Hydra vs. PXL
–
An evaluation of two approaches towards the classification of semi-structured data

Stefan Betzmeir

April 17, 2012

Contents

1 Introduction

Semi-structured data, especially in the form of XML documents, has become the de-facto standard for data exchange as monolithic all-purpose applications gave way to a network of service providers and service consumers, connected via the world wide web.

Apart from acting as an interface for applications today, XML will help to shape the world wide web of the future into the semantic web, in which ontologies and other kind of meta-data encode the semantics behind data, to make it machine-comprehensible beyond the symbolic level. The reason why XML lends itself for so many different tasks, is that by representing information as a tree, the structure itself encodes a lot of information like context and connectivity.

Due to its heavy usage, XML data offers great potential for data mining approaches to discover previously unknown correlations and patterns. Although research in data mining and information retrieval has come a long way, most of the techniques used today are based on a feature based approach for querying and comparing semi-structured data. But by processing semi-structured data in a feature based way, we lose valuable information implicitly stored in the structure of the data.

In this report, we will present the approach of *Hydra* by Bader (2009) towards automatic classification of software incidents in the domain of SAP Business ByDesign, which had been reimplemented and adapted into the general-purpose library *PXL* for the classification of semi-structured (i.e. tree based) data by Betzmeir (2010). We will focus on the findings made during a project, in which we compared the performance of *Hydra* and *PXL*.

2 SAP Business ByDesign

SAP is offering more and more products for small and medium sized companies. As these companies usually do not have key-account users offering first-level-support to other users, they rely on the support provided by SAP. SAP is therefore confronted with a rising demand for expensive help-desk support services. Consequently SAP seeks to automate the process of finding the cause of a certain problem and delegating it to the right persons.

2.1 SAP's Help Desk Workflow

In case of the software SAP Business ByDesign, the help desk support is based on an incident, which is a snapshot of the software's state at the time an error occurred. The state is represented as a tree structure consisting of a hierarchy of application objects and is stored in an XML file along with the users textual description of the error, to provide the support staff with context information concerning the incident.

Said XML file is being examined by a help desk expert, to determine the cause for the error in order to give the user advice based on experience with similar cases. If the problem cannot be solved this way, the incident can still provide valuable information to narrow down which part of the software is responsible for the malfunction, so that the problem can be delegated to the developers who are in charge of the flawed module. Eventually the incident will be classified and archived for future queries.

2.2 Automated incident classification

This workflow lends itself to a *Case Based Reasoning (CBR)* approach, where an incoming incident is being automatically classified based on the similarity to several clusters of incidents belonging to the same class, respectively. That means, that the incoming incident has to be compared to each of the incidents in the database. As the set of pre-classified incidents grows, the classification precision increases, but at the same time the classification process also takes more time. To cut down on the runtime, while allowing the database of incidents to grow, the *Cognitive Systems Group* suggested a classification based on a prototype for each cluster, which represents the structure and

features of the cluster members in an aggregated form. The basic idea was to count the matching nodes of each incident in a cluster and to weigh each node in the prototype according its frequency in the cluster. Therefore such a prototype can be described as the weighed set union of the incidents of a cluster. To compare an incident to a prototype, again the matching nodes of incident and prototype have to be determined and evaluated depending on their weight in the prototype.

3 Requirements Analysis

In the last chapter we presented a concrete use case for prototyping and classification of tree structures. Although prototyping and classification are completely different tasks, both rely on a matching strategy to determine the corresponding parts between two or more trees. In this chapter we will show how the properties of a tree model influence the structure the tree instances derived from it and how this in turn influences the matching. Based on these findings, we will identify requirements towards matching strategies. Finally we will present a generic outline for both prototyping and classification, which illustrates the central aspect of matching.

3.1 Tree-Model Properties

Let us assume a market research institute collects data about companies and stores it in a semi-structured database, e.g. a set of XML files. We will see, that data can be represented in various ways, which require different approaches towards the comparison of trees, independent of the chosen data-mining technique. We will define a tree-model, which determines the structure of our data. In an XML context this would be done using DTD or a XML Schema Definition (XSD).

A tree-model can be considered a set of rules for the construction of trees, which can be expressed e.g. as a formal grammar. For the sake of simplicity, we assume that every nonterminal (production rule), except the start symbol σ, implicitly leads to the creation of a node, with a label containing the name of said nonterminal.

3.1.1 Static Structure

In our first example we would like to store the name of the company, the name of its CEO, as well as the heads of the departments Purchasing, Production and Sales. One possible model which suits our needs might look like this:

$\sigma \rightarrow$ **COMPANY**
COMPANY $\rightarrow$ **CNAME CEO PURCHASING**
PRODUCTION SALES
CNAME $\rightarrow$ [a-Z]+
CEO $\rightarrow$ [a-Z]+
PURCHASING $\rightarrow$ [a-Z]+
PRODUCTION $\rightarrow$ [a-Z]+
SALES $\rightarrow$ [a-Z]+

Figure 3.1: Model 1: No quantifiers

Now we can create an instance of that model like, the one depicted in figure 3.2.

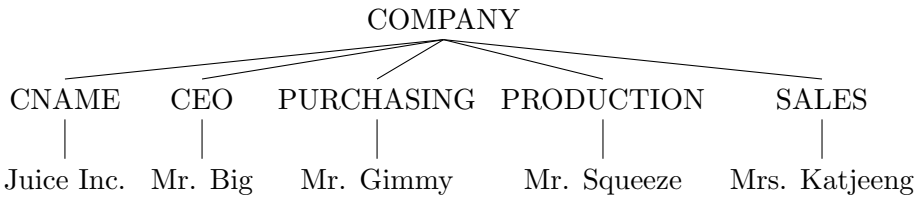


Figure 3.2: An instance of model 1

As the model in figure 3.1 has no quantifiers, all its instances will share the same structure. In this case, the tree structure can be considered a constant, in a way that instead of using a tree structure, we could also have used a simple vector representation, without losing any information.

3.1.2 Dynamic Structure

Although semi-structured databases are often being used for statically structured data, like in the example above, their strength lies in storing data which can't be represented in a tabular manner.

Optional Production Rules

Let us assume we also want to store data about companies, which lack one or the other department. Therefore we introduce the quantifier "?", allowing for zero or one occurrences of a nonterminal (i.e. inner node) or terminal (leaf node).

$\sigma \rightarrow$ **COMPANY**
COMPANY $\rightarrow$ **CNAME CEO PURCHASING? PRODUCTION? SALES?**
CNAME $\rightarrow$ [a-Z]+
CEO $\rightarrow$ [a-Z]+
PURCHASING $\rightarrow$ [a-Z]+
PRODUCTION $\rightarrow$ [a-Z]+
SALES $\rightarrow$ [a-Z]+

Figure 3.3: Model 2: Optional departments

Instances of this model could still be translated into a vector representation, but this would require an intermediate step to check which nodes are missing.

Repetition of Production Rules

Now we will extend out model to fully utilize the benefits of a semi-structured data-representation. We we might encounter companies which have additional departments like "Research" or "Customer Service", so we will just define a node *DEPT*, which has a child node containing the department name. Each company can now have zero up to an arbitrary number ($*$) of departments. The head of department is now represented by the node *HDEPT* and his/her name is being stored as its child node.

$\sigma \rightarrow$ **COMPANY**
COMPANY $\rightarrow$ **CNAME CEO DEPT***
CNAME $\rightarrow$ [a-Z]+
CEO $\rightarrow$ [a-Z]+
DEPT $\rightarrow$ [a-Z]+ **HDEPT**
HDEPT $\rightarrow$ [a-Z]+

Figure 3.4: Model 3: Allows for unlimited departments

If we want to store the information of the tree depicted in figure 3.2 the resulting tree would look like this:

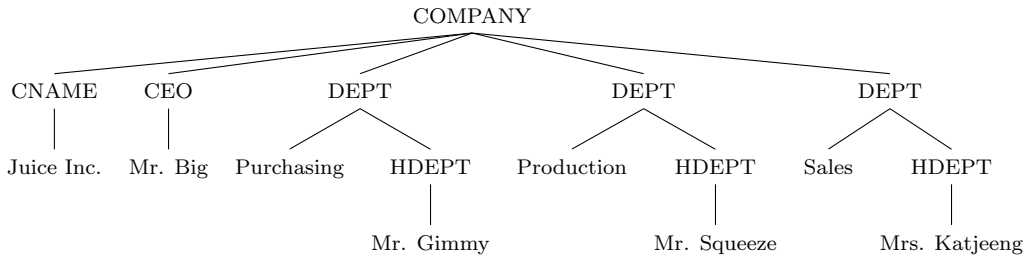


Figure 3.5: Tree A: An instance of model 3

Now that we can create trees with greater structural variance, the problem of comparing two trees also gets more complex, because now we have to find a way of determining which *DEPT* node from tree A to compare with which node from tree B.

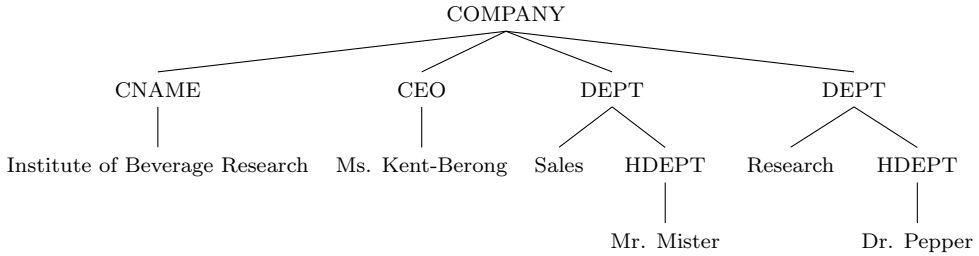


Figure 3.6: Tree B: Second instance of model 3

To illustrate that thought, we consider the data of a second company. If we compare tree A (figure 3.5) and tree B (figure 3.6), we could not decide at the level of the *DEPT* node, which department of tree A should be compared with which department of tree B. It is obvious, that we have to consider the whole subtree starting at each *DEPT* node now. If we do so, we find out that trees A and B can be considered similar to a certain degree, as both feature two *DEPT* nodes with partially matching subtrees.

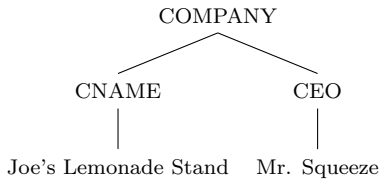


Figure 3.7: Tree C: Third instance of model 3

Due to bad working conditions, one of the department heads of Juice Inc. now decides to secretly start his own one-man start-up business. If we compare tree C to the trees A and B, we must conclude that tree C is as similar to tree A as it is to tree B, because in both cases only the nodes *COMPANY*, *CNAME* and *CEO* match. The fact that both tree A and tree C contain the leaf node "Mr. Squeeze" has no meaning, because our model differentiates between a CEO and a department head.

Reuse of Production Rules

Up till now, all nonterminals/inner nodes we defined were bound to a specific context i.e. type of parent node. Now we will loosen that restriction by generalizing *CEO* and *HDEPT* to *HEAD*.

$\sigma \rightarrow$ **COMPANY**
COMPANY $\rightarrow$ **CNAME HEAD DEPT***
CNAME $\rightarrow$ [a-Z]+
DEPT $\rightarrow$ [a-Z]+ **HEAD**
HEAD $\rightarrow$ [a-Z]+

Figure 3.8: Model 4: Reuse of nodes

We can do that, because the information about the rank of an employee is implicitly encoded in the tree-structure: The CEO will always be stored right below the *COMPANY* node and the department heads will be stored right below an *DEPT* node. We can transform tree A, B and C into tree A', B' and C' without losing information.

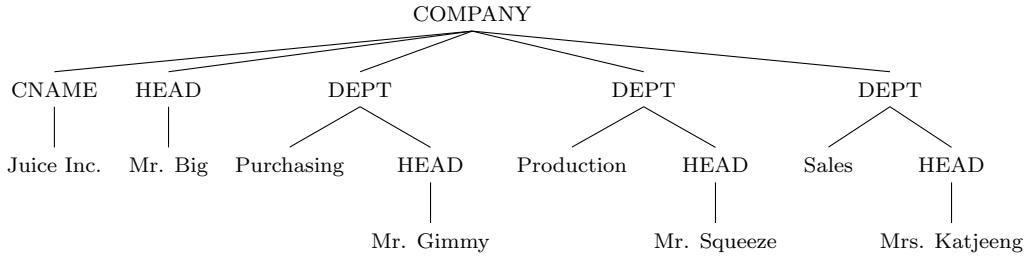


Figure 3.9: Tree A': An instance of model 4

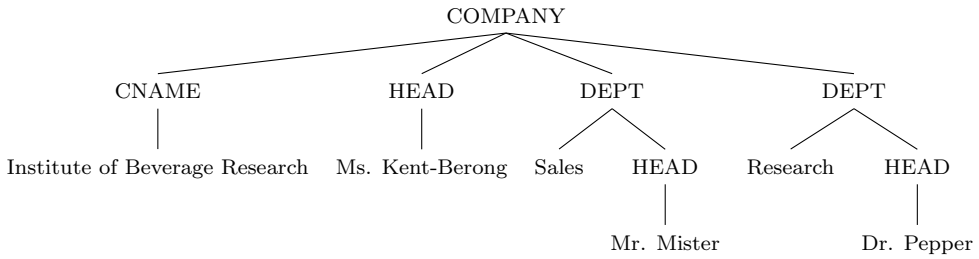


Figure 3.10: Tree B': Second instance of model 4

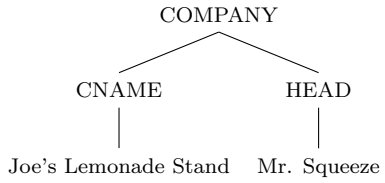


Figure 3.11: Tree C': Third instance of model 4

By generalizing the production rules *CEO* and *HDEPT* to *HEAD*, we have created a superclass and it is now semantically correct to compare a *HEAD* node in the context of a *COMPANY* node to a *HEAD* node in the context of a *DEPT* node.

Now the similarity of the trees A' and C' is greater than that of tree B' and tree C', because now we would compare the *HEAD* node in the *DEPT* context (namely the production department) from tree A' to the *HEAD* node in the *COMPANY* context of tree C'.

3.2 Matching

As we can see from our little example, the prerequisite to compare two datasets is to find a mapping between their elements. Now we can derive some fundamental requirements towards matching strategies.

3.2.1 Position Based

If we compare two vectors, we assume that that value 1 of vector 1 is of the same type and that it has the same semantics as value 1 of vector 2. So matching single features of

datasets with static structure, be they vectors or trees, is trivial and therefore usually done implicitly based on the order/position of elements.

3.2.2 Dynamic Tree-Structures

If the data model allows a dynamic data structure, the matching cannot be guided by a predefined order anymore, but must incorporate information gathered from the model (*Model Based Matching*) and/or from the tree instances (*Instance Based Matching*).

Horizontal Matching

Whenever the model allows repetition of nodes, *horizontal matching* must decide which node instance from tree a to match with which node instance from tree b.

Vertical Matching

If node definitions are being reused in different contexts, i.e. nodes with the same labels can have parents with different labels, or a production rule leads to direct or indirect recursion, *horizontal matching* decides on which level of the tree-hierarchy of tree b to look for a matching node for a node of tree a.

Ambiguity and Best Match

When matching two trees, usually there are more than one possible configuration for matching/associating the nodes of the trees. Therefore a matching strategy must make assumptions about how to determine the *best match*.

3.3 Generic Outline

By identifying matching as a preprocessing step for both prototyping and classification (or tree comparison in general), the process of creating the prototype boils down to simply counting/weighing the matching parts of the trees, while classification can bee seen as the process of calculating the difference of an incoming tree to each prototype by, again, counting the matching parts between tree and prototype and weighing them according to how often they occurred in the cluster represented by the prototype.

3.3.1 Prototyping

In figure 3.12 we can see that at the beginning of the generic prototyping pipeline a matching step is necessary to determine which nodes from the trees of each cluster correspond. These corresponding nodes are then examined and incorporated into the prototype during the prototyping phase. The result is a prototype for each cluster.

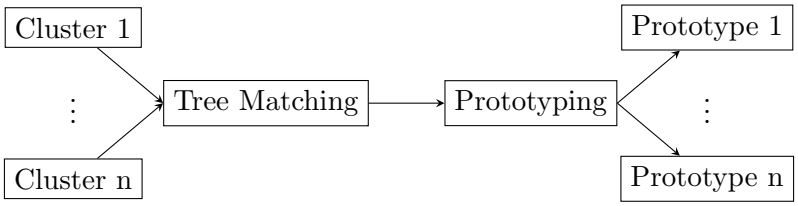


Figure 3.12: The generic prototyping pipeline

3.3.2 Classification

Classification based on precalculated prototypes can be alternatively described as finding the most similar prototype to an incoming tree. Again, the first step is to match the nodes of the incoming tree to the nodes of a prototype. In the classification phase, the corresponding parts are being evaluated and a value representing the similarity of the tree to the prototype is being calculated. This is being repeated for each prototype. The whole generic classification pipeline is depicted in figure 3.13.

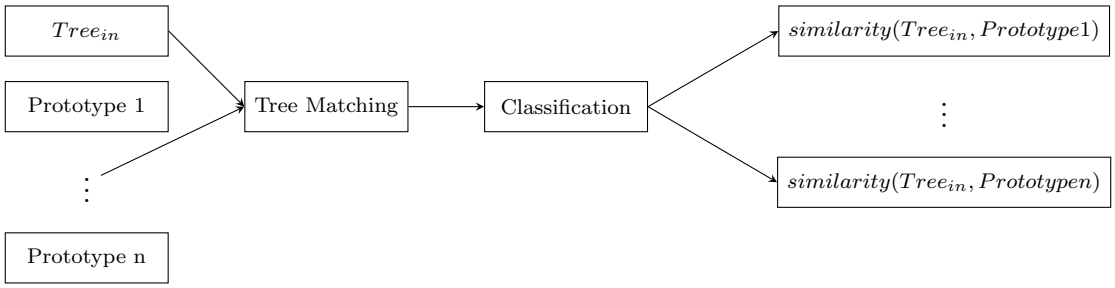


Figure 3.13: The generic classification pipeline

4 Evaluation

In the previous chapter we have seen how rules defined in the data model influence the structure of the tree instances, which results in different requirements towards matching strategies. Over the last years, two approaches towards prototype extraction and classification of semi-structured data, have been turned into a concrete software product, at the Cognitive Systems Group (University of Bamberg). Hydra had been developed as part of Florian Bader's masters thesis (Bader, 2009) and is optimized for processing SAP incidents. Bader's benchmark results showed, that classification of semi-structured data based on pre-calculated prototypes works very well in the domain of SAP incidents (Bader, 2009, p. 91f).

In the context of Stefan Betzmeir's bachelor thesis (Betzmeir, 2010), an attempt has been made to generalize the findings from Bader's work into the Prototype Extraction Library (PXL) for prototyping and classifying generic tree-based data structures. Although both Hydra and PXL are based on the same idea of extracting a prototype from a cluster of tree-structured data and classifying incoming data based on the similarity to these prototypes, they differ greatly when it comes to matching nodes.

4.1 Hydra

Based on the aspects regarding matching identified in section 3.2, Hydras matching strategy is instance based and incorporates neither horizontal, nor vertical matching. To be more precise, Hydra matches nodes implicitly by counting path-occurrences in a cluster.

Therefore the prototype calculated by Hydra's prototyping algorithm (Bader, 2009, p. 65) is basically a lookup-table which stores, how often a specific path has been encountered while traversing the trees of a cluster.

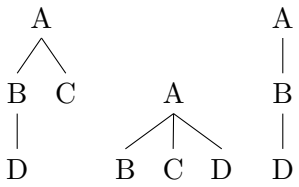


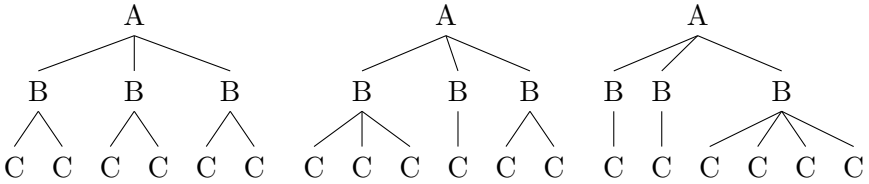
Figure 4.1: Cluster 1

The cluster from figure 4.1 would result in the prototype depicted in figure 4.2.

Path	Count	Cluster Size
`/A`	3	3
`/A/B`	3	3
`/A/C`	2	3
`/A/D`	1	3
`/A/B/D`	2	3

Figure 4.2: Prototype of cluster 1

While this strategy works well for the SAP domain and performs very good in terms of runtime, it leads to problems if the data model allows for extensive repetition of nodes, as we lose information about the concrete instances of the same path.



(a) Cluster 2 (b) Cluster 3 (c) Cluster 4

Figure 4.3: Three clusters featuring identical paths

The problem becomes apparent if Hydra calculates the prototype of cluster 2, 3 and 4 depicted in figure 4.3, each of which consists of only one tree. The trees differ at the third level, but due to the fact, that the path `/A` is contained once, `/A/B` is contained three times and `/A/B/C` is contained exactly six times in each cluster, the prototypes of those clusters would be identical and therefore the classification of an incoming tree would not

be better, than when done randomly. The reason is, that we lose the information about which instance of the C-node is the child of which instance of the B-node.

Therefore, to be able to classify trees based on a model which allows for repetition of nodes, the matching has to be more sophisticated.

4.2 PXL

The Prototype Extraction Library has been designed in a way, that each aspect of the prototyping and classification process can be altered and replaced to assess different techniques. Due to this modularity, we will first describe, how PXL has been configured before running the tests discussed in this paper.

4.2.1 Matching

The matching strategy currently used is purely *instance based* and best described as *Progressive Hierarchical Tree Alignment*, as it progressively matches two trees at a time and only incorporates *horizontal matching*, thus it only matches nodes from the same hierarchy level without "skipping" nodes (i.e. *vertical matching*).

4.2.2 Similarity

Matching and similarity are two inseparable concepts. One cannot calculate the similarity of two sets of objects, without deciding on which object from the first set to compare with which object from the other set, first (i.e. the matching problem). As there might be more than one valid configuration to do so, it is necessary to find the best match, i.e. the configuration with maximal similarity.

Trees are recursive data-structures, consisting of a hierarchy of nodes, so the similarity of two trees can be expressed as the similarity of two nodes, consisting of the *local similarity*, i.e. the similarity of the node labels, and the *recursive similarity*, i.e. the similarity of the two nodes children, and their children's children, etc. For the local similarity, we will use a simple binary similarity measure: if two labels are equal (i.e. they represent the same sequence of characters) the local similarity is 1, if not, it is 0. As the recursive similarity can be defined as the similarity of two sets of nodes (i.e. the child nodes), the best match of nodes has to be computed first, which again is based on the similarity of a pair of nodes, consisting of the local similarity and the recursive similarity. The recursion ends, when one or both sets of nodes are empty, in which case the recursive similarity is 0.

Our tests showed that the sum of weighed nodes, i.e. $local\_similarity + recursive\_similarity$ works best as similarity measure because this guarantees that the trees/subtrees with the most matching nodes get merged first, when progressively "combining" the trees of a cluster into one prototype. Originally we used the normalized function $local\_similarity \cdot parent\_weight + recursive\_similarity \cdot (1 - parent\_weight)$ which led to inferior results.

4.2.3 Prototyping

Currently in PXL there is no difference between trees and prototypes, as both are represented as trees, with nodes consisting of a weight and a label, which can be of any type, although we will use strings throughout the examples. The weights can have values between 0 and 1 and default to 1 for incoming trees. In a prototype tree the weights are used to encode how often a certain node has been encountered in the cluster. We used a prototyping strategy which calculates the weighed set union of nodes, by simply counting the number of matching nodes used to create one prototype node and applying the weight for each node accordingly. This corresponds with the *Structure Dominance Tree Generalization* described by Schmid, Hofmann, Bader, Häberle, and Schneider (2010).

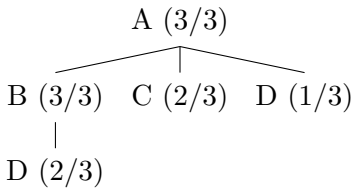
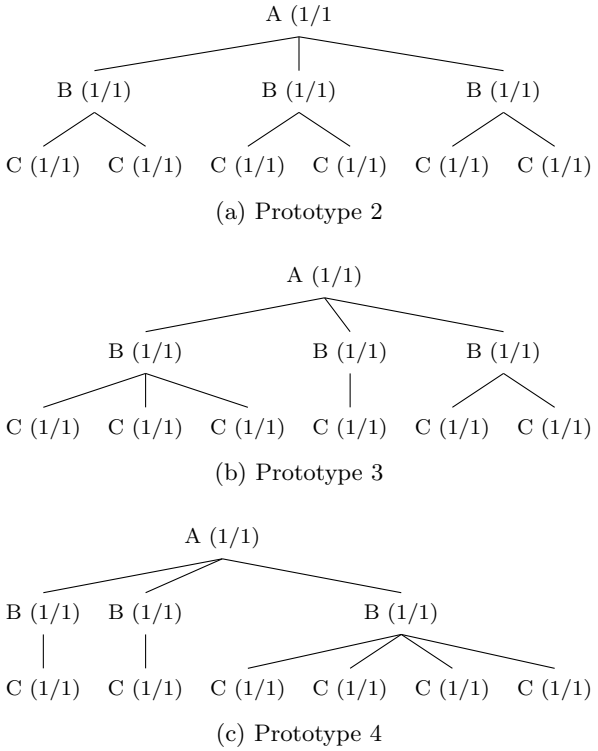


Figure 4.4: Prototype of cluster 1

For example, if we consider cluster 1 from figure 4.1, the resulting prototype, calculated by our PXL setup described above, would look like the tree depicted in figure 4.4.



(a) Prototype 2

(b) Prototype 3

(c) Prototype 4

Figure 4.5: The prototypes 2, 3 and 4 calculated by PXL

The advantage of horizontal matching and the representation of the prototype as a tree becomes apparent, if we examine the prototype of cluster 2, 3 and 4 calculated by PXL depicted in figure 4.5. Now nodes are being matched based on their local similarity (i.e. their labels) and the similarity of their children and because of the fact that the prototype is being represented as a tree, all the structural information is retained.

4.2.4 Classification

Classification of a tree is simply the process of calculating the similarity of said tree to each prototype and choosing the class represented by the prototype with the highest similarity.

4.3 Tests

To compare the performance of Hydra and PXL, we decided to keep close to the test setup and data used by Bader (2009). We compared the results of our PXL setup with three different versions of Hydras *Structure Dominance Tree Generalization (SDTG)*: SDTG without Anti-Unification, SDTG with Anti-Unification and SDTG with a reduced information base (Bader, 2009, p 61-63). The first and second test have been adopted unaltered, whereas the third test had been redesigned. The fourth test is supposed to show the limitations of the approach implemented in Hydra. The test data used for the first three tests consists of the same 57 SAP Business ByDesign incidents used by Bader (2009), which have been partitioned into 11 clusters by an SAP expert. For the fourth test, a small set of incidents have been created, based on the SAP Business ByDesign incident model.

4.3.1 First Test

The first test setup can be considered a clean room test. A prototype is being calculated for each cluster and all of the 57 incidents are being classified, so that each incident of the training set is also included in the test set.

Approach	**Hits**	**Misses**	**Precision Rate**
Hydra SDTG	53	4	0.9298
Hydra SDTGAU	54	3	0.9474
Hydra SDTGRI	55	2	0.9649
PXL	54	3	0.9474

Figure 4.6: First test

The results show that PXL's classification precision of nearly 95 percent is comparable to that of Hydra. Although it is necessary for real-world applications to classify data which has been already classified before, this situation should be considered a corner case.

4.3.2 Second Test

The second test one incident of each cluster is being removed from the training/prototyping set and these 11 incidents are being used for the classification. Hence, the incidents to be classified are not part of the training set anymore. This test is divided in three sub-tests, where in the first one, the first incident of each cluster is being removed and in

the second and third one the same is being repeated with the second and third incident respectively.

Sub-Test 1			
Approach	**Hits**	**Misses**	**Precision Rate**
Hydra SDTG	8	3	0.7273
Hydra SDTGAU	11	0	1.0000
Hydra SDTGRI	11	0	1.0000
PXL	8	3	0.7273
Sub-Test 2			
Approach	**Hits**	**Misses**	**Precision Rate**
Hydra SDTG	6	5	0.5455
Hydra SDTGAU	7	4	0.6364
Hydra SDTGRI	9	2	0.8182
PXL	9	2	0.8182
Sub-Test 3			
Approach	**Hits**	**Misses**	**Precision Rate**
Hydra SDTG	8	3	0.7273
Hydra SDTGAU	11	0	1.0000
Hydra SDTGRI	11	0	1.0000
PXL	9	2	0.8182
Average Sum			
Approach	**Hits**	**Misses**	**Precision Rate**
Hydra SDTG	7.3333	3.6667	0.6667
Hydra SDTGAU	9.6667	1.3333	0.8788
Hydra SDTGRI	10.3333	0.6667	0.9394
PXL	8.6667	2.3333	0.7879

Figure 4.7: Second test

Here PXL performs not as good as Hydra, although we can see that PXL is always as good as or better than Hydra using SDTG without Anti-Unification.

4.3.3 Third Test

The third test is based on the setup of the second test, but this time the clusters are going to be "contaminated": In the first sub-test the first incident of each cluster is being moved to the next cluster respectively, in the second and third sub-test the same scheme is being repeated with the second and third incident respectively.

Sub-Test 1			
Approach	**Hits**	**Misses**	**Precision Rate**
Hydra SDTG	7	4	0.6364
Hydra SDTGAU	8	3	0.7273
Hydra SDTGRI	7	4	0.6364
PXL	7	4	0.6364
Sub-Test 2			
Approach	**Hits**	**Misses**	**Precision Rate**
Hydra SDTG	8	3	0.7273
Hydra SDTGAU	8	3	0.7273
Hydra SDTGRI	9	2	0.8182
PXL	8	3	0.7273
Sub-Test 3			
Approach	**Hits**	**Misses**	**Precision Rate**
Hydra SDTG	6	5	0.5455
Hydra SDTGAU	8	3	0.7273
Hydra SDTGRI	7	4	0.6364
PXL	8	3	0.7273
Average Sum			
Approach	**Hits**	**Misses**	**Precision Rate**
Hydra SDTG	7	4	0.6364
Hydra SDTGAU	8	3	0.7273
Hydra SDTGRI	7.6667	3.3333	0.6970
PXL	7.6667	3.3333	0.6970

Figure 4.8: Third test

Test three simulates a more realistic setting which is similar to the situation in SAP's help desk, where data is being manually classified, which is an error prone process. The results suggest that PXL error compensation capabilities are comparable to that of Hydra.

4.3.4 Fourth Test

For this test we have created three clusters, each consisting of one tree based on the SAP incident model each. A prototype is being learned from each cluster and then the same three trees are being classified. As mentioned above, the sole purpose of the fourth test is to show that Hydra it is not suitable as a general purpose approach towards semi-structured data, although it works well in the SAP domain. It illustrates the problems of Hydra's path-based matching, which have already been discussed with the help of the

example depicted in figure 4.3.

Approach	**Hits**	**Misses**	**Precision Rate**
Hydra SDTG	1	2	0.3333
Hydra SDTGAU	1	2	0.3333
Hydra SDTGRI	1	2	0.3333
PXL	3	0	1.0000

Figure 4.9: Fourth test

Due to the fact that the three prototypes calculated by Hydra are equal, the classification of the trees is completely random, resulting in 1 hit and 2 misses for each SDTG, SDTGAU and SDTGRI. PXL on the other hand is able to classify the trees correctly.

5 Conclusion

We have seen that classification of semi-structured data based on prototypes is indeed possible and that it can help to free help desk experts from the tedious, repetitive task of manually classifying incident reports. By introducing the model based perspective we have shown how production rules manifest themselves in the tree instances and which challenges arise by introducing quantifiers. With the implementation of horizontal matching in our *Prototype Extraction Library (PXL)*, prototype based classification has become available for a much broader field of domains.

The next step to be taken must be the incorporation of vertical matching, in order to improve the classification precision for tree instances of models where the reuse of production rules has a major influence on the structure of the trees. By identifying matching as the central problem, we can also assess generic matching algorithms which could fit our needs. For example, a tree-matching approach featuring horizontal and vertical matching has been developed by Zhang and Shasha (1989) which is based on the edit distance of trees. As it might be a suitable matching strategy for our prototype based classification, it should be tested, whether it can be incorporated in our current framework.

Currently, *PXL's* prototyping algorithm is based on a strategy of calculating the alignement of multiple trees based on the alignment/match of pairs of trees. This approach should be compared to more sophisticated techniques developed, e.g. in the field of bioinformatics (Notredame, 2002), because when deriving the global alignment from locally optimal matches, there is a high risk of local maxima leading to bad global alignments.

An important question is: How can we further exploit the information cotained in tree model? Although a lot of our reasoning is based on the idea of a model from which tree instances are derived, the actual algorithms for matching, prototyping and classification are instance based and do not take advantage of additional knowledge which could be provided by the model.

References

Bader, F. (2009). *Modellbasierte Klassifikation von Störungsmeldungen in betriebswirtschaftlichen Softwaresystemen - Ein Ansatz zum Prototyp-Lernen durch strukturbasierte Generalisierung.*

Betzmeir, S. (2010). *Lernen von Prototypen über Baumstrukturen – Entwicklung und Evaluation eines Verfahrens zum Structural Incident Mining* (Tech. Rep.).

Notredame, C. (2002). Recent progress in multiple sequence alignment: a survey. *Pharmacogenomics*, *3*(1), 131–144.

Schmid, U., Hofmann, M., Bader, F., Häberle, T., & Schneider, T. (2010). *Incident mining using structural prototypes* (Tech. Rep.). University of Bamberg.

Zhang, K., & Shasha, D. (1989). Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM Journal on Computing*, *18*(6), 1245–1262.