RC24847 (W0908-100) August 19, 2009
Computer Science

IBM Research Report

Problem Classification Method to Enhance the ITIL Incident, Problem and Change Management Process

Yang Song
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802

Anca Sailer, Hidayatullah Shaikh
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Problem Classification Method to enhance the ITIL Incident, Problem and Change Management Process

[1]Yang Song,* [2]Anca Sailer, [2]Hidayatullah Shaikh,

[1]Department of Computer Science and Engineering,
The Pennsylvania State University,
University Park, PA 16802, USA

[2]IBM TJ Watson Research Center,
Hawthorne Research Lab,
19 Skyline Drive,
Hawthorne, NY 10532, USA

ABSTRACT

Problem determination and resolution PDR is the process of detecting anomalies in a monitored system, locating the problems responsible for the issue, determining the root cause and fixing the cause of the problem. The cost of PDR represents a substantial part of operational costs, and faster, more effective PDR can contribute to a substantial reduction in system administration costs. In this paper, we propose to automate the process of PDR by leveraging machine learning methods. The main focus is to effectively categorize the problem a user experiences by recognizing the problem specificity leveraging all available training data such like the performance data and the logs data. Specifically, we transform the structure of the problem into a hierarchy which can be determined by existing taxonomy in advance. We then propose an efficient hierarchical incremental learning algorithm which is capable of adjusting its internal local classifier parameters in real-time. Comparing to the traditional batch learning algorithms, this online learning framework can significantly decrease the computational complexity of the training process by learning from new instances on an incremental fashion. In the same time this reduces the amount of memory required to store the training instances. We demonstrate the efficiency of our approach by learning hierarchical problem patterns for several issues occurred in distributed web applications. Experimental shows that our approach substantially outperforms previous methods.

Categories and Subject Descriptors

H.4.m [**Information Systems**]: Miscellaneous; D.2 [**Software**]: Software Engineering; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Delphi theory

Keywords

ACM proceedings, LaTeX, text tagging

*This work was done when Yang was an intern at IBM Research.

Copyright is held by the author/owner(s).
WWW2008, April 21–25, 2008, Beijing, China.
.

1. INTRODUCTION

The cost of problem determination and resolution (PDR) usually represents more than half of the operational costs in IT services. Specifically, PDR is the process of detecting anomalies in a monitored system, locating the problems responsible for the issue, determining the root cause and fixing the cause of the problem. Traditionally, once the user (customer or technical personnel) detects a problem, he first tries to identify the type of problem in order to search for the relevant fix. However, in case of software problems in multi-tier IT environment with complex distributed system dependencies, the same front-end issue that the user experienced may be caused by different back-end system or application problems. Thus, the problem may be only the effect of an underlying issue within the IT environment and the fixes found are not addressing the root cause. An example of such a multi-tier environment is an e-business system which is supported by an infrastructure consisting of the following subsystems connected by local and wide area networks: web based presentation services, access services, application business logic, messaging services, database services and storage subsystems.

Therefore, faster, more effective PDR can contribute to a substantial reduction in system administration costs. Most existing works that address the issue of PDR either require human involvement [11], or monitor the system performance and set threshold for warning [4]. The main limitation of this prior art are that most are problem specific and as such lack the potential of being applied to wider type of issues: e.g., [6] addresseds network problems, [5] addressed hardware problems, [14] addressed database problems and so on.Because of the specificity, the methodologies above use only one type of the available information, ignoring information that may be relevant to the problem at hand, e.g., [5] only considered log files and [8] only focused on system performance metrics.

More recently, machine learning techniques have been introduced to problem determination. Among all, [7] analyzed fault logs and trace logs using statistical methods to determine the root causes. However, this approach did not take into consideration the taxonomy of the problem causes and used a flat structure for detection, which is not scalable to real-world large systems. In [15], the authors proposed a decision tree to find the root causes in distributed systems which leveraged the hierarchical structure of the problem taxonomy. Nevertheless, decision tree is known to give preference to features (or attributes) with a large number of po-

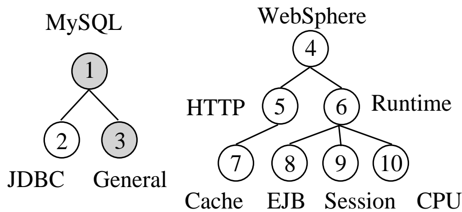

Figure 1: An example of problem taxonomy represented using two-tree forest with $M = 10$ nodes. Each node is associated with a class label. Gray nodes indicate a problem trace of an error which is formed using a multilabel assignment in our learning algorithm.

tential values [13], which may cause bias during the learning process and reduce the classification performance.

1.1 Our Contribution

In this paper, we propose a framework to address PDR by classifying the problems into a predefined hierarchical structure of taxonomies, using an incremental online learning algorithm that learns the pattern of errors from a set of manually-labeled training data. Specifically, we make the following contributions:

- By transforming the structure of a flat problem space into a hierarchy (see Figure 1) which is predefined according to an existing taxonomy, we reduce the complexity of the pattern search for problem determination, using a more efficient tree structure.

- We propose an incremental learning algorithm which is capable of adjusting its internal local classifier parameters whenever a new training instance is available. Comparing to the traditional batch learning algorithms, this on-line learning framework can significantly decrease the computational complexity of the training process by learning from new instances on an incremental fashion. In the same time this reduces the amount of memory required to store the training instances.

- For efficiency, we train a linear classifier at each node in the hierarchy. Since linear classifiers usually have simpler decision boundaries, which exhibit better generalization toward large data sets. Meanwhile, linear classifiers require less internal parameters to learn, which potentially avoids the problem of over-fitting caused by large number of model parameters, and also makes it scalable to large-scale enterprize-level systems.

- Instead of simply using error logs or system performance data as training data, we combine these two types of data to be a more powerful feature set for error classification. The log data contains word features while the performance data are numerical values. These two types of data are basically uncorrelated. In machine learning, it has been shown that combining features that are less correlated will generally exhibit better predictive performance for the model [10], which becomes the motivation of this combination.

2. OUR APPROACH

The hierarchical approach that we consider is suitable for problem determination since it allows the problem categorization in an automatic fashion. Each problem will be labeled with a set of labels that covers the name of the application/product as well as the specific cause of the problem. This set of inter-related labels can be well-represented using a hierarchical forest structure as shown in Figure 1, where each tree in the forest specifies possible causes of a specific application/product. The name of the application/product is assigned to the root of the tree whose level-structure is decided by the complexity of the application/product. Regardless of the tree structure of an application/product, however, the path of the labels always starts at a root node and ends at a leaf node. For each individual problem there is one and only one such path in the entire forest. Note that whenever a certain node is included in the label set, all the nodes on the path to the root node must also be included.

When a problem occurs, either the system/application will generate some error logs, or the administrator will notice some change of the monitoring data. To automate the process of problem determination, a set of training data is needed for the classifiers to learn the patterns of the problem taxonomy. This process requires a manual labeling of each problem instance that is used for training purpose. In our problem setting, each problem instance contains a set of labels which is referred to as *multilabel.* E.g., {MySQL, General}, {WebSphere, Runtime, CPU}. The presence of a multilabel indicates whether a problem instance has been labeled by all nodes relevant to the problem. A simple representation of multilabel is the binary coding schema, where 1 indicates the presence of the problem instance at a node, and 0 indicates the absence of the problem instance at a node. In the example shown in Figure 1 where a total of $M = 10$ labels exist, the problem instance (in grey nodes) has two labels: label 1 and 3. Thus, the associated multilabel can be represented as $\{1, 0, 1, 0, 0, 0, 0, 0, 0, 0\}$.

For training classifiers, each problem instance is transformed into a feature vectors x, where the features in the vector correspond to the words that appear in the error logs, the numerical thresholds monitored by the administrator and so on. Given a set of problem instances x_i with their associated labels y_i, we train M classifiers for the taxonomy, where each node in the taxonomy has a local classifier. The process is shown in Algorithm 1, where each x_i is a d-dimensional feature vector and y_i an M dimensional multilabel vector. Each problem instance x_i is passed to all M nodes for classification. If the predicted label is different from the actual label at a specific node m, the associated decision functions are then updated given the feature vector x_i.

We use linear classifier at each node for pattern classification. Usually, a linear classifier can be represented as $f(x) = w^T x + w_0$, where w is called weight vector that has the same dimensionality as the feature vector x. w_0 indicates the bias or offset of the classifier. w and w_0 are the internal parameters of the classifier which are learnt during the training phase. For binary classification where each label y_m is either 0 (negative) or 1 (positive), an input vector x_i is classified to the positive class if $f(x) \geq 0$ and to the negative class otherwise.

There are many choices of linear classifiers, including discriminant classifiers like Fisher's linear discriminant, and

Algorithm 1 Hierarchical Classification: Training Stage

1: **Input** training data $\{\mathbf{x}_i, \mathbf{y}_i\}_1^N$, $\mathbf{x}_i \in \mathbb{R}^d$ d-dimensional feature vector, $\mathbf{y}_i = \{y_1, ..., y_M\} \in \{0,1\}^M$ multilabel, where
N: number of instances, M: number of labels (nodes)
2: **Initialize** local classifiers for all M nodes $\{f_1(\mathbf{x}), ..., f_M(\mathbf{x})\}$
3: **for** each training data $\mathbf{x}_i$ $(i = 1...N)$
4: **for** each label m $(m = 1...M)$
5: classify $\mathbf{x}_i$ using $f_m(\mathbf{x}_i)$
6: **if** $\mathbf{x}_i$ is misclassified **then** update $f_m(\mathbf{x}_i)$
7: **end for**
8: **end for**
9: **Output** M classifiers $\{f_1(\mathbf{x}), ..., f_M(\mathbf{x})\}$

generative classifiers like naive Bayes. We use the Perceptron algorithm [12] for our online learning purpose because the algorithm can incremental updates the parameters w and w_0 whenever a new training instance becomes available. Specifically, the Perceptron updates at each node m the parameters w and w_0 when the instance x_i is misclassified. In the case that the true label of x_i is positive (1) but predicted to be negative (0), the parameters are updated by adding $x_i : w = w + x_i$; if the true label of x_i is negative (0) but predicted to be positive (1), the parameters are updated by subtracting $x_i : w = w - x_i$.

In Algorithm 2, we use a variant of the general Perceptron which is able to update the parameters more efficiently [12]. In this algorithm, the weight vectors for each node are initialized to be zero or small random variables. We combine the parameters w and w_0 into one vector $\tilde{w}$, and augment each feature vector x_i by adding 1 to its first column to be $\tilde{x}$, so that the update of the decision function can be simplified to $f(x) = \tilde{w}^T \tilde{x}$.

Each time an instance is misclassified at a node m, the associated weights are updated by adding $\tilde{x}_i$ with a constant factor (line 7). The algorithm stops when all instances have been classified. We define $\delta = -\tilde{w}^T \tilde{x}$ and use it as the learning rate for each Perceptron update. ϵ is a very small positive number. In this way we accelerate the update of Perceptron by guarantying to reduce the error rate after each step. When all instances have been classified, the algorithm stops and outputs the learnt weight vectors.

After training, each node needs to store only the weight vectors $\tilde{w}_m$ in the memory. The trained classifiers can be use for real-time problem determination when a new problem is reported. Specifically, when a new problem $x*$ occurs, the multilabel $y*$ is computed using the same top-down process as used for training, except for the condition that a node m is visited only if its parent node $parent(m)$ classifies the instance $x*$ to be positive. The detailed algorithm is listed in Algorithm 3. The label decision rule is shown below:

$$y_m = \begin{cases} 1, & \text{if } \mathbf{w}^m \mathbf{x}_* \geq 0 \text{ and } m \text{ is a root node,} \\ 1, & \text{if } \mathbf{w}^m \mathbf{x}_* \geq 0 \text{ and } y_{parent(m)} = 1, \\ 0, & \text{otherwise.} \end{cases}$$

Given a set of N training instances and M predefined labels, the online learning algorithm iterates $N * M$ steps to update the parameters for learning. During each update, the prediction is calculated by multiplying two d-dimensional vectors into a scalar. Thus, the computational complexity of our algorithm is bounded by $O(MNd)$.

3. EXPERIMENTS

Algorithm 2 Perceptron

1: **Initialize** the parameters for each $f_m(\mathbf{x})$
$\mathbf{w}^m = \{0, ..., 0\}^d, w_0^m = -1$, augment $\tilde{\mathbf{w}}^m = \{w_0^m, \mathbf{w}^m\}$
2: **for** each training data $\mathbf{x}_i$ $(i = 1...N)$
3: augment $\tilde{\mathbf{x}}_i = \{1, \mathbf{x}_i\}$
4: **for** each label m $(m = 1...M)$
5: classify $\tilde{\mathbf{x}}_i$ using $f_m(\tilde{\mathbf{x}}_i) = \tilde{\mathbf{w}}^m \tilde{\mathbf{x}}_i$
6: **if** $\mathbf{x}_i$ is misclassified
7: $\tilde{\mathbf{w}}^m = \tilde{\mathbf{w}}^m + \frac{\delta + \epsilon}{||\tilde{\mathbf{x}}_i||^2} \tilde{\mathbf{x}}_i$, where
8: $\delta = -\tilde{\mathbf{w}}^m \tilde{\mathbf{x}}_i$ and ϵ is a small positive number
9: **end if**
10: **end for**
11: **end for**

Algorithm 3 Hierarchical Classification: Test Stage

1: **Input** M classifiers $\{f_1(\mathbf{x}), ..., f_M(\mathbf{x})\}$, a test instance $\mathbf{x}*$
2: **for** each classifier $f_m(\mathbf{x})$ $(m = 1...M)$
3: **if** $f_m(\mathbf{x})$ classifies $\mathbf{x}*$ to be negative (0)
4: assign $y_m = 0$
5: **else if** $f_m(\mathbf{x})$ classifies $\mathbf{x}*$ to be positive (1)
6: **if** $y_{parent(m)} = 1$ then assign $y_m = 1$
7: **else** assign $y_m = 0$
8: **end for**
9: **Output** predicted class label y_* for $\mathbf{x}*$

In this section we present some empirical analysis of our approach. We used a similar experimental setting as presented in [4]. Specifically, we used Trade 6 [1] as a test-bed for web applications. For simulating user operations, we used IBM Websphere Workload Simulator [2] to perform multi-user activities on Trade 6. Four types of errors were injected in order to generate errors so that we can collect the training data, i.e., database shutdown, network failure, Websphere port change and exceed user connection limit.

Both log data and system performance data were collected. The Websphere error log data were collected as textual information for the errors, while the metrics from Snappimon [3] were collected as numerical data for the system performance (e.g., CPU usage). Overall, we collected a total of 1,700 error samples, which consists approximately 2,800 word features and 80 numerical features.

For performance evaluation, we measure both *effectiveness* and *efficiency* of our algorithms. The effectiveness is calculated by using the *precision* score, i.e., the percentage of correctly classified data in the entire test data set. To evaluate the efficiency, we consider the computational complexity for training the classifiers.

Figure 2 presents the precision on the test data set. We split the data into training set and test set with different proportions. It can be observed that with the combination of both log data and numerical data, our algorithm performs best in all three cases where the training data contains 10%, 50% and 90% of the entire data. It is evident that with 90% of the training data, the performance of the algorithm is generally much better than those with few training data. With the combination of log and numerical, our algorithm achieves a 96.35% of precision with 90% of the training data.

To justify our hierarchical classification method, we made a comparison to an approach that uses a flat structure to classify errors. Specifically, the algorithm performs a binary classification for each class and chooses the one with the highest confidence (i.e., highest predictive score) as the class label. Figure 3 shows the results for the two algo-

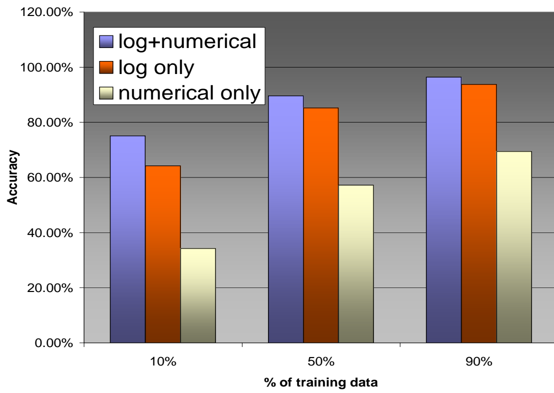

Figure 2: Precision on the test data set. The combination of log and numerical data performs the best.

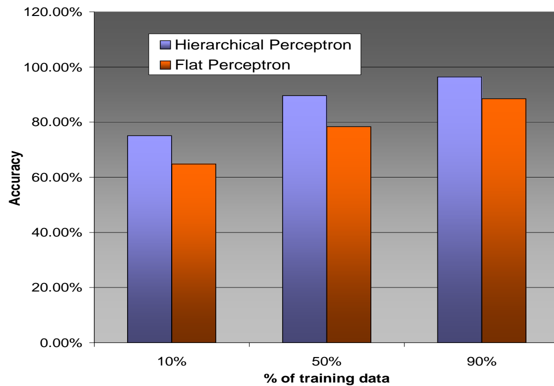

Figure 3: Comparison of the test performance between hierarchical classification vs. flat classification. Our Hierarchical approach shows better results in all three cases.

rithms. Our hierarchical structure clearly outperforms the flat stricture in all cases. As explained in the algorithm details, the hierarchical structure reduces the candidate classes for a test sample by approximately half after each step of classification, which potentially reduces the probability of making errors as well.

Finally, we compare the efficiency of our online Perceptron with a batch learning algorithm. We choose a popular discriminant batch learning support vector machine (SVM) [9] for comparison. From Figure 4, it can be seen that the online Perceptron has a constant learning time with the increase data. While for the batch learning algorithm, the computational cost of training a classifier is linear to the number of training instances, which cannot be scalable for large enterprize-level applications.

4. CONCLUSION AND FUTURE WORK

We presented a hierarchical online classification framework for automatically determine the root causes of problems in IT services. Our approach shown high quality of classification as well as fast training time for the classifiers. We believe that this approach is suitable for large-scale enterprize-level systems. Future work involves finding more numerical and log features to improve the precision, and better classifiers for online learning.

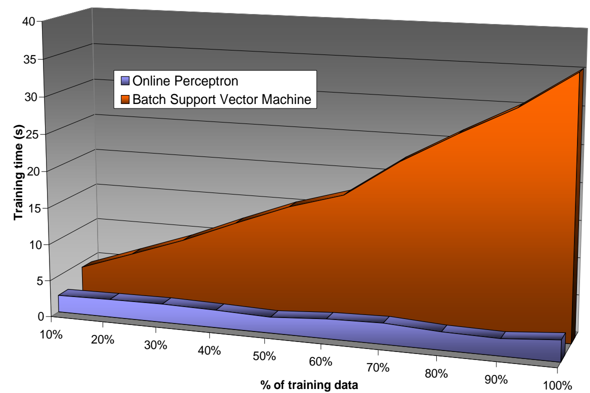

Figure 4: Computational complexity of two learning algorithm. Our online learning algorithm (shown in blue) exhibits a constant learning time with the increase of the training data, while traditional batch learning algorithm requires linear learning time.

5. REFERENCES

[1] IBM trade performance benchmark sample, http://www-306.ibm.com/software/webservers/appserv/was/performance.html.

[2] IBM websphere studio workload simulator, http://www-306.ibm.com/software/awdtools/studioworkloadsimulator/.

[3] Snappimon monitoring suite, http://www.snappimon.com/.

[4] M. K. Agarwal, N. Sachindran, M. Gupta, and V. Mann. Fast extraction of adaptive change point based patterns for problem resolution in enterprise systems. In *DSOM*, 2006.

[5] D. M. Benignus, M. S. Edwards, and A. J. Tysor. Method and system for performing problem determination procedures in hierarchically organized computer systems. *U.S. patent application Publication No.US6532552*, 2003.

[6] D. D.-H. Chen, W. F. M. Jr., E. DePaolis, and L. Temoshenko. System and method for collecting and retrieving network problem determination data with a generic collection subsystem reporting to an agent on demand. *U.S. patent application Publication No.US5682523*, 1997.

[7] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: problem determination in large, dynamic internet services. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 595–604, 2002.

[8] I. Cohen and M. Goldszmidt. Determining and annotating a signature of a computer resource. *U.S. patent application Publication No.US7184935*, 2007.

[9] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[10] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, 2003.

[11] W. D. Pauw and C. E. Williams. Methods for adaptive problem determination in distributed service-based applications. *U.S. patent application Publication No.US7360120*, 2008.

[12] R. Rojas. *Neural networks: a systematic introduction.* Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[13] A. M. Tsvetkov. Development of inductive inference algorithms using decision trees. *Cybernetics and Systems Analysis*, 29:141–145, 1993.

[14] V. R. R. Tummalapalli. Multidimensional repositories for problem discovery and capacity planning of database applications. *U.S. patent application Publication No.US6804714*, 2004.

[15] A. X. Zheng, J. Lloyd, and E. Brewer. Failure diagnosis using decision trees. In *ICAC '04: Proceedings of the First International Conference on Autonomic Computing*, 2004.