Canada.
Copyright 2007 ACM 978-1-59593-776-6/07/0008...$5.00.

## 1. INTRODUCTION

Version is the description of an object in a period of time or under a certain point of view, whose recording is important for the considered application. The applications of the version concept are many and diverse. Previous works focused on version management and querying rather than version detection [1][2][3][4]. However, the version detection problem is critical for many applications, such as plagiarism detection, Web page ranking, software clone identification, and peer-to-peer searching. For plagiarism detection, comparing file checksums is enough for detecting exact replicas, but totally useless for detecting partial copies [5][6]. However, such plagiarism can be detected by considering partial copies as versions. Likewise, the Web page ranking process will greatly benefit from such detection mechanism by ranking highly new versions of existent top-ranked pages [7]. The software clone problem often arises during the development of systems with negative impact on their maintenance [8]. By considering such clones as versions, this problem is managed and reduced. Finally, traditional peer-to-peer systems that are not aware of the existence of resource versions often face increasing complexity at the logical level and inefficiency at the physical level. These drawbacks can be reduced by using an automatic version detection mechanism.

This paper focuses on version detection of XML documents. By *version* we denote different but very similar representations of the same real-world object. Similarity can be measured by several metrics, such as content, structure or related subject. Considering this assumption, the general idea is that two files with high similarity are considered versions of the same document[1]. However, the version detection problem poses two important issues: the first is how to measure similarity between files and the second is how to define the minimum degree of similarity required for a file to be considered a version of another. To solve the first problem, several similarity functions are available for atomic values (e.g. string,

[1] The term *file* refers to a physical representation; *document* refers to the representation of an object in the real world. In other words, one document can be stored as many files.

integer) [9][10] and complex values (e.g. collections, tuples) [11][12][13]. Similarity functions for atomic values are domain-dependent, while similarity functions for complex values are usually based on the structure of the object [14]. However, finding similarity functions for XML documents that are effective for version detection represents an open research issue. The similarity function for version detection must consider the content and structure similarity, and also other features that match the requirements for each specific application. Based on the typical behavior of the document evolution in specific applications, some features can be more relevant for detecting versions.

The use of similarity functions in version detection poses a second challenge: this is the problem of determining the threshold value that should be used [15]. This difficulty occurs because the distribution of the score values generated by distinct similarity functions may be completely different – it may even vary when the same similarity function is applied to different data sets [16]. Moreover, the threshold definition is usually a task performed by the user, and it tends to be an error-prone activity. The user has to try several different values to get a satisfactory result. If the threshold value is too high, the number of false negatives can be high. On the other hand, if the threshold value is too low, the number of false positives will be high.

To overcome these two problems, our work proposes a new version detection mechanism for XML documents. The mechanism is part of *DetVX*, an environment for **Det**ection, management and querying of replicas and **V**ersions of **X**ML documents in a peer-to-peer context [17]. The main contributions of this paper are:

- The definition of a similarity function for XML files used as the basis for detecting versions: the function is not restricted to a specific application and it can be adapted to consider other relevant similarity features for specific scenarios.
- The definition of a version detection mechanism based on classification techniques: the detection mechanism relies on the use of *Naïve Bayesian* classifiers, which eliminates the previously discussed disadvantages of using thresholds.

The paper is organized as follows. Section 2 presents some related works. In Section 3 we present the main guidelines for the version detection mechanism. The similarity function is discussed in Section 4. Section 5 details the detection mechanism based on *Naïve Bayesian* classifiers. Experimental results are shown in Section 6. Metadata management is described in Section 7. Finally, concluding remarks are discussed in Section 8.

## 2. RELATED WORK

The version detection mechanism proposed in this paper relies on the use of a similarity function. A similarity function $f(f1,f2) \rightarrow sim$ generates a score *sim* to a pair of files *f1* and *f2*, where $0<=sim<=1$. The higher the *sim* value, the more similar are the files. Based on the similarity values, versions can be detected by using a threshold: files whose *sim* is greater than a given threshold are considered versions; files whose *sim* is lower than the threshold are considered different documents. Some existent works address the similarity functions, while others address the threshold definition, as discussed below.

Existent researches on change detection can be used as a basis for measuring similarity. Some approaches use *diff* algorithms to detect differences between files [18][19]. However, *diff* results are a delta script with no semantic information regarding the similarity between files. Another possibility is to analyze their ordered tree representations by calculating the edit distance, i.e., the minimum cost to transform one tree into another using basic operations [20][21]. Also, tree edit distance results do not contain meaningful information related to the similarity level that could be used to detect resource versions. Moreover, some approaches for similarity assessment consider only the textual content of documents [11], while others consider the structure [12][13].

Besides choosing a proper similarity function, it is also fundamental to understand its behavior to select the best threshold. As identified by the experiments in [16] and also in [22][23], some similarity functions are more adequate than others in that the values of recall and precision tend to be higher and less dependent from the threshold values. But the threshold definition also depends on the data set and the quality of results expected by the user. A high threshold value will consider as versions only files that present high similarity (almost replicas). On the other hand, a low threshold value can consider as version a large number of files that present low similarity (i.e., different documents).

The discussed works lead us to two conclusions. The first conclusion is that there are a variety of similarity functions, either for atomic values (e.g. *Levenshtein* or *Edit Distance* (Edit) [9], *Guth* [10] and *N-grams* [9]) or for documents [11][12][13]. However, these functions are often suitable for some specific requirements or interests. For example, some XML similarity functions focus more on structure rather than content; others focus more on content than structure. However, for the version detection problem many different features must be considered together. In this paper, we define a similarity function that considers several characteristics with different weights to achieve a more flexible approach.

The second conclusion is that the threshold definition is not a trivial task, even if it is automatically defined or manually chosen by a user. Some works address threshold definition [24][25], but there is not a widely accepted approach. Usually the similarity value is semantically poor. Different functions produce different distributions, which results in different interpretations for the similarity values [14]. In other words, the result quality (measured by recall and precision) can vary among functions when a specific threshold is chosen. This means that a threshold chosen for one set of files may not be adequate for another set of files. In order to obtain more robust methods we need to eliminate the dependence of threshold. Thus, we turn the version detection mechanism into a classification problem, for which no threshold definition is necessary. This is done by using *Naïve Bayesian* classification technique, a simple probabilistic classifier with strong independence assumptions [26].

There are some works on document classification based on *Naïve Bayesian* classification [27][28]. In these works, the goal is to assign a single electronic document to the category that is most relevant. In our proposal, the goal is to categorize pairs of documents in two classes (i.e., versions and non-versions). There are a set of features that must be taken into account when defining the necessary requirements for versions. This paper defines these requirements in a similarity function and applies the classification technique for version detection.

# 3. VERSION DETECTION

The version detection mechanism seeks to verify if two files are versions of the same document. In this proposal, version detection is based on file similarity. The general idea is that two files with high similarity are considered versions of the same document. For measuring similarity, we define a set of attributes (i.e., features) that must be assessed to be considered a version.

Our proposed procedure for version detection performs three activities. The first activity is named *similarity analysis* and it is responsible for applying a similarity function to each pair of files. The second activity is the *classification*, which is responsible for detecting the versions, based on data generated in the previous task using *Naïve Bayesian* classifiers. Finally, some metadata collected during the version detection process are maintained by the activity named *metadata management.*

In this work, versions are managed as separate files. The same detection mechanism can be used for both linear and branched versioning. The behavior of the mechanism is the same for both versioning types. It only differs in the structure of the tree that is traversed during the *similarity analysis* phase. Let us assume that the similarity function $f(f1,f2) \rightarrow sim$ generates a score *sim* to a pair of files *f1* and *f2*. The *sim* values are based on discrete mathematics (varying in 0.01 scale). Thus, there are hundred different possible values for the similarity function result.

The structure of the traversed tree depends on the type of versioning used, as discussed below.

## 3.1 Linear Versioning

In this type of versioning, the document's evolution creates a linear sequence of versions: $V_1$, $V_2$, . . ., $V_j$, where $V_j$ is the current version. A new version ($V_{j+1}$) is established by applying a number of changes (insertions, deletions or updates) to the current version ($V_j$ ) [29]. This leads to a single sequence of consecutive versions.

In linear versioning, the detection mechanism compares pairs of files (e.g. *f1* and *f2*), where *f1* is the version candidate and *f2* is the current version of an existent document. The current version can be either specified by the user or assumed by default. By default, we consider the modification date of the file versions; the file that was last modified is considered the current version. Consider that we have two sets of version sequences, as depicted in Figure 1: the first sequence (i.e., *f1*, *f2* and *f3*) corresponds to three versions of a document *D1*; the second sequence (i.e., *f4* and *f5*) corresponds to two versions of a document *D2*. Consider that the current versions of these sets are, respectively, *f3* and *f5* (assumed by default or informed by the user). The similarity value between files is represented over the arrows.

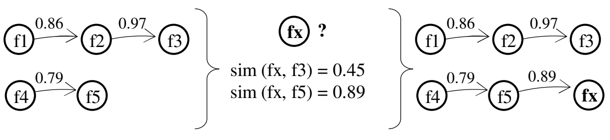

**Figure 1. Linear versioning sequences**

We compare the candidate file *fx* with *f3* and *f5*, producing the similarity values, such as *sim (fx, f3) = 0.45* and *sim (fx, f5) = 0.89*. The pair of files with higher similarity (i.e., *fx* and *f5*) is considered to be linear versions.

## 3.2 Branched Versioning

In this type of versioning, a new version can be derived from any previous one, creating a tree of versions [30]. When branched versioning is used, the detection mechanism compares pairs of files (e.g. *f1* and *f2*), where *f1* is the version candidate and *f2* is any existent version belonging to the tree. Consider the tree of versions shown in Figure 2. Let us assume that this tree corresponds to five versions of a document *D1*.

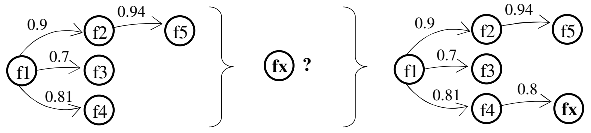

**Figure 2. Branched versioning tree**

Similar to the previous example, the candidate file *fx* is compared to some existent versions. However, according to the branched versioning definition, the candidate file *fx* can be derived from any previous version *fy* (i.e., *f1*, *f2*, *f3*, *f4*, or *f5*). Thus, in an optimal solution, *fx* must be compared with all the tree nodes. The pair of files with higher similarity is considered to be versions. However, there are different approaches for traversing the tree and choosing the order to compare the files, as follows:

1. Compare *fx* with all the files following a random order, using depth-first or breadth-first search order. This approach is simple and optimal, since *fx* is compared with all the previous versions. However, it can have a high cost depending on the number of the nodes the tree presents;

2. Compare *fx* with all the files that are leaf nodes in the tree. This approach is not optimal, since we do not compare the candidate file with intermediary nodes. However, it has a lower cost, since only a subset of the tree is compared with *fx*;

3. Compare *fx* with all the files following a depth-first or breadth-first search in a reverse order (i.e., starting by a leaf node) and stop traversing when the comparison achieves a *stop-condition*. A *stop-condition* can be: time of processing, minimum level of similarity expected by the user, number of visited nodes, or number of *K*-visited nodes (where *K* is the number of files that were last modified). This approach is not optimal, but it also produces a lower cost.

One of these three approaches must be chosen on the basis of the behavior of the document evolution for a specific application. For example: if the versions tend to evolve from one of the last versions that were generated, the second approach could be chosen, since the leaf nodes represent the last versions created from a specific branch. If the versions tend to evolve from the last modified version, then the third approach could be chosen (for *K=1*). However, if the document does not follow a typical evolving behavior and the user seeks an optimal solution, then the first approach shall be chosen. Moreover, a combination of approaches could be used for specific applications.

For instance, consider the example in Figure 2, and let us assume that the second approach is chosen. Thus, *fx* is compared to all the leaf nodes (i.e., *f3*, *f4*, and *f5*). Suppose that the similarity function produces the following values: *sim (fx, f3) = 0.41*, *sim (fx, f4) = 0.8* and *sim (fx, f5) = 0.07*, then the two files with higher similarity (i.e., *fx* and *f4*) are considered to be versions.

Similarity analysis between files is described in the next section.

# 4. SIMILARITY ANALYSIS

The similarity analysis task is responsible for generating a similarity value *sim* for each pair of files. Consider a set $Rs = \{x \in R: x >= 0 \wedge x <=1\}$ as the set of all real values in $\mathbb{R}$ in the interval [0,1]. The similarity function *s* receives a set of pair of files *P* and generates a similarity value in the *Rs* interval: $s: P \rightarrow Rs$. Let $G = \{(f1, f2), \ldots , (fn, fm)\}$ be a set of pair of files, the similarity value for all pairs can be computed by $\forall(fn, fm)\in G : s(fn,fm)$. The basic types of evolution that are considered for similarity analysis are:

1. **Content evolution:** in this type of evolution, the element content changes between versions, but the element structure keeps the same. The example in Figure 3 shows that the element *x* changes its content from version (a) to version (b). In terms of implementation, we consider that the element *x* is updated.
2. **Structure evolution:** in this type of evolution, the element structure changes between versions, but the element content keeps the same. The example in Figure 3 shows that the element *x* changes its structure from version (a) to version (c). In terms of implementation, we consider that the element *x* is removed and the elements *y* and *z* are added.
3. **Structure and content evolution:** in this type of evolution, both element content and structure changes between versions. The example in Figure 3 shows that the element *x* changes its structure and content from version (a) to version (d). In terms of implementation, we consider that the element *x* is removed and the elements *y* and *z* are added.

```
      (a)                (b)                (c)              (d)
<root>             <root>             <root>            <root>
  <x>A St, 7</x>     <x>B St, 8</x>     <y>A St</y>       <y>B St</y>
</root>            </root>              <z>7</z>          <z>8</z>
                                      </root>           </root>
```

**Figure 3. Different representations in XML document versions**

In the proposed mechanism, *structure* evolution and *structure and content* evolution are grouped together. Let us first discuss the content evolution in the next section.

## 4.1 Content Evolution

In order to evaluate the content similarity between files with content evolution, the following features are observed:

- ***Diff* results:** to assess the content similarity level between two files, we first consider the use of a *diff* algorithm. The *diff* algorithm outputs the differences between two files *f1* and *f2* in a delta representation, as a set of basic edit operations. By using a *diff* algorithm, differences between the files are detected. Analyzing the input files and the delta representation, we can determine the percentage of elements that have not changed in *f2*.

Suppose the files *f1* and *f2* shown in Figure 4. Since only the content has changed in these files, the number of elements keeps the same. Figure 4 shows that each file has 6 elements (i.e., the root element *employee* has 6 direct and indirect descendants).

```
<employee>                                  <employee>
     <name>Marcos</name>                         <name>Marcos</name>
     <hiringDt>10/10/03</hiringDt                <hiringDt>10/10/03</hiringDt>
     <job>engineer</job>                         <job>manager</job>
     <salary>3700</salary>                       <salary>4900</salary>
     <address>7 St</address>                     <address>7 St</address>
     <phone>65982541</phone>                     <phone>65982541</phone>
</employee>                                 </employee>
```

**Figure 4. XML files (*f1* and *f2*) with element content changes**

As Figure 5 shows, the content of the elements *salary* and *job* do not match in the second file. In other words, 67% of the original elements kept unchanged in the second file. The assumption is that the bigger percentage of unchanged elements, the larger chance the files are versions of the same document.

```
<delta> <Deleted update="yes"  pos="0:0:3:0">3700</Deleted>
        <Deleted update="yes" pos="0:0:2:0">engineer</Deleted>
        <Inserted update="yes"  pos="0:0:2:0">manager</Inserted>
        <Inserted update="yes" pos="0:0:3:0">4900</Inserted> </delta>
```

**Figure 5. *Diff* result for XML files (*f1* and *f2*)**

We are currently using the *XyDiff* algorithm [18], but the architecture allows changing to other *diff* implementations. *XyDiff* is very efficient in terms of speed and memory space; also, it considers, besides insertions, deletions and updates, a move operation on subtrees that is essential in the context of XML. According to [18], the complexity is no more than the expected $O(n*log(n))$ time. Regarding the algorithm quality, the computed changes are very close in size to the synthetic (perfect) changes.

- **Matched and unmatched elements:** the previous feature analyses the percentage of elements that are the same between files. On the other hand, another interesting feature is to look at the elements that could have changed completely between files or just some of its characters could have changed. In other words, we analyze the differences and similarities between elements that have changed. The more similar the respective unmatched elements, the larger is the chance that the files are versions of the same document.

In this paper, we consider the term *matched* to refer to an element that has the identical content in both files (for example, *name*); *unmatched*, otherwise (for example, *salary*). Let us take a look at the unmatched elements *salary* and *job*. Using a (combination of) string similarity function(s), we calculate a value that demonstrates how similar the unmatched elements are. Our goal is not to describe string similarity functions and quality results. An extensive analysis of similarity functions and threshold definition can be found in [15].

- **Element change relevance:** Another important feature to be considered for similarity analysis is the relevance of individual changes. Some domain concepts can change more frequently than others. Let us suppose that we have an *address* element. Two different addresses can easily refer to the same person; however, two different birthdates suggest that we are analyzing two different objects in the real world. In other words, the change relevance is differently weighted for different concepts. In our approach we use different weights, such as *high* (1), *medium* (0.5) and *low* (0). The average of weighted relevances is used to calculate file similarity. The smaller change relevance they present, the larger chance the files are versions of the same document.

Based on the *diff* results, matched and unmatched elements, and element change relevances, the content similarity function *simC* between two files *f1* and *f2* is defined as:

$$simC(f1,f2) = (w_1*F_1 + w_2*F_2 + w_3*F_3 + ... + w_n*F_n)$$

Where $w_n$ is a factor that weights the importance of a specific feature $F_n$. A factor may be positive or negative (if it influences the similarity growth or reduction, respectively). Considering $w_x$, $w_{x+1}, ...w_y$ as positive factors and $w_z$, $w_{z+1}, ...w_q$ as negative factors, we assume that $w_x + w_{x+1} + ... + w_y = 1$ and $-1 <= w_z + w_{z+1} + ... + w_q <= 0$.

In our approach, three features are combined to produce the following content evolution similarity function:

$$\mathrm{simC(f1,f2)} = w_1*P + w_2*S + w_3*R$$

Where: *P* is the percentage of matched elements, *S* is the mean similarity of the unmatched elements and *R* is the average of domain relevances of the unmatched elements. *P* and *S* factors ($w_1$ and $w_2$, respectively) are positive values (the greater these values, the more similar the files) and *R* factor ($w_3$) is a negative value (the smaller this value, the less relevance the change and the more similar the files). The factors ($w_1, w_2, \ldots, w_n$) must be defined based on the importance of the three features in specific applications/domains and recall/precision measures.

The intervals of the defined variables are defined as: $\{P|P \in [0,1]\}$, $\{S|S \in [0,1]\}$, $\{R|R \in [0,1]$. Analyzing the minimum and maximum values of *P*, *S* and *R*, and the sum restrictions for positive and negative factors, we conclude that the similarity function produces a value *simC* that ranges from -1 to 1, i.e., $\{\mathrm{simC}|\mathrm{simC} \in [-1,1]\}$.

To calculate *P*, we use a function *calcP* that returns the percentage of matched elements based on the *diff* result. *S* is calculated by using a (combination of) string similarity function(s) *(StrSim())* and it is defined as the average of the similarity values for the unmatched elements (*ue*). Consider that the number of unmatched elements is denoted by *t*. The function is detailed as follows:

$$\mathrm{simC(f1,f2)} = w_1*\mathrm{calcP(diff(f1,f2))} + w_2*\frac{\sum_{x=1}^{t}\mathrm{StrSim}(\mathrm{ue1}_x,\mathrm{ue2}_x)}{(t)} + w_3*\frac{\sum_{x=1}^{t}R(\mathrm{ue}_x)}{(t)}$$

Let us consider that *P*, *S* and *R* have the same importance for similarity analysis in a specific application (i.e., $w_1=0.5$, $w_2=0.5$ and $w_3=-0.5$). Figure 6 shows the distribution for the function similarity values. The similarity function values are not uniformly distributed. To uniformly distribute the values, we sort and map the *m* similarity function results into *n* classes. The mapping, represented in a transformation table, categorizes *m/n* members in each class. Since we have 100 different similarity values, this transformation generates *0.01*m* members in each class.

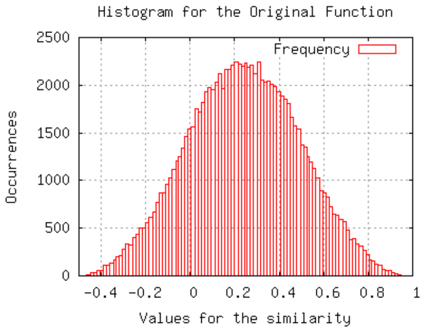

**Figure 6. Histogram for the original function**

Figure 7 shows the distribution of the mapped uniform transformation. We generated 1.000.000 values according to the original similarity function, using 0.5, 0.5 and -0.5 as the weight values, and grouped them into 100 classes. These classes were mapped to values $\in [0,1]$, in order to uniformly distribute the function values. To ensure that the mapping is correct, we generated 100.000 more values and mapped them to this table.

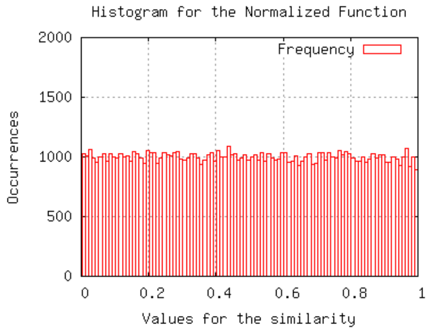

**Figure 7. Histogram for the normalized function**

The next section discusses the similarity analysis for structure and content evolution.

## 4.2 Structure and Content Evolution

In addition to *diff results* and *element change relevance* discussed in the last section, there is another important feature that must be considered, as presented below.

- **Added and removed elements:** using a *diff* algorithm, the differences between the files are detected. Analyzing the files and the *diff* results, we can observe the number of added elements and the number of removed elements between the first and the second file. Let *added* denote the new elements (for example, *address*) and let *deleted* denote the removed elements (for example, *job*). The concepts of *added* and *removed* are similar to the ideas presented in [31], which consider *plus*, *minus* and *common* elements for measuring similarity between a document and a DTD. As an example, consider two files, *f3* and *f4*, shown in Figure 8. Analyzing the files and the *diff* results (the *diff* result is not presented here), we can see that *f4* has added one element (*address*) and has removed two elements (*job* and *hiringDt*).

```
<employee>
    <name>Marcos</name>
    <hiringDt>10/10/03</hiringDt>
    <job>engineer</job>
    <salary>3700</salary>
    <phone>65982541</phone>
</employee>
```

```
<employee>
    <name>Marcos</name>
    <salary>4500</salary>
    <address>7 St</address>
    <phone>65982541</phone>
</employee>
```

**Figure 8. XML files (*f3* and *f4*) with element content and structure changes**

As *XyDiff* considers the *move* operation, the *added* concept refers only to new elements; movement actions are not identified as addition and deletion in our proposal.

For content and structure evolution, we redefine the concept *matched* used in the last section. Here we will use the term *matched* to refer to an element that has the same structure and content in both files (for example, *name* and *phone*). The term *unmatched* will still be used for denoting changes in the element content (for example, *salary*), since changes on the structure are classified as *removed* and *added* elements. Thus, the function used to compute the similarity including structure evolution is as follows:

$$\mathrm{simE(f3,f4)} = \mathrm{simC(f3,f4)} + w_4*A + w_5*D$$

Where: *simC* is the content similarity value, *A* is the percentage of added elements and *D* is the percentage of deleted elements. The weights for $A$ and $D$ ($\mathtt{w}_4$ and $\mathtt{w}_5$, respectively) are negative values (the smaller these values, the more similar the files).

For these, variables range over the following interval: $\{A|A \in [0,1]\}$, $\{D|D \in [0,1]\}$. Analyzing the minimum and maximum values of *simC*, A, *D*, and the sum restrictions for positive and negative factors, we conclude that the similarity function produces a value *simE* that $\in [-3,2]$. To calculate *A*, we use a function *calcA* that returns the percentage of added elements, based on the *diff* result. To calculate *D*, we use a function *calcD* that returns the percentage of removed elements, based on the *diff* result. Thus we obtain the similarity function is detailed as follows:

$$\mathtt{simE(f3,f4) = simC(f3,f4) +}$$
$$\mathtt{w_4*calcA(diff(f3,f4)) + w_5*calcD(diff(f3,f4))}$$

Similarly to the function for content evolution, the values of this function are also not uniformly distributed. The process detailed in the previous section is applied on the results to uniform these values. After measuring the similarity between files, the version detection task is performed. The version detection is done by the activity described in the next section.

## 5. CLASSIFICATION

The second task in the version detection mechanism is responsible for deciding if two files *f1* and *f2* are versions. A typical classification task receives as input a training set of tuples, each labeled with a class label. The output is a model (i.e., classifier) which assigns a class label to each tuple based on the other attributes. The model can be used to predict the class of new tuples, for which the class label is missing or unknown. The classification is defined as a supervised learning technique, since the mechanism uses training samples with known classes to classify new data. The tuples (i.e., samples) are partitioned in *training* set and *test* set.

The classification is also performed in two steps: training and test phases. The *training* step is responsible for building the model from the training set; the *test* step is responsible for checking the accuracy of the model using the testing set. The accuracy of the generated model can be measured by matching the test samples against the class predicted by the model. The accuracy rate is given by the percentage of test set samples correctly classified by the model.

In our proposal, the training set consists of a set of file pairs with feature values (i.e., *P, S, R* for content evolution or *P, S, R, A, D* for structure/content evolution), the similarity value and a class *k* (k | k ∈ {"version", "no version"}). The testing set also consists of the same structure, but the value of *k* is unknown. Thus, the function *c* computes the class *k* for each pair. Let *G = {(f1, f2), … , (fn, fm)}* be a set of pair of files, where each pair is associated with a *6-tuple* F={*P, S, R, A, D, sim*}. The classification value *k* for all pairs can be computed by $\forall(fn, fm)\in G : c(fn, fm)$.

In this work, we use *Naïve Bayesian* classification technique. A *Naïve Bayesian* classifier is a simple probabilistic classifier with strong independence assumptions [26]. This technique requires a small amount of training data to estimate the parameters (means and variances of the variables) necessary for classification. Because independent variables are assumed, only the variances of the variables for each class need to be determined. The *Naïve Bayesian* classifier assumes attribute independence ($P(x_1,\ldots,x_k|C) = P(x_1|C)*\ldots*P(x_k|C)$). If the *i-th* attribute is categorical, then $P(x_i|C)$ is estimated as the relative frequency of samples having value $x_i$ as *i-th* attribute in class *C*. If the *i-th* attribute is continuous, then $P(x_i|C)$ is estimated thru a *Gaussian* density function. In our approach, the attributes are categorical. The categories are defined in an interval, from 0 to 1. The *i-th* element is given by $f(x) = i * 0.01$. Therefore, there are 100 different categories. Experimental results on the classification task are presented in Section 6.

For measuring the quality of the *Naïve Bayesian* classifier, our approach calculates the accuracy rate by using recall and precision, metrics widely used in information retrieval [11]. The classical definition for recall is the proportion of relevant documents that are retrieved, out of all relevant documents available. Precision is defined as the proportion of retrieved and relevant documents to all the documents retrieved. Thus, let *A* be the set of file pairs that are truly versions and *B* be the set of file pairs that were detected as versions by the classifier. Let "|" be the cardinality of a set (i.e., the number of elements of the set). Then, the recall is defined as *Recall:* $|A\cap B|/|A|$, and the precision is defined as *Precision:* $|A\cap B|/|B|$. The $|A\cap B|$ expression is the number of versions that were correctly identified as versions by the classifier. In other words, the recall and precision are defined as:

*Recall: (n$^o$. of versions correctly detected)/(n$^o$. of existent versions)*

*Precision: (n$^o$. of versions correctly detected)/(n$^o$. of detected versions)*

Using the proposed similarity functions and the version detection mechanism based on *Naïve Bayesian* classifiers, several experiments were carried out. The quality of the proposed detection version mechanism is quite high and it is going to be discussed in the next section.

## 6. EXPERIMENTAL RESULTS

This section presents the results of various experiments that show that our approach produces very good results, both in terms of recall and precision measures. In order to assess the accuracy of the similarity function and the version detection mechanism based on *Naïve Bayesian* classification, we divided the experiments into two groups. The first group considers only content evolution. The second group considers content and structure evolution, as presented in Section 4.

For each type of evolution, the experiments were divided in four phases: data acquisition, data training, data testing, and result analysis. In the first phase, we acquire the data to be used as the training set. In the second phase, the classifier uses the acquired data and train on these data, in order to get a classification model. In the third phase, we apply a set of data to be tested by the classifier. Finally, the analysis phase measures the accuracy of the results, using recall and precision metrics.

The first group of experiments was carried out for files with content evolution, as described in the next section. The experiments were conducted on synthetic data.

### 6.1 Content Evolution

For these experiments, we consider the similarity function presented in Section 4.1. The experiments presented in this paper were intentionally based on simulated values (i.e., synthetic data), in order to assess the classifier scalability while analyzing the quality of the results. The experiments were carried out as follows:

- **Data Acquisition:** this phase is responsible for acquiring the necessary data set to be used as training data set for the classifier. Some activities were performed, as described.

1. We randomly generated 9000 values for the attributes (features) considered in the similarity function: *P*, *S* and *R*, where *P*, *S, R* $\in$ *[0,1]*. For these experiments, we set the weights in *0.5*, *0.5* e *-0.5*, respectively.

2. We applied the similarity function *simC* in these values, obtaining similarity values for 9000 pairs of files. In other words, we calculated 9000 similarity values for a set of pair of files. These samples were generated equally distributed between versions and non-versions (around 50% each one).

3. Finally, the similarity values were uniformly distributed, using the mapping uniform transformation described in Section 4.1.

- **Data Training:** this phase is responsible for providing the acquired data to the classifier. The classifier uses this data set to generate a model that is later used in the test phase. Performing the activities 1, 2 and 3, above described, the training set is generated and used by the *Naïve Bayesian* classifier. The training set is stored in a database, following the structure:

*trainingTable (pairID, P, S, R, simCNormalized, class)*

Where: *pairID* is the identifier of a pair of files; *P*, *S* and *D* are the features considered in the similarity function; *simCNormalized* is the similarity value after the mapping transformation; and *class* is the category of the each pair of files that was compared (e.g. version or non-version).

From the training set, we generated the probability for each feature value for the expected classes. These probabilities are used by the classifier and they were stored using the structure:

*featureProbability (<u>class, feature, value</u>, probability)*

Where: *class* is the category (e.g. version or non-version), *feature* is one of the features considered in the similarity function (i.e., *P, S* or *R*), *value* is the possible value for a feature (from 0 to 1, varying in 0.01 scale) and *probability* is the probability that a certain value appear in a feature for a specific class (e.g. 0.000648). The probability is defined as described in Section 5 and it is computed by SQL queries over the *trainingTable* relation.

- **Data Testing:** this phase is responsible for testing several sets of data in the model generated by the classifier in the training phase. Some activities were carried out, as described. In order to evaluate the result quality in different sizes for the testing data sets, we have chosen a large, a medium and a smaller group: 3 data sets with 1000 samples each, 3 data sets with 500 samples each and 3 data sets with 100 samples each. The generation of these data sets followed the same steps described in data acquisition phase. The testing set was stored in a database, according to the structure:

*testingTable (pairID, P, S, R, simCNormalized)*

For each pair of files (*pairID*), we compute the probability of the value *xi* appears in the *i-th* attribute (i.e., feature) in the class *C* (i.e., version and non-version). For example, let us consider the following tuple in *testingTable* relation:

*testingTable (1, 0.15, 0.13, 0.55, 0.07)*

The calculated probabilities are shown below:

$$P(0.15,0.13,0.55|non\text{-}version) = 0.0000018102$$

$$P(0.15,0.13,0.55|version) = 0.0000000709$$

Since the first probability is larger than the second probability, the pair of files is classified as *non-version*. We apply the classifier for each set. The classifier returns the category for each pair of files (i.e., version or non-version).

- **Data Analysis:** this phase is responsible for measuring the accuracy of the results in terms of recall and precision. Several experiments were performed using the proposed similarity function and the classifier. The experiments are as follows: *e1*, *e2*, *e3*: 1000 pairs of files; *e4*, *e5*, *e6*: 100 pairs of files; *e7*, *e8*, *e9*: 500 pairs of files. The results are presented in Figure 9. As Figure 9 shows, the mean recall and precision rates were, respectively, 92.13% and 92.49%. Even the worst cases for recall (88.89% in experiment 4) and precision (87.27% in experiment 4) were still good. In other words, the classifier correctly detected over 92% of the existent versions and over 92% of the detected versions were correctly classified.

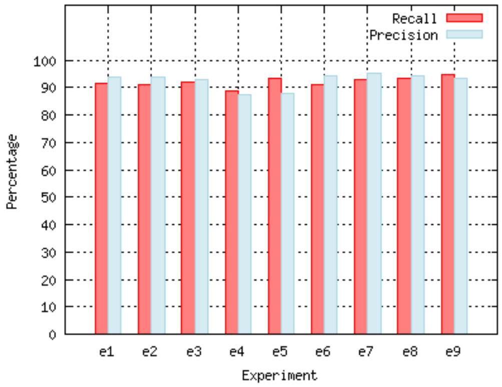

**Figure 9. Recall and precision results for group 1**

For the results presented in Figure 9, we considered the same proportion (around 50%) of versions and non-versions for the training and testing sets. In order to evaluate how the classifier behaves with different proportions of versions and non-versions in the training and testing sets, we made other experiments. We used the same data configuration described above. However, in these experiments, we considered that 80% of the training and testing sets were represented by versions. The results of these experiments are shown in Figure 10.

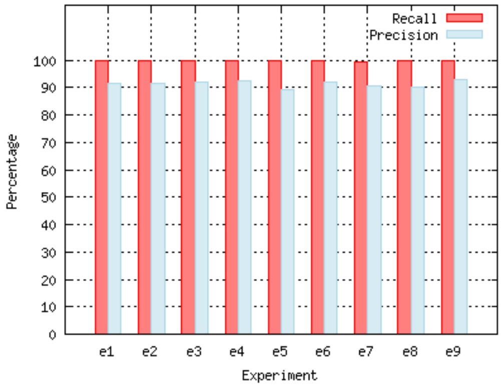

**Figure 10. Recall and precision results for group 2**

As Figure 10 shows, the recall rates were better in this case. The mean recall and precision rates were, respectively, 99.83% and 91.35%. Even the worst cases for recall (99.28% in experiment 7) and precision (89.13% in experiment 5) were still very good. In other words, the classifier correctly detected over 99% of the existent versions and over 91% of the detected versions were correctly classified.

## 6.2 Structure and Content Evolution

For these experiments, we consider the similarity function presented in Section 4.2. Similar to the previous section, the experiments were carried out as follows:

- **Data Acquisition:** similar to the activity presented in Section 7.1, some activities were executed, as described.

  1. We randomly generated 9000 values for the attributes (features) considered in the similarity function: *P, S, R, A* and *D*, where *P*, *S*, *R*, *D* and *A* ∊ *[0,1]*. For these experiments, we set the weights in *0.5*, *0.5*, *-0.33*, *-0.33* and *-0.33*, respectively.
  2. We applied the similarity function *simE* in these values, obtaining similarity values for 9000 pairs of files. These samples were also generated equally distributed between versions and non-versions (around 50% each one).
  3. Finally, the similarity values were uniformly distributed, using the mapping uniform transformation described in Section 4.1.

- **Data Training:** this activity is similar to the *data training* activity presented in Section 7.1. The training set is stored in a database, following the structure:

  *trainingTable (pairID, P, S, R, A, D, simENormalized, class)*

  Also, the table for the feature probabilities was generated.

- **Data Testing:** similarly to the *data testing* activity presented in Section 7.1, some activities were performed, as described. Again, in order to evaluate the result quality in different sizes for the testing data sets, we have chosen a large, a medium and a smaller group: 3 data sets with 1000 samples each, 3 data sets with 500 samples each and 3 data sets with 100 samples each. The generation of these data sets followed the same steps described in data acquisition phase. The test set was stored in a database, following the structure:

  *testingTable (pairID, P, S, R, A, D, simENormalized)*

  We apply the classifier for each set. The classifier returns the category for each pair of files (i.e., version or no version).

- **Data Analysis:** similarly to the *data analysis* activity presented in Section 7.1, several experiments were performed using the proposed similarity function and the classifier. The experiments are as follows: *e1*, *e2*, *e3*: 1000 pairs of files; *e4*, *e5*, *e6*: 100 pairs of files; *e7*, *e8*, *e9*: 500 pairs of files. The results are presented in Figure 11. As Figure 11 shows, the mean recall and precision rates were, respectively, 91.11% and 91.05%. Even the worst cases for recall (87.23% in experiment 5) and precision (90.04% in experiments 1 and 7) were still good. In other words, the classifier correctly detected over 91% of the existent versions and over 91% of the detected versions were correctly classified.

All the presented results (groups 1 to 3) were executed over a training set with 9000 samples. We also would like to evaluate how the recall and precision values behave using a different size for the training set. So, we ran again the experiments above (*e1* to *e9*) using a training set with only 3000 samples (group 4). The same configuration was used, i.e., equal distribution between versions and non-versions. Figure 12 shows that the results are not as good as previous results, but still good.

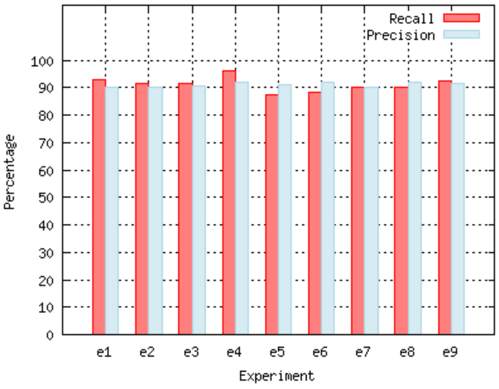

**Figure 11. Recall and precision results for group 3**

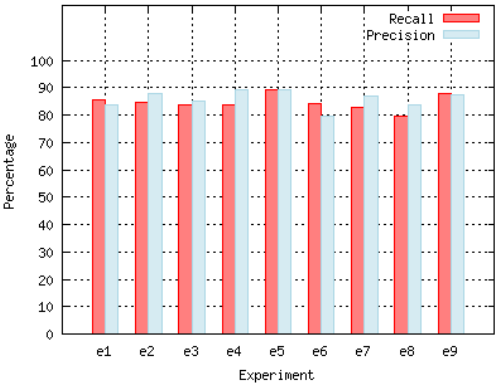

**Figure 12. Recall and precision results for group 4**

The mean recall and precision rates were, respectively, 84.66% and 85.87%. Even the worst cases for recall (79.66% in experiment 8) and precision (79.63% in experiment 6) were reasonably good. In other words, the classifier correctly detected over 84% of the existent versions and over 85% of the detected versions were correctly classified. The results in recall and precision rates (lower than in previous experiments) lead us to conclude that the classifier produces better results if the training set has more samples. Considering the results presented in Figure 11 and Figure 12, the recall and precision rates were, respectively, 7.07% and 5.68% higher with a larger training set.

Finally, all the experiments (considering groups 1 to 4) have shown the accuracy of the proposed mechanism. We used 2 different training sets with some configuration changes (e.g. the proportion of versions and non-versions, and size), resulting in 4 training sets. Also the testing sets were diverse: 27 different testing sets, whose sizes varied from 100 to 1000 samples. The average recall rate considering the first 3 groups of experiments (groups 1 to 3) was 94.35% and the average precision rate was 91.63%. Including the forth group in this analysis (with worse rates because of the smaller size of the training set), the average rates were still good: 91.93% for recall and 90.19% for precision rates.

As presented in Section 6, the experiments were executed on synthetic data, where we have randomly simulated float values for the similarity function features (e.g., *P*, *S*, and *R*). We are currently implementing the similarity functions in order to run those experiments on the features acquired from XML documents crawled on the Web. The good results for evaluating recall and precision obtained in this paper are expected to be maintained in real scenarios. The new results are going to be presented in the conference.

The next section shows how the version detection allows performing very interesting metadata management.

## 7. METADATA MANAGEMENT

Finally, the last activity performed by the version detection mechanism is the metadata management. This metadata aims to describe information related to the detected versions. However, the metadata model also supports the representation of non-versioned files (i.e., files that do not match as version of existent files. Non-versioned files can be represented in the metadata model as roots for future lineages. The metadata information is structured as shown in Figure 13.

```
<!ELEMENT metadata (node*)>
<!ELEMENT node (features*, node*)>
<!ATTLIST node file CDATA #REQUIRED>
<!ELEMENT features EMPTY>
<!ATTLIST features  P CDATA #REQUIRED    S CDATA #REQUIRED
                    R CDATA #REQUIRED    sim CDATA #REQUIRED
                    D CDATA #IMPLIED     A CDATA #IMPLIED>
```

**Figure 13. Metadata structure**

By using the metadata model, the logical structure of linear and branched versions can be represented. Figure 14 describes the metadata XML representation for Figure 1.

```
<metadata>
  <node file="f1">
      <node file="f2">
          <features P="0.7" S="0.9" R="0.1" sim="0.86"/>
          <node file="f3">
              <features P="0.8" S="0.9" R="0.1" sim="0.97"/>
          </node>
    </node>
  </node>
  <node file="f4">
      <node file="f5">
          <features P="0.6" S="0.7" R="0.1" sim="0.79"/>
          <node file="fx">
              <features P="0.7" S="0.8" R="0.1" sim="0.89"/>
          </node>
      </node>
  </node>
</metadata>
```

**Figure 14. Metadata example for linear versioning**

The lineage of versions is represented as a *parent-children* relationship. By traversing the tree following the relationships, the entire lineage of a specific version can be retrieved. For example, there are two groups of linear versions in Figure 1. The first group is formed by the files *f1, f2* and *f3*. In Figure 14, this group is represented by following the *parent-children* relationship:

*/metadata/node[@file="f1"]/node[@file="f2"]/node*

Another example is depicted in Figure 2, which contains three branches of versions. The third branch is formed by the files *f1, f4* and *fx*. In Figure 15, this branch is represented by the following *parent-children* relationship:

*/metadata/node[@file="f1"]/node[@file="f4"]/node*

```
<metadata>
    <node file="f1">
        <node file="f2">
            <features P="0.7" S="0.9" R="0.1" sim="0.9"/>
            <node file="f5">
                <features P="0.8" S="0.9" R="0.1" sim="0.94"/>
            </node>
        </node>
        <node file="f3">
            <features P="0.6" S="0.7" R="0.1" sim="0.7"/>
        </node>
        <node file="f4">
            <features P="0.7" S="0.8" R="0.1" sim="0.81"/>
            <node file="fx">
                <features P="0.7" S="0.8" R="0.2" sim="0.8"/>
            </node>
        </node>
    </node>
</metadata>
```

**Figure 15. Metadata example for branched versioning**

The file identifiers (denoted by the attribute *file* in the metadata representation) are generated by using hash functions, a frequently used approach for file identification. We are currently using *MD5* (*Message-Digest Algorithm* 5), a cryptographic hash function with a 128-bit hash value. For simplicity, the hash-based identifiers are not presented in the examples above. The hash function result is substituted by a simpler notation (e.g., the hash result *ece50ed4d6d48dac839bfe8fa719fcff* is denoted by *f5*).

## 8. CONCLUDING REMARKS

This paper focused on version detection of XML documents. The significance of such problem is quite evident in many scenarios, such as plagiarism detection, Web page ranking, software clone identification, assuring link permanence in Web documents, and enhancing search in peer-to-peer systems. In this paper, we defined a similarity function that considers several characteristics that must be taken into account when considering a version. The function is not restricted to a specific application and it can be adapted to consider other relevant similarity features for specific scenarios. Moreover, each feature can be differently weighted, which turns our proposal into a more flexible approach.

A detection mechanism based on classification techniques is also presented. By using a classifier, the hard task of defining the threshold value is eliminated. In our proposal, we used the *Naïve Bayesian* classification technique, a simple probabilistic classifier. An advantage of this classification technique is the assumption of independent variables; thus, only the variances of the variables for each class need to be determined.

The version detection problem is not a new issue. Neither is the use of *Naïve Bayesian* classifiers to categorize documents. However, the use of this technique for solving the mentioned problem requires the definition of variables (i.e., attributes or features) that describe each category, which usually is a hard domain-dependent task for version detection. In the response of these requirements, in this paper we proposed a similarity function and used a classification technique, which provides a very accurate solution for an aged problem. The experiments produced very good results, over than 90% for recall and precision rates (even considering smaller training sets). In the absence of other similar approaches for version detection, the values of precision and recall cannot be compared. However, over than 90% rate alone is a good indicative of precision and recall.

As future work, we are going to apply the mining method to classify the documents using testing sets derived from a particular application domain. Therefore, we will also optimize the similarity function, the coefficients used (i.e., weights) and the mining method. The good results for evaluating recall and precision obtained so far are expected to be maintained in different scenarios. Further enhancement of metadata management is also planned.

# 9. ACKNOWLEDGMENTS

This work has been partially supported by CNPq under grant No. 142396/2004-4 and Capes under grant No. 1451/06-5 for Deise de Brum Saccol. It is also supported by CNPq under grant No. 481516/2004-2, Fapergs under grant No. 0412264 and Digitex (CNPq No. 550.845/2005-4) for Renata de Matos Galante.

# 10. REFERENCES

[1] Westfechtel, B., Munch, B. P., and Conradi, R. A Layered Architecture for Uniform Version Management. IEEE Trans. Software Eng., 27(12):1111–1133, 2001.

[2] Chien, S-Y., Tsotras, V. J., Zaniolo, C. (2001). XML Document Versioning. SIGMOD Records, Vol. 30 Number 3, Sept.

[3] Ronnau, S.; Scheffczyk, J. e Borghoff, U.M.. Towards XML Version Control of Office Documents. DocEng '05: Proc. of the 2005 ACM symposium on Document engineering, ACM Press, 10-19, 2005.

[4] Katz, R. e Chang, E.. Managing Change in a Computer-Aided Design Database. Proceedings of VLDB Conference, 1987.

[5] Schleimer, S., Wilkerson, D., Aiken, A.. Winnowing: Local Algorithms for Document Fingerprinting. Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, San Diego, CA, p. 76-85, 2003.

[6] Chen, X., Francia, B., Li, M., McKinnon, B., Seker, A.. Shared information and program plagiarism detection. IEEE Transactions on Information Theory, v. 50, n. 7, p-1545-1551, 2004.

[7] Baeza-Yates, R., Castillo, C.. Relating Web Characteristics with Link based Web Page Ranking. Proc. of the 8th Intl. Symposium on String Processing and Information Retrieval, 2001.

[8] Ducasse, S., Niertrasz, O., Rieger, M.. On the effectiveness of clone detection by string matching. Journal of Software Maintenance and Evolution: Research and Practice, v. 18, n. 1, p. 37-58, 2006.

[9] Navarro, G.: A guided tour to approximate string matching. ACM Computing Surveys 33, 31–88, 2001.

[10] Guth, G.J.: Surname spellings and computerized record linkage. Historical Methods Newsletter 10, 10–19, 1976.

[11] Baeza-Yates, R.A. e Ribeiro-Neto, B.A.. Modern Information Retrieval. ACM Press / Addison-Wesley, 1999.

[12] Flesca, S. e Pugliese, A.. Fast Detection of XML Structural Similarity. IEEE Transactions on Knowledge and Data Engineering, 17, 160-175, 2005.

[13] Nierman, A. e Jagadish, H.V.. Evaluating Structural Similarity in XML Documents. Proc. of the 5th Intl. Workshop on the Web and Databases (WebDB 2002), 2002.

[14] Dorneles, C. F. ; Heuser, C. A. ; Lima, A. E. N.; Silva, A. S.; Moura, E. S. . Measuring similarity between collections of values. In: Proc. of the 6th ACM Intl. Workshop on Web Information and Data Management (WIDM), Washington, DC, 2004. p. 56-63.

[15] Silva, R. ; Stasiu, R. K. ; Orengo, V. M. ; Heuser, C. A. . Measuring quality of similarity functions in approximate data matching. Journal of Informetrics, v. 1, p. 4, 2007.

[16] Stasiu, R. K. ; Heuser, C. A. ; Silva, R. . Estimating Recall and Precision for vague queries in Databases. In: 17th International Conference Advanced Information Systems Engineering (CAISE), Porto, Portugal, 2005. v. 3520. p. 187-200.

[17] Saccol, D.B., Edelweiss, N., Galante, R.M.. Detecting, Managing and Querying Replicas and Versions in a Peer-to-Peer Environment. In: 1st IEEE TCSC Doctoral Symposium, in conjunction with the 7th IEEE Intl. Symposium on Cluster Computing and the Grid, Rio de Janeiro, 2007.

[18] Cobena, G., Abiteboul, S. and Marian, A.. Detecting Changes in XML Documents. Proc. of 18th Intl. Conf. on Data Engineering, 41-52, 2002.

[19] Wang, Y., DeWitt, D. J., Cai, J. X-Diff: An Effective Change Detection Algorithm for XML Documents. Intl. Conf. on Data Engineering, 519-530, 2003.

[20] Chawathe, S. S.. Comparing Hierarchical Data in External Memory. Proc. of the 25th Intl. Conf. on Very Large Data Bases, Morgan Kaufmann Publishers Inc., 90-101, 1999.

[21] Wan, X. and Yang, J.. Using Proportional Transportation Similarity with Learned Element Semantics for XML Document Clustering. WWW '06: Proc. of the 15th Intl. Conf. on World Wide Web, ACM Press, 961-962, 2006.

[22] Cohen, W.W., Ravikumar, P., Fienberg, S.: A comparison of string distance metrics for name-matching tasks. In: Proc. of IJCAI-03 Workshop on Information Integration on the Web, Acapulco, Mexico, Morgan Kaufmann, 73–78, 2003.

[23] Schallehn, E., Sattler, K.U., Saake, G.: Efficient similarity-based operations for data integration. Data Knowl. Eng. 48 (2004) 361–387.

[24] Bilenko, M.; Mooney, R.; Cohen, W.; Ravikumar, P.; Fienberg, S. Adaptive Name Matching in Information Integration. IEEE Intelligent Systems, [S.l.], v.18, n.5, p.16–23, September/October 2003.

[25] Sarawagi, S.; Bhamidipaty, A. Interactive deduplication using active learning. In: International Conference on Knowledge Discovery and Data Mining, ACM SIGKDD, 8. New York, NY, USA. Proc. New York:ACM Press, 2002. p.269–278, 2002.

[26] Langley, P., Iba, W., & Thompson, K. An analysis of Bayesian classifiers. Proc. of the 10th National Conference on Artificial Intelligence (pp. 223-228). San Jose, CA: AAAI Press, 1992.

[27] Wang. Y., Hodges, J., Tang, B.; Classification of Web Documents Using a Naive Bayes Method. Proc. of the 15th IEEE Intl. Conf. on Tools with Artificial Intelligence. IEEE Computer Society Washington, DC, USA, 2003.

[28] Pon, R.K., Cárdenas, A.F., Buttler, D., Critchlow, T.. iScore: Measuring the Interestingness of Articles in a Limited User Environment. In: IEEE Symposium on Computational Intelligence and Data Mining, Honolulu, HI, 2007.

[29] Chien, S.-Y., Tsotras, V.J., Zaniolo, C.; Efficient schemes for managing multiversion XML Documents, The VLDB Journal, Dec. 2002.

[30] Vagena, Z., Moro, M.M., Tsotras, V.J.; Supporting Branched Versions on XML documents. In: 14th International Workshop on Research Issues on Data Engineering, held with 20th Intl. Conf. on Data Engineering (ICDE), Boston, USA, 2004.

[31] Bertino, E., Guerrini G., Mesiti, M.. A Matching Algorithm for Measuring the Structural Similarity between a XML Document and a DTD and its Applications. Information Systems, v. 29, n. 1, Special issue on web data integration, p. 23-46, 2004.