Using Analytical Models to Bootstrap Machine Learning Performance Predictors

Diego Didona and Paolo Romano
INESC-ID / Instituto Superior Técnico, Universidade de Lisboa

***Abstract*—Performance modeling is a crucial technique to enable the vision of elastic computing in cloud environments. Conventional approaches to performance modeling rely on two antithetic methodologies: white box modeling, which exploits knowledge on system's internals and capture its dynamics using analytical approaches, and black box techniques, which infer relations among the input and output variables of a system based on the evidences gathered during an initial training phase.**

In this paper we investigate a technique, which we name *Bootstrapping*, which aims at reconciling these two methodologies and at compensating the cons of the one with the pros of the other. We analyze the design space of this gray box modeling technique, and identify a number of algorithmic and parametric trade-offs which we evaluate via two realistic case studies, a Key-Value Store and a Total Order Broadcast service.

I. Introduction

The vision of elastic computing is probably the main driver at the basis of the disruptive success of the Cloud paradigm. By acquiring resources based on actual applications' demands, both upfront capital investments and operational costs can be significantly reduced. On the other hand, materializing the potential gains of elastic computing demands the development of performance prediction methodologies capable of accurately estimating both the actual resource demands of cloud applications, as well as the impact on their performance (or other relevant metrics such as availability, reliability, etc) of a number of factors, including workload characteristics and intensity, configuration parameters, and deployment alternatives (e.g., public vs private cloud, number and capacity of virtualized servers over which the system is deployed).

Classical approaches to performance modeling rely on two techniques, typically regarded as antithetic: Machine Learning (ML) [1] and Analytical Modeling (AM) [2][1].

ML-based techniques embody the *black box* approach, which infers performance models based on the relations among the input and output variables of a system that are observed during an initial training phase. ML-based performance models can typically achieve a very good accuracy when working in *interpolation*, i.e., in areas of the feature space that have been sufficiently explored. On the downside, the accuracy of such techniques is typically hindered when used in *extrapolation*, i.e., to predict values in regions of the parameter space not observed during the training phase [1]. Another major issue of ML techniques is that the number of configurations to be explored can grow, in the worst case, exponentially with the number of variables (often referred to as features, in the ML literature) characterizing the application — the so-called *curse of dimensionality* [1]. This affects the time needed to gather a sufficiently representative training set, which can quickly become large enough to make the usage of such techniques cumbersome or even prohibitive in complex systems.

Even employing dimensionality reduction techniques, such as Principal Component Analysis [3], and optimized techniques for the exploration of the parameter space, e.g., adaptive sampling [4], the training phase of a black box learner to predict the performance of complex applications can take days [4]. This represents a major threat to the adoption of pure ML-based approach for performance prediction in Cloud environments: a change in a component of the deployment infrastructure, e.g., virtual CPU processing power, or the shift to another Cloud provider could severely hamper the accuracy of a black box model trained over a data-set, and force a new, potentially long training phase over the new infrastructure [5].

AMs, conversely, are based on the *white box* approach, according to which the model designer exploits knowledge about the dynamics of the target system in order to mathematically express its input/output relations. AMs require no or minimal training phase; on the other hand, in order to allow for mathematical tractability, they rely on approximations and simplifying assumptions. Hence, the accuracy of AMs can be challenged in scenarios in which such approximations and assumptions are not valid [2]. In addition, the virtualization layer of Cloud infrastructures hides low-level details of the physical platform that hosts an application; this makes it cumbersome to derive detailed AMs of Cloud applications [6].

Being based on antithetic approaches, AM and ML are typically regarded as alternative techniques to model the performance of computer applications. In this paper we investigate a technique, which we name *Bootstrapping*, that aims at reconciling these two paradigms and achieve the best of the two worlds, namely the extrapolation capabilities and low training time of AM, combined with the high accuracy of ML when working in interpolation.

The key idea at the basis of the Bootstrapping technique consists in relying on an AM to generate a *synthetic* training set over which a complementary machine learner is initially trained. The synthetic training set is then updated over time to incorporate new samples collected from the operational system. By exploiting the knowledge of the white box AM, the resulting model inherits its initial prediction capabilities, avoiding, unlike traditional ML-based approaches, the need for long, and potentially expensive, training phases. At the same time, by updating the synthetic knowledge base with

This work has been supported by FCT through projects UID/CEC/50021/2013 and Incentivo/EEI/LA0021/2014.

[1]For the sake of brevity, the acronyms ML and AM will also be used, respectively, with the meaning "machine learner" and "analytical model".

samples coming from the actual system, the Bootstrapping technique allows for progressively correcting initial errors due to inaccuracies/approximations of the AM. The white box model also allows for enhancing the robustness of the resulting *gray* box predictor, by improving its accuracy in regions of the feature space not observed enough during the training phase.

Note that the Bootstrapping technique relies on both the white and the black box approaches, yet does not entail doubling the effort of modeling the target application's performance. Conversely, it reduces the cost of building either of the two models. On the one hand, one can use simpler/less accurate AMs, as they are progressively enhanced by the corrections applied via *off-the-shelf* ML. On the other hand, the availability of a base AM reduces the costs of gathering training set data in large multi-dimensional feature spaces.

Contributions. This paper makes the following contributions:

1) We introduce the Bootstrapping technique, and we provide a detailed algorithmic formalization thereof.

2) We identify two key factors in the design of the Bootstrapping technique: *i*) which samples of the output of the AM should compose the initial synthetic training set, and *ii*) which technique should be used to update the (initially fully) synthetic knowledge base with new evidences gathered from the operational system. We thoroughly investigates these issues and propose a set of alternative approaches to tackling them.

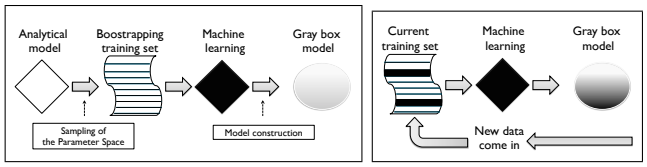
3) We provide an extensive experimental study aimed to assess the effectiveness of the proposed technique based on two realistic case studies: a popular distributed Key-Value Store (Infinispan by Red Hat [7]) and a Total Order Broadcast (TOB) service [8]. The former is representative of typical cloud data stores, whose performance exhibits complex non-linear trends and is affected by a large number of parameters. The latter represents an incarnation of the consensus problem [8] and is used as a fundamental building block in a number of fault-tolerant platforms. Our experimental results highlight the potentiality of the proposed Bootstrapping technique, and allow us to derive insights regarding its sensitivity to several non-trivial factors, such as the accuracy and density of the initial synthetic training set, and the choice of the technique used to incorporate new knowledge into it.

II. Related work

This paper is related to performance modeling techniques that leverage AM and ML in synergy. The 2 solutions that are closest to ours rely either on on-line [10] or off-line ML [11].

In the on-line domain, Romano and Leonetti propose the use of UCB, a popular Reinforcement Learning algorithm, to determine the optimal batching level for a TOB implementation [10]. In this solution, a UCB instance is responsible to learn the optimal batching level for a given workload; an AM is used to initialize the state of the UCB learners.

The Bootstrapping technique significantly differs from these solutions because the issue of how to update the training set is not a problem for on-line learning algorithms. In fact, such approaches do not maintain a training set, as they already specify how the state/knowledge of the learner (e.g., a UCB instance) is updated whenever a new sample is received. Conversely, updating the training set to include factual knowledge



(a) Initialization phase. (b) Updating phase.

Fig. 1. Main phases of the Bootstrapping technique.

coming from the real system is a key function in off-line bootstrapped predictors. In this work, we propose and evaluate several algorithmic variants to implement it.

In the off-line category, IRONMODEL [11] is, to the best of our knowledge, the first attempt to combine AM and regression techniques. IRONMODEL relies on human supervision to detect the deviation of an application/component's performance from the expected one (according to the model) and to trigger a corrective phase. This phase consists in the execution of a set of tests aimed to replicate the conditions (i.e., workload) that led to the deviation between observed and predicted performance and that aim to provide enough data to induce the black box learner to correct such discrepancy.

The proposed technique, instead, relies on a totally automated work-flow, which corrects the inaccuracies of the base AM with no human intervention. In addition, the work in [11] does not address two crucial issues, which, as shown in our evaluation, strongly impact the accuracy of a bootstrapped model, namely how to initialize and update the knowledge base of the ML model. Conversely, not only we describe different implementations for these operations, but we also thoroughly discuss and analyze their impact on the accuracy of the resulting gray box model.

This work has also relations with other gray box modeling techniques, which we classify in *Parameter Fitting*, *Divide et impera* and *Ensemble* approaches. In *Parameter fitting*, black box techniques (e.g., regression [12] and Kalman filters [13]) are used to estimate parameters of a white box model, which is ultimately used to forecast performance. Bootstrapping is orthogonal to these ones: as we shall detail in Section IV, the AM employed for the TOB use case incarnates this approach.

The *Divide et impera* technique builds specialized models for different components of the target system, that rely either on AM or on ML [9], [14], [15]. The Bootstrapping approach is orthogonal with respect to this methodology. Indeed, as detailed in Section IV, the base predictor employed in the KVS case study is based on the *divide et impera* approach.

In the *Ensemble* approach, the output of multiple white and black box performance models is combined with the aim of minimizing predictive error: either a black box classifier is used to identify which is the best predictor to use depending on the incoming query [16], [17], or black box models are employed to learn how to correct the inaccuracies of a base white box one [17], [18]. The Bootstrapping technique is orthogonal also to these last solutions: thanks to their hybrid nature, bootstrapped models can be seamlessly employed as part of the aforementioned ensemble techniques.

Algorithm 1 Bootstrapping main loop

```
1: function MAIN()
2:     ML ml                                              ▷ The machine learner
3:     AM am                                              ▷ The analytical model
4:     DataSet ST = initKB()                              ▷ Generate the synthetic training set
5:     ml.train(ST)                                       ▷ Train the ML over the synthetic training set
6:     while true do
7:         DataSet D = collectSamples()                   ▷ Collect samples at runtime
8:         updateKB(ST, D)                                ▷ Incorporate the new samples in the knowledge base
9:         ml.train(ST)                                   ▷ Re-train the ML over the updated training set
10:    end while
11: end function

12: function QUERY(Configuration x)
13:     return ml.query(x)
14: end function
```

Algorithm 2 Initialization phase

```
1: function INITKB(ML ml, AM am, int initSize, double ε)
2:     double error                                       ▷ Fitting error of ml over am's function
3:     int currSize = initSize                            ▷ Current size of the synth. training set
4:     do
5:         Set T = SampleConfigSpace(currSize)            ▷ Training configurations
6:         DataSet ST = ∅                                 ▷ AM-based training set
7:         for all x ∈ T do
8:             ST = ST ∪ {x, am.query(x)}                 ▷ Query the analytical model
9:         end for
10:        error = estimateFittingError(ST, ML)           ▷ Evaluate ml fitting over am
11:        currSize = nextSize()                          ▷ Select a new value for the size of T
12:    while (!isAccurate(error, ε))                      ▷ Ensure ml has learnt am's function
13:    return ST
14: end function
```

III. The Bootstrapping Technique

In this section we describe the Bootstrapping technique in a top-down fashion: first, a specification of the technique is presented, in which several relevant building blocks are encapsulated into abstract primitives. Next, in Sections III-A and III-B, we shall discuss in detail the key parametric and algorithmic trade-offs associated with each of these primitives.

As reported in the pseudo-code of Algorithm 1, the Bootstrapping technique consists of two main phases: the initialization of the black box model based on the predictions of the analytical one (Lines 4-5), and its update, which is performed every time that new samples from the running application become available (Lines 6-10).

The initialization phase, depicted in Figure 1(a) and detailed in Section III-A, is composed, in its turn, of two steps:

i) *generation of the synthetic training set (Line 4):* a subset T of the AM's parameter space (*PS*) is generated and is used to bootstrap the knowledge base of a machine-learner. As already mentioned, the number of samples in *PS* that are necessary to characterize an arbitrary function defined over such space grow, in the worst case, exponentially with the dimensionality of *PS*. A first challenge addressed in this step is, thus, to determine *which* samples to include in the initial synthetic training set such that *PS* is sufficiently covered.

Once T has been obtained, the AM is queried to compute a prediction of the performance of the application for each of its elements. The output of this phase is a new set ST, whose elements are tuples of the form $< x, am.query(x) >$, where $x \in T$ and $am.query(x)$ is the corresponding prediction computed by the AM. This step will be detailed in Section III-A;

iii) *black box model construction (Line 5)*: the ML is trained on ST and produces a statistical model of the application's performance; note that the ML can be based on alternative algorithms, e.g., Decision Trees (DT), Artificial Neural Networks (ANN) and Support Vector Machines (SVM) [1].

The update phase, shown in Figure 1(b), consists of 3 steps:

i) *collection of real samples (Line 7):* a new dataset D of $< x, perf(x) >$ tuples is collected, where x is a configuration/workload of the target application, and $perf(x)$ is the real performance, i.e., measured on the live system, corresponding to x. Such factual knowledge can either be spontaneously originated by the on-line production system (e.g., corresponding to workloads that are generated by end users), or can be collected during a dedicated off-line training phase, thus steering the workloads and configurations to experiment with;

ii) *update of the training set (Line 8)*: the ST set is updated in order to incorporate knowledge represented by the samples in D. There are several ways to perform this operation, which will be discussed in Section III-B;

iii) *black box model update (Line 9)*: the ML is trained on the updated ST and outputs a new application performance model.

A. Synthetic Knowledge Base Initialization

The first step of the Bootstrapping technique is embodied by the INITKB function, whose pseudo-code is reported in Algorithm 2. This function iteratively performs two main operations. The first one consists in selecting a subset T of samples from the whole space of possible configurations for the application. The second one consists in generating a training set ST by exploiting the predictions output by the AM for each of the elements in T. The goal of the function is to output a synthetic training set ST that is representative of the target performance function to be modeled, i.e., such that the black box learner trained over it is able to accurately represent the performance function embedded in the base AM.

The *initKB* function takes as input an AM of the target application (noted *am*), a ML algorithm (*ml*), an initial value for the size of the synthetic training set (*init*) and a threshold value (ε). Then, proceeding iteratively, it aims to find a value x such that training *ml* over a synthetic training set of size x produces a black box model which well approximates *am*.

To this end, the *initKB* function relies on 4 primitives. We introduce their high-level functionalities the following, and describe how we implemented them to conduct our experimental evaluation in Section IV-C.

- SAMPLECONFIGSPACE: it determines *which* samples of the feature space to include in the synthetic training set, given its size. This function can embody arbitrary sampling strategies, based, e.g., on random sampling or Active Learning [19].
- ESTIMATEFITTINGERROR: it estimates how much the performance model encoded by *ml* is similar to the one embedded by *am*. Also this primitive lends itself to several possible instantiations, e.g., leave-one-out or cross-validation [20].
- ISACCURATE: it returns *true* if *ml* approximates *am* sufficiently well; *false* otherwise.
- NEXTSIZE: it determines the size of the set to sample from the whole parameter space at the next iteration. The *initKB* function basically aims at minimizing the difference between

Algorithm 3 Update phase

```
1: function UPDATEKB(DataSet D)
2:     setWeight(D,w)                          ▷ Set the weight to the new samples
3:     function update = any function in {MERGE,RNN,RNR,RNR2}
4:     update(D);          ▷ Include real samples of D into the synthetic training set
5:     ml.train(ST)                            ▷ Retrain the ML with the new dataset
6: end function
```

am's and *ml*'s predictions as a function of *ST*'s size. Therefore, this primitive can implement different search techniques, e.g., iterative or binary search, to identify a cardinality of *ST* such that, under the provided sampling algorithm, $ml \simeq am$.

These primitives are employed in the *initKB* function as follows: *i*) sample the parameter space via the *sampleConfigSpace* primitive, to obtain a set T of cardinality *currSize* (Line 5); *ii*) for each element in T, query *am* to obtain the corresponding performance prediction, and generate the synthetic set *ST* (Lines 6-9); *iii*) train *ml* over *ST* and evaluate its accuracy in predicting the performance function encoded by *am*, by means of the *isAccurate* primitive (Line 10); *iv*) if the fitting error (or its improvement over past iterations) is less than a given threshold ε, then return *ST*; else determine the next value for *currSize* and go to step *i*) (Lines 11-13)[2].

Note that, given a sampling algorithm, the cardinality of *ST* plays a role of paramount importance in determining the effectiveness of the Bootstrapping methodology. It represents, in fact, a key trade-off between the accuracy with which the function encoded by the white box model can be approximated by the black box learner, and the effectiveness with which the latter can incorporate new knowledge deriving from the availability of samples collected from the operational system.

Reducing the number of samples can, in general, yield several benefits. These include reducing the duration of the initial training phase of the black box learner; also, it may favor the subsequent update phase of the training set: the lower the number of synthetic samples, the higher the relative density of the real samples in the updated training set. This can reduce the time it takes for the real samples to outweigh the synthetic ones, and correct possible errors of the AM.

Using a lower number of synthetic samples, however, also yields the black box model to approximate more coarsely the original white box one, which may degrade accuracy. On the other hand, a very large training set provides more detailed information to the black box learner on the function embodied by the AM, and can favor a better approximation of such function. However, it comes with the downside of an increased training time and may induce a longer transient phase before runtime samples can take over synthetic ones.

Note that the initial training phase of the ML is performed over the output of the AM on a sampling of the *whole* parameter space *PS* of the target application. Hence, even if provided only with a set R of real samples corresponding to narrow regions of *PS*, the bootstrapped learner still inherits the predictive power of the base AM when working in extrapolation with respect to R.

[2]For simplicity, we do not show how to handle cases in which the fitting error never goes below the ε. Coping with this case could simply entail returning the *ST* that minimizes the error after a given number of attempts.

B. Update of the Knowledge Base

The UPDATEKB function, reported in Algorithm 3, is the core of the Bootstrapping methodology, as it allows for the incremental refinement of the initial performance model. This function is responsible for incorporating real samples coming from the running application into the initial synthetic training set, thus allowing the black box model to gradually correct inaccurate performance predictions by the white box model.

The UPDATEKB function takes as input the dataset D containing new samples and injects them into the current training set. The key issue here is that the new samples contained in D may contradict the synthetic samples generated by the AM that are in the training set. This happens whenever D contains samples belonging to regions of the feature space in which the AM achieves unsatisfactory accuracy: in such a case, in fact, the AM generates outputs (i.e., performance predictions) that may differ significantly from the corresponding values in D (i.e., having similar or identical input mapped to different output). In this work, we consider two techniques that aim at reconciling possible divergences between synthetic and actual samples: *weighting* and *replacing*.

Weighting is a well-known and widely employed technique in the ML area [21]: the higher the weight for a sample, the more the ML will try to minimize the fitting error around it when building the statistical model. In the Bootstrapping case, weighting can be used to suggest the ML to give more relevance and trust to real samples than to synthetic ones. The replacing approach consists in removing preexisting "close enough" (synthetic) samples from the training set, whenever new real samples are incorporated. We consider four implementations of the UPDATEKB function, which incorporate new knowledge according to different principles.

1) Merge. This is the simplest considered variant, and it consists in adding the new samples to the existing set *ST*. This implies the possible co-existence of real and synthetic samples that map very similar (or equal) input features to very different performance. Hence, the use of weights is the only means to induce the ML to give more importance to real samples over (possibly contradicting) synthetic ones.

2) Replace based on Nearest Neighbor (RNN). This variant consists of two steps, which are repeated for each element (x,y) in D: i) find the element (x_r,y_r) that is closest (according to a distance function) to (x,y) in *ST* and ii) replace (x_r,y_r) with (x,y). Also in this case the newly injected sample is given a weight w. Note that, once an element from D is inserted in *ST*, it becomes eligible to be evicted from the set, even in favor of another sample contained in D itself. This algorithm aims to progressively replace all the synthetic samples from *ST* with real ones; by switching a real sample with its nearest neighbor in *ST*, moreover, this algorithm aims at keeping unchanged the density of samples in *ST*.

3) Replace based on Nearest Region (RNR). This algorithm represents a variant of RNN. A first difference is that, in order to avoid "losing" knowledge gathered from the running system, RNR policy only evicts synthetic samples from the training set. Moreover, instead of replacing a single sample in *ST*, a sample in D replaces all the ones in *ST* whose distance from it is less than a given cut-off value c. If a sample in D does not replace any sample in *ST*, it is added to *ST*, as it is

considered representative of a portion of the feature space that is not covered by pre-existing elements in ST. On one side, this implementation speeds up the process of replacement of synthetic samples with real ones; on the other side, depending on the density of the samples in ST and on the cut-off value, it may cause imbalances in the density of samples present in the various regions of the feature space for which T contains information. In fact, a single sample from D may potentially take the place of many others in ST.

4) Replace based on Nearest Region 2 (RNR2). This algorithm represents a variant of RNR. Also RNR2 policy, in fact, only evicts synthetic samples from the training set; however, it differs from RNR in the way samples corresponding to actual measurements are incorporated in the training set. For each element $(x,y) \in ST$, the closest neighbor $(x_r,y_r) \in D$ is found: if the distance between the two is less than a cut-off value c, then the output relevant to x is changed from y to y_r. Like in RNR, if a sample in D does not match any sample in ST, it is added to ST. This implementation inherits from RNR the speed in replacing samples in ST with real, new ones, but avoids its downside of changing the density of samples in ST: instead of removing samples from ST, for each element (x_r,y_r) in D, the target value of all the points in the training set for which it is nearest neighbor and within distance c is approximated with y_r.

IV. Experimental Evaluation

In this section we evaluate the algorithmic and parametric trade-offs discussed in the previous section via an experimental evaluations based on two case studies: a distributed Key-Value Store and a Total Order Broadcast primitive.

A. Case studies

As already mentioned, we consider two case studies: Infinispan [7], a popular distributed Key-Value Store (KVS) and a sequencer-based Total Order Broadcast (TOB) service [22].

1) Key-Value Store: In this study we consider Infinispan, a popular NoSQL data store, which provides a key-value data model. Infinispan is a in-memory transactional platform that relies on replication to ensure data durability.

The performance of this type of platforms is affected by several factors: contention on physical (e.g., CPU) and logical (i.e., data items) resources, workload characteristics (e.g., transactional mix) and platform configuration (e.g., scale and replication degree). This case study is, thus, an example of a modeling/learning problem defined over a large dimensional space (spanning 7 dimensions in our case) and characterized by a complex performance function.

Base AM. The reference model used as base predictor for this case study is PROMPT [9]. In PROMPT, an AM captures the effects of workload and platform configuration on CPU and data contention. On the other hand, it relies on ML to predict latencies of network bound operations. Therefore, PROMPT represents by itself an instance of gray box modeling (*divide et impera*); in order to treat PROMPT as a plain white box model, we fix its black box network model, by training it with the same samples of [9].

Experimental dataset and test bed. We consider a dataset composed by approximately 900 samples, collected by deploying Infinispan on a private Cloud infrastructure, consisting of 140 Virtual Machines (VM) hosted on a cluster composed by 18 physical servers equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) processors and 32 GB of RAM and interconnected via a private Gigabit Ethernet. The employed virtualization software is Openstack Folsom. The VMs are equipped with 1 Virtual CPU and 2GBs of RAM.

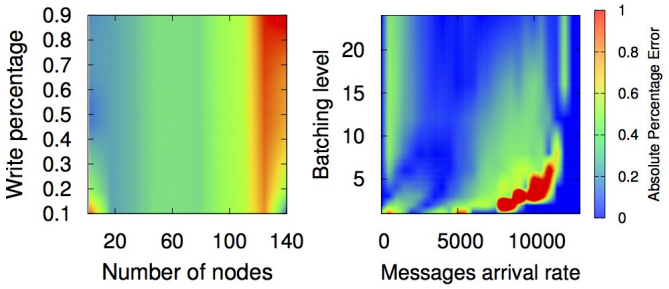
The considered application is a transactional porting of YCSB [23], the *de facto* standard benchmark for key-value stores. The dataset consists of YCSB workloads A, B and F, generated using a local thread that injects requests in closed loop. Performance samples correspond to different workloads' throughputs, while varying the number of nodes N in the platform in the set $\{2,5,10,25,50,75,100,125,140\}$ and the replication factor in $\{1,2,3,\frac{N}{2},N\}$.

2) Total Order Broadcast: TOB [8] is a primitive that allows a group of processes to achieve consensus on a common delivery order of messages that can be broadcast (possibly concurrently) by processes in this group. TOB is a fundamental building block at the basis of a number of fault-tolerant replication mechanisms. Specifically, we consider a sequencer-based implementation of TOB [22], which relies on a single process, called *sequencer*, to impose a common total order of messages delivery. Batching [24] is a well-known optimization technique for STOB algorithms: by buffering messages, and processing them together, the sequencer can amortize the sequencing cost and achieve higher throughput; the message delivery latency, however, can be negatively affected at low load, due to the additional time spent by the sequencer waiting (uselessly) for the arrival of additional messages.

Base AM. The AM used as starting point to implement the Bootstrapping algorithm is described in [10]: the sequencer node is modeled as a $M/M/1$ queue, for which each job corresponds to a batch of messages of size b. The message delivery latency is computed as the response time for a queue that is subject to an arrival rate λ equal to the frequency of arrival of a batch of messages of size b and whose service time μ accounts both for the CPU time spent for sequencing a message of size b and for the average time waited by a message to see its own batch completed.

This AM takes as input the CPU costs of processing the first and subsequent messages in a batch. In this study, the estimation of these costs has been performed by finding the values that minimize the AM's prediction error over *all* the samples in our data-set. Therefore, this baseline allows us to assess the effectiveness of the Bootstrapping technique in improving the accuracy also of *parameter estimation*-based gray box models, discussed in Section II. Since the fitting has been performed over the whole set of available measurements, the accuracy of this baseline already represents an upper-bound of the one achievable by the *parameter estimation* technique. The following evaluation, therefore, shows that Bootstrapping can also improve over this other gray box modeling technique.

Experimental dataset and test bed. We consider a data set containing a total of 250 observations, corresponding to a uniform sampling of the aforementioned bi-dimensional space, and drawn from a cluster of 10 machines equipped with two Intel Quad-Core XEON at 2.0 GHz, 8 GB of RAM and



(a) **KVS** (b) **TOB**

Fig. 2. Error distribution of the base AMs.

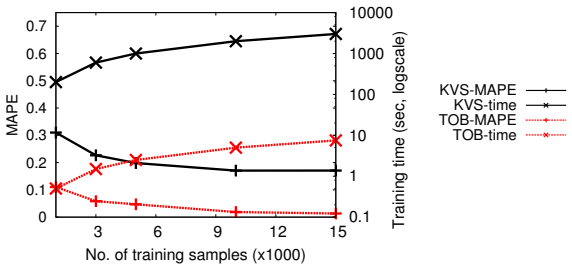


Fig. 3. Fitting an AM via ML: training size vs training time and MAPE.

interconnected via a private Gigabit Ethernet. In the experiment performed to collect the samples, the batching level was varied between 1 and 24, and 512 bytes messages were injected at arrival rates ranging from 1 msgs/sec to 13K msgs/sec.

3) Relevance of the Case Studies: We selected the two aforementioned case studies for three main reasons.

1. Relevance and wide adoption: using a KVS and a TOB primitive allows us to assess the viability of Bootstrapping when applied to mainstream distributed platforms and applications.

2. Diversity of the performance functions: the feature spaces of the two case studies have very different dimensionality (2 for TOB vs 7 for KVS), and the corresponding base performance models exhibit different distribution of errors, as depicted in Figure 2. This allows us to evaluate the proposed solution in very heterogeneous scenarios, increasing the representativeness of our experimental study.

3. Diversity of the base performance models: the AM for the TOB case describes the system performance at a high level of detail by means of a simple $M/M/1$. The KVS model, conversely, goes to great lengths to capture the complex dynamics of a number of internal components of the system (e.g., concurrency control, data locality and replication). Hence, the selected case studies allow us to assess the Bootstrapping technique using base models that differ significantly in terms of design complexity and level of detail at which they capture the dynamics of the target system.

B. Black box modeling

We employ, as black box learner, Cubist, a DT regressor that approximates non-linear multivariate functions by means of piece-wise linear approximations [25]. As already mentioned, the Bootstrapping technique can be implemented with any black box learner. One may argue that the choice of the learner to couple with the AM can be considered another tuning parameter of the Bootstrapping technique. However, identifying the learner that maximizes the prediction accuracy given a training and a test sets is a general challenge, which falls beyond the sole boundaries of the Bootstrapping technique, and that can be addressed with standard techniques [26][3].

C. Initialization

We begin our study by evaluating the impact on the gray model's accuracy and construction time depending on the number of samples of the feature space used to populate the initial synthetic training set. We start by describing the implementation of the primitives invoked in the initKB function, used to create the initial synthetic training set ST.

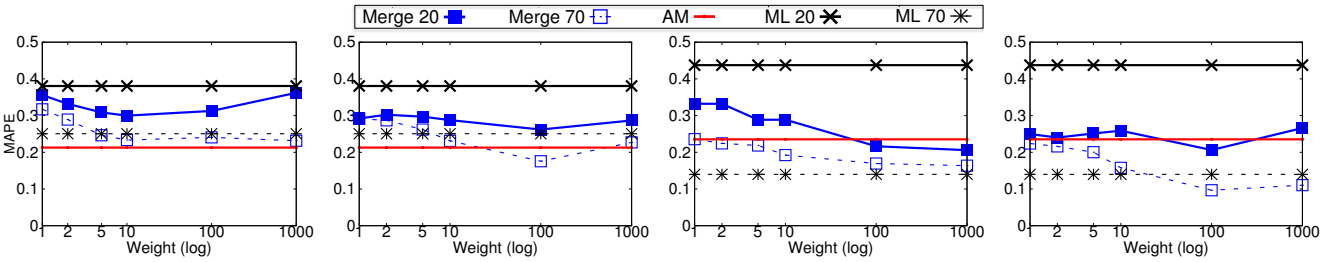
- SAMPLECONFIGSPACE is based on a uniform random sampling of the parameter space: the rationale behind this choice is that a random policy is the most simple sampling strategy to implement, yet it has been shown to be very effective [26].
- ESTIMATEFITTINGERROR implements 10-fold cross-validation, i.e, it i) partitions ST into 10 bins $ST_1 \dots ST_{10}$; ii) iteratively $\forall i = 1 \dots 10$, trains the ML over $ST \setminus S_i$ and evaluates its accuracy against S_i and iii) returns the average accuracy.
- ISACCURATE is based on a simple predicate that returns *true* if the accuracy of the black box model has not increased enough (i.e., more than ε in relative value) over the last n iterations (with $\varepsilon = 0.01$ and $n = 3$ in our evaluation).
- NEXTSIZE returns the size of ST at the current iteration plus a fixed value (set to 500 in our evaluation).

Therefore, the implemented initKB evolves by increasing the size of ST by a fixed step and randomly sampling the parameter space to build the new ST; it terminates when the accuracy on the black box learner trained over ST plateaus.

Figure 3 reports, for both case studies, the gray box model building time and the Mean Average Percentage Error (MAPE), computed as $Avg.(\frac{|real-pred|}{pred})$, of the gray box model with respect to the predictions produced by the AM evaluated by means of ten-fold cross validation. On the x-axis we let the number of initial synthetic samples included in the training set of the gray box model. For both case studies, our *initKB* function returns a synthetic training set of 10K samples, as it detects no noticeable improvements in the black box model's accuracy. Indeed, Figure 3 shows that even proceeding up to 15K samples the accuracy gain is negligible.

The model building time portrayed in the plots corresponds to the sum of the time needed to query the AM in order to generate the synthetic data set of a given cardinality plus the time needed to train the ML over such set. We report that, in our experiments with Cubist, the training time for both case studies has been less than half a second; the gray box model building time in the plots is, thus, largely dominated by the cost needed to query the AM. As shown by Figure 3, in the KVS case this cost is much higher than in the TOB one, as the corresponding AM is solved through multiple iterations [9]. However, the cost to query the AM has to be paid only once, upon initializing the bootstrapped learner, as the update phase only requires to re-train the black box learner.

[3]Analogous considerations apply for the issue of feature selection, which is an intrinsic, and well studied [27], problem of any ML technique.



(a) **KVS**: 1K synthetic samples (b) **KVS**: 10K synthetic samples (c) **TOB**: 1K synthetic samples (d) **TOB**: 10K synthetic samples

Fig. 4. Impact of the weight parameter for the Merge updating policy, using 1K and 10K synthetic samples.

Figure 3 shows that, by fitting the AM using ML techniques, a loss of accuracy is unavoidable. The actual extent of this accuracy degradation depends on factors such as the number of samples used to construct the initial synthetic training set and the intrinsic capability of the learner to approximate the target function. The plot shows that, as expectable, larger training sets yield a lower approximation error, at the cost of a longer training time; it also shows that Cubist is able to fit the TOB response time function encoded in the AM very well (3% of MAPE with a 10K samples training set) but it is unable to achieve similar accuracy for the KVS case. We argue that this depends on the fact that Cubist approximates non-linear functions by means of piece-wise linear approximation in the leaves of the decision tree that it builds. Such model may be unable to properly approximate the performance function of PROMPT, which is defined over a multi-dimensional space and exhibits strongly non-linear behaviors. On the other hand, a preliminary experimentation with alternative learners (ANN and SVM) provided significantly worse approximation errors, especially for the KVS case.

Overall, these results highlight that ML techniques may introduce approximation errors with respect to the original AM. This initial accuracy loss of the gray box model, as we shall see, can actually make it less accurate than the base AM.

D. Updating

We now evaluate the alternative algorithms for the updating of the knowledge base that we presented in Section III-B. We first assess the sensitivity of each algorithm to its key parameters. Finally, we compare their accuracy assuming an optimal tuning of such parameters.

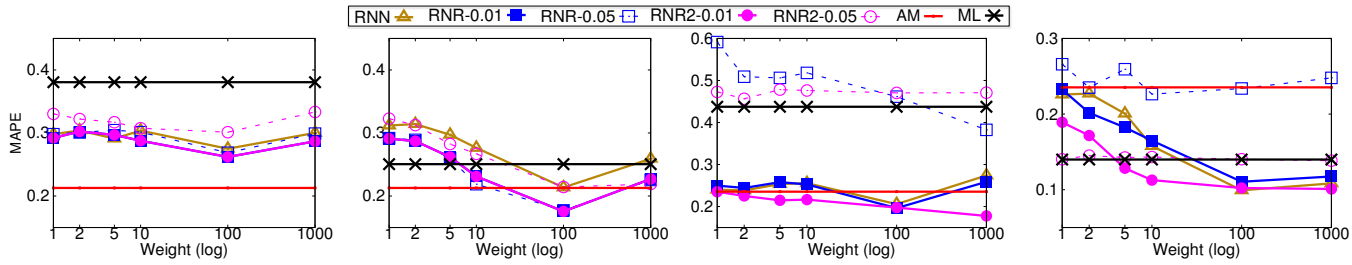
We start by showing in Figure 4 the results of a study aimed at assessing the impact of the weight parameter on the resulting accuracy of the bootstrapped model, while considering synthetic training sets of different initial sizes, namely 1K (Fig. 4(a) and 4(c)) and 10K samples (Figure 4(b) and 4(d)). We consider two scenarios, in which we assume the availability of 20% and 70% of the entire data set composed of collected, real samples, which we feed as input to both the Merge algorithm and to Cubist (non-bootstrapped) that serves as first baseline. As a second reference, we show also the accuracy achieved by using the AM, which incurs a MAPE that is independent of the initial size of the synthetic training set. On the x-axis we vary the weight parameter of the Merge algorithm, and report on the y-axis the MAPE computed with respect the whole set of actual samples (i.e., unlike in the previous section, here the MAPE is not computed with respect to the output of the AMs).

The first finding revealed by the plots is that employing 10K synthetic samples (rather than 1K) is beneficial for the accuracy achieved by the bootstrapped learner. This happens because, as already discussed, larger synthetic training sets allow the black box learning algorithm to learn more accurately how to approximate the base performance models; this, in turn, yields to inherit a predictive accuracy that is closer to the one exhibited by the white box model itself.

The other finding highlighted by the plots is the relevance of correctly tuning the weighting parameter, regardless of the size of the initial synthetic training set. However, it is possible to observe, by comparing Figures 4(a) and 4(b), that the best setting of this parameter may be relatively larger in the case of larger synthetic training set than for the case of smaller one. This can be explained by considering that, by increasing the size of the initial training set, the ratio of real vs synthetic samples correspondingly decreases. From the ML perspective this translate into decreasing the relevance of the real samples with respect to that of the "surrounding" synthetic samples. In fact, the Merge update method never evicts synthetic samples from the knowledge base: if the initial synthetic training set is significantly larger than the number of available real samples, these are always surrounded by a large number of synthetic ones, which end up obfuscating the information conveyed by the real ones. By increasing the weight of the samples gathered from the running system, the statistical learner is guided to minimize the fitting error on these points. On the other hand, using excessively large weight values can be detrimental, as it makes the learner more prone to over-fitting.

Overall, the experimental data show that both with large and small initial synthetic training set, Merge achieves significantly higher accuracy than both Cubist and the white box model, when provided with 70% of the data in their training set. When the training set percentage is equal to 20%, the scenario is rather different. In both scenarios, the gray box model still achieves a higher accuracy than a pure ML-based technique. However, the gray box is only marginally better than the AM with the large initial synthetic training set, and slightly worse than AM with small initial synthetic training set. This can be explained by considering that the gain achievable using the 20% training set is relatively small, and can be even outweighed by the loss of accuracy introduced by the learning of the initial white box model (see Section IV-C).

In Figure 5 we focus the comparison on the updating



(a) **KVS:** 20% of *real* samples. (b) **KVS:** 70% of *real* samples. (c) **TOB:** 20% of *real* samples. (d) **TOB:** 70% of *real* samples.

Fig. 5. Impact of the weight and cut-off parameters for RNN, RNR, and RNR2, using 10K synthetic samples.

policies RNN, RNR, and RNR2. We recall that, unlike Merge, these techniques strive to avoid the coexistence in the training set of "neighboring" synthetic and real samples, by removing or replacing synthetic samples close enough to the real ones. The intuition underlying these approaches is that the AM may be erroneous, and hence contradict the actual samples and confuse the learner. With the exception of the RNN method, which uses exclusively the weight parameter, RNR and RNR2 also use a cut-off parameter, which defines the relative amplitude (normalized over a maximum distance) of the radius that is used to determine which synthetic points are to be removed (RNR) or updated (RNR2), whenever a new real sample is incorporated in the training set.

For the sake of presentation, only two cut-off values are considered, namely 1% and 5%, and the weight parameter is treated as the independent variable. The choice of reporting results with these cut-off values is motivated by the fact that they suffice to illustrate the main dynamic related to the setting of this parameter: the higher is the cut-off, the more the training set is composed by only real samples, thus making the bootstrapped model eventually collapse to the pure black box one. In all the performed experiments, the employed distance function is the Euclidean one, but any could be used. Before computing the distance between two samples, a feature normalization process is performed, i.e., the value of every feature is normalized so as to lie in the range [0,1]. When using scale-sensitive distance functions like the Euclidean one, this avoids features that naturally assume larger absolute values to have more weight in determining the distance than other ones.

Figure 5(a) and 5(c), resp. Figure 5(b) and 5(d), report the MAPE achieved when using 20%, resp. 70%, of the real data set as training set, reporting, as before, the reference values achieved by the white box model and by Cubist (non-bootstrapped). The first result highlighted by these plots is that, also in the replace-based update variants, the weight parameter plays a role of paramount importance. Also the cut-off parameter has a huge impact on the final accuracy of the hybrid model, when implementing RNR and RNR2.

Overall, the cut-off based replace policies result more effective into increasing predictive accuracy in the two considered case studies. The improvement with respect to RNN is more evident in the KVS case: this is because RNN entails the possibility of evicting real samples from the training set, whereas RNR and RNR2 do not. As a result, RNN discards some of the information conveyed by real samples, thus losing some of its corrective power.

This effect is tightly related to the characteristics of the target performance function and of the distribution of the real samples. For the TOB case, in fact, both real and synthetic samples are drawn uniformly at random from the whole space of possible arrival rates and batching levels. Moreover, the 10K synthetic samples are very cluttered in the two-dimensional space in which they lie, thus reducing the probability that RNN evicts a real sample instead of a synthetic one.

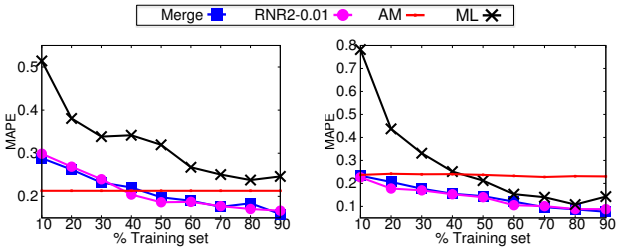
Conversely, for the KVS case, the samples in the synthetic training set are drawn uniformly at random but the real ones are not as they are, instead, representative of typical configurations and workloads for that kind of platforms. For example, the density of the points characterized by a number of nodes smaller than 25 is higher than the one relevant to points corresponding to more than 100 nodes in the platform; in the same guise, the replication degree for data items is defined over the set $\{1,2,3,\frac{N}{2},N\}$, being N the number of nodes. This, together with the relative sparseness of the 10K synthetic samples in the seven-dimensional space in which they lie, induces RNN to evict, in some cases, real samples.

E. Lessons learnt

In this section we summarize the most important and insightful findings that our evaluation has highlighted.

Importance of parameterization. Previous sections have highlighted the sensitivity of the Bootstrapping technique to the setting of its internal parameters: if properly tuned, this technique can yield considerable gains in terms of accuracy with respect to AM and ML employed singularly; conversely, if poorly parametrized, the resulting hybrid model can even be worse than the individual black and white box models. It is important to underline that this is not an idiosyncrasy or flaw of the Bootstrapping technique; rather, it is an inherent issue for any black box model: the accuracy of any ML algorithm is strictly dependent on the quality of its parameterization. Identifying proper values for the parameters of a Bootstrapping-based model can be accomplished via standard techniques for the optimization of the hyper-parameters of ML algorithms, e.g., random search or Bayesian Optimization [26].

On the updating policy. Our study suggests that it may be preferable to opt for a simple update heuristic, such as Merge, over more convoluted ones, such as Replace-based variants. This conclusion is backed by Figure 6, which compares the accuracy achieved by the two best performing updating heuristics, Merge and RNR2, with that achieved by pure white and black box approaches. The size of the initial synthetic training set is 10K, and the parameters used by the update



(a) KVS (b) TOB

Fig. 6. Merge vs Replace (weight=100)

policies are the ones that resulted in the best performance in the evaluation cases considered so far. The accuracy of the various predictors is evaluated while varying the size of the available training set from 10% to 90%.

Figure 6 clearly highlights the advantages that the Bootstrapping technique can provide, eventually outperforming both the reference white and black box models. It also shows that, in the considered case studies, and for the considered parameters values, there is no clear winner between the two updating variants, at least provided that their parameters are properly tuned (e.g., using cross-validation). In particular, the weighting parameter results to be the one that affects accuracy the most, up to the point that its careful tuning allows the Merge updating policy to perform similarly to the —relatively more complex— RNR2.

Therefore, our evaluation indicates that the Merge update policy has the potential to deliver an accuracy that is comparable to the one achieved by Replace-based methods, while requiring the tuning of only one parameter, namely the weight given to real samples, over the two that characterize Replace-based policies.

Relevance of the synthetic training set's size. The size of the synthetic training set *ST* influences a key trade-off between the accuracy of the initial statistical model[4] and the timeliness with which errors/biases of the AM can be corrected by incorporating real samples. A small *ST* set yields a coarse approximation of the analytical function, but real samples can rapidly outweigh fabricated ones (since the density of the latter is low). Conversely, a larger *ST* allows the ML to better approximate the AM; this, however, comes at the cost of a higher training time and possibly slower correction process, given the high density of synthetic samples in the training set.

V. Conclusions

In this paper we have proposed a technique, named Bootstrapping, that aims to reconcile the white box and black box methodologies by compensating the cons of the one with the pros of the other. We have identified several crucial design choices in Bootstrapping algorithms, proposed a set of alternative approaches to tackling these issues, and evaluated the impact of these alternatives by means of an extensive experimental study targeting two popular distributed platforms (a distributed Key-Value Store and a Total Order Broadcast service). Our results highlight the potentiality of this technique, but also shed light on a number of trade-offs that have to be taken into account in order to maximize its effectiveness, and has also opened interesting research avenues.

[4]More precisely the accuracy with which the initial ML model fits the AM.

References

[1] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., 2006.

[2] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge Univ. Press, 2013.

[3] J. Shlens, "A tutorial on principal component analysis," *CoRR*, vol. abs/1404.1100, 2014.

[4] S. Duan, V. Thummala, and S. Babu, "Tuning database configuration parameters with ituned," *VLDB Endowment*, vol. 2, 2009.

[5] M. B. Sheikh, U. F. Minhas, O. Z. Khan, A. Aboulnaga, P. Poupart, and D. J. Taylor, "A bayesian approach to online performance modeling for database appliances using gaussian models," in *ICAC*, 2011.

[6] O. Tickoo, R. Iyer, R. Illikkal, and D. Newell, "Modeling virtual machine performance: Challenges and approaches," *SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 3, pp. 55–60, Jan. 2010.

[7] F. Marchioni and M. Surtani, *Infinispan Data Grid Platform*. Packt Publishing, 2012.

[8] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[9] D. Didona and P. Romano, "Performance modelling of partially replicated in-memory transactional stores," in *MASCOTS*, 2014.

[10] P. Romano and M. Leonetti, "Self-tuning batching in total order broadcast protocols via analytical modelling and reinforcement learning," in *ICNC*, 2012.

[11] E. Thereska and G. R. Ganger, "Ironmodel: Robust performance models in the wild," in *SIGMETRICS*, 2008.

[12] Q. Zhang, L. Cherkasova, and E. Smirni, "A regression-based analytic model for dynamic resource provisioning of multi-tier applications," in *ICAC*, 2007.

[13] C. M. Woodside, T. Zheng, and M. Litoiu, "Performance model estimation and tracking using optimal filters," *IEEE Transactions on Software Engineering*, vol. 34, no. 3, pp. 391–406, 2008.

[14] D. Didona, P. Romano, S. Peluso, and F. Quaglia, "Transactional auto scaler: Elastic scaling of replicated in-memory transactional data grids," *ACM Trans. Auton. Adapt. Syst.*, vol. 9, no. 2, pp. 11:1–11:32, Jul. 2014.

[15] P. Di Sanzo, F. Quaglia, B. Ciciani, A. Pellegrini, D. Didona, P. Romano, R. Palmieri, and S. Peluso, "A flexible framework for accurate simulation of cloud in-memory data stores," *Simulation Modelling Practice and Theory*, available online doi:10.1016/j.simpat.2015.05.011, 2015.

[16] J. Chen, S. Ghanbari, G. Soundararajan F. Iorio, A. Hashemi and C. Amza,"Ensemble: A Tool for Performance Modeling of Applications in Cloud Data Centers", *IEEE Transactions on Cloud Computing* available online doi: 10.1109/TCC.2015.2469656, 2015

[17] D. Didona, F. Quaglia, P. Romano, and E. Torre, "Enhancing performance prediction robustness by combining analytical modeling and machine learning," in *ICPE*, 2015.

[18] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker, "Identifying the optimal level of parallelism in transactional memory applications," *Springer Computing*, pp. 1–21, 2013.

[19] B. Settles, "Active learning literature survey," University of Wisconsin–Madison, Computer Sciences Technical Report 1648, 2009.

[20] S. Arlot and A. Celisse, "A survey of cross-validation procedures for model selection," *Statistics surveys*, vol. 4, pp. 40–79, 2010.

[21] S. Cost and S. Salzberg, "A weighted nearest neighbor algorithm for learning with symbolic features," *Mach. Learn.*, v. 10, no. 1, Jan. 1993.

[22] H. Miranda, A. Pinto, and L. Rodrigues, "Appia, a flexible protocol kernel supporting multiple coordinated channels," in *ICDCS*, 2001.

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010.

[24] T. Friedman and R. V. Renesse, "Packing messages as a tool for boosting the performance of total ordering protocls," in *HPDC*, 1997.

[25] J. R. Quinlan, "Rulequest Cubist," http://www.rulequest.com, 2012.

[26] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for Hyper-Parameter Optimization," in *NIPS*, 2011.

[27] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *J. Mach. Learn. Res.*, vol. 3, pp. 1157–1182, Mar. 2003.

[28] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009.