Using Pattern Recognition Techniques for Server Overload Detection

Cor-Paul Bezemer, Veronika Cheplygina, Andy Zaidman

Report TUD-SERG-2011-009
Abstract—One of the key factors in customer satisfaction is application performance. To be able to guarantee good performance, it is necessary to take appropriate measures before a server overload occurs. While in small systems it is usually possible to predict server overload using a subjective human expert, an automated overload prediction mechanism is important for ultra-large scale systems, such as multi-tenant Software-as-a-Service (SaaS) systems. An automated prediction mechanism would be an initial step towards self-adaptiveness of such systems, a property which leads to less human intervention during maintenance, resulting in less errors and better quality of service.

In order to provide such a prediction mechanism, it is important to have a solid overload detection approach, which is (1) a first step towards automated prediction and (2) necessary for automated testing of a prediction mechanism. In this paper we propose a number of steps which help with the design and optimization of a statistical pattern classifier for server overload detection. Our approach is empirically evaluated on a synthetic dataset.

I. INTRODUCTION

The success of web applications stands or falls with their customer satisfaction, and one of the key factors in customer satisfaction is the application performance [1]. In traditional settings, it is usually not very difficult to manually detect a performance problem, however, with the advent of ultra-large-scale (ULS) systems [2], manual performance monitoring and prediction becomes tedious and would thus ideally require automation. Automating performance prediction is typically hard, because many factors influence performance, and it is typically the human mind that excels at taking the right (subjective) decisions based on multiple factors. It is our aim to automate performance prediction, for which we have two distinct goals in mind: (1) warn the system administrator for the need of an impending hardware upscaling and (2) provide an automatic overload prevention mechanism.

One of the application domains where these ultra-large-scale systems will come to existence are the so-called multi-tenant software systems [3]. Multi-tenant software systems, being protagonists of the Software as a Service (SaaS) paradigm, are systems where large groups of users are working with a similar base-application, but where each tenant (∝ organization, which groups a number of users) has specific requirements towards the application. In essence, the core ideas behind multi-tenancy are (1) to make use of the economy of scale, which is typical for SaaS, and (2) provide advanced configurability, so that each tenant can adapt the application to their own needs. This highly-configurable multi-tenant model will likely replace the traditional application service provider (ASP) model, in which each organization has a deployment of the hosted application running on a dedicated server.

A trend in current software engineering research is the investigation of self-adaptive software [4]. Such self-adaptive systems, which are capable of adapting their own behavior according to changes in the environment and the system itself [5], [6], can benefit greatly from an automated server overload prediction mechanism. Firstly, having such a mechanism allows them to take action before they will actually be overloaded. This will result in a more consistent quality of service as an overloaded server will likely negatively impact the performance of the entire system. Secondly, such a mechanism also allows to avoid overload, by taking the decision to implement an overload prevention mechanism just before overload is likely to occur.

Combining the ideas from the realms of multi-tenancy and self-adaptiveness, it is our aim to investigate performance prediction solutions, which will ultimately assist in preventing server overload conditions, thus keeping the users of the application happy.

In this paper, we take a first step towards an automated overload prediction mechanism by implementing and evaluating an overload detection approach. Our approach is based on measuring a wide variety of so-called performance counters [7], such as the Memory\Available Mbytes and Processor\%Processor Time counters. Rather than defining exact threshold values for the monitored performance counters, we propose to use statistical pattern recognition, which can assist with classifying complex performance counter patterns. This allows us to recognize complex correlations between performance counters, rather than just simple overload caused by an extreme value of a performance counter, for example, not enough free memory.

The outline of this paper is as follows. In Section II, we will motivate the problem and present our research questions. Section III gives a high-level description of our approach for server overload detection using pattern recog-
Section IV, the implementation of our approach will be discussed. The empirical evaluation of this implementation will be presented in Section V and discussed in Section VI. We will conclude our paper with a discussion of related work in Section VII and future work in Section VIII.

II. PROBLEM STATEMENT

While ideally a SaaS application can be installed on one server, both the complexity of the application and the limited performance capabilities of the underlying hardware platform are likely to require that the application is distributed over a number of servers.

An interesting problem is then, when the application should be scaled up, i.e., when the underlying hardware should be replaced and/or expanded. Intuitively, we want to scale the application before the user notices a decrease in performance. Because scaling an application usually takes time, due to factors like hardware relocation and/or installation, we must be able to predict a server overload with a margin in the order of several days to several weeks ahead. While the actual upscaling, i.e., ordering new hardware and installing it, is not part of the core challenges of self-adaptiveness, the underlying principle of “sensing” [6] the overload situation is.

An important step towards overload prediction is the ability to detect it. This paper will focus on overload detection using performance metrics and thus the following research question:

RQ: How can we detect server overload by monitoring performance metrics?

Overload is defined as the point when the demand on at least one of the servers resources exceeds the capacity of that resource [8]. A first challenge lies in automatically detecting overload. As our goal is to predict server overload in order to provide a better customer experience, we focus on detecting it from a customer’s point of view, which can be regarded as enforcing a Service Level Agreement. An important metric for application users is server response time. If an application regularly takes too long to respond, the customer may become unsatisfied and he may eventually switch to another service provider. Therefore, we will use the response time as an important indication for server performance. However, doing this has some challenges, which will be discussed in the following paragraphs.

Feasibility. To get details about the current performance of a server, we can inspect its performance counters [7]. These performance counters exhibit information about all the components of the server, such as CPU usage, harddrive writes per second or the number of requests in the web server queue. Because accessing these performance counters is relatively cheap, they provide a platform for performance monitoring on regular intervals.

For a human expert it is usually possible to detect a performance problem using intuition, but it is costly and error-prone to monitor a server using human expertise only. However, automating this process is not straightforward. We must first decide on whether the performance counter values of an overloaded state differ enough from those of a normal state, so that an automated detection mechanism can function robustly enough.

RQ1: Do the performance metrics of a server provide enough information to detect that it is in an overloaded state?

Selecting the metrics. Assuming we can use performance metrics to detect overload, the next step is to select the set of performance metrics to measure.

RQ2: Which server performance metrics should be monitored?

An observation is that this set may be different for different types of servers, such as web and database servers.

Defining thresholds. After identifying the set of metrics to monitor, it is necessary to define the thresholds at which they represent an overload state.

RQ3: What are the thresholds for these server performance metrics?

Defining thresholds for these metrics is non-trivial, because overload may be indicated by different performance metrics patterns, e.g., a memory overload will exhibit different performance metric values than a CPU overload. Note that we define a (performance counter) pattern as a set of performance counter values, which represent the state of a server at a specific time. In addition, specifying a threshold using concrete values is difficult because many combinations of performance metrics exist and defining all combinations which lead to overload can be problematic.

In the rest of this paper we will propose and evaluate an approach for automatically detecting server overload, taking the aforementioned challenges into account.

III. OUR APPROACH

The goal of our approach is to be able to automatically decide whether a server is in an overloaded state using performance metrics measurements only. In this paper, we propose to use statistical pattern recognition for overload detection.

A. Statistical Pattern Recognition

Statistical pattern recognition is a term used to cover all stages of an investigation from data collection to classification and interpretation of the results [9]. A special case of pattern recognition is classification, which deals with the classification of patterns into classes. In this study, we are specifically looking into classifying performance patterns into two classes, namely normal and overload.
In pattern recognition a pattern is described by features. Features are often measurements of real-world properties of an object, such as the performance counters of a server. An untrained classifier must first be trained with a dataset with patterns of all classes to be able to classify patterns. The result, a trained classifier, can be used to classify new patterns. Throughout this paper the term classifier will be used for both the trained and untrained classifier when the meaning is clear from the context.

An example of a simple classification problem is shown in Figure 1. There are 100 objects, 50 in each class, and 2 features. Based on these measurements, a linear classifier is trained. The trained classifier is shown by the bold line. A new object or pattern can be classified by looking at where the object will be with respect to the line (top left = class 1, bottom right = class 2).

B. Stages of Our Approach

The exact parameters used during statistical pattern recognition, such as which algorithms to use, are dependent of the dataset under investigation. Therefore, our approach consists of a number of steps, which help with setting and configuring these parameters, rather than defining a predefined algorithm for overload detection. In our approach we distinguish the following stages:

1) Data collection
2) Investigate whether the normal and overload classes in this dataset are separable using pattern recognition (RQ1)
3) Classifier design (RQ3)
4) Classifier evaluation
5) Classifier optimization1 (RQ2)

Please note that after optimization of the classifier it is necessary to reevaluate it to make sure the optimizations had the desired result. In the next section we will explain our implementation of each of these stages.

1Note that by optimization we mean optimization of the problem with respect to classifier performance or speed, not optimization of the classification procedure itself.

IV. DETECTING SERVER OVERLOAD

In this section, we will discuss the implementation of our approach following the five stages defined in the previous section.

A. Data Collection

In order to train a classifier to detect overload patterns, we need performance counter patterns which describe normal and overloaded states. Therefore, the first step of our approach is collecting data from the server to monitor. Ideally, data can be collected from a production server with real overloaded states. However, it can be difficult to collect enough such data as these are states we are trying to avoid in a real system. In addition, it is difficult to control which overloaded states are reached. Therefore, we have created synthetic load tests which generate overload on a similar server, so that 1) the specifications of the servers are (almost) equal but the production server does not have to reach the overload state and 2) we have more control over reaching an overload state.

For our training data, we are interested in two measurements:

- The average response time – This is used to decide whether a performance counter pattern should be classified as normal or overload state
- The performance counter pattern – This is used to describe the state

After collecting the training data, we must first transform it for use with a classifier. For a dataset containing response time and performance counter pattern as described in the previous section, this means the performance counter patterns must be assigned to a target class (overload or normal) based on the response time.

B. Feasibility

To get a hint of whether pattern recognition can be used on the dataset we can inspect the data recorded in a graph. Figure 2 graphs part of the data recorded during an overload state. The bar below the graph indicates the normal and overload states. When we look at these metrics, intuitively we see a large difference between the normal and overloaded states, which should allow to use classification based on performance counter patterns.

C. Classifier Design

A number of different classifiers exist [9] and some are more suitable for particular datasets than others. For instance, the data in Figure 1 can be separated relatively well using a linear classifier. However, for a different data distribution, this might not be the case. Understanding the data distribution is therefore helpful in selecting a suitable classifier. In 2 or 3 dimensions, much can be understood by plotting the data, but in more dimensions, this analysis is not as straightforward. Therefore, we have decided to
heuristically search for the best classifier. First we have tried the cheaper classifiers, to test whether they would give sufficiently good results. After this, the more expensive ones were tried for comparison.

For our implementation we have used PRTools [10], a pattern recognition toolbox for MATLAB 2, developed at the Delft University of Technology. PRTools is suitable for our study, as it has implemented a number of classifiers. We implemented Algorithm 1 in MATLAB and use PRTOOLS to train a classifier on a random part of the training data (trainSet) and to validate it on the remaining part (testSet). This is repeated several times for all classifiers we want to try, so that we can calculate the mean error (mean) and standard deviation (std) of their classification and to avoid bias of a training set. Finally, we select the classifier with the smallest error and standard deviation as our baseline performance classifier. Note that the standard deviation is important as this describes the stability of the classifier over a couple of runs on different portions of the data. Table I shows the classifiers tried by our implementation. This table contains some commonly used classifiers in the pattern recognition field.

2http://www.mathworks.com/products/matlab/

### Table I
**CLASSIFIERS TRIED**

<table>
<thead>
<tr>
<th>Function name</th>
<th>Classifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldc</td>
<td>Linear Bayes Normal Classifier</td>
</tr>
<tr>
<td>qdc</td>
<td>Quadratic Bayes Normal Classifier</td>
</tr>
<tr>
<td>nmc</td>
<td>Nearest Mean Classifier</td>
</tr>
<tr>
<td>perlc</td>
<td>Linear Perceptron Classifier</td>
</tr>
<tr>
<td>udc</td>
<td>Uncorrelated Normal Based Quadratic Bayes Classifier</td>
</tr>
<tr>
<td>kldc</td>
<td>Linear Classifier built on the KL expansion of the common cov. matrix</td>
</tr>
<tr>
<td>loglc</td>
<td>Logistic Linear Classifier</td>
</tr>
<tr>
<td>knnc</td>
<td>K-Nearest Neighbor Classifier</td>
</tr>
<tr>
<td>parzenc</td>
<td>Optimized Parzen Classifier</td>
</tr>
</tbody>
</table>

Algorithm 1 Search best classifier

```matlab
function classifier = GetBestClassifier(err, std, classifiers)
    classifiers = GetClassifiersList();
dataSet = GetDataSet();
split = GetNumberOfRepetitions();
for i = 0 to repeat do
    dataSet.Shuffle();
    (trainSet, testSet) = dataSet.Split(split);
    classifier = TrainClassifier(c, trainSet);
    err[] = TestClassifier(classifier, testSet);
end for
mean[c.Name()] = mean(err);
std[c.Name()] = std(err);
empty(err);
return GetBestClassifier(err, std, classifiers);
end for
```

1) **Optimizing the Classifier and Dataset:** After determining the baseline performance for a classifier, the classifier and dataset can be optimized. We have used three forms of optimization. Note that it is necessary to reevaluate a classifier after applying such a measure, as the possibilities of optimization are dependent on the dataset. Evaluation can be done on the following variables:

- Mean error (classifier accuracy)
- Standard deviation (classifier stability)
- Time (classifier speed)

Selecting the best classifier based on these variables is a tradeoff between accuracy and time and dependent of the application. For example, a real-time performance monitoring application may prefer a faster but slightly less accurate classifier, while a medical application may prefer accuracy over time.

**Data Scaling:** Some classifiers are sensitive to very large or small values, such as the nearest neighbor classifier [11]. As this classifier calculates the distance between objects, the results become heavily biased when features...
with very large or small values are used. Therefore, the dataset should be scaled. The data values are transformed such that for each feature, the mean is 0 and the variance is 1.

**Lower Number of Samples:** Our baseline classifier is trained using performance counter patterns which were recorded every second. This has 2 disadvantages:

- The dataset becomes large and the classifier becomes slower
- Monitoring is expensive

Part of our research deals with the question whether we can optimize the dataset by recording less samples.

**Feature Selection:** In addition to reducing the number of samples in a dataset, we can also optimize the number of columns, which represent features. One way to do this is using feature selection. Using PRTOOLS, we can rank all the features of our dataset based on the 1-Nearest Neighbour leave-one-out metric. The top entries of this ranking are likely to have a larger influence on the classifier than the lower entries. Therefore, the dataset may be optimized by removing the lowest-ranked features. Please note that, as feature selection is done without knowledge of a classifier, the ranking may not be optimal for all classifiers.

**One-Class Classification:** A special form of classification is one-class classification [12]. In one-class classification, one class is considered as the target class, while all other classes are outliers. An advantage of one-class classification is that no outlier patterns are needed to train the classifier, as everything that cannot be classified as an element of the target class is automatically considered an outlier. This makes this type of classification interesting for overload detection, as no examples of overload patterns have to be collected. Therefore, we are going to investigate whether one-class classification inducers should be included in the search for the best classifier. For our implementation we have used the MATLAB toolbox DD_TOOLS [13], a toolbox specialized in one-class classification. Table II shows the classifiers we have included in our heuristic search.

The performance of a one-class classifier is measured using the area under the ROC-curve (AUC) [14]. The ROC-curve ("receiver operating characteristic"-curve) graphs the true positive rate (or sensitivity) against the false positive rate (or 1-specificity). This means that the closer the AUC is to 1, the better the test as this indicates high sensitivity and high specificity.

Our MATLAB implementation for searching for the best one-class classifier implements Algorithm 1, with the small adjustment that functions from DD_TOOLS are used instead of those of PRTOOLS. In addition, we have evaluated all the optimization measures discussed in this section.

### V. Empirical Evaluation

We have evaluated our approach by applying it to a synthetically generated dataset. In this section, the results of this evaluation are presented.

#### A. Setup

We have generated our dataset on a Windows server, which allows to use PerfMon, Microsoft’s integrated system monitor, to read performance counters. As a portion of the complete set of performance counters is very unlikely to be of interest when monitoring for overload, we have selected a list of counters of which we feel they should be significant for overload detection (see Table III). An example of a group of performance counters which is not of interest is the Windows Media Player Metadata group, a group of performance counter which gives more information about the Windows Media Player. As this player is very unlikely to be used on a server, this counter does not tell anything about the state of the server, and it can be safely ignored by our approach.

To collect performance counter patterns which represent overload, we have synthetically created overload situations on a test server for this paper as a proof of concept. We have created 8 test cases in the form of web pages which should cause an overload situation when requested by a large number of users. These test cases were created with the intention of impacting the values of the performance counters listed in Table III. Table IV shows the type of impact caused by each test case. Note that these test cases are not a definite list of causes for server overload but just a small subset used as a proof of concept.

For every test case, we have created a load test in Apache JMeter, an open source performance testing tool. During this load test the load was increased until the average response time was over 20 seconds. This threshold was chosen for our proof of concept to make sure a large amount of performance data was available.

We have configured PerfMon to record a performance counter pattern every second. The average response time was calculated from the log of our webserver, Microsoft’s Internet Information Server (IIS).

<table>
<thead>
<tr>
<th>Function name</th>
<th>Classifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>gauss_dd</td>
<td>Gaussian Data Description</td>
</tr>
<tr>
<td>kmeans_dd</td>
<td>K-Means Data Description</td>
</tr>
<tr>
<td>kcenter_dd</td>
<td>K-Center Data Description</td>
</tr>
<tr>
<td>kmdd</td>
<td>K-Nearest Neighbour Data Description</td>
</tr>
</tbody>
</table>

Table II

**ONE-CLASS CLASSIFIERS TRIED**

For information, see the help of the PRTOOLS MATLAB toolbox available at http://www.prtools.org/

---

3For information, see the help of the PRTOOLS MATLAB toolbox available at http://www.prtools.org/

4See http://jakarta.apache.org/jmeter/

5See http://www.iis.net/
### Algorithm 2 Transform recorded data for classifier

```plaintext
overloadThreshold ← 20
avgResponseTime ← 0
samples ← GetPerfMonData()
samples ← GetIISDataIterator()

for all s in samples do
    sampleTime ← s.GetTime()
    while (rt ← responseTimes.Next()) < sampleTime do
        avgResponseTime ← (avgResponseTime + rt)/2
    end while
    overload ← avgResponseTime > overloadThreshold
    AddToTrainingsSet(overload, s)
end for
```

The classifier. This is repeated $n$ times, after which the dataset is shuffled and $n$-fold cross-validated $(m - 1)$ times.

After all classifiers were validated, the best one was selected based on mean error, standard deviation and time elapsed for the classification. Finally, all other classifiers were compared to the best one ($\text{clasf}_{best}$) using a t-test [16] to show that their results were indeed significantly different from the best one. Note that due to the nature of the t-test this is for $p < 0.05$ and $p > 0.95$. The t-test was setup as follows:

$$H_0: \text{clasf}_{best} = \text{clasf} \quad \text{(on the same train and test set)}$$

$$H_1: \text{clasf}_{best} > \text{clasf} \quad \text{(on the same train and test set)}$$

$H_0$ is rejected for $p < 0.05$ or $p > 0.95$.

The t-tests were implemented in MATLAB using `RESTOOLS`[^6].

1) **Baseline Performance Classifier:** The t-test is most reliable for comparing two classifiers, therefore we need a baseline classifier to compare all the other classifiers to. We have defined this by searching for the best classifier for the original dataset. Table V shows the search and t-tests for this dataset. During the heuristic search the $\text{qdc}$ classifier performed the best with low mean error and standard deviation values, but our results also show that $\text{qdc}$ to be relatively expensive, which is why we decided to select the second best classifier. Although the $\text{qdc}$ classifier has a larger mean error than the $\text{knnc}$ classifier, it is much faster ($\approx 48$), which is convenient for our overload detection application as we would like to use it in real-time eventually.

The t-tests (gray area in Table V) show that this is indeed the second best choice, as $H_0$ is rejected for all classifiers except $\text{knnc}$, which indicates that $\text{knnc}$ performs at least as good as $\text{qdc}$ [16]. This means that our baseline classifier has a mean error of 0.0659 with a standard deviation of 0.0145 and can perform $5x$-fold cross-validation in 3.0934 seconds, so any optimization made to the classifier or dataset should yield values lower than these. The $\text{parzen}$ classifier is very slow for non-scaled data, so this classifier was skipped in the t-test.

2) **Optimization 1: Data Scaling:** The first optimization tried was data scaling as explained in Section IV-C1. The results of 5x10-fold cross-validation are depicted in Table VI.

It is clear that most classifiers give better results on a scaled dataset. The best classifier, knnc was dismissed for the same reasons as explained in the previous section. Although the table shows good results for parzenc, the calculations were relatively expensive with respect to time. Therefore, we selected the ldc classifier as best classifier for the t-test. The t-test shows that this classifier is indeed better than most other classifiers, but performs similarly to the klldc classifier. As can be seen in the table, the mean error and its standard deviation are almost equal, but klldc is slower. As a result, scaling has improved our baseline performance of 5x10-fold cross-validation to a mean error of 0.0502 with a standard deviation of 0.0112 in 2.9977 seconds.

3) Optimization 2: Lower Number of Samples: To investigate whether a lower number of performance counter pattern samples would result in at least similar performance, we have regenerated the dataset by taking every 5th sample. In addition, we have recalculated the average response times for requests made between the samples. Table VII shows a drop in performance for most classifiers. Especially the standard deviation of the error becomes larger, which means that the classifier becomes less stable. Therefore, using every 5th sample does not optimize this dataset. However, it is possible that using a larger portion (but not all) of the samples may give satisfactory results.

4) Optimization 3: One-Class Classification: We have converted our dataset to a one-class dataset using DD_TOOLS. This type of classification uses different classifiers than for a dataset that is not converted to a one-class dataset. Table VIII shows the results of 5x10-fold cross-validation using these one-class classifiers. Note that, one-class classifiers are not compared using the error but the AUC, for which a value closer to 1 means a higher accuracy. From the table, we see that the knn gives high accuracy with a low standard deviation. As we cannot compare different metrics using a t-test, we will use these values as the baseline performance for one-class classification.

5) Optimization 4: Data Scaling in a One-Class Dataset: The results of scaling the one-class dataset are shown in Table IX. It is interesting to see that scaling has a negative effect on most one-class classifiers for our dataset. Although it improves the results of the gauss_dd classifier, it does not improve our baseline one-class classifier.

6) Optimization 5: Lower Number of Samples in a One-Class Dataset: Like with the other dataset, our one-class dataset with 5-second instead of 1-second samples deteriorates the classification results. Therefore, we did not lower the number of samples in the one-class dataset.

7) Optimization 6: Feature Selection: Although reading performance counter values is relatively cheap, recording x-
**Performance counter**

| Memory\Pages/sec | ASP.NET Applications\(\_Total\)\RequestSec | Memory\Pool\Nonpaged\Bytes | ASP.NET Applications\(\_Total\)\Cache\Total\Turnover\Rate | Processor\Total\)\%Privileged\Time | Processor\Total\)\Thread\Count | Network\Interface\(\text{Intel(R)}\)\\(\text{Wi-Fi\ Link\ 5100\ AGN}\)\\Output\Queue\Length | Memory\%\Committed\Bytes\In\Use | \(\text{.NET\ CLR\ Memory}\(\_\text{Global}\)\)\%\Time\in\GC | ASP.NET\Applications\(\_\text{Total}\)\\Errors\Total | Memory\%Available\MByte | \(\text{.NET\ CLR\ Memory}\(\_\text{Global}\)\)#\ Gen\ 1\ Collections | Processor\Total\)\Private\Bytes | Network\Interface\(\\text{\Local\ Area\ Connection*}\ 21)\\Bytes\Total/sec |

| highest ranked performance counters |

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Name</th>
<th>Value</th>
<th>Name</th>
<th>Value</th>
<th>Name</th>
<th>Value</th>
<th>Name</th>
<th>Value</th>
<th>Name</th>
<th>Value</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private\Bytes</td>
<td>1622143464</td>
<td>Available\MByte</td>
<td>2784544</td>
<td>%\Committed\Bytes\In\Use</td>
<td>100</td>
<td>Pages/sec</td>
<td>11433</td>
<td>#\Gen\1\Collections</td>
<td>29635</td>
<td>%\Privileged\Time</td>
<td>100</td>
<td>%\Time\in\GC</td>
<td>100</td>
</tr>
<tr>
<td>%\Committed\Bytes\In\Use</td>
<td>100</td>
<td>Pages/sec</td>
<td>11433</td>
<td>#\Gen\1\Collections</td>
<td>29635</td>
<td>%\Privileged\Time</td>
<td>100</td>
<td>%\Time\in\GC</td>
<td>100</td>
<td>%\Time\in\GC</td>
<td>100</td>
<td>%\Time\in\GC</td>
<td>100</td>
</tr>
<tr>
<td>%\Privileged\Time</td>
<td>100</td>
<td>%\Time\in\GC</td>
<td>100</td>
<td>%\Time\in\GC</td>
<td>100</td>
<td>%\Time\in\GC</td>
<td>100</td>
<td>%\Time\in\GC</td>
<td>100</td>
<td>%\Time\in\GC</td>
<td>100</td>
<td>%\Time\in\GC</td>
<td>100</td>
</tr>
</tbody>
</table>

**VI. DISCUSSION**

In this section we will map the research questions that we set out in Section II to the results we obtained from our experiment. Furthermore, we also identify threats to validity that might impact our conclusions.

A. The research questions revisited

Feasibility (Revisiting RQ1). In our problem statement, we questioned whether performance metrics alone provide enough information to decide if a server is an overloaded state. After our evaluation, we can conclude that they do and that we indeed can use pattern recognition to do overload detection. We were able to create a classifier for our test dataset which was able to classify patterns with a mean error of less than 5% and a low standard deviation, which should be accurate enough for most applications.

However, it is necessary to be careful while selecting and optimizing a classifier. As our evaluation demonstrated, some classifiers performed much worse than others and not all optimization measures worked out as intended.

One can raise the question why we used performance counters instead of only the average response time in our approach. The reason for this is that in future work we will monitor applications that are spread over multiple servers. Using only the response time will not give us enough information about which server is in overloaded state, while a server-specific performance counter pattern can give us this information.

Selecting the Metrics (Revisiting RQ2). In our evaluation we have shown that it is possible to use feature selection on a dataset, which results in less variables with similar performance. This is an interesting optimization, because it saves resources both during the training phase of the classifier and during the monitoring of the production server.

Defining Thresholds (RQ3). The approach we follow does not need fixed thresholds for performance counters, but instead follows a more flexible approach that determines patterns of performance counters. Preliminary evidence shows that we can correctly classify performance patterns with a mean error of less than 5%, but we acknowledge that we should expand our research on larger and more diverse datasets.

An interesting possibility is the use of one-class classification, especially because it does not need overload states to train the classifier. For our dataset the results looked very promising.

Scalability. Our test dataset is approximately 8 MB in size with nearly 3500 training patterns. In most cases, classification was very fast. We expect that the classifiers which were slow for our dataset are also slow for larger datasets. To ensure that the faster classifiers perform fast for larger datasets as well, more research has to be done. Because both the design of the classifier and the classification can be done on an isolated server, the overhead on the production server will be low.

Different Applications. Our approach is lightweight and transparent; it requires no modification of application code as measurements are done at the operating system level. The approach is also application-independent. By monitoring a different subset of performance counters, it can also do overload detection for different types of servers, such as database servers. Although we have evaluated our approach on Windows, it can be used on any type of operating system, as long as it offers a way of reading performance counter values.

B. Threats to Validity

We have already touched upon some of the issues concerning external validity in the above discussion. As far as the internal validity is concerned, we have used 5x10-fold cross-validation to avoid bias in our classifier results. In addition, we have used t-tests to compare classifiers on the same training and test set to avoid bias in the comparison.

In our experimental setup we used synthetic scenarios to create normal and overload situations. While we ac-
knowledge that these synthetic scenarios do not exhaustively cover all possible scenarios, we did create the synthetic scenarios to be representative of realistic overload scenarios. In particular, the test cases were designed together with a performance expert with the goal of impacting the monitored performance counters.

With respect to reliability, the MATLAB code used to do the cross-validation and t-tests will be made available for download from our website\(^7\). PRTOOLS, DD_TOOLS and RESTOOLS are open source toolboxes for MATLAB.

A final validity threat is the size of our dataset. In particular, during our experiment we noticed that the dataset was too small to optimize it by lowering the number of samples. In order to fully understand the effect of lowering the sampling rate, we should expand on our current research and investigate larger datasets.

**VII. RELATED WORK**

Existing research on using statistical pattern recognition and machine learning for server overload detection and prediction is mostly focused on preventive mechanisms, such as admission control and load balancing. This section provides an overview.

Admission control is a procedure, which allows a certain number of requests and rejects remaining requests to avoid overload, as this can lead to system failure. Fontaine et al. use a combined model of a neural network and feedback control to implement admission control for the Apache web server [17].

Load balancing algorithms try to spread the load of requests equally over a number of servers. Jia and Sun [18] use performance counter values as input for a neural network. Dantas and Pinto [19] use a classifier system to do load balancing for processes in a cluster of systems. A classifier system is a special type of genetic algorithm which can learn rules to guide the evolution.

The main concern we have with using only a server overload prevention mechanism such as admission control or load balancing, is that such mechanisms do not advise us on when to scale a system up. While they can reassure that all servers have to handle roughly the same workload, they cannot tell us when this workload becomes too high. Our approach could function complementary to these mechanisms to improve feedback from the server. In addition, our approach can be used for automated testing of such mechanisms.

Correa and Cerqueira [20] use statistical approaches to predict and diagnose performance problems in component-based distributed systems. For their technique, they compare decision tree, Bayesian network and support vector machine approaches for classifying. In contrast to our own work, Correa and Cerqueira’s work focuses on distributed systems, making network traffic an important part of the equation.

**VIII. CONCLUSION**

In this paper we have investigated whether statistical pattern recognition can be used to successfully classify performance counter patterns into the *normal* or *overload* class. We have proposed an approach for searching for a well-performing classifier and we have empirically evaluated this approach on a synthetically generated dataset. In short, our paper makes the following contributions:

- An approach for finding a suitable, well-performing classifier for a dataset
- Steps for improving the accuracy and speed of this classifier
- An empirical evaluation of our approach
- An open source implementation of our approach as a toolbox for MATLAB, Java tools and a JMeter test suite

\(^7\)http://swerl.tudelft.nl/bin/view/Main/MTS
Revisiting our research questions:

**RQ1** Do the performance metrics of a server provide enough information to detect that it is in an overloaded state? Using synthetically created load test scenarios, we have shown that performance counters alone provide enough information to decide whether a server is in an overloaded state.

**RQ2** Which server performance metrics should be monitored? We have also shown how to define and optimize the set of performance counters to monitor. In particular, using feature selection we managed to isolate the most important performance counters for classification purposes, thereby reducing the number of performance counters that we need to measure from 66 to 14, while keeping good performance.

**RQ3** What are the thresholds for these server performance metrics? Finally, we have shown that by using statistical pattern recognition for overload detection, we do not need to define concrete thresholds for performance counter metrics, but can use statistics to reason about their values instead.

### A. Future Work

In future work we will evaluate our approach on a large dataset, collected by monitoring an industrial SaaS application. In addition, we aim to extend our approach with prediction abilities. During this industrial case study, we will also extend our approach with support for overload detection and prediction for applications running on multiple servers. Finally, we will investigate the impact of more optimization techniques, such as combining features or classifiers.

**ACKNOWLEDGMENT**

The authors would like to thank Exact for providing the funds and opportunity to perform this research. Further support came from the NWO Jacquard ScaleItUp project.

**REFERENCES**


