Hierarchical Online Problem Classification for IT Support Services

Yang Song, Anca Sailer, and Hidayatullah Shaikh

Abstract—The overwhelming amount of various monitoring and log data generated in multitier IT systems makes problem determination one of the most expensive and labor-intensive tasks in IT Services arena. Particularly the initial step of problem classification is complicated by error propagation making secondary problems surfacing on multiple dependent resources. In this paper, we propose to automate the process of problem classification by leveraging machine learning. The main focus is to categorize the problem a user experiences by recognizing the real root cause specificity leveraging available training data such as monitoring and logs across the systems. We transform the structure of the problem into a hierarchy using an existing taxonomy. We then propose an efficient hierarchical incremental learning algorithm which is capable of adjusting its internal local classifier parameters in realtime. Comparing to the traditional batch learning algorithms, this online solution decreases the computational complexity of the training process by learning from new instances on an incremental fashion. Our approach significantly reduces the memory required to store the training instances. We demonstrate the efficiency of our approach by learning hierarchical problem patterns for several issues occurring in distributed web applications. Experimental results show that our approach substantially outperforms previous methods.

Index Terms—Machine learning, artificial intelligence, computing methodologies, services technologies, principles of services, services computing.

---

1 Introduction

Most organizations rely nowadays on IT environments to provide or receive business critical services. The problem determination in such environments is one of the most expensive and labor-intensive tasks in the IT incident and problem management. Traditionally, once the user (customer or technical personnel) detects a problem, they first try to identify the type of problem in order to search for the relevant fix. However, in case of problems in multitier (e.g., web) environments 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 initial symptom may be only the effect of an underlying issue propagated through the complex chain of dependent systems and the fixes found are not addressing the real root cause. An example of such a multitier web environment is illustrated by the IT environment in Fig. 1. The distributed e-business system supported by an infrastructure consists of the following interdependent subsystems connected by local and wide area networks: web-based presentation services, access services, application business logic, messaging services, database services, and storage subsystems. Attempting to collect and manually analyze all the available monitoring and log data in a failing multitier environment turns the problem classification in an overwhelming effort. Therefore, only an effective automatic problem classification can significantly improve the incident and problem management experience and thus contribute to substantial reduction in system administration costs. Most of the existing solutions for problem classification use either only the monitoring data or only logs, without getting the full benefit of both worlds. Steinder and Sethi [24] review the existing approaches to fault classification and also presents the challenges of managing modern e-business environments. The most common approaches to fault classification are AI techniques (e.g., rule-based, model-based, neural networks, decision trees), model traversing techniques (e.g., dependency-based), and fault propagation techniques (e.g., codebook-based, Bayesian networks, causality graphs). Our solution falls in the category of AI (machine learning) techniques. Another limitation of the prior art is that most solutions are problem specific and as such lack the potential of being applied to wider type of issues: e.g., network problems [5], database problems [12], and misconfiguration problems [27], [28], [10], [14].

More recently, machine learning techniques have been introduced to problem determination [15]. For instance, fault logs and trace logs are analyzed using statistical methods to determine the root causes [7]. 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. Decision tree are used as well to find the root causes in distributed systems leveraging a hierarchical structure of the problem taxonomy [31]. Nevertheless, decision tree is known to give preference to features (or attributes) with a large number of potential values [26], which may cause bias during the learning process and reduce the classification performance.

In this paper, we propose a framework to address problem determination enhancement through automation

- *Y. Song is with Microsoft Research, One Microsoft Way, Redmond, WA 98034. E-mail: yangsong@microsoft.com.*
- *A. Sailer and H. Shaikh are with the IBM T.J. Watson Research Center, Hawthorne Research Lab, 19 Skyline Dr, Hawthorne, NY 10532. E-mail: {ancas, hshaikh}@us.ibm.com.*

Manuscript received 27 Aug. 2010; revised 8 Dec. 2010; accepted 25 Dec. 2010; published online 2 Feb. 2011.
For information on obtaining reprints of this article, please send e-mail to: tsc@computer.org, and reference IEEECS Log Number TSC-2010-08-0121.
Digital Object Identifier no. 10.1109/TSC.2011.3.

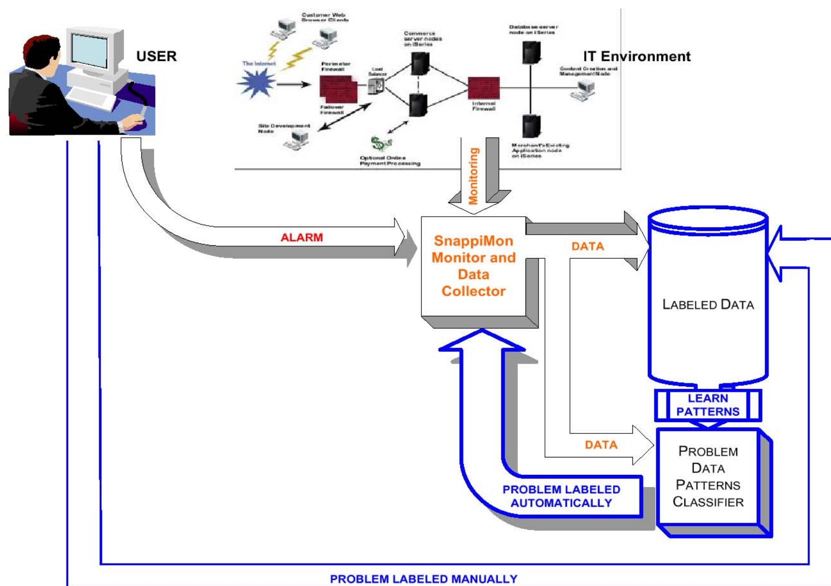


Fig. 1. Problem classification process in multitier web environment. The monitoring data and error logs are manually labeled with the real underlying problem that occurred when the data were collected. The patterns specific for an exiting problem taxonomy are learned from this historical labeled data. Our system recognizes problems when given a new set of monitoring and log data based on the learned patterns.

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 Fig. 2) which is predefined according to an existing taxonomy, we reduce the complexity of the pattern search for problem determination, since using a more efficient tree structure.
- We propose an incremental Perceptron 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 online learning framework can significantly decrease the computational complexity of the training process by learning from new instances on an incremental fashion. Additionally, 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 in the hierarchical structure [6]. Moreover, linear classifiers require less internal parameters to learn, which potentially avoids the problem of overfitting caused by large number of model parameters, and also makes it scalable to large-scale enterprise-level systems.
- We combine error logs with usage and performance monitoring data for training data as a more powerful feature set for error classification, instead of simply using partial data. The log data are typical generated information for any software resource in an IT environment and is collected by IT management tools as events. The monitoring data can be either natively generated by the software resource themselves and when available pulled by IT monitoring tools or it can be generated by monitoring agents deployed in the IT environment and stored by IT monitoring tools in repositories for reporting and postevent analysis. In the log data we are mainly looking for word features since it typically consists of text and code, while in the monitoring data we are looking for trends since it contains 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 [11], which becomes the motivation of this combination.

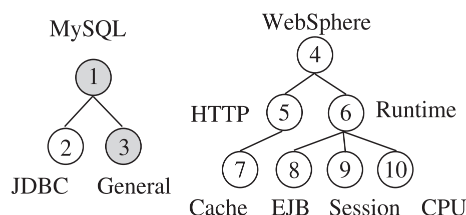


Fig. 2. An example of problem taxonomy represented using two-tree forest with $M = 10$ nodes. Each tree in the forest represents a different application. Each node is associated with a class label. Gray nodes indicate a problem trace of an error which is formed using a multilabel assignment in our learning algorithm.

Note that the creation of the problem taxonomy is beyond the scope of this paper. The problem taxonomy used in this paper is mainly extracted from the IBM Support Assistant (ISA) tool.[1] ISA is an IBM serviceability workbench for software products that allows the users to analyze an IT problem and manage a problem request. The main goal of ISA is to accelerate the problem resolution for customers. ISA contains a predefined problem taxonomy

1. http://www-01.ibm.com/software/support/isa/index.html.

for IBM products: DB2, WebSphere, Lotus, Tivoli, and Rational. Currently, in order to invoke the problem analysis with its specialized data collectors (one for each problem in the ISA taxonomy), the customer has to know the real problem they are dealing with, not only its front-end effects. Due to the large ISA coverage of IT problems and to the significant volume of issues solved by the IBM personnel using ISA, the taxonomy defined within ISA is highly relevant and the use cases sufficient for the training data required by our classifiers.

2 Related Work

2.1 Hierarchical and Online Classification

Hierarchical classification on one hand decomposes traditional multiclass classification into hierarchical structures by discovering the dependencies between class labels. The new representation facilitates both classifier learning and new label prediction. Hierarchical classification is closely related to multilabel classification, where each instance contains more than one class of labels. Most of the existing solutions assume that the multilabel of an instance only includes one path in the hierarchy, starting from the root and ending at one of the leaves.

One approach was to automatically organize the content of search results by leveraging the outputs from different levels of hierarchical classifiers [9]. The authors leveraged support vector machines as the classifiers. Similarly, the neural networks were used as a hierarchical mixture of expects (HME) for text classification [21]. Alternatively, the single path assumption in hierarchical classification can be relaxed [20]. The authors introduced a kernel method that is capable of assigning an instance to more than one path in the hierarchy.

On the other hand, online learning aims at incrementally updating the classifiers when the training data are collected continuously over time. Compared to traditional batch learning algorithms, online learning exhibits significantly faster training effort at comparably predictive performance. Among online classifiers such as online linear discriminant analysis (LDA) [16], online support vector machines (SVMs) have been proved the most successful classifiers [23], [17]. The idea behind online-SVMs is simple and effective: keep a set of previously used training data instead of the entire training data and use only this data to proximate the training space for an optimal online SVMs. Every time an instance is misclassified, the algorithm updates the parameters by using the SEEN data. This approximation can be treated as a tradeoff between time/space complexity and the predictive performance of the classifiers. We leverage this effective technique to train our online classifier.

More recently, both hierarchical and online incremental learning have been combined to create more powerful classifiers. A general framework was introduced that can be implemented by using arbitrary classifiers at each node in the taxonomy [6]. From a machine learning perspective, the proposed model minimizes the H-loss of the training set which is a loss function defined over the taxonomy. The authors also theoretically analyzed the performance of their framework where more than one path is allowed for classification.

In the IT problem determination context, a technique based on statistical modeling, clustering and inference, that automatically extracts signatures from system metrics during problem events was described in [8]. These signatures are constructed so that they can be matched against signatures for similar previous events. In our approach we combine the system metrics and logs to identify signatures and use supervised machine learning techniques.

3 Our Approach

We propose a novel machine learning algorithm that uses a hierarchical classification framework for problem classification. 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 cover the name of the application/product (A/P) as well as the specific cause of the problem. This set of interrelated labels can be well represented using a hierarchical forest structure as shown in Fig. 2, where each tree in the forest specifies possible causes of a specific A/P. The name of the A/P is assigned to the root of a tree whose level-structure is decided by the complexity of that A/P, while its nodes represent specific A/P aspects or subcomponents that can be sources of problems. Regardless of the tree structure, 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 A/P will generate error logs, or the administrator will notice some change of the monitoring data. To automate the process of problem determination, set of training data are 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., $\{\mathrm{MySQL}, \mathrm{General}\}$, $\{\mathrm{WebSphere}, \mathrm{Runtime}, \mathrm{CPU}\}$. 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 Fig. 2 where a total of $M = 10$ labels exist, the problem instance (in gray nodes) has two labels: labels 1 and 3. Thus, the associated multilabel can be represented as $\{1, 0, 1, 0, 0, 0, 0, 0, 0, 0\}$.

3.1 Hierarchical Classification Framework

For training classifiers, each problem instance is transformed into a feature vector x, where the features in the vector correspond to words that appear in the error logs, the numerical thresholds monitored by the administrator and so on. Details of feature representation and selection will be discussed in Sections 4 and 5, respectively. Given a set of problem instances x_i with their associated labels y_i, we train M *binary* classifiers for the taxonomy, where each node in the taxonomy has a local classifier. Each binary classifier

simply makes a decision whether y_i is presented at that node (positive) or not (negative). Note that the choice of classifiers can be arbitrary at each node. For simplicity, we restrict the classifiers to be the same for all nodes in the taxonomy in this paper.

The training 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. The associated decision functions at each node are updated given the feature vector x_i and the predictive results from the local classifiers. The algorithm also maintains the relationship between classifiers, i.e., knowing the parent and children nodes of each classifier f_m. In the situation when x_i is classified to be negative (0) by the node f_m, all the nodes in the subtree of f_m will update their parameters by assuming a negative label of x_i, without making further computation. Otherwise, x_i will visit all nodes in the subtree of f_m iteratively. After training, each node retains a locally trained binary classifier.

Algorithm 1. Hierarchical Classification: Training
1: **Input** training data $\{\mathbf{x}_i, \mathbf{y}_i\}_1^N$, $\mathbf{x}_i \in \mathcal{R}^d$ d-dimensional feature vector, $\mathbf{y}_i = \{y_1, \ldots, y_M\} \in \{0,1\}^M$ multilabel, where
   N: number of instances, M: number of labels (nodes)
2: **Initialize** local classifiers for all M nodes $\{f_1(\mathbf{x}), \ldots, f_M(\mathbf{x})\}$
3: **for** each training data $\mathbf{x}_i$ $(i = 1 \ldots N)$
4: **for** each label m $(m = 1 \ldots M)$
5: **if** $y_{parent(m)} = 0$
6: then update the parameters of f_m given $y_m = 0$
7: **else** classify $\mathbf{x}_i$ using $f_m(\mathbf{x}_i)$, output the predicted y_m
8: update the parameters of f_m according to y_m
9: **end for**
10: **end for**
11: **Output** M classifiers $\{f_1(\mathbf{x}), \ldots, f_M(\mathbf{x})\}$

3.2 Online Learning Classifiers

For training efficiency, we propose to use classifiers that are capable of incremental learning from the training data received continuously over time. Specifically, an online-learning classifier should be able to adjust the classification parameters once new training data are available, without resolving to previously used training data. For simplicity, we only consider linear classifiers for online learning in our hierarchical framework.

Normally, a linear classifier can be represented as $f(\mathbf{x}_i) = \mathbf{w}^T\mathbf{x}_i + \mathbf{w}_0$, where $\mathbf{w}$ is called weight vector and has the same dimensionality as the feature vector $\mathbf{x}_i$. $\mathbf{w}_0$ indicates the bias or offset of the classifier. $\mathbf{w}$ and $\mathbf{w}_0$ together are the internal parameters of the classifier which are learned during the training phase. For binary classification where each label $\mathbf{y}_m$ is either 0 (negative) or 1 (positive), an input vector $\mathbf{x}_i$ is classified to the positive class if $f(\mathbf{x}_i) \geq 0$ and to the negative class otherwise.

We propose an online Perceptron algorithm as shown in Algorithm 2. Traditionally, the Perceptron algorithm is known to be a simple and very effective discriminative classifier [19]. The learning phase of Perceptron only involves two parameters. Specifically, the Perceptron only updates the parameters $\mathbf{w}$ and $\mathbf{w}_0$ when an instance $\mathbf{x}_i$ is misclassified. In case the true label of $\mathbf{x}_i$ is positive (1) but predicted to be negative (0), the parameters are updated by adding $\mathbf{x}_i : \mathbf{w} = \mathbf{w} + \alpha\mathbf{x}_i$, where α indicates the learning rate; if the true label of $\mathbf{x}_i$ is negative (0) but predicted to be positive (1), the parameters are updated by subtracting $\mathbf{x}_i : \mathbf{w} = \mathbf{w} - \alpha\mathbf{x}_i$. The Perceptron algorithm guarantees to converge if the data are linearly separable, i.e., the algorithm is capable of finding a hyperplane to separate two classes well. However, the classic Perceptron algorithm is required to iterate many runs on the training data until its convergence, which makes it unsuitable for online learning purpose.

Algorithm 2. Online Perceptron Learning
1: **Initialize** the parameters for each $f_m(\mathbf{x})$
   $\mathbf{w}^m = \{0, \ldots, 0\}^d, w_0^m = -1$, augment
   $\tilde{\mathbf{w}}^m = \{w_0^m, \mathbf{w}^m\}$,
   RESERVE = {}.
2: **for** each training data $\mathbf{x}_i$ $(i = 1 \ldots N)$
3: augment $\tilde{\mathbf{x}}_i = \{1, \mathbf{x}_i\}$
4: **for** each label m $(m = 1 \ldots M)$
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: **for** each $\mathbf{x}_j$ in RESERVE // *the adding phase*
8: augment $\tilde{\mathbf{x}}_j = \{1, \mathbf{x}_j\}$
9: $\tilde{\mathbf{w}}^m = \tilde{\mathbf{w}}^m + \frac{\delta+\epsilon}{\|\tilde{\mathbf{x}}_j\|^2}\tilde{\mathbf{x}}_j$, where $\delta = -\tilde{\mathbf{w}}^m\tilde{\mathbf{x}}_j$
   and ϵ is a small positive number
10: **end for**
11: add $\mathbf{x}_i$ to RESERVE
12: **for** each $\mathbf{x}_j$ in RESERVE // *the removing phase*
13: **if** $\mathbf{x}_j$ is classified correctly
14: remove $\mathbf{x}_j$ from RESERVE
15: **end for**
16: **end if**
17: **end for**
18: **end for**

Thus, in Algorithm 2, we propose a variant of the general Perceptron, which is able to update the parameters more efficiently [19]. In this modified algorithm, the weight vectors for each node are initialized to be zero or small random variables. We combine the parameters $\mathbf{w}$ and $\mathbf{w}_0$ into one vector $\tilde{\mathbf{w}}$, and augment each feature vector $\mathbf{x}_i$ by adding 1 to its first column to be $\tilde{\mathbf{x}}_i$, so that the update of the decision function can be simplified to $f(\mathbf{x}) = \tilde{\mathbf{w}}^T\tilde{\mathbf{x}}_i$. We also maintain a set RESERVE that keeps all the misclassified instances.

The algorithm has two major learning phases: 1) the adding phase and 2) the removing phase. In phase 1, each time an instance is misclassified at a node m, the algorithm adds the data into the RESERVE set. The algorithm then updates the parameters by adding a constant factor to $\tilde{\mathbf{w}}_m$ by iterating through all previously misclassified data in RESERVE. This way the hyperplane gets another chance to correct errors on previously misclassified instances. In phase 2, during the removing phase, the algorithm checks if a data in RESERVE can now be classified correctly after

the parameters update. If this is the case, the data are then removed from the RESERVE set. The algorithm stops when all data instances have been classified. We define $\delta = -\tilde{\mathbf{w}}^T\tilde{\mathbf{x}}$ and use it as a part of the learning rate for each Perceptron update. ϵ is a very small positive number. 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 learned weight vectors for each classifier f_i at node i.

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 for all data in the RESERVE set. Thus, the computational complexity of our algorithm is bounded by $O(MNRd)$, where R is the maximum size of the RESERVE set.

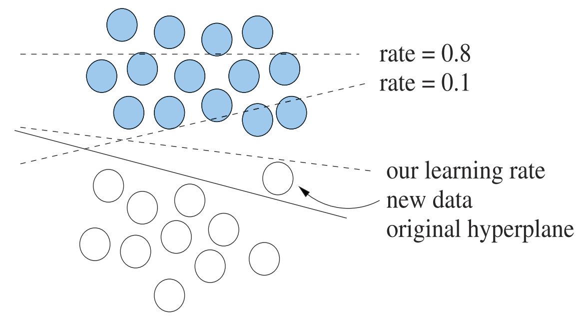
3.3 The Online Perceptron Learning Rate

The learning rate $\alpha = \frac{\delta+\epsilon}{\|\mathbf{x}\|^2}$ of the Perceptron technique is the most important parameter during learning, since it controls the algorithm convergence rate. Historically, the learning rate was chosen heuristically to be either 1 or a small positive number without prior knowledge. Instead of using a fixed learning rate for online learning, we propose to adjust the learning rate according to every new instance received dynamically. Theoretically, it has been shown that the learning rate $\frac{\delta+\epsilon}{\|\mathbf{x}\|^2}$ is guaranteed to correct the classification of each instance $\mathbf{x}$ by at least ϵ at each step [19]. Consequently, the misclassified instances in the RESERVE set will have a chance to get corrected after new training data are introduced (during the adding phase), and will eventually be removed from the RESERVE set (during the removing phase).

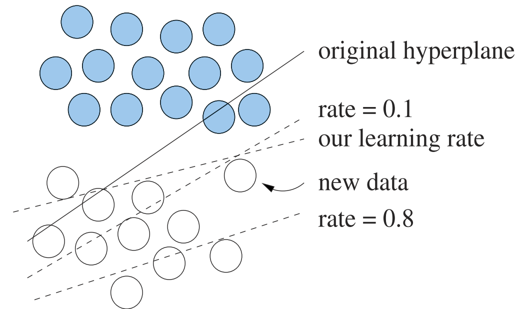
Fig. 3 demonstrates two true examples to show the importance of the learning rate. Two class examples that are separable are presented, with a new data arriving on the margin. In the case where the original hyperplane has already separated two classes before the introduction of a new data, our learning rate guarantees that the new weight vector moves to the region where it is just able to separate the new data without introducing new misclassified data, while the learning rates of 0.1 and 0.8 both create new misclassified instances in this case. On the other hand, when the original hyperplane cannot separate the two classes (in Fig. 3b three instances are misclassified), our learning rate will make sure that no more misclassified instances will be introduced. Specifically, after the adding and removing phases, the hyperplane that uses our learning rate reduces the number of misclassified instances to 1. The other two hyperplanes, unfortunately, both introduce new misclassified instances which are not present in the RESERVE set.

3.4 Explanations on One Pass Learning

Our online learning framework resembles the online SVM learning as in [23], where the model only gets retrained when an example is misclassified. With a good initial hypothesis parameters from previous trained models, the training time on new examples is usually sufficiently short so that it can be considered a real-time algorithm.



(a) Original hyperplane separates the two classes.



(b) Original hyperplane does not separate the two classes.

Fig. 3. The impact of the learning parameter. The original hyperplanes are shown in solid lines. The new hyperplanes with different learning rates are in dash lines.

Similarly, our algorithm has an adding phase where poorly classified examples are included in the RESERVE set. Whenever a new example is misclassified, the algorithm will iterate through all the examples in the RESERVE set and adjust the parameter. This step is very similar to the online SVM algorithm. However, our algorithm also has a removing phase where the correctly classified example by using the updated parameters can be removed from the RESERVE set. This makes sure that the size of the RESERVE set R does not grow monotonically but will be adjusted after each update, so that its size is relatively smaller than the entire data set N. It is true that during the adding phase, the parameter updating needs several rounds to converge. But considering $R \ll N$, our algorithm can be considered as a one-pass online learning framework.

Fig. 4 illustrates an example of the online learning process. We randomly generated two multivariate normal distributions, $N_1 \sim (\mathbf{0}, [1-0.4; -0.41])$ (in red) and $N_2 \sim (\mathbf{0}, [10.8; 0.81])$ (in blue), each with 1,000 examples, respectively. We then randomly shuffled the data points and added them to the online learning framework sequentially. We demonstrate the change of the error rate as well as the size of the RESERVE set when every 200 examples are added to the training set in Fig. 4. We limit the size of the RESERVE set to be 50 in this case. Thus, if more than 50 examples are misclassified, only the first 50 will be kept in the RESERVE set. It can be observed that the error rate is generally less than 3.5 percent in all scenarios, while the size of the RESERVE set fluctuates from 0 to 50. During each round, only the examples that are the most difficult to

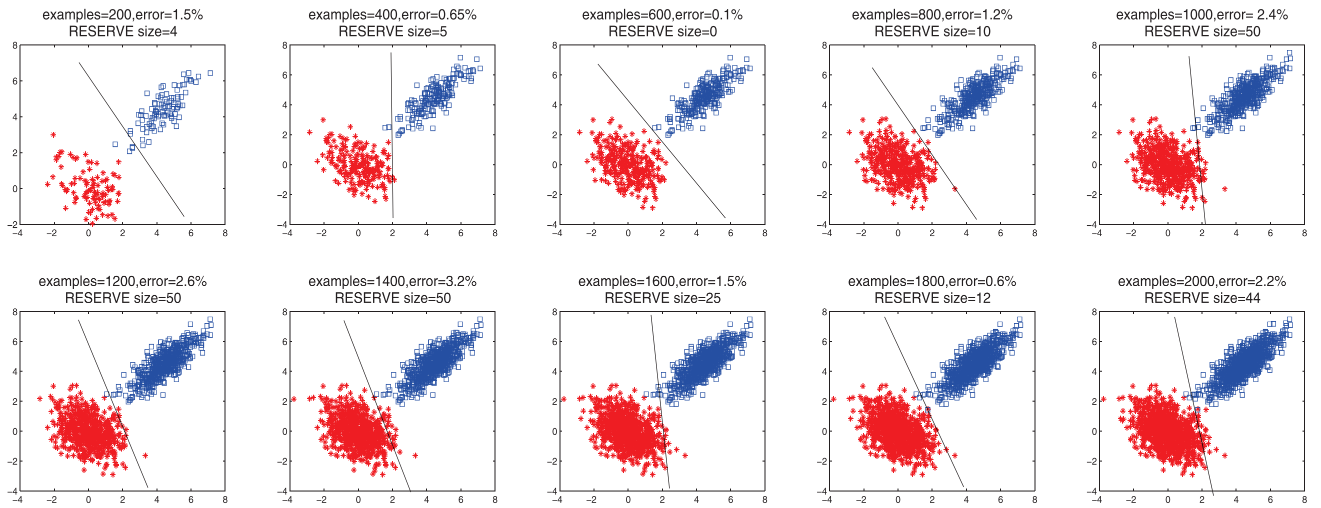


Fig. 4. A demonstration of our framework. Size of RESERVE set is limited to be 50. Data are added incrementally from 0 to 2,000.

classify are added to the RESERVE set. Some of those examples are correctly classified in the next round so that the RESERVE set will shrink, e.g., from 1,400 to 1,600, half of the examples in the RESERVE set were classified successfully. The error rate then dropped from 3.2 to 1.5 percent. It is obvious that the size of the RESERVE set plays an important role in balancing the efficiency and the classification performance. If the size of the RESERVE set is set to be infinite (or equal to the size of training set), then the algorithm is essentially the same as the original Perceptron algorithm. The influence regarding the size of the RESERVE set will be discussed in the experiments Section.

3.5 Online Hierarchical Prediction

After training, each node in the hierarchical framework needs to store only the weight vectors $\tilde{\mathbf{w}}_m$ in the memory. The trained classifiers can now be used for real-time hierarchical classification when a new instance is reported. Specifically, given a new $\mathbf{x}*$, its multilabel $\mathbf{y}*$ is predicted 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 $\mathbf{x}*$ to be positive. The detailed algorithm is listed in Algorithm 3. The label decision rule is shown below

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

Algorithm 3. Hierarchical Classification: Prediction

1: **Input** M classifiers $\{f_1(\mathbf{x}), \ldots, f_M(\mathbf{x})\}$, a test instance $\mathbf{x}*$
2: **for** each classifier $f_m(\mathbf{x})$ $(m = 1 \ldots M)$
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 $\mathbf{x}*$

The online prediction is extremely effective using our algorithm. Given a taxonomy with K trees, with the largest tree having depth D, a traditional multiclass classifier needs to decide the labels for $K * 2^D$ possible classes. However, with our algorithm only D steps are required to determine the label of an instance. Because during every step, a tree or a subtree with root predicted to be zero will be removed from the path. For example, given the taxonomy in Fig. 2 with $K = 2$ and $D = 3$, at most three steps are needed for deciding the root cause of a problem.

4 Features For Problem Classification

Two types of data are considered important for problem classification. 1) The first type of data is the error log or trace generated by the A/P under failure. This semistructured data usually contains important information regarding the errors, including, for example, error timestamps, ID of the instance which caused the error, error signatures, and error trace messages. An example of such an error log trace generated by IBM WebSphere Application Server is shown in Fig. 5.

Using the text information of the error logs for problem classification has been extensively studied [7], [4], [5], [12], [27], [28], [10], [14], [30], and these techniques often exhibit good accuracy in practice. Intuitively, two errors that share similar log information such as the error signatures are likely to be caused by the same root issue. However, this is not always reliable in the real-world scenarios. Since the format of error logs are often predefined varying from application to application, one technique may not be able to capture all problem patterns for a particular problem taxonomy, which often evolves over time.

2) Consequently, we suggest leveraging a second type of data, which is the monitoring data, as additional features. We collect important performance metrics at system, middleware and application level, e.g., CPU usage ratio, average application response time, etc., which exhibit significant changes when a related problem happens in the system. For example, in Fig. 6, we illustrate 17 hours of CPU usage for a WebSphere Application Server. An error

```
[6/16/08 13:34:59:388 EDT] 000042a6 ServletWrappe E SRVE0068E: Uncaught exception thrown
in one of the service methods of the servlet: TradeAppServlet. Exception thrown :
        at com.ibm.websphere.samples.trade.web.TradeServletAction.doLogin(TradeServletAction.java:377)
        at com.ibm.websphere.samples.trade.web.TradeAppServlet.performTask(TradeAppServlet.java:122)
        at com.ibm.websphere.samples.trade.web.TradeAppServlet.doPost(TradeAppServlet.java:84)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:763)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
        at com.ibm.ws.webcontainer.servlet.ServletWrapper.service(ServletWrapper.java:966)
        at com.ibm.ws.webcontainer.servlet.ServletWrapper.service(ServletWrapper.java:907)
```

Fig. 5. A sample of error log trace from the IBM WebSphere application. The error contains timestamp, application ID, originating class as well as the error trace message.

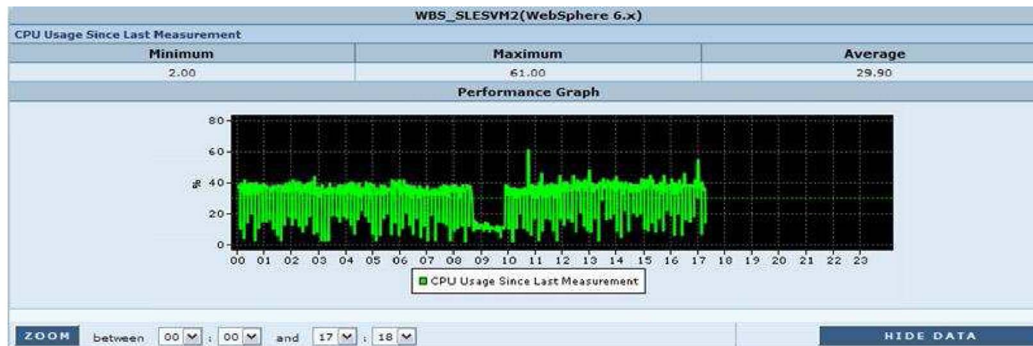


Fig. 6. An example of CPU usage graph from the Snappimon monitoring suite. An error was injected between 08:00 am and 10:00 am that causes a low CPU usage.

that alters the connection between clients and this application server was injected between 08:30 am and 10:00 am. Evidently, the CPU usage is significantly lower during this time period than it normally should be. Note that this type of errors can also be discovered by extracting and analyzing the system error logs with some effort. However, the value of the CPU usage is clearly a better indicator of CPU related failures.

In our experimental setup, we use the IBM Snappimon monitoring suite [3] to collect all important network, the application servers and the database servers are set up. Specifically, integrated infrastructure, network, system, middleware, and application metrics.

5 Feature Selection

Feature selection is an important postprocessing step after the feature generation and the construction of the instance-feature matrix. The benefits of feature selection include reducing the storage requirement of the training data, easing the visualization of the data for a better understanding, and most importantly, feature selection techniques are believed to improve the prediction performance of classification tasks, especially for text classification tasks [13], [18]. This is generally due to the sparseness issue of the bag-of-words representation in text classification, where each word is treated as a feature without considering the relations between different words. Fig. 7 illustrates an example of instance-feature matrix in our experiments, where each row is a training instance and each column represents a (word or numerical) feature. Each dot indicates an appearance of the feature in the instance. The feature space contains both text features and numerical features in no particular order.

Theoretically, word features that are less correlated or orthogonal to each other are more likely to show better predictive performance than features that are highly correlated [11]. Consequently, the objective of our feature selection task is to keep the most important word features, or so called *signatures* from the error logs, as well as the most useful numerical features of system performance, which have been successfully used in the literature [4] by monitoring their change points to detect system failures.

Similarly, numerical features have the same issues as word features. Fig. 8 presents an illustration of three features, where *committed rows* and *message count* are highly correlated, while *message count* and *method ready count* are less correlated. By scatter plotting, it is clear to

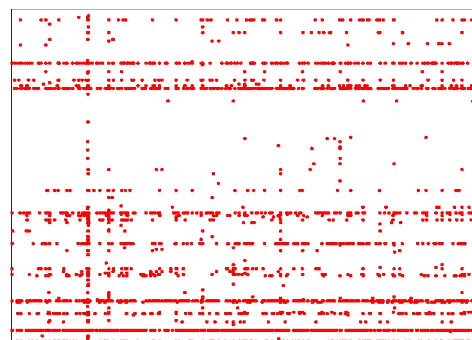


Fig. 7. An example of error-feature matrix from one of the data sets used in this paper. X-axis corresponds to feature IDs, and Y-axis is error IDs. Each red dot indicates an appearance of a particular feature in an error message. The features contain both text features (logs) and numerical features (system metric). The matrix is more than 90 percent sparse.

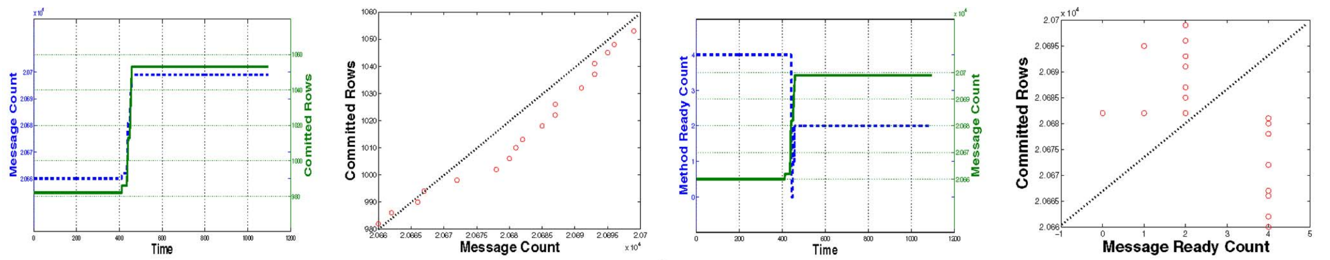


Fig. 8. Two examples of feature correlations. (Left) Time and correlation plots of two highly correlated features. (Right) Time and correlation plots of two less correlated features.

see that the correlation values of highly correlated features lie very close to the dash line of perfect correlation. Including these variables, therefore, will be redundant in terms of training an efficient classifier. We describe in the next sections the three supervised feature selection methods used in this paper.

5.1 Information Gain-Based Feature Selection

Information gain is an entropy-based method. Researchers have found that information gain is among the most effective statistical feature selection methods, which removes unnecessary features without losing accuracy for classification [18]. Specifically, given a set of training examples X with d features $\{F_1, \ldots, F_d\}$ in M classes $Y = \{Y_1, \ldots, Y_M\}$, we want to select features that give high information gain to the class labels. We define $IG(Y|F_i)$ to be the information gain of Y given feature F_i:

$$\begin{aligned} IG(Y|F_i) &= H(Y) - H(Y|X) \\ &= -\sum_{m=1}^{M} p(Y_m) \log_2 p(Y_m) \\ &\quad + p(F_i) \sum_{m=1}^{M} p(Y_m|F_i) \log_2 p(Y_m|F_i), \end{aligned} \tag{2}$$

where $p(Y_m)$ and $p(F_i)$ are the marginal probabilities of class labels and features, while $p(Y_m|F_i)$ defines the conditional probability of class label Y_m given a feature F_i. In our experiments, a predefined threshold t is specified to remove features whose IG values are less than t. The time complexity for this preprocessing step is $O(dM)$.

5.2 SVM-Based Feature Ranking

Support vector machines (SVMs) have been one of the most effective classifiers for text classification in the past years. Recently, researchers have proposed the use of SVMs as a tool for feature selection [29]. The authors first train an l_1-norm SVM on the training data. The output of the classifier contains weights for each feature where higher weighted features are more important. The authors find that many features are weighted zero and therefore they can be eliminated from the training set.

5.3 Document-Specific Feature Selection

A similar approach was taken by ranking the features based on their within-class weights obtained by the classifier [13]. This full induction method retains the top-k ranked features for each training instance, and then retrains a classifier with the new representation of feature space. The method is referred to as the document-specific feature selection (DSFS) method.

These feature selection methods have been used in our experiments for comparison with the IG-based method.

6 Experiments

In this section, we present some empirical analysis of our approach. We used an experimental setting similar to the one presented in [4]. Specifically, we used Trade 6 [1] as a testbed for web applications. Trade 6 serves as a WebSphere performance benchmark that simulates a brokerage trading system. Users of this application are able to perform typical trading operations provided by brokers. This application contains Java servlets, JDBC, JSPs, and other resources which are critical sources of failures for both WebSphere and its hosted applications and, thus, are covered by ISA's problem taxonomy.

For simulating user operations, we used IBM Websphere Workload Simulator (WSWS) [2] to perform multiuser activities on Trade 6. We recorded and repeated multiple user activities on different sessions. User actions include *registration, view account, view portfolio, buy stock, sell stock, logoff*, etc. Details can be found in [4].

To replicate a monitored multitier working environment, we installed the WebSphere Application Server v6 and DB2 Server v9 on different machines. We deployed Trade 6 benchmark web application on the WebSphere Application Servers. We installed Snappimon [3] on a separate server to monitor this distributed e-commerce environment. Fig. 9 shows our experimental settings. Since our goal is learning failure patterns for future recognition, it was sufficient to have a single server exposed to a particular failure under varying conditions of loads and additional failures rather than having a cluster of multiple load balanced servers under the same error injection; hence, we conducted our experiments focusing on a single pair of WebSphere/DB2 servers.

To generate failure training data, we injected commonly occurring errors into the system and collected the necessary data. More precisely, we ran multiuser sessions simultaneously (10 and 20 users for two experiments, respectively) for 30 minutes. The first 20 minutes correspond to normal operations and, hence, are error free, while during the last 10 minutes of each session we injected seven different errors. For the validity of our evaluation, we selected from the most prevalent issues in actual IT deployments those

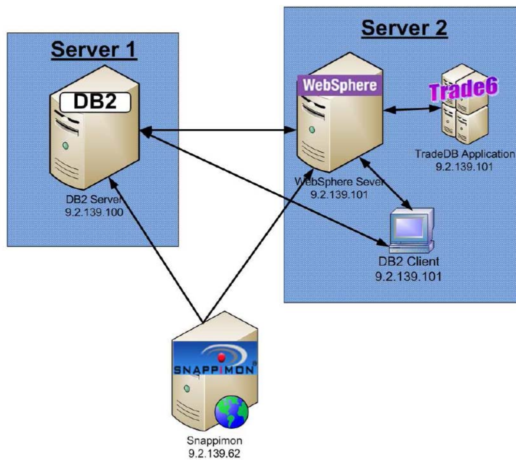


Fig. 9. Servers and client setup for our experiments. Server 1 contains the database server. Server 2 installs web application server, database client, and a web application. A monitoring suite is installed in a separate machine.

errors that appear similarly in production as well as on our testbed [4].

1. Iptables routing: drop packages and cause a network failure between WebSphere and DB2.
2. DB2 drop index: drop a DB2 table index.
3. CPU hog (DB): run a malicious program that consumes 100 percent of CPU on the DB2 Server.
4. DB2 shutdown: shutdown the DB2 server by force.
5. User limit exceed: connect many clients at the same time to the Trade 6 web application until WebSphere exceeds the configured limit of accepted connection.
6. WebSphere port change: switch the default port to an unknown port to the web applications.
7. CPU hog (WebSphere): run a malicious program that consumes 100 percent of CPU on WebSphere Application Server.

Overall, we collected a total of 1,700 failure samples (each sample corresponds to a failure), which consists of approximately 2,800 word features and 80 numerical features in seven classes. Fig. 10 shows the taxonomy in our experiment.

For performance evaluation, we measure both *effectiveness* and *efficiency* of our algorithms. The effectiveness is calculated by using the F_1 score, defined as $F_1 = 2 * P * R/(P + R)$, where P and R are the *precision* and *recall* metrics [22]. The efficiency is evaluated considering the computational complexity for training the classifiers.

We compare the effectiveness of our algorithm to two online-learning algorithms and one batch learning algorithm. The first online algorithm, online-LDA, is an online extension of the Fisher linear discriminant analysis [16], which exhibited good performance for text classification. The other online algorithm, online-SVM, is a relaxed version of support vector machine classifiers [23], which keeps a window of previously trained data to reduce the training complexity. The batch learning algorithm is SVM^{struct}, which is an SVM algorithm for predicting structured outputs.[2] It has been successfully applied to

2. http://svmlight.joachims.org/svm_struct.html.

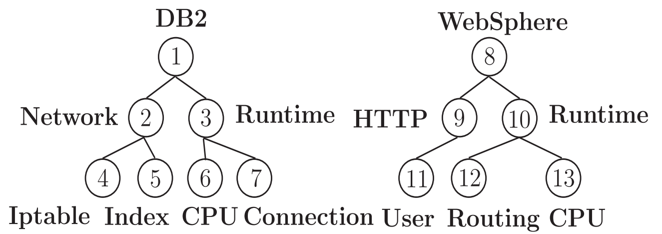


Fig. 10. Problem taxonomy for our experiment. Two applications with seven different errors are presented in the tree.

both hierarchical classification and natural language parsing [25]. Since it has been shown that linear classifiers usually perform well for text classification [13], we choose the linear kernel for both SVM algorithms.

For feature selection, we compare the three algorithms described in Sections 5.1, 5.2, and 5.3.

We split the error data into both training and testing data with different proportions from 10 to 90 percent. Additionally, for feature selection, we evaluate the performance of classifiers with 10, 40, and 70 percent of features selected.

In all our experiments, we preset the size of the RESERVE set to be 5 percent of the training data and perform adding and removing phrase according to Algorithm 2. The choices of the parameters will be discussed later.

6.1 Results and Discussions

Table 1 reports the effectiveness results of F_1 score after 10 independent runs of experiments. Our algorithm (Online-Perc) clearly outperforms online-LDA in all cases. Our algorithm also outperformed online-SVM in many cases, and in general, shows very competitive results against SVMs. The batch learning SVM^{struct} algorithm achieves the best predictive performance as expected in all three cases, and with an edge of only 1.5 percent over online-Perc on average.

Comparing the three feature selection methods, the SVM-based feature selection method shows evident advantage over the other two. The document-specific feature selection (DSFS) performs somewhat better than information gain (IG), but not very significant. Interestingly, the SVM-base feature selection method shows bigger improvement when applied to online-perc and online-LDA. For the two SVM classifiers, i.e., online-SVM and SVM^{struct}, however, SVM-based feature selection exhibits less improvement over IG and DSFS. Note that using 70 percent features demonstrates the best performance among all algorithms.

Next, we compare the learning efficiency of the four classifiers. From Fig. 11, it can be seen that the online Perceptron has a constant learning time as the number of training data increases. 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 enterprise-level applications. It is also evident that the online-SVM algorithm requires more time for training. The reason is that every time an instance is misclassified, the online-SVM has to find new parameters by performing quadratic programming on a set of previously reserved data. While for our algorithm, updating the parameters only involves linear operations for both adding and removing phases.

TABLE 1
Experimental Results on F_1 Score of Four Classifiers

# of features	**Online-Perc**	**Online-LDA**	**Online-SVM**	**SVM**^{struct} (Batch)
10% (IG)	61.77% ± 1.43%	53.21% ± 1.15%	62.92% ± 2.23%	64.78% ± 1.21%
40% (IG)	71.54% ± 1.75%	68.33% ± 1.60%	71.56% ± 2.73%	73.69% ± 2.17%
70% (IG)	83.25% ± 2.11%	76.14% ± 1.75%	83.22% ± 2.32%	84.35% ± 2.51%
100% (IG)	80.75% ± 2.08%	73.75% ± 1.21%	81.07% ± 2.91%	81.78% ± 2.17%
10% (SVM)	63.77% ± 1.11%	55.13% ± 1.22%	63.25% ± 2.07%	65.17% ± 1.16%
40% (SVM)	72.54% ± 1.21%	69.98% ± 1.57%	72.58% ± 2.12%	74.17% ± 2.00%
70% (SVM)	85.25% ± 2.35%	79.33% ± 1.85%	85.35% ± 2.45%	85.38% ± 2.38%
100% (SVM)	82.84% ± 1.79%	74.59% ± 1.34%	84.39% ± 2.66%	81.89% ± 2.18%
10% (DSFS)	61.95% ± 1.57%	53.87% ± 1.07%	62.16% ± 2.56%	64.71% ± 1.25%
40% (DSFS)	71.84% ± 1.09%	68.97% ± 1.81%	71.48% ± 2.09%	73.85% ± 2.05%
70% (DSFS)	83.22% ± 2.50%	79.35% ± 1.55%	83.22% ± 2.04%	84.85% ± 2.44%
100% (DSFS)	78.76% ± 2.09%	72.76% ± 1.34%	77.90% ± 2.32%	81.85% ± 2.05%

Three feature selection methods are evaluated. IG = information gain. SVM = support vector machine ranking. DSFS = document-specific feature selection.

Furthermore, we examine the performance of combine text features and numerical features. We choose to use different proportion of the training data, 10, 40, and 70 percent. We also select features using three different thresholds with three different methods. We use online-perc and online-SVM as the classifiers. This results a total of $2 * 3^{3*3} = 27^3$ different experiments. For brevity, we only show the 27 results by using IG-based feature selection. Fig. 13 presents the predictive precisions. It can be observed that with the combination of both log data and numerical data, both algorithms show the best predictive performance in most scenarios. It is evident that with 90 percent 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 93.35 percent of precision with 90 percent of the training data. It is also observable that using only log features constantly outperforms using only numerical features. Moreover, combining log and numerical features usually indicate better performance than log features itself. In our case, even with 40 percent features selected (86.02 percent, top right figure), the combination of both features outperforms log features with 70 percent features selected (85.09 percent, top middle figure). A rough conclusion drawn from the results is that, by combining log and numerical features, the performance of classifiers can be further improved.

To show the advantage of our hierarchical classification framework, we made a simple 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 path with the highest confidence (i.e., highest predictive score) as the class label. In our experiments, this involves classifying an instance into one of the 13 classes as shown in Fig. 10, and choosing the leaf node with the highest confidence score. An example has been shown in Fig. 12 to classify instances into the "Iptable" issue category. We choose to use Perceptron as the classifier for the flat (binary) case for fair comparison. Fig. 14 shows the results for the two frameworks. Our hierarchical structure clearly outperforms the flat stricture in all cases. As explained in the algorithm details, the hierarchical structure reduces the

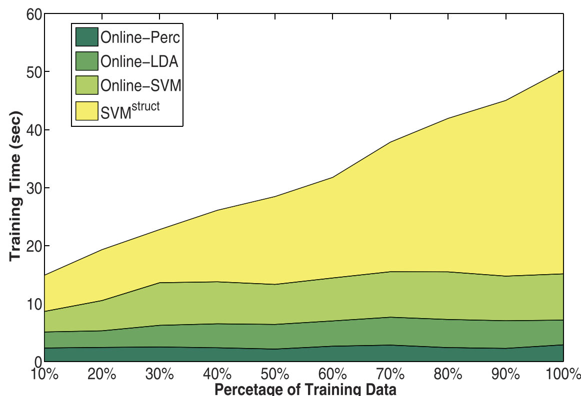


Fig. 11. Computational complexity of four learning algorithm. Our algorithm (online-Perc) exhibits a constant learning time with the increase of the training data, while traditional batch learning algorithm (SVM^{struct}) requires linear learning time.

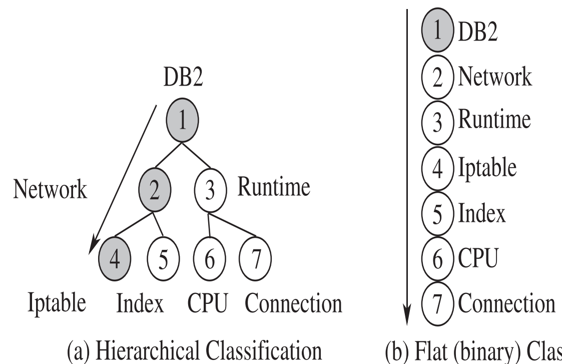


Fig. 12. Comparison of hierarchical classification and flat classification. To classify an instance into the "Iptable" category, hierarchical classification only has to perform three cases of classification while flat classification has to perform binary classification in all nodes which potentially increases the error rate.

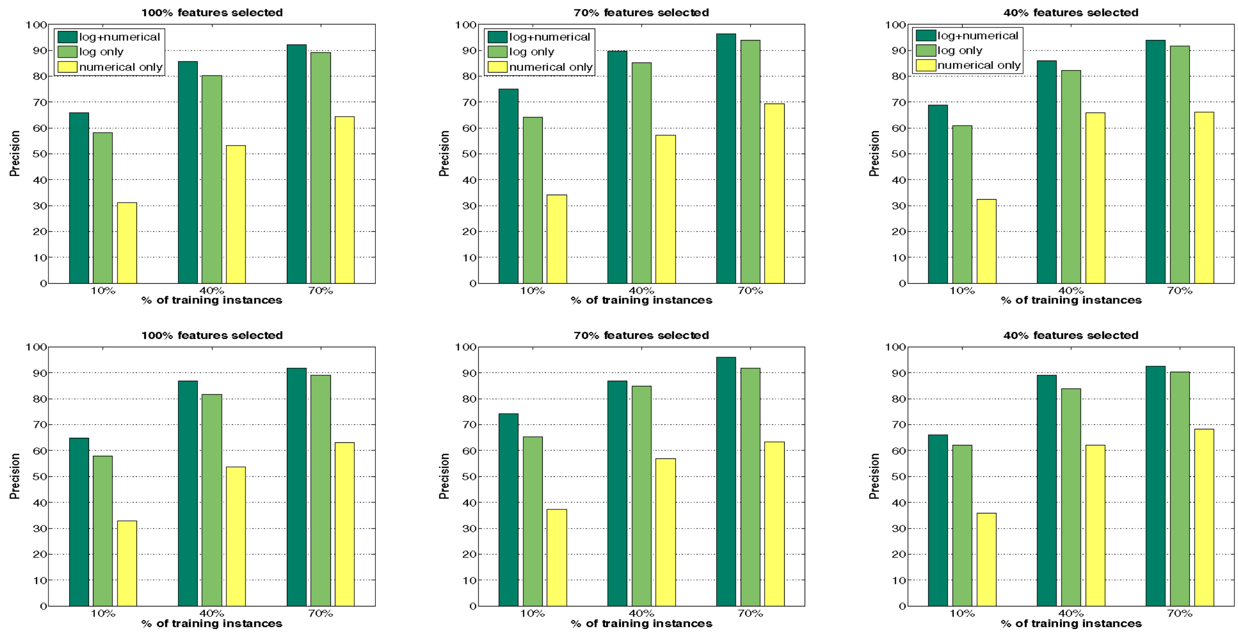


Fig. 13. Precision on the test data set. The combination of log and numerical data performs the best. Top: results using online Perceptron classifier. Bottom: results using online SVM classifier. Left: 100 percent features used. Middle: 70 percent features selected. Right: 40 percent features selected.

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 are also interested in discovering which text and numerical features are most useful in problem determination. By examining the IG-based and SVM-based scores, we find that *originating class* and *instance ID* are among the most informative text features, while *CPU usage* and *(network) response time* are the most discriminative numerical features. Table 2 lists the best features and their scores.

Note that in machine learning, hierarchical online classification has been a very challenging problem in the last decades, e.g., the subcategories in two different tree structures can have similar labels (take Fig. 10 as an example, both DB2 and WebSphere have an CPU category). Comparing to the published works in other application areas [9], [21], [6], the results of our top-down approaches indicate a significant improvement in terms of both performance and efficiency.

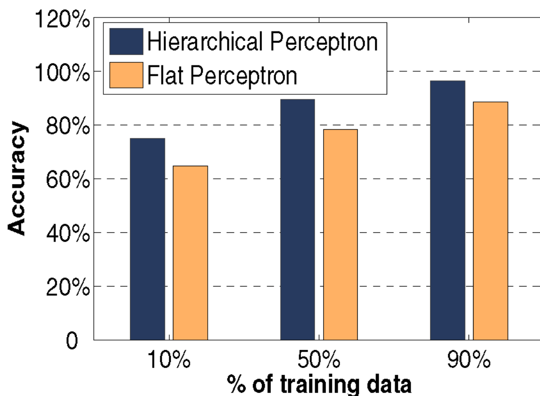


Fig. 14. Comparison of the test performance between hierarchical classification versus flat classification. Our Hierarchical approach shows better results in all three cases.

6.2 The Influence of RESERVE Set Size

Next, we demonstrate the influence of the size of the RESERVE set. By setting the size to be 5, 20, 45, and 65 percent, we compare the performance of the online Perceptron with the original Perceptron algorithm which iterates many rounds to converge. In this experiment, we empirically set the iteration to be 200. We test also whether the removing phase in Algorithm 2 is efficient and effective when removing correctly classified instances.

Fig. 15 sketches the results. Two observations can be made here. First, with the increase of the size, the accuracy increases as well, but not very significantly. The improvement of 65 percent over 5 percent as RESERVE set size is only 1.5 percent on average. Furthermore, the accuracy starts to converge even when the RESERVE size is only 20 percent, indicating that the size doesn't need to be large because the most difficult examples have already been included in the set. This observation indicates the effectiveness of the RESERVE set in our online learning framework.

Second, we can also observe that without the removing phrase, the accuracy of the online learning does not increase

TABLE 2
The Most Discriminative Text and Numerical Features

feature name	**IG score**	**SVM score**
originating class (text)	0.21705597	0.57662841
instance ID (text)	0.09327432	0.24273866
timestamp (text)	0.05267098	0.13238833
CPU usage (numerical)	0.13290475	0.44714795
response time (numerical)	0.08123898	0.21245921
method ready count (numerical)	0.04002738	0.13219308
committed rows (numerical)	0.01784899	0.04259606

Their IG scores and SVM scores are listed.

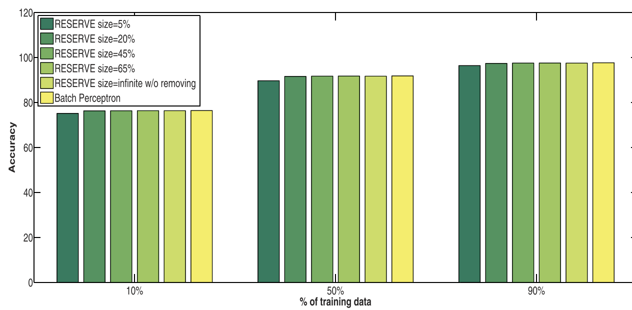


Fig. 15. Comparison of different size of the RESERVE set with the original Perceptron algorithm.

at all. Indeed, the size of the RESERVE set may only increase without removing, which could potentially benefit the learner to learn better parameters with more examples. But in practice, it only increase the learning cost by adding more examples which compromises the efficiency of the algorithm without increasing the accuracy. This observation indicates that the removing phase in Algorithm 2 can efficiently remove examples to be further classified, without affecting the accuracy of our framework.

7 Conclusion and Future Work

We presented a hierarchical online classification framework to automatically determine the root causes of problems in IT services. We proposed a novel online Perceptron algorithm which is capable of learning an optimal decision boundary for the training data by making only one pass through the data. Our approach showed similar quality of classification when comparing with state-of-the-art online SVM classifier, as well as faster training time than online SVM. By comparing with two other online algorithms and a batch learning algorithm, our framework shows better generalized performance on a real-world data set. We believe that this approach is suitable for large-scale enterprise-level systems.

In the future, we will try to collect more system performance data to boost the predictive results by using numerical features. We will also test our algorithms on a larger data sets that contain more problem patterns to make sure our algorithms scale to the real-world applications. Finally, we are looking forward to integrating our framework with the IBM support assistant (ISA) tool.

Acknowledgments

We would like to thank Narendran Sachindran and Manish Gupta from IBM IRL for providing the error injection methodology and code.

References

[1] IBM Trade Performance Benchmark Sample, http://www-306.ibm.com/software/webservers/appserv/was/performance.html, 2012.
[2] IBM Websphere Studio Workload Simulator, http://www-306.ibm.com/software/awdtools/studioworkloadsimulator, 2012.
[3] Snappimon Monitoring Suite, http://www.snappimon.com, 2012.
[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," *Proc. Distributed Systems, Operations and Management (DSOM)*, 2006.
[5] A. Brown, G. Kar, and A. Keller, "An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment," *Proc. Seventh IFIP/IEEE Int'l Symp. Integrated Network Management*, 2001.
[6] N. Cesa-Bianchi, C. Gentile, and L. Zaniboni, "Incremental Algorithms for Hierarchical Classification," *J. Machine Learning Research*, vol. 7, pp. 31-54, 2006.
[7] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services," *Proc. Int'l Conf. Dependable Systems and Networks (DSN)*, pp. 595-604, 2002.
[8] I. Cohen, M. Goldszmidt, A. Fox, J. Symons, J. Symons, S. Zhang, S. Zhang, T. Kelly, and T. Kelly, "Capturing, Indexing, Clustering, and Retrieving System History," *Proc. Capturing, Indexing, Clustering, and Retrieving System History (SOSP '05)*, pp. 105-118, 2005.
[9] S. Dumais and H. Chen, "Hierarchical Classification of Web Content," *Proc. 23rd Ann. Int'l ACM SIGIR Conf. Research and Development in Information Retrieval (SIGIR '00)*, pp. 256-263, 2000.
[10] A. Ganapathi, Y.M. Wang, N. Lao, and J.R. Wen, "Why PCs Are Fragile and What We Can Do About It: A Study of Windows Registry Problems," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '04)*, p. 561, 2004.
[11] I. Guyon and A. Elisseeff, "An Introduction to Variable and Feature Selection," *J. Machine Learning Research*, vol. 3, pp. 1157-1182, 2003.
[12] J. Hellerstein and V.R Tummalapalli, "Using Multidimensional Databases for Problem Determination and Planning of a Networked Application," *Proc. IEEE Third Int'l Workshop Systems Management (SMW '98)*, p. 117, 1998.
[13] A. Kolcz and W. tau Yih, "Raising the Baseline for High-Precision Text Classifiers," *Proc. 13th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD '07)*, pp. 400-409, 2007.
[14] N. Lao, J.R Wen, W.Y. Ma, and Y.M. Wang, "Combining High Level Symptom Descriptions and Low Level State Information for Configuration Fault Diagnosis," *Proc. 18th USENIX Conf. System Administration (LISA '04)*, pp. 151-158, 2004.
[15] M.A. Munawar and P.A.S. Ward., "A Comparative Study of Pairwise Regression Techniques for Problem Determination," *Proc. Conf. Center for Advanced Studies on Collaborative Research (CASCON '07)*, pp. 152-166, 2007.
[16] S. Pang, S. Ozawa, and N. Kasabov, "Incremental Linear Discriminant Analysis for Classification of Data Streams," *IEEE Trans. System, Man and Cybernetics*, vol. 35, no. 5, pp. 905-914, Nov. 2005.
[17] L. Ralaivola and F. d'Alché-Buc, "Incremental Support Vector Machine Learning: A Local Approach," *Proc. Int'l Conf. Artificial Neural Networks (ICANN '01)*, pp. 322-330, 2001.
[18] L. Ralaivola and F. d'Alché-Buc, "Incremental Support Vector Machine Learning: A Local Approach," *Proc. Int'l Conf. Artificial Neural Networks (ICANN '01)*, pp. 322-330, 2001.
[19] R. Rojas, *Neural Networks: A Systematic Introduction.* Springer, 1996.
[20] J. Rousu, C. Saunders, S. Szedmak, and J. Shawe-Taylor, "Learning Hierarchical Multi-Category Text Classification Models," *Proc. 22nd Int'l Conf. Machine Learning (ICML '05)*, pp. 744-751, 2005.
[21] M.E. Ruiz and P. Srinivasan., "Hierarchical Text Categorization Using Neural Networks," *Information Retrieval*, vol. 5, pp. 87-118, 2002.
[22] G. Salton and M.J. McGill, *Introduction to Modern Information Retrieval.* McGraw-Hill, 1986.
[23] D. Sculley and G.M. Wachman, "Relaxed Online Svms for Spam Filtering," *Proc. 30th Ann. Int'l ACM SIGIR Conf. Research and Development in Information Retrieval (SIGIR '07)*, pp. 415-422, 2007.
[24] M. Steinder and A.S. Sethi, "The Present and Future of Event Correlation: A Need For End-To-End Service Fault Localization," *Proc. Fifth World Multiconf. Systemics, Cybernetics, and Informatics (SCI)*, pp. 124-129, 2001.
[25] I. Tsochantaridis, T. Hofmann, T. Joachims, and Y. Altun, "Support Vector Machine Learning for Interdependent and Structured Output Spaces," *Proc. 21st Int'l Conf. Machine Learning (ICML '04)*, p. 104, 2004.
[26] A.M. Tsvetkov, "Development of Inductive Inference Algorithms Using Decision Trees," *Cybernetics and Systems Analysis*, vol. 29, pp. 141-145, 1993.

[27] H.J. Wang, J.C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic Misconfiguration Troubleshooting With Peerpressure," *Proc. Sixth Conf. Symp. Opearting Systems Design and Implementation (OSDI '04)*, pp. 17-17, 2004.

[28] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H.J. Wang, C. Yuan, and Z. Zhang, "Strider: A Black-Box, State-Based Approach to Change and Configuration Management and Support," *Proc. 17th USENIX Conf. System Administration (LISA '03)*, pp. 159-172, 2003.

[29] J. Weston, A. Elisseeff, B. Schölkopf, and M. Tipping, "Use of the Zero Norm with Linear Models and Kernel Methods," *J. Machine Learning Research*, vol. 3, pp. 1439-1461, 2003.

[30] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma., "Automated Known Problem Diagnosis with Event Traces," *SIGOPS Operating System Rev.*, vol. 40, no. 4 pp. 375-388, 2006.

[31] A.X. Zheng, J. Lloyd, and E. Brewer, "Failure Diagnosis Using Decision Trees," *Proc. First Int'l Conf. Autonomic Computing (ICAC '04)*, 2004.

Yang Song received the BS degree in computer science from Zhejiang University, China, in 2003 and the PhD degree in computer science from The Pennsylvania State University in 2009. His research interests include machine learning, data mining, information retrieval, and search engines. He is a research member at Microsoft Research, Redmond.

Anca Sailer received the PhD degree in computer science from Pierre et Marie Curie University of Paris, France, in 2000. She was a research member at the Networking Research Laboratory at Bell Labs from 2001 to 2003, where she specialized in Internet services traffic engineering and monitoring. In 2003, she joined IBM, where she is currently a research staff member in the Service Products Department. Her interests include cloud computing management, self-healing technologies, and information analysis.

Hidayatullah Shaikh is a senior technical staff member and senior manager at the IBM T.J. Watson Research Center. He is currently the IBM lead for business support services for cloud offerings. His research interests and expertise include virtualization, cloud computing, remote services delivery, business process modeling and integration, service-oriented architecture, grid computing, e-commerce, enterprise Java, database management systems, and high-availability clusters.