A Machine Learning-based Framework for Building Application Failure Prediction Models

Alessandro Pellegrini
pellegrini@dis.uniroma1.it
DIAG – Sapienza, University of Rome

Pierangelo Di Sanzo
disanzo@dis.uniroma1.it
DIAG – Sapienza, University of Rome

Dimiter R. Avresky
autonomic@irianc.com
IRIANC – Munich, Germany

***Abstract*—In this paper, we present the *Framework for building Failure Prediction Models* (F^{2}PM), a Machine Learning-based Framework to build models for predicting the Remaining Time to Failure (RTTF) of applications in the presence of software anomalies. F^{2}PM uses measurements of a number of system features in order to create a knowledge base, which is then used to build prediction models. F^{2}PM is application-independent, i.e. it solely exploits measurements of system-level features. Thus, it can be used in differentiated contexts, without the need for any manual modification or intervention to the running applications. To generate optimized models, F^{2}PM can perform a feature selection to identify, among all the measured system features, which have a major impact in the prediction of the RTTF. This allows to produce different models, which use different set of input features. Generated models can be compared by the user by using a set of metrics produced by F^{2}PM, which are related to the model prediction accuracy, as well as to the model building time. We also present experimental results of a successful application of F^{2}PM, using the standard TPC-W e-commerce benchmark.**

I. Introduction

In computing systems, one significant cause of availability and performance degradation is the accumulation of anomalies of different nature. A large part of these anomalies is often associated with errors and/or sub-optimal implementations of applications, which may lead to the occurrence of, e.g., memory leaks, unterminated threads, unreleased locks, file fragmentation. These phenomena have been widely observed in different contexts, revealing an average percentage of 40% of anomalies being due to errors in the software development [**?**]. The accumulation of these kinds of anomalies can cause exhaustion of system resources over time, and might lead to incremental loss of performance, or even hang/crash of the hosting system. The effect of resource exhaustion can be particularly strengthened in the case of long-running applications (as in the case of many applications hosted in web/application servers), which can be subject to the accumulation of a large number of anomalies over time. Most of the time, the occurrence of anomalies in applications is unpredictable and their causes are complex to figure out. Additionally, in some cases, unearthing the causes would require a huge effort or, even, could be cost-ineffective. In these cases, alternative approaches to address accumulation of anomalies are preferred, such as executing proper correcting actions aimed at removing their effects. For example, a largely adopted technique is *software rejuvenation* [**?**], which consists of forcing the state of the application/system to a "clean" state, i.e. a state where the system/application is known to work without the presence (or with reduced number) of anomalies. Typical actions for cleaning up the state include restarting the application or rebooting the system.

Generally, being able to predict the occurrence of anomalies (as well as the effects associated with their accumulation) can help to improve both the performance and the availability of systems. In fact, proper actions could be executed in advance to prevent upcoming system failures or excessive performance degradation. In the case of software rejuvenation, this kind of proactive approach is referred to as *proactive rejuvenation*, which consists of preventively forcing the application or the hosting system to a clean state before the time when, e.g., a crash is predicted to occur.

In this paper, we present the *Framework for building Failure Prediction Models* (F^{2}PM), a Machine Learning (ML) based Framework aimed at building system failure prediction models. F^{2}PM allows to generate optimized ML models to predict when a given (abnormal) condition is expected to occur. We generally call this time *Remaining Time To Failure* (RTTF). The condition can identify different kind of system failures. Specifically, it can be defined by the user on the basis of the values of one or more selected system features, which can reveal that the system is approaching, e.g., a hang/crash point or is working in a sub-optimal way (e.g., it is providing very low performance). F^{2}PM operates in a non-intrusive way, i.e. any kind of instrumentation of applications (such as inserting probes for collecting data) is not required, thus being completely *application-agnostic*. In fact, F^{2}PM only exploits system-level features, which can be monitored by using simple tools typically included also in the basic versions of all common operating systems. This makes F^{2}PM of general usability. Since our proposal is based on Machine Learning techniques, its applicability ranges to all contexts where it is possible to collect in advance a sufficient number of observations of the monitored phenomena. Essentially, the higher the occurrence rate of anomalies/faults in the monitored system, the quicker is the generation of a reliable prediction model (coherent with the system under monitoring).

F^{2}PM relies on a preliminary system observation phase. During this phase, a number of system features are monitored, and their values are recorded, while the application responsible of generating anomalies runs. Every time the condition defined by the user is met, F^{2}PM logs the occurrence time, and the system is restarted. Then, collected data are used for building and validating a number of models generated by using different ML algorithms. Before building the models, F^{2}PM also performs two additional steps: 1) the values of a number of derived metrics, calculated on the basis of the collected system

features, are added to the collected data, and 2) a data selection step is executed, where a number of training sets (including different sub-sets of features and metrics) are extracted from the data set. Then, a number of models are generated by applying ML algorithms to the different training sets. For each model, F^{2}PM provides a set of metrics measuring the accuracy and the model training/validation time, thus offering the user the possibility to select the best suited model on the basis of the model accuracy and building time. Particularly, the set of metrics also includes the Soft-Mean Absolute Error (S-MAE), which evaluates the absolute prediction error assuming that an error below a given user-defined threshold T is acceptable (i.e., an error less than T is not considered in the evaluation of the metric). S-MAE is particularly useful when performing proactive management of system failures. In fact, in this case, if a correcting action is executed at time T before the system is predicted to fail, a prediction error less than T would be tolerated.

We also show how F^{2}PM can be instantiated in the case of web applications. However, we note that F^{2}PM is designed to be used independently of a specific kind of application and type of anomaly. In fact, by changing the set of observed system features and defining a proper condition to be met for considering the system as failed, F^{2}PM can be customized for different systems and applications.

Finally, in order to highlight the usefulness of F^{2}PM, we note that software rejuvenation has been shown to be en effective technique when using virtualization [**?**]. Given the widespread adoption of virtualized and cloud computing architectures, this extremely broadens the scope of usage of F^{2}PM.

The remainder of this paper is structured as follows. In Section II we discuss related work. Section III describes the organization of F^{2}PM, and the design principles behind its development. Finally, in Section IV we present and discuss experimental results collected by using F^{2}PM in the case of a web application, where an implementation of the TPC-W benchmark [**?**] hosted on top of Apache Tomcat is used.

II. Related Work

Predicting the effect of application anomalies is not a new idea. Along this path, several works have already proposed prediction techniques and models [**?**].

In [**?**], the authors propose a proactive prediction and control system for large clusters. The proposal relies on logs containing six types of events categorized into classes (e.g. the availability of specific systems, or performance violation thresholds) and collected during one year of activity of a large (350 nodes) system. By using time series, rule-based classification, and Bayesian networks, the authors filter the initial data, selecting only the entries, which are useful to carry on a prediction. Essentially, as opposed to the above-mentioned work, in this paper we devise a framework, which is able to autonomously derive a set of different prediction models, enabling the user to select the best-suited one.

A framework to automatically detect anomalies and track the performance of an application is proposed in [**?**]. The framework relies on a regression-based transaction model, which reflects the resource consumption model of the application. The proposal explicitly monitors CPU usage per transaction, thus requiring a more in-depth analysis of the application. The main difference with respect to F^{2}PM is that it does not require to insert any kind of probe in the applications, thus extremely simplifying its instantiation ad usage.

In [**?**], a framework to support rejuvenation of virtualized systems upon the prediction of upcoming crashes is proposed. This framework has some common features with ours, including the calculation of derived metrics and the feature selection, based on Lasso Regularization. Anyway, there are substantial differences. In the framework in [**?**] decision rules are generated at runtime in order to support software rejuvenation. Conversely, F^{2}PM is able to produce different ML models, including non-linear ones, whose training is performed starting from different sets of (filtered or unfiltered) training data. Hence, F^{2}PM enables the user to compare different models and select the best one on the basis of a set of metrics, as we have discussed in Section I. Additionally, prediction models produced by the framework in [**?**] have been validated using only synthetic anomalies injected in the system, and in the case of memory leaks only. Instead, we show F^{2}PM to be effective with a combination of different anomalies, also occurring at different rates, and we demonstrate F^{2}PM in the case of a real-world application.

There is a set of commercial tools [**?**], [**?**], [**?**] which can be used to monitor Java applications by instrumenting the JVM. While this allows for a non-intrusive instrumentation to analyze transactions' performance via the reconstruction of their execution path, the approach is non-general, as it focuses only on Java-based applications. We keep the ability to monitor the evolution of this kind of applications, while broadening the application scope to *any* language used to build the (virtualized) service, as we operate in an application-agnostic way.

A work similar in spirit to what we show here is the one in [**?**], where an evaluation of a set of well-known Machine Learning classifiers for software anomalies is presented. In [**?**], three different states (*all ok*, *warning*, and *danger*) of the life of a software system are defined, and the generated models are only able to predict in what state a system is currently expected to be. Differently, we are able to generate models to precisely estimate the RTTF.

III. The F^{2}PM Framework

F^{2}PM is intended to build optimized ML models to predict the occurrence of system failures. A model receives as input the values of the set of selected system features and provides the RTTF, where the failure condition is defined by the user. F^{2}PM is based on the workflow shown in Figure 1. For the sake of simplicity, in our presentation we refer to the case of a generic web application, where we assume that memory leaks and unterminated threads may occur at different rates during the lifetime of the application. However, as we discussed in Section I, F^{2}PM is not limited to this kind of applications and this kind of anomalies. In the following, we go into details by describing F^{2}PM's workflow through a sequence of phases.

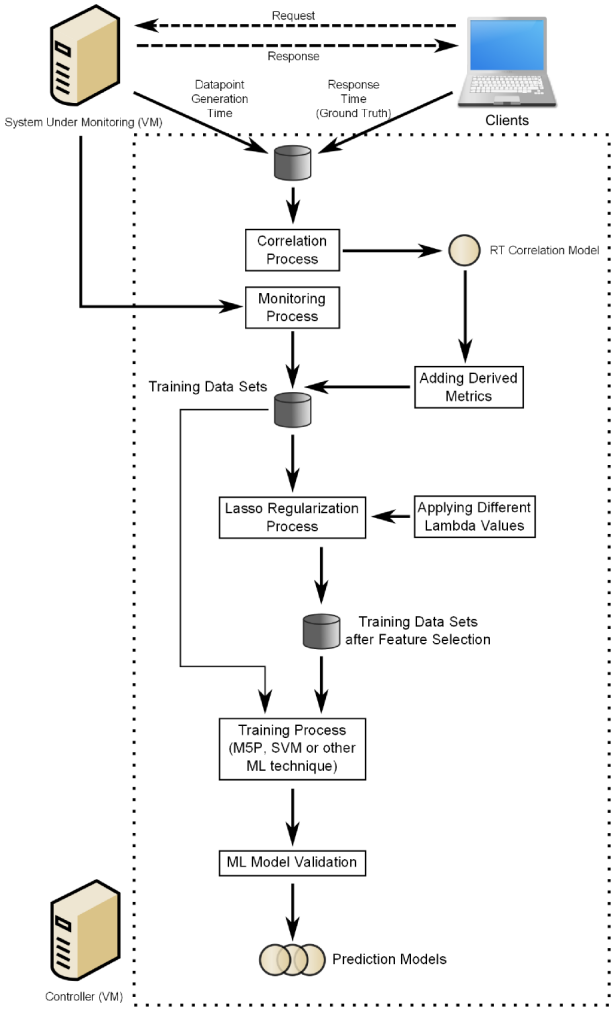


Fig. 1. F^{2}PM Architecture

A. Initial System Monitoring

As suggested in Section I, the initial system monitoring phase consists of collecting measurements of a number of system features while the system runs the application generating anomalies. All monitored system features are periodically measured, with an interval established by the user. Each measurement (datapoint) is timestamped with the elapsed time from system start. Hence, a data history is created by F^{2}PM, including the sequence of all datapoints. Every time the system failure condition is met, a *fail* event is added to the data history and the system is restarted. This gives rise to a number of runs of the system. We note that the duration of this phase can be very different for different systems, mainly depending on the anomaly occurrence rate(s) and the amount of available resources in the system. Particularly, a given amount of data, which would be sufficient to build ML models with a given accuracy, has to be collected. Determining the size of the dataset to be collected in this phase could require a long period of training time. F^{2}PM can support this task incrementally, via the set of metrics that allow the user to evaluate the accuracy of the produced models. If the estimated accuracy is not sufficient, further system runs can be executed to collect new data into the training set, and to produce new models.

Each datapoint consists of a tuple including the following set of values:

- T_{gen} is the timestamp denoting the elapsed time since the system has started;
- n_{th} is the number of active threads in the system;
- M_{used} is the amount of memory used by applications running in the system;
- M_{free} is the amount of memory freely available for usage by applications;
- M_{shared} is the amount of memory used for buffers shared by applications;
- M_{buff} is the amount of memory used by the underlying operating system to buffer data;
- M_{cached} is the amount of memory used to cache disk data;
- SW_{used} is the amount of swap space, which is currently used;
- SW_{free} is the amount of swap space, which is currently free;
- CPU_{us} is the percentage of CPU time dedicated to userspace processes;
- CPU_{ni} is the percentage of CPU time occupied by user-level processes with a positive nice value (lower scheduling priority);
- CPU_{sys} is the percentage of CPU time spent in kernel mode;
- CPU_{iow} is the percentage of CPU time spent waiting for a I/O operations to complete;
- CPU_{st} is the percentage of time a virtual CPU waits for a real CPU while the hypervisor is servicing another virtual processor;
- CPU_{id} is the percentage of CPU time spent doing unfruitful work (i.e., the system is underloaded).

We note that we selected the above listed system features because, on basis of them, we can potentially measure the effect on the system of the kind of anomalies affecting the application that we are studying (i.e. memory leaks and unterminated threads). However, on the basis of the kind of anomalies to be address, the user can change the system features to be used. The output of this phase includes a set of row data representing the evolution of the system feature along a number of system runs.

B. Datapoint Aggregation and Added Metrics

In the first step of this phase, aggregated datapoints are generated on the basis of a user-defined time interval. The aggregation is done according to the scheme shown in Figure 2. Each input datapoint (shown in black in the figure) is placed, on the basis of the value of T_{gen}, on the time axis. Hence, all datapoints falling in the same time interval are used to generate one aggregated datapoint, meaning that all the values of the system features are averaged in the aggregated datapoint. The subsequent step consists of adding some metrics to each the aggregated datapoint. Specifically, for each system feature j, the slope is calculated according to the following formula:

$$slope_j = \frac{x_j^{end} - x_j^{start}}{n}, \quad (1)$$

where x_j^{start} and x_j^{end} are the values of the feature j of the first and the last original datapoint falling in the time interval

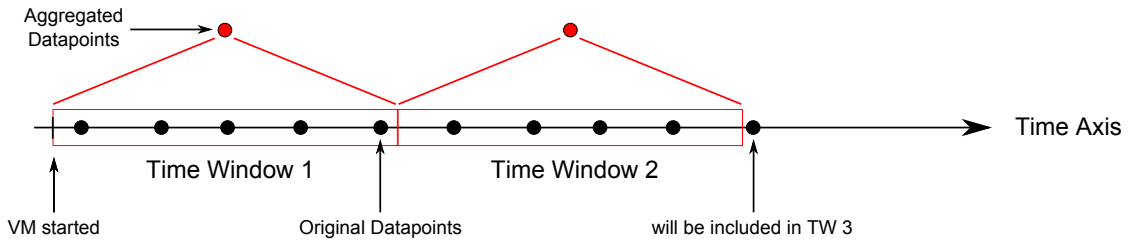


Fig. 2. Datapoint Aggregation

of the aggregated datapoint. Values of slopes are added to the aggregated datapoints. Introducing these slopes adds further knowledge about the dynamics of the system, which may show a highly variable behaviour. For example, some systems could show a constant increment of the resource usage over time until the crash point is actually approaching. At that time, some parameters could grow very quickly, even exponentially. The slopes, which could be interpreted as a simple approximation of a derivative function, in such a scenario might be proven effective to promptly detect an upcoming crash point. As a specific case, let us consider the above-specified SW_{used} feature. If the system crashes due to memory exhaustion, SW_{used} will start growing faster when approaching the crash point. Therefore, the slope can be used effectively to build the prediction model.

One motivation of aggregating datapoints along subsequent fixed-sized time intervals is that a more precise picture of the system behaviour along the time is achieved. In fact, the generation of original datapoints might incur in some skewing due to, e.g., the scheduler of the operating system, depending on the current workload. Particularly, as soon as the system is approaching the crashing point, this skew could have a higher impact, thus not providing a regular representation of the system behaviour along the time. An additional motivation is related to the fact that a large number of datapoints can require much time for a prediction model to be generated. Considering that many runs are required to build an accurate model, and taking into account the large number of datapoints in one single run, this time can be very high. Datapoint aggregation reduces the number of datapoints, without affecting the accuracy of the model.

Among the features stored in a datapoint, we include T_{gen}, namely the timestamp denoting the elapsed time since the system has started. F^{2}PM derives from T_{gen} an additional derived metric, namely the *inter-generation time* among two consecutive datapoints. This derived metric allows to capture the overload taking place in the monitored system. In fact, in case the system is overloaded, this inter-generation time is expected to grow, due to the increased load of the system. To demonstrate this, we have used WEKA [**?**] to carry on (using the fast Linear Regression [**?**]) a correlation process among the inter-generation time and the response time of the clients using the application[1] used to assess the proposed F^{2}PM (which will be later discussed in Section IV).

The correlation process builds a model for the Predicted

[1]The RT (ground truth) is measured from the clients, which have been instrumented using software probes just for this study. In fact, F^{2}PM does not require any modification to the software involved in the usage of the application.

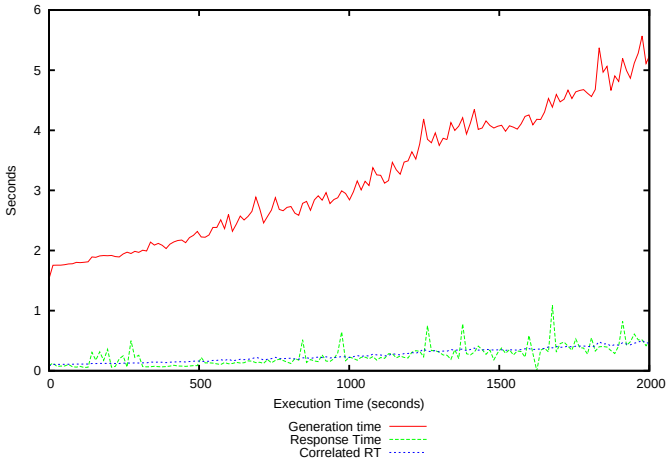


Fig. 3. Response Time Correlation

RT, namely an estimation of the actual RT experienced from remote clients, starting only from the measurement of the inter-generation time. In Figure 3, we report the results of this correlation process. Three curves are shown: the measured RT (referred to as Response Time), the measured inter-generation time of datapoints (referred to as Generation Time), and the outcome of the correlation model, evaluated on the inter-generation time of datapoints (referred to as Correlated RT). As it can be seen by the plots, both Generation Time of data points and RT (ground truth) are increasing when the system is experiencing memory leaks and unterminated threads. Therefore, the inter-generation time is a significant measure to capture the effects of the overloading of the system under monitoring. The user can specify a specific threshold for this additional feature, in order to fine tune the condition according to which F^{2}PM considers the system as failed. This correlation has a large impact, and can be applied to differentiated contexts. In fact, this technique can be effectively used, for example, to have a pragmatic estimation of the response time seen by end users, without any modification to the software at the end point.

Finally, at the end of the aggregation process, for each aggregated datapoint, the RTTF is calculated. This is done by exploiting the *fail* events, which have been added to the data history during the initial monitoring phase.

C. Features Selection

In this phase, the issue of selecting an optimal subset of system features and added metrics is addresses. For large/complex systems, where even thousands of features could

be involved (e.g., in some Cloud-related contexts [**?**]), this selection could significantly reduce the complexity of the model generation phase, having a simpler (likely more effective) representation of the system behaviour. In fact, this phase aims at identifying those features having (incrementally) more impact (weight) in the prediction of the RTTF. As shown in Figure 1, the execution of this phase is optional, so that the user is able to choose whether in the next phase F^{2}PM should consider the whole set of parameters, or only the ones selected during this phase. As it will be later discussed in Section IV, this choice can have effects on both the timeliness of the generation of the model, and on its accuracy.

Feature selection is based on Lasso Regularization [**?**]. By applying the Lasso Regularization method to our case, for a given vector $\bar{\lambda}$ of factors λ, we achieve as output a vector β, whose elements are the weights of the vector x_j, which minimizes the following objective function:

$$\frac{1}{n}\sum_{j=1}^{n} V(y_j, \langle \beta, x_j \rangle) + \lambda ||\beta||_1 \tag{2}$$

where n is the number of data points from the aggregation step, x_j is a vector of values of input features (independent variables) of each data point, y_j is the associated value of the dependent variable (RTTF) for the specific data point, and $V(y_j, \langle \beta, x_j \rangle)$ is equal to $(y_j - \beta^T x_j)^2$.

For each value of $\lambda \in \bar{\lambda}$, the calculated vector β includes a (sub-)set of non-zero elements. All features and added metrics associated with a zero element of the vector β are filtered out from the training set. Generally, while increasing the value of λ, more elements of the vector β are likely equal to zero. Particularly, these elements are likely those which have a smaller weight in the evaluation of the RTTF. Thus the effect of using higher values of λ is the reduction of the number selected features to be used in the ML models. The output of this phase is a number of training sets, each one including a sub-set of selected features and added metrics.

D. Model Generation and Validation

This phase aims at generating and validating a set of prediction models, which are built by using the training sets produced in the previous phases. We included in F^{2}PM six ML methods for building prediction models, namely Linear Regression [**?**], M5P [**?**], REP-Tree [**?**], Lasso as a Predictor [**?**], Support-Vector Machine (SVM) [**?**], and Least-Square Support-Vector Machine [**?**]. This set of methods includes both linear and non-linear ones. However, the set can be customized by the user by adding other methods or removing some of them. In the following, we provide a description of the above-mentioned methods.

Linear Regression [**?**] is an approach for modeling the relationship between a (scalar) dependent variable and one or more explanatory variables. Given a data set defined as $\{y_i, x_{i1}, \ldots, s_{ip}\}_{i=1}^{n}$ of n statistical units, a linear regression model assumes that the relationship between the dependent variable y_i and the p-vector of regressors x_i is linear. This relationship is modeled through a disturbance term or error variable ε_i—an unobserved random variable that adds noise to the linear relationship between the dependent variable and regressors. Thus the model takes the form:

$$y_i = \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \varepsilon_i = \mathbf{x}_i^{\mathrm{T}} \beta + \varepsilon_i, \quad i = 1, \ldots, n \tag{3}$$

M5P [**?**] is a decision tree with the possibility of linear regression functions at the nodes. First, a decision-tree induction algorithm is used to build a tree, but instead of maximizing the information gain at each inner node, a splitting criterion is used that minimizes the intra-subset variation in the class values down each branch. The splitting procedure in M5P stops if the class values of all instances that reach a node vary very slightly, or only a few instances remain. Second, the tree is pruned back from each leaf. When pruning, an inner node is turned into a leaf with a regression plane. Third, to avoid sharp discontinuities between the subtrees a smoothing procedure is applied that combines the leaf model prediction with each node along the path back to the root, smoothing it at each of these nodes by combining it with the value predicted by the linear model for that node.

REP-Tree [**?**] is a fast decision tree learner. It builds a decision/regression tree using information gain/variance and prunes it using reduced-error pruning (with backfitting). Only sorts values for numeric attributes once. Missing values are dealt with by C4.5's method [**?**] of using fractional instances.

Lasso as a Predictor [**?**] generates, depending on a parameter λ, a vector β whose elements are the weights of the vector x_j, which minimizes the objective function shown in Equation (2). The application of Lasso as a Predictor is grounded on the same mathematics used for Lasso Regularization, but the goal is different. In fact, while during the regularization process we are interested in determining the β vector, during the prediction we exploit the already-computed β vector to evaluate the prediction model, which is expressed as a closed-form equation.

Support-Vector Machine [**?**] constructs a hyperplane or a set of hyperplanes in a high- or infinite-dimensional space, which can be used for classification, regression, or other tasks. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training data point of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier.

Least-Square Support-Vector Machine [**?**] Given a training set $\{x_i, y_i\}_{i=1}^{N}$ with input data $x_i \in \mathbb{R}^n$ and corresponding binary class labels $y_i \in \{-1, +1\}$, the SVM classifier, according to Vapnik's original formulation [**?**], satisfies the following conditions:

$$\begin{array}{lll} w^T \phi(x_i) + b > 1, & \text{if} & y_i = +1, \\ w^T \phi(x_i) + b < -1, & \text{if} & y_i = -1. \end{array} \tag{4}$$

which is equivalent to $y_i \left[w^T \phi(x_i) + b \right] \geq 1, \quad i = 1, \ldots, N$ where $\phi(x)$ is the non-linear map from original space to the high (and possibly infinite) dimensional space.

Once generated the set of models by using the different ML methods applied to the different training sets, the validation phase takes place. This phase consists of computing a number of metrics by using a sub-set (validation set) of samples (possibly not used for the model training) included in the

training sets. Hence, for each model, the following metrics are provided:

Mean Absolute Prediction Error (MAE): it is the average of the differences between predicted and real RTTF. It is calculated as:

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n}|f_i - y_i|\,, \tag{5}$$

where f_i is the predicted value, y_i is the observed value, and n is the number of samples in the validation set.

Relative Absolute Prediction Error (RAE): it is relative to a simple predictor, namely the average of the actual measurement. RAE normalizes the total absolute error by dividing it by the total absolute error of the simple predictor.

$$\mathrm{RAE} = \frac{\sum_{i=1}^{n}|f_i - y_i|}{\sum_{i=1}^{n}|Y - y_i|}, \tag{6}$$

where

$$Y = \frac{1}{n}\sum_{i=1}^{n}|y_i|\,. \tag{7}$$

Maximum Absolute Prediction Error (MAE): it is the maximum prediction error, i.e. the maximum value in the set $|f_i - y_i|$ for each sample i in the validation set.

Soft-Mean Absolute Prediction Error (S- MAE): it is calculated as the MAE, except that when the value $|f_i - y_i|$ is less a given threshold it is considered to be equal to zero.

Training Time: it is the time required by the learning method for building the model.

Validation Time: it is the time required for completing the validation of the model, including the calculation of the above mentioned errors.

The above metrics provide the user with useful information for comparing the different models produced by F^{2}PM.

E. Additional Utilities of F^{2}PM

To further enhance the applicability of F^{2}PM, we provide additional utilities, which can be used along the main Framework. Two of these utilities can be used to inject anomalies in the system in a synthetic way. This could be used, e.g., either for testing F^{2}PM in a synthetic environment, or to speed up the collection of datapoints for later training. The goal of these utilities is to generate artificial memory leaks and detach unterminated threads, according to uncorrelated distribution functions. In this case, the hardware system could be stressed under different anomaly loads, which allows F^{2}PM to explore many of the possible system configurations, which lead to a crash.

Memory leaks are generated by allocating periodically a variable-size contiguous chunk of memory and writing dummy data into it. Writing data is essential to mimic a faulty implementation, as otherwise the underlying operating system kernel might not really allocate physical memory for the buffer—depending on its internal implementation—yet only virtual memory would be allocated, which on its turn would not occupy actual physical space.

Each activation of the leak generator exploits two statistical distribution: On the one hand, we rely on a uniform distribution to define what is the size of each leak (in an interval specified by the user at startup). By doing so, we are able to mimic a general behaviour where applications require both small-size and large-size buffers to carry on their work, and both these kinds of buffers could be not released by a faulty implementation. The second statistical distribution is an exponential one, which is used to draw the time to wait before the next memory leak occurs. The mean of this exponential distribution is drawn uniformly at randomly (again in an interval specified by the user at startup). This allows us to mimic the execution of the "faulty portions" of the software more or less often.

Similarly to the case of memory leaks, we have resorted to one exponential distribution to draw the time spanning between two consecutive generations of unterminated threads. The average of the exponential distribution is drawn uniformly at random at startup, again in a range defined by the user.

Again, we emphasize that only relying on these utilites might not allow F^{2}PM to build prediction models, which are correct under any kind of utilization of the real (non-synthetic) application, but they can be used as a complement of the actual data collection on the real system application, in a controlled environment.

Additionally, F^{2}PM comes along with a thin client, called *Feature Monitor Client* (FMC), which can be installed on the monitored system. The goal of this client is to periodically gather system feature measurements and to generate datapoints. This custom thin client can be used in place of other third-party tools like collectd [**?**], e.g. when the application under monitoring is hosted on a machine different from the one where the training process is carried out. In our implementation, the FMC waits about[2] 1.5 seconds between the generation of one datapoint and the next one. This allows us to catch as well very high variability of the system features, and to avoid stealing too much computing power only for the sake of monitoring.

The FMC is connected to the last utility, namely the *Feature Monitor Server* (FMS), using standard TPC/IP sockets. This technological solution allows to deploy FMC/FMS both on the same machine under monitor, or on different machines. This flexibility gives the user the possibility to monitor the behaviour of both local and remote applications, in case other system monitoring tools are not directly available.

IV. Experimental Data

To evaluate the feasibility of our proposal, we carried out a controlled experiment on a virtual architecture, which we built on top of a 32-core HP ProLiant NUMA server. The server is equipped with a Debian GNU/Linux distribution (kernel version 2.6.32-5-amd64). VMware Workstation 10.0.4 is the virtual environment hypervisor. All virtual machines of the experimental environment were equipped with Ubuntu 10.04 Linux Distribution (kernel version 2.6.32-5-amd64). While we

[2]Fluctuations are related to CPU scheduling variability and to the current load of the system.

emphasize that a virtual architecture is not mandatory for the applicability of F^{2}PM, this is a technical simplification, which allowed us to quickly take control over a near-to-faulty machine. In this way, the system can be quickly restarted, so as to collect data related to a large number of executions in a limited amount of time.

To perform our experiments we use two different virtual machines (VM). One VM runs our FMS (to collect the hardware features), and generates the workload targeting the second VM. The second VM hosts the application, experiencing occurrence of anomalies.

A. Test-Bed Application

We test F^{2}PM in the case of a multi-tier e-commerce web application that simulates a on-line book store, following the standard configuration of TPC-W benchmark [**?**]. We use the Java version [**?**], developed using servlets, and relying on Mysql [**?**] as a data base server.

In order to generate TPC-W requests, we have used an emulated browser. To evaluate the RT of the web application, we have introduced software probes in the emulated browsers to store on a database file the response time of every web interaction.

The TPC-W implementation has been modified in order to generate the anomalies of interest for this experimentation, namely memory leaks and unterminated thread. Specifically, we have modified the *Home Web Interaction* class (which implements the beginning of a TPC-W session) so that, when the servlet is started up, two different rates (for memory leaks and unterminated threads) are generated. Then, whenever an emulated browser connects to the initial page, some memory is leaked or a new thread is spawn, according to the corresponding probability. By using this approach, the generation rate of anomalies directly depends on the load of the TPC-W server (i.e., on the number of connections by the emulated browsers).

We have continuously run the experiment for one week, having the emulated browsers continuously issue requests to the TPC-W server. Upon a crash, the VM hosting the TPC-W is automatically restarted, so as to start serving again requests by the emulated browsers as soon as possible. During the execution of the experiment, we have used our FMS/FMC architecture to gather the features discussed in Section III-A.

B. Results

To evaluate the effectiveness of the features reduction using Lasso Regularization, we report in Figure 4 the number of selected parameters when increasing the value of λ. As it can be seen, higher values of λ are generally associated with a smaller number of features selected by Lasso (namely, Lasso associates a higher number of features with a zero weight in the β vector). Therefore, the regularization allows to select only the features of high interest for the generation of the prediction model. For the sake of completeness, we report in Table I the features selected by Lasso when $\lambda = 10^9$, along with the corresponding weights in the β vector.

As it can be seen, slopes play an important role to build the prediction model. In fact, as we have already discussed, they allow for the detection of sudden changes in the system

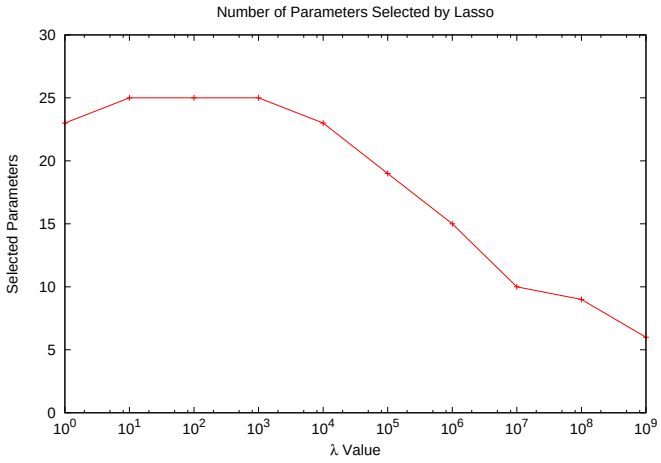


Fig. 4. Parameters selected by Lasso

TABLE I. WEIGHTS ASSIGNED WHEN $\lambda = 10^9$

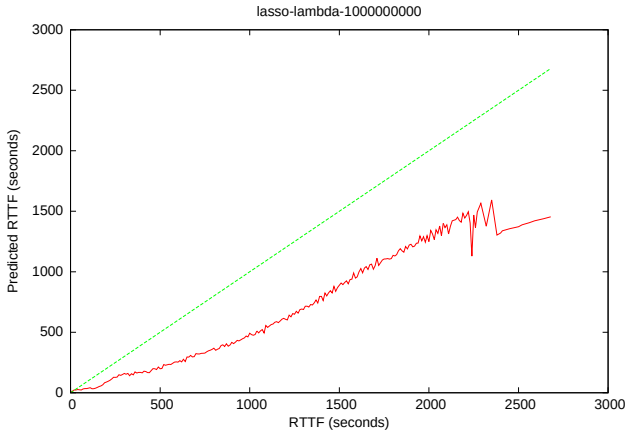
Parameter	Weight
mem_used_slope	0.000019235560086
mem_free_slope	0.000236946638676
swap_used_slope	0.000263386541515
swap_free_slope	0.000263386541515
mem_free	0.000263386541515
mem_buffers	0.000263386541515

features, which can be used, e.g., to promptly detect an approaching failure point. Similarly, memory is a predominant factor, as the system becomes immediately unavailable when there is no more free memory and the swap space is used completely.

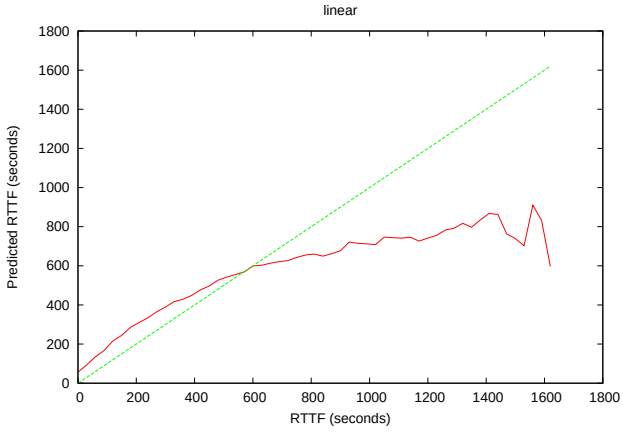
In order to evaluate the accuracy of the prediction models, we only show, for the sake of brevity, the results for the *Soft-Mean Absolute Error* (S-MAE). In Table II, we report S-MAE error values in seconds, for the cases when prediction models are built using all parameters and only parameters selected by Lasso. In both cases, we can see that the best accuracy is provided by REP-Tree. In comparison with REP-Tree, M5P increases the error in the order of 10%. All other ML methods show higher errors. We note that this could be due to the fact that both REP-Tree and M5P divide the model space in smaller portions, and evaluate for each portion a different linear approximation.

The training times for all learning methods are shown in Table III. The results are obtained by using all parameters and only parameters selected by Lasso. It is evident that when using all parameters the training times are significantly higher. Based on the presented results, the user can make a choice between less time in training or having a higher accuracy of the prediction model. Similarly, as we can see in Table IV, more time is required for validating prediction models when all parameters are used.

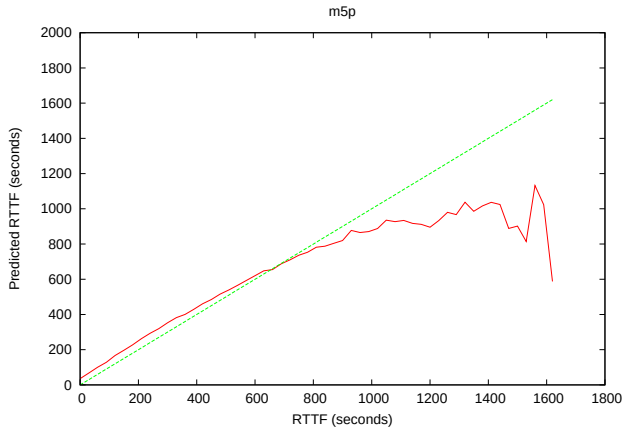
A graphical representation of the prediction models generated by F^{2}PM is presented in Figure 5. Specifically, we show the generated models by using all parameters. In the plots, the red curves represent the predicted RTTF (y axes) vs. the real one (x axes). The green lines represent the ground truth. Results show that, generally, the prediction error is



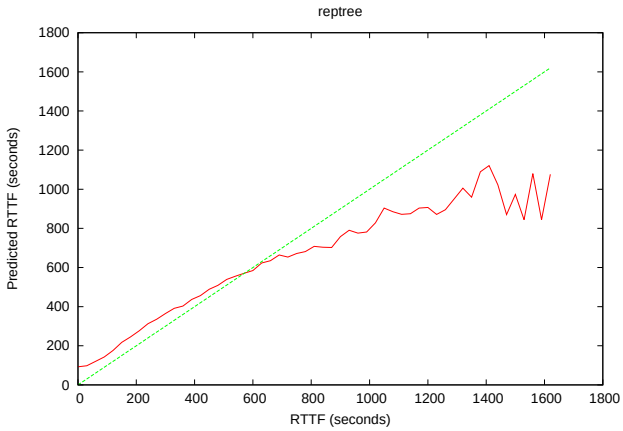
(a) Lasso ($\lambda = 10^9$)



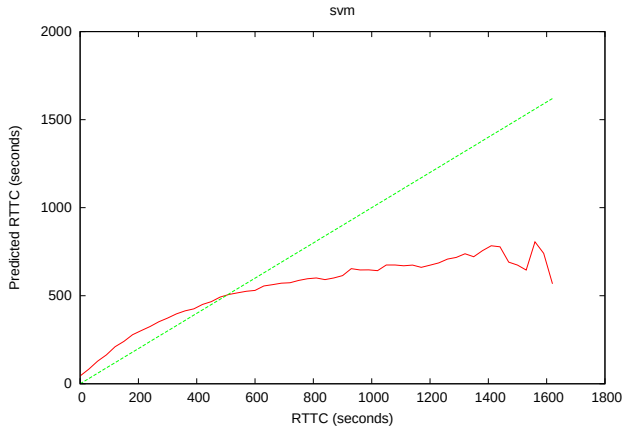
(b) Linear Regression



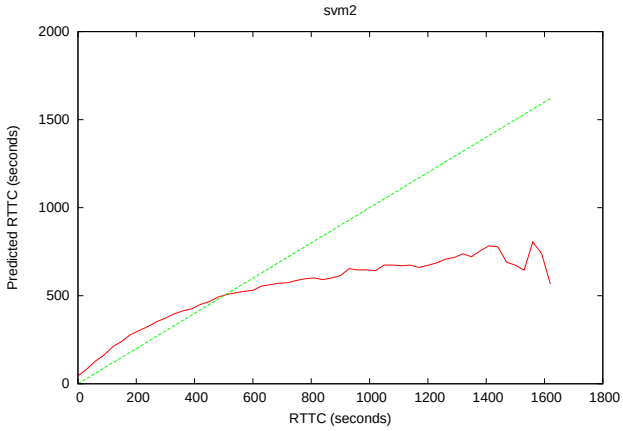
(c) M5P



(d) REP-Tree



(e) SVM



(f) LS-SVM

Fig. 5. Fitted Models using all Parameters

TABLE II. SOFT MEAN ABSOLUTE ERROR—10% THRESHOLD

Using all parameters		Using only parameters selected by Lasso	
Algorithm	**Error (seconds)**	**Algorithm**	**Error (seconds)**
Linear Regression	137.600	Linear Regression	156.603
M5P	79.182	M5P	118.292
REP Tree	69.832	REP Tree	108.476
SVM	132.668	SVM	146.594
SVM2	132.675	SVM2	146.607
Lasso ($\lambda = 10^0$)	405.187	Lasso ($\lambda = 10^0$)	405.187
Lasso ($\lambda = 10^1$)	405.187	Lasso ($\lambda = 10^1$)	405.187
Lasso ($\lambda = 10^2$)	405.186	Lasso ($\lambda = 10^2$)	405.186
Lasso ($\lambda = 10^3$)	405.178	Lasso ($\lambda = 10^3$)	405.178
Lasso ($\lambda = 10^4$)	405.124	Lasso ($\lambda = 10^4$)	405.124
Lasso ($\lambda = 10^5$)	404.823	Lasso ($\lambda = 10^5$)	404.823
Lasso ($\lambda = 10^6$)	404.041	Lasso ($\lambda = 10^6$)	404.041
Lasso ($\lambda = 10^7$)	399.023	Lasso ($\lambda = 10^7$)	399.023
Lasso ($\lambda = 10^8$)	399.240	Lasso ($\lambda = 10^8$)	399.240
Lasso ($\lambda = 10^9$)	392.469	Lasso ($\lambda = 10^9$)	392.469

TABLE III. TRAINING TIME

Using all parameters		Using only parameters selected by Lasso	
Algorithm	**Error (seconds)**	**Algorithm**	**Error (seconds)**
Linear Regression	0.30	Linear Regression	0.08
M5P	3.10	M5P	1.58
REP Tree	0.56	REP Tree	0.17
SVM	417.41	SVM	164.96
SVM2	391.69	SVM2	205.65

TABLE IV. VALIDATION TIME

Using all parameters		Using only parameters selected by Lasso	
Algorithm	**Error (seconds)**	**Algorithm**	**Error (seconds)**
Linear Regression	0.42	Linear Regression	0.12
M5P	0.36	M5P	0.09
REP Tree	0.55	REP Tree	0.11
SVM	0.39	SVM	0.13
SVM2	0.38	SVM2	0.13

higher when the system is far from the failure time, showing a lower prediction of the real RTTF. One motivation of this behaviour of the models is that when the amount of accumulated anomalies increases, the system performance (in particular the system throughput) tends to decrease. As a consequence, also the rate of occurrence of anomalies decreases, thus delaying, with respect to the prevision, the actual time when the system fails. In any case, results show that models provide low error while approaching the actual failure time, where more accuracy would be required in order to, e.g., trigger proper actions to address the upcoming system failure. For example, graphs show that, except for the case of Lasso as a predictor, the prediction error becomes very low when the actual RTTF is around 600 seconds.

As a final observation, we note that, also on the basis of results presented in Figure 5, REP-Tree and M5P are the best methods in our test-bed application.

V. CONCLUSIONS

In this paper, we have proposed the innovative F^{2}PM Framework for generating RTTF prediction models of applications. One advantage of our approach is that F^{2}PM can be used out of the box, without any need for manual modification/intervention in the applications. Additionally, it can be customized by the user according to a specific class of application and/or type of anomalies. F^{2}PM uses different machine-learning methods to generate the models, allowing the user to decide, on the basis of a set of metrics, the best suited one for his needs.

We demonstrate how F^{2}PM can be used, in the case of a web application, based on a popular e-commerce benchmark. We present a set of experimental results for evaluating F^{2}PM. In our specific test-bed scenario, we found that F^{2}PM allows us to select prediction models for application failure, with small training time and high accuracy.

ACKNOWLEDGEMENTS

The research presented in this paper has been supported by the European Union via the EC funded project PANACEA, contract number FP7 610764.