A Learning-Based Method for Combining Testing Techniques

Domenico Cotroneo*, Roberto Pietrantuono*, Stefano Russo*†
*Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione, Universita' di Napoli Federico II, Naples, Italy.
†Laboratorio CINI-ITEM "Carlo Savy", Complesso Universitario Monte Sant'Angelo, Ed. 1, Naples, Italy.

***Abstract*—This work presents a method to combine testing techniques adaptively during the testing process. It intends to mitigate the sources of uncertainty of software testing processes, by learning from past experience and, at the same time, adapting the technique selection to the current testing session. The method is based on machine learning strategies. It uses offline strategies to take historical information into account about the techniques performance collected in past testing sessions; then, online strategies are used to adapt the selection of test cases to the data observed as the testing proceeds. Experimental results show that techniques performance can be accurately characterized from features of the past testing sessions, by means of machine learning algorithms, and that integrating this result into the online algorithm allows improving the fault detection effectiveness with respect to single testing techniques, as well as to their random combination.**

I. Introduction

Software testing techniques are known to perform well when applied in combination [1], [2], [3]. However, selecting the best set of techniques tailored for a specific software application is a hard task. Indeed, techniques may exhibit very different performance when applied to different software applications and within diverse organizational contexts. Hence, their once-for-all characterization is a challenge difficult to cope with. Most often, practitioners use, informally, their knowledge about the software under test in order to apply "proper" combinations, but they do not have a structured approach to select and combine testing techniques.

This paper presents a learning-based method for online testing techniques combination. The method aims at maximizing the number of faults detected, by selecting test cases dynamically from the technique with the greatest likelihood of exposing a failure for the specific software under test. To this aim, we address the inherent uncertainty that each testing session is subject to, by exploiting both historical information on techniques performance and data coming from the test cases outcome as the testing session proceeds. The best techniques are suggested at the beginning of the process, and adjusted during the testing session by regulating the impact of historical information with respect to online data. The method includes two phases. The first phase provides an initial characterization of techniques within a reference context (e.g., within a company), by exploiting available empirical data. To this aim, an *offline learning* approach is adopted to infer a relationship between the experienced techniques performance in past testing sessions and features describing the application conditions in which that performance is observed. This will orient techniques selection in a new software application. Accounting for the past is, however, not sufficient for having the best selection. Performance of a technique depends on so many variables, unrelated to the past, that it is unfeasible to have a universally valid characterization (e.g., the specific features of the software under test, the time and order of application of techniques, the state of the software at the time of application, and so on). This may cause totally different behaviours from what observed in the past. Thus, offline learning is complemented by an *online learning* phase, whose goal is to adapt the techniques choice to the software under test and to the observations coming directly from the field. The goal is to direct the attention toward the best techniques iteratively, being at the same time robust to noisy observations. *Offline learning* is driven by an outlined procedure, which first defines the features of a testing session potentially related to the techniques performance, and then uses several machine learning approaches for classification and prediction of the techniques behaviour. *Online learning* is based on a Bayesian algorithm that integrates offline learning results and uses the importance-sampling approach to learn the distribution of the "importance" of each testing technique.

From applying this method, this paper highlights that: *i)* it is possible to predict, with high accuracy, the techniques performance by machine learning algorithms. We defined features of the testing session potentially related to its performance; we then produced empirical data for such features by testing three software programs, and used them to train classifiers and build prediction models. The most impacting features are also identified; *ii)* integrating the result of prediction into the online Bayesian algorithm for techniques selection, with the goal of considering historical data in the online selection, leads to an improved fault detection effectiveness with respect to single testing techniques as well as to their random combination; *iii)* at higher level, formulating the problem compatibly with machine learning schemes helps to highlight unknown facets of the testing techniques behaviour, such their relationship with program and testing process features, or their sensitivity to historical data with respect to data coming from the current testing session. In the rest of the paper, Section II provides some background; Section III describes the method; Section IV applies the approach to case studies; Section V surveys the existing related literature; Section VI concludes the paper.

978-1-4673-3076-3/13/$31.00 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA

II. Background

The use of machine learning in software engineering has a long history, and it served many purposes [4], including defect prediction [5], debug/fault localization [6], software development [7], and software testing [8]. There are many machine learning techniques, and their usage in a specific context depends on the type of the formulated problem and on its underlying assumptions. Techniques can be grouped into: *i) offline learning* techniques, which train models with observed data in order to identify patterns in the test data; *ii) online learning* techniques, which learn models incrementally as more samples become available, and refine models based on the observed data. Examples of offline learning methods include decision/regression trees, random forests, logistic regression, (multiple) linear regression, support vector machines. Online learning is primarily based on probabilistic representation and on the concept of adaptation; examples are: Markov decision processes, reinforcement learning, stochastic sampling, and Bayesian reasoning schemes (e.g., Bayes networks)[1]. In this context, the problem addressed by *offline* machine learning is formulated as a binary classification problem. It consists in learning a predictive model from known data samples belonging to two classes in order to classify new unknown samples as belonging to one of them. The problem addressed by *online* learning is instead formulated as a Bayesian inference problem. An algorithm is defined to prioritize testing techniques iteratively. It is based on a Monte Carlo sampling method, named *importance sampling*, which has been used successfully in many domains [10]. Importance sampling is an inference method to approximate the computation of the true distribution of variables of interest, which in many practical tasks is intractable. The method samples from the true (unknown) distribution, and thus represents the beliefs (i.e., hypotheses) about the state of the system by sets of samples. Each sample is associated with a probability that the belief is true, and at each iteration: (1) the hypotheses are modified to account for changes in the system, (2) the probability of each hypothesis is updated by examining some samples of that hypothesis; and (3) a larger number of samples are drawn from hypotheses with a larger (relative) probability, to be analyzed in the next iteration [9]. The goal is to converge, in few iterations, to the true probability distribution over the set of hypotheses, identifying the ones more likely to be true.

III. The On-Off Learning-Based Strategy

The objective of the proposed strategy (hereafter *On-Off* testing method) is to run test cases drawn from the technique with the greatest expected probability to expose a failure at testing time t. This probability depends on several variables, which may be regarded as describing the program under test (e.g., its static features and its state when the technique is applied), and the state of the testing session itself (e.g., the number of test cases executed up to time t, the number of failures already exposed by that technique, the type of residual faults). Two are the key elements of the proposed strategy: *i)* to exploit historical data for providing indications on techniques performance; *ii)* to "modulate" the gathered information to the new software program, by an adaptive testing procedure. In the following, variables are introduced to formulate the problem; then offline and online learning procedures are defined.

A. Problem Definition

A testing technique in this context is a test case selection criterion[2]; given a number B of test cases as maximum budget for testing a program, a testing session is the execution of a sequence of B test cases chosen according to several criteria. Consider a set of testing techniques $T = \{\tau_1, \tau_2 \dots \tau_n\}$ and the current program to test denoted as P_0. This is assumed to be the set of available testing criteria that practitioners refer to. For the problem definition, consider the following variables.

- Denote with H=$\{h_1, h_2, \dots, h_m\}$ the past testing sessions applied to software programs $P_1, P_2, \dots, P_m$. The testing session to run on P_0 is denoted as h_0.
- $S_i(t)$: the state of a testing session on a program P_i at testing time t. It is described by variables $\rho_{1,i}, \rho_{2,i}, \dots, \rho_{g,i}$, representing features of the testing session that change as testing time t goes on. We measure the testing time t as number of executed test cases[3].
- O_i: the set of values assumed by features of a program P_i, denoted as $\varphi_{1,i}, \varphi_{2,i}, \dots, \varphi_{f,i}$; these are *static* features of the program, and do not change during the testing session.
- $d_{i,j}$: Euclidean distance between normalized O_i and O_j;
- $N_{\tau_j,i}$: number of test cases drawn from the technique τ_j and applied to P_i;
- NT_i: total number of test cases applied to P_i;
- $F_{\tau_j,i}$: number of failures exposed by τ_j when applied to P_i; $F_i = \sum_{j \in T} F_{\tau_j,i}$ is the total number of failures exposed in testing P_i.
- $\theta_{\tau_j,i} = \frac{F_{\tau_j,i}}{N_{\tau_j,i}}$: the detection rate of the criterion τ_j applied to the program P_i;
- $\theta_i = \frac{F_i}{NT_i}$: the total detection rate;
- $N_{\tau_j,i}(t)$, $NT_i(t)$, $F_{\tau_j,i}(t)$, $F_i(t)$, $\theta_{\tau_j,i}(t)$, and $\theta_i(t)$ are the defined variables evaluated at testing time t.

Regarding the testing session h_0, the following parameters are of interest in addition to the previous ones:

- $\theta^k_{\tau_j,0}(t)$ is the detection rate of criterion τ_j in iteration k; the "iteration" refers to the online learning algorithm.
- p_j: a probability measure associated with each testing criterion τ_j, representing the ability of τ_j of exposing a failure; it will be proportional to the detection rate of τ_j.

The value to be maximized is the total number of exposed failures F_0, given a testing budget NT_0. For simplicity, we will omit the subscript 0 from the notation when referring to the software program under test P_0 (e.g., $\theta_{\tau_j,0} = \theta_{\tau_j}$).

[1]Bayesian-based techniques are also successfully used as offline learning methods; relevant examples include Naive Bayes, Bayesian network classifiers, Bayesian Logistic regression.

[2]Testing criterion is used synonymously to testing technique

[3]Measuring testing time differently, e.g., by calendar time, or CPU execution time, does not impact the strategy definition and its application, because the metrics to characterize the techniques (next sub-section) are not influenced

B. Preliminary Technique Characterization: Offline Learning

The offline learning procedure uses $< P_i, h_i >$ pairs as historical data. A sample x in the learning process refers to a test case applied at time t, and it is a pair $< S_i(t), outcome_i(t) >$ composed by the $S_i(t)$ features vector, $\rho_{1,i}, \rho_{2,i}, \ldots, \rho_{g,i}$, at testing time t as predictors (we call them *dynamic metrics*, since they are time-dependent), and the outcome of the test case as predicted binary variable (0/1). Features at testing time t are intended to capture conditions in which a test case is applied, and are assumed to impact the outcome of a test case. In particular, we conjecture that the outcome of a test case mainly depends on the state of the system at the test case execution time. For instance, if many test cases have already been executed with a given technique, exposing many failures, and the detection rate referred to that technique at time t is considerably decreased, it is unlikely that a test case chosen according to that technique will still expose any failure; the same test case, executed in different conditions, may, instead, provide a positive outcome. Based on this concept, we identified a set of features describing the "evolution" of a testing session potentially impacting test cases outcome. In particular, referring to a program P_i, we define:

- $NT_{\tau_j,i}(t)$, and $\overline{NT}_{\tau_j,i}(t) = \frac{NT_{\tau_j,i}(t)}{t_i}$ which is the normalized value of the number of test cases already drawn from τ_j at time t ($NT_{\tau_j,i}(t)$) to the current number of executed test cases.
- $F_{\tau_j,i}(t)$, and $\overline{F}_{\tau_j,i}(t) = \frac{F_{\tau_j,i}(t)}{F_i(t)}$, i.e., the total number of failures uncovered by τ_j at time t and its normalized value to the total number of failures exposed;
- $NT_i(t)$, $F_i(t)$, i.e., the total number of test cases and failures exposed at time t;
- $MTTF_i(t)$: mean time to failure from time 0 up to time t; it is the inverse of the detection rate $\theta_i(t)$;
- $\overline{MTTF}_{\tau_j,i}(t) = \frac{MTTF_{\tau_j,i}(t)}{MTTF_i(t)}$, i.e., the MTTF referring only to τ_j over to the total MTTF.
- $LTTF_i(t)$: last time to failure at time t, given by: $LTTF_i(t) = t - t(F)$, with $t(F)$ denoting the time in which the last failures occurred; $\overline{LTTF}_{\tau_j,i}(t) = \frac{LTTF_{\tau_j,i}(t)}{LTTF_i(t)}$, i.e., the last time that τ_j exposed a failure, over the last time that a failure was exposed.
- $\theta_{\tau_j,i}(t)$, i.e., the detection rate referred to τ_j.
- τ_j, the applied testing technique

These metrics complement measures like *detection rate* or *total number of failures found*, which are not sufficient alone to judge a technique performance in a testing session. Considering such metrics, the following procedure is defined:

1) Sample Collection: to build a sample, the set of features $\rho_{1,i}, \rho_{2,i}, \ldots, \rho_{g,i}$ are collected from past testing sessions. These features, as well as the values $outcome_i$, are extracted from test results. The information needed for each test case is: test case number, criterion used, and outcome, from which all the features can be computed. Samples are separated per program, obtaining $|P|$ datasets: $D_i =< S_i(t), outcome_i(t) >$.

2) Selection of Predicting Program and Classifier: to enable prediction from programs in the repository to the program undergoing test, P_0, a predicting program and a classifier algorithm need to be selected. To this aim, two solutions are considered with a two levels of accuracy/overhead trade-off. The former is called *basic cross-program prediction*; as for the latter, we refer to the one suggested by Zimmermann et al. [12]. We first describe the main steps of these procedures, and then go into detail of each step. For the *basic* procedure, we define the following steps:

1) Let us denote with $pred(i)_j$, the prediction for program P_i from P_j. We are interested in figuring out how to perform $pred(0)_j$, being P_0 the current program. We chose, as P_j, the program with the minimal distance $min_{j \in P}(d_{0,j})$, where the distance is computed based on the set of normalized static features $\varphi_{1,0}, \varphi_{2,0}, \ldots \varphi_{f,0}$ (O_0) and $\varphi_{1,j}, \varphi_{2,j}, \ldots \varphi_{f,j}$ (O_j).
2) Once the program has been chosen, the classifier is selected. A validation of a set of classifiers is carried out on the dataset D_j, which refers to P_j. Results are compared, and the ones statistically worse are discarded, whereas the remaining ones are *validated* classifiers[4].
3) A series of cross-program evaluations is performed by using P_j as training set and the other programs (i.e., P_i with $i \neq 0 \neq j$) as test set, and all the validated classifiers. The classifier with the best average result, weighted by number of samples, is chosen[5]. The output is the pair $< P_j, C >$, with C the selected classifier.

if a sufficient number of programs is available, the procedure from Zimmermann et *al.* [12] can be applied, which has the advantage of allowing a preliminarily estimate of the expected prediction quality. It foresees: *i)* the cross-program prediction among all the program pairs (with all classifiers C_k), obtaining effectiveness measures for each pair; *ii)* the computation of distances among all the program pairs $d_{i,j}$; *iii)* the construction of decision trees with samples $< d_{i,j}, R(pred(i)_j) >$, with $R(\cdot)$ denoting the effectiveness measure of the prediction (there will be a tree for each measure R and classifier C_k); *iv)* from the new program P_0, compute the distances $d_{0,j}$ and navigate the tree to estimate the effectiveness of the prediction the would have the program j as predictor with the classifier C_k. In this way, before actually doing the prediction, one can establish if it will be successful or not. Of course, the best pair $< pred(0)_j, C_k >$ is chosen. Further details of the Zimmermann's procedure are found in [12]. Since this procedure may be expensive, the *basic cross-program prediction* is preferred if one of the following conditions holds: *i)* a sufficient number of programs to build the decision tree is not available; *ii)* the obtained predictions with the basic procedure are satisfactory; *iii)* there are no (time/cost/computational) resources to run all the mentioned cross-prediction tests. As for the *basic cross-program prediction*, let us detail the defined steps.

[4]If there are no statistically relevant differences among them, all of them can be chosen for the successive step.

[5]If performance turns out to be poor (e.g., compared to past studies prediction, as [11]), the best one is still chosen, but its poor prediction will be compensated in the online learning phase, and the weight of historical data will be low. In other terms, the approach is robust to poor predictions.

TABLE I: Software metrics considered

Metrics	Description	Metrics	Description	Metrics	Description
CountDeclFunction	Number of Function	C Code File	Number of C Code files	AvgVocabulary	Average Halstead's Vocabulary
CountLine	Number of all lines	CountLineCodeDecl	Declarative source code lines	AvgDifficulty	Avg. Halstead's Difficulty
CountLineBlank	Blank lines	CountLineCodeExe	Executable source code lines	AvgEffort	Avg. Halstead's Effort
CountLineCode	Lines containing source code	AvgCyclomatic	Average cyclomatic complexity	AvgBugs	Avg. Halstead's Bugs Deliv.
CountLineComment	Lines containing comments	MaxCyclomatic	Max. cyclomatic complexity	VVolume	Variance of Halstead's Vol.
CountLineInactive	Number of lines inactive from the view of preprocessor	MaxNesting	Maximum nesting level of control constructs	VLength	Var. of Halstead's Length
CountStmtDecl	Number of declarative statements	SumCyclomatic	Sum of cyclomatic complexity	VVocabulary	Var. of Halstead's Vocabulary
CountStmtExe	Number of executable statements	SumEssential	Sum of essential complexity	VDifficulty	Var. of Halstead's Difficulty
RatioCommentToCode	Ratio of the number of code lines to the number of comments line	AvgVolume	Average Halstead's Volume	VEffort	Var. of Halstead's Effort
				VBugs	Var. of Halstead's Bugs Deliv.
C Header File	Number of C Header files	AvgLength	Average Halstead's Length	CountPath	Number of unique paths

In the program selection (step 1) we use, as features $\varphi_{i,j}$, the metrics in Table I. Among the wide variety of metrics, it is difficult to claim that a given set of metrics is the best one in characterizing the program. Authors in [5] survey several metrics at various levels of granularity (method-level, class-level, component-level, process-level). Although there are studies using every type of metrics, they show that the most used and reliable ones are the method-level metrics, followed by the class-level ones. We based our choice on this observation, considering the metrics in Table I. These express the complexity and size of the program, the most common ones being the McCabe's complexity, the lines of code, the Halstead's metrics, and the object-oriented CK metrics. It is worth noting that engineers can select other metrics more suitable for their organization and for the software they produce.

The procedure is based on the conjecture that the "closest" program to P_0 is the best one at predicting techniques performance in P_0; if this is not true, during the online learning phase the prediction will be discovered to be poor, and the weight of history is accordingly lowered. The variant of the Zimmermann's procedure allows for predicting this performance, giving more confidence on the expected prediction performance, at the expense of more cross-predictions and more samples required [6]. In the step 2 and 3, classifier algorithms needs to be chosen. Several algorithms can be adopted; some common algorithms for binary classification problems, which we adopted later on, are *Decision Trees*, *Bayesian Network*, *Naive Bayes*, *Logistic Regression*. Algorithms undergo a cross-validation, to assess their individual performance. Performance for each run of cross-validation is assessed by indicators derived from the number of *true/false positives* (TPs/FPs), and *true/false negatives* (TNs/FNs)[7]. In particular, we use the following indicators:

Probability of Detection (**PD**): $PD = \frac{TP}{TP+FN} \cdot 100\%$.
It denotes the probability that a failure-exposing test case will actually be identified as such. A high *PD* is desired.

Probability of False Alarms (**PF**): $PF = \frac{FP}{TN+FP} \cdot 100\%$.
It denotes the probability that a non-failure-exposing test case is identified as failure-exposing. A low *PF* is desired.

Balance (**Bal**): $Bal = 100 - \frac{\sqrt{(0-PF)^2+(100-PD)^2}}{\sqrt{2}}$.
PD and *PF* are usually contrasting objectives, and a trade-off between them is needed. *Bal* represents this trade-off and it is based on the Euclidean distance from the ideal objective PD = 100% and PF = 0% [11].

Given these indicators[8], the cross-validation is performed, in step 2, in the following way: the evaluation is repeated 100 times for each classifier on the dataset, and the average results are computed. In each repetition, 66% of random samples are used for the training set and the remaining for the test set. The non-parametric *Wilcoxon* signed-rank hypothesis test [16] is then applied to identify the best classifier(s) ($\alpha = 0.05$)[9]. The procedure tests the null hypothesis that the differences Z_i between repeated measures from two classifiers have null median. Under the null hypothesis, the distribution of the test statistic tends to the normal distribution since the number of samples is large (N = 100). The hypothesis is rejected if *p-value*$\leq \alpha$. Classifiers statistically worse are discarded. In the step 3, using P_j as training and all the others as test set, a further evaluation of remaining classifiers is carried out, and the best one in terms of mean *Balance* is chosen.

3) Usage of Prediction Results: The chosen program/classifier pair $< P_j, C >$ will be used during the testing session to weight the acquired belief about the techniques performance with respect to the current performance. Prediction will be done in two phases: *i)* before starting the testing session h_0, in order to start the online algorithm by prioritizing techniques that behaved better in the past (Section III-C step 1); *ii)* at the end of each iteration k of the online algorithm, using samples observed in the iteration *k-1*, in order to figure out if the prediction from historical data is working well; this will be used to weight the impact of historical data with respect to current observations (Section III-C step 2). These are described in the online learning step.

[6]E.g., for a decision tree with 30 samples, at least 6 programs are required

[7]Samples of the test set belonging to the *target class* (i.e., a test case is successful in exposing a failure) are TPs if they are correctly classified, whereas they are FNs. Similarly, samples belonging to the other class are TNs if they are correctly classified, and are FPs otherwise.

[8]The defined measures are commonly adopted in machine learning studies [11], [13], [14]. Again, there exist other indicators that can be obtained from true/false positives/negatives, such as accuracy and precision, but they are known to be highly unstable for datasets where the target class occurs infrequently [15], like ours (our target class is the exposure of a failure that is very infrequent)

[9]This test has been shown to be robust with non-normal testing distributions.

C. Adapting to the Software under Test: Online Learning

As mentioned, although past history can provide good predictions of techniques performance, for the extremely uncertain nature of software testing, historical information is complemented by online adaptation. To this aim, Bayesian reasoning is exploited, and in particular, the importance-sampling method with KLD sampling adaptation [17].

1) Initial Assignment: As first step, the importance sampling requires a relatively small initial set of samples, which in our problem are test cases. Hence, the number of test cases for each criterion needs to be decided. This step is the first phase in which historical data are used. In particular, before starting the session, prediction is done on the variables $outcome_0(0), \ldots, outcome_0(t^*)$, where t^* is the number of test cases in the first iteration of the online algorithm. It will represent the expected performance of techniques at the beginning of the session, given the past results. To perform the prediction before starting the session, we should hypothesize the features' values for these initial t^* test cases in the new program (e.g., the $LTTF_0(t)$, $MTTF_0(t)$ and so on, for $t = 0, \ldots, t^*$). Since we do not know them before starting, the values of the first t^* test cases run on the program P_j are used. This means that without still knowing anything about the program under test, we initially give the maximum importance to past history, that will be then adjusted by online observations. Thus, using these features as input to the selected classification model, the prediction is carried out for each testing technique[10], obtaining $outcome_{0,\tau_i}(0)$, $\ldots, outcome_{0,\tau_i}(t^*)$ values, denoting the predicted outcome when the feature "testing technique" is set to τ_i. Given these values, we distribute test cases as follows:

$$NT^0_{\tau_i} = NT^0 \cdot \frac{\sum_{j=0}^{j=t^*} outcome_{0,\tau_i}(j)}{\sum_{i \in T} \sum_{j=0}^{j=t^*} outcome_{0,\tau_i}(j)} \quad (1)$$

Here, NT^0 is the total number of test cases to be executed in the first iteration, and $NT^0_{\tau_i}$ is the resulting number of test cases to be assigned to τ_i. This means that more test cases will be given to the techniques that are expected to perform better in the first t^* test cases, considering the impact of the past at this stage as equal to 100%. The term $p^0_i = \frac{\sum_{j=0}^{j=t^*} outcome_{0,\tau_i}(j)}{\sum_{i \in T} \sum_{j=0}^{j=t^*} outcome_{0,\tau_i}(j)}$ is the corresponding initial probability assigned to criterion τ_i. An alternative policy is to sample from a uniform distribution: $NT^0_{\tau_i} = \frac{NT^0}{|T|}$, meaning that the same number of test cases are drawn from each criterion; the initial probabilities in this case are $\frac{1}{|T|}$. This means that no historical information is available; we will use this policy as comparison with the history-based distribution.

2) Probability Update: After the initial assignment, the algorithm iteratively updates the likelihood that a criterion τ_i exposes a failure, p_i. This probability is based on the proportion of test cases that exposed a failure (i.e., detection rate) in each iteration:

$$p^k_i = \alpha \cdot p^{k-1}_i + (1 - \alpha) \cdot (\theta^k_{\tau_i} - p^{k-1}_i) \quad (2)$$

The update rule is obtained by weighting the difference by the *smoothing factor* α, which can be tuned in order to give more or less importance to the past with respect to the current iteration. To determine the latter value, the offline prediction comes into play again. In particular, as more and more data become available with testing execution, samples (i.e., test cases) are used as test set, with the training set being again P_j test data. Results of the prediction are evaluated in terms of *Balance* at each iteration: the impact of historical data is regulated based on the result of such prediction. For instance, if at the end of the first iteration, the *Balance* computed with samples from $t = 0$ to $t = t^*$ are below 0.5, it means that prediction is working bad and the impact of history-based prediction will be low compared to the online learning. This makes the approach robust to poor predictions. This information is put in Equation 2 by assigning a value to α (between 0 and 1) proportional to the last *Balance* measure ($\alpha = Balance \cdot 1/c$, with $c > 1$ constant[11]). The values of p^k_i are then normalized, since they are probabilities: $p^k_i = (p^k_i)/(\sum_{i \in T} p^k_i)$. *These probabilities represent the estimate at iteration k of the relative importance of criterion τ_i.*

3) Importance Sampling Algorithm: The importance sampling algorithm chooses, at each iteration, the number of test cases to execute in the next iteration, with the goal of giving more test cases to techniques with larger probabilities of detecting failures. The focus is progressively shifted toward criteria more able to detect residual failures. We use the importance sampling scheme modified to target our specific problem. The algorithm uses of the KLD-sampling variant, which adapts the number of sample in each iteration to the desired error and confidence. The number of samples to generate at iteration $k + 1$ is given by [17]:

$$\eta^{k+1} = \frac{1}{2\epsilon}\chi^2_{q-1,1-\delta} \approx \frac{q-1}{2\epsilon}\{1 - \frac{2}{9(q-1)} + \sqrt{\frac{2}{9(q-1)}} z_{1-\delta}\}^3 \quad (3)$$

where: ϵ represents the error between the sampling-based estimate and the true distribution that we want to tolerate; $1 - \delta$ is the confidence that we have in this approximation; q is the number of test criteria from which at least one test case has been drawn in iteration k; $z_{1-\delta}$ is the normal distribution evaluated with significance level δ. η^{k+1} represents the number of test cases to execute in the $(k + 1)$-th iteration. With such a number available and probability vectors, we implemented the following algorithm (***Algorithm 1***) to distribute test cases at each iteration. The algorithm first computes the cumulative probability distribution of the testing criteria, with probabilities set in descending order. Then it computes the number of test cases for the next iteration, and distributes, as output, test cases to criteria proportionally to their relative importance. It is executed until the number of available test cases ends.

[10]Namely, prediction for each value of $t = 0 \ldots t = t^*$ are repeated with each testing criterion τ_i, i.e., $|T|$ times

[11]We chose a constant $c = 1$ in our experiments

Algorithm 1

The importance sampling algorithm. **Inputs**: τ_i, p_i^k: $i \in [1, |T|]$
//sort such that $p_i^k \geq p_{i+1}^k$
$b_1 = p_1^k$; *//Initialize Cumulative Distribution*
for i=1 to $|T|$
    NT_i^{k+1}=0; *//initialization*
end for
for i=2 to $|T|$
    $b_i = b_{i-1} + p_i^k$; *//Compute Cumulative Distribution*
end for
//Compute η^{k+1} *according to Eq. 3*
$r_1 \sim U[0, \frac{1}{\eta^{k+1}}]$ *//Draw sample from uniform distribution*
//Distribute test cases to each criterion
i = 1;
for j = 1 **to** η^{k+1}
    while $r_j > b_i$ **do**
        $i = i + 1$;
    end while
    NT_i^{k+1}=NT_i^{k+1} + 1;
    r_{j+1}= $r_j + \frac{1}{\eta^{k+1}}$
end for
//Return re-ordered $\{NT_i^{k+1}\} : i \in [1, |T|]$

It is finally worth to remark that the entire procedure is based on a learning framework, whose setting up incurs a cost. The minimal cost procedure foresees: an offline learning step, including *i)* the extraction of metrics of the program under test, *ii)* the computation of the distance with programs stored in the base of knowledge, *iii)* and the selection of the best classifier (by cross-validation); an online phase where the costs are due to: *iv)* the initial test cases assignment for each testing criterion, requiring *outcome(t)* predictions for each criterion, *v)* the probability updates, which includes a further prediction step for the smoothing factor update, *vi)* the importance sampling algorithm. Steps *i)*, *iii)*, *iv)* anf *v)* are fully supported by existing tools and thus easily made automatic[12]. Steps *ii)* and *vi)*, as well as the probability update rule, are also relatively simple to implement[13]. Therefore, the procedure execution can be easily automated; a further potential source of cost comes from historical data collection. Indeed, the technique makes more sense in organizations already tracking historical data about testing sessions (at least, as mentioned, the test case number, the criterion used, and the outcome). Of course, tracking testing data is a practice suggested (e.g., by CMMI) also for other purposes, related to the overall process quality. Thus, the effort required by this step is negligible for organizations that already collect them, whereas, for organization not adopting this practice, it corresponds to the cost of setting up an activity for data tracking (in which case the advantage would go beyond testing). It is finally worth to note that, by gathering data iteratively, the accuracy of predictions will increase with time, with further benefits in terms of defect detection ability.

[12]In particular, we used *Understand 2.0*©, available at: *http://www.scitools.com/*, for metric extraction, and *WEKA*, available at *http://www.cs.waikato.ac.nz/ml/weka*, also providing an API for its usage by Java programs, for all (cross-)prediction steps.

[13]We implemented them by *Matlab*© procedures

IV. Case Study

A. Objective

By means of a case-study, this Section investigates the following research questions: *i)* does *On-Off* testing expose more failures with respect to other techniques? *ii)* how much the availability of historical information can impact the performance of *On-Off* testing? To address these questions, the performance of the *On-Off* strategy in terms of number of failures exposed given a testing budget B is evaluated. The *On-Off* testing will rely on three different testing criteria from which to draw test cases. The strategy will be compared against: *i)* the application of the three criteria singularly, *ii)* a strategy, which we call *All-RAN*, that combines criteria randomly, and *iii)* a strategy, that we called the *full* method, in which one test case per each criterion is executed at each test case execution time. We assumed the latter strategy as an upper bound to compare with, since it employs each criterion at each test case execution, with $|T| \cdot B$ (i.e., three times) more resources than the others. Finally the impact of historical information is evaluated, running test sessions at two extremes: by considering no information from offline learning, and no information from online learning, respectively.

B. Subject Programs and Testing Criteria

To experiment both offline and online learning phases, we took a sample of 4 software programs, 3 of which used to build the historical data repository, and one used as subject to test (i.e., sample P_0). Programs are taken from a publicly available repository[14], and chosen based on the availability of test case specification and/or test suites, and with at least 5KLoC (lines of code). Selected programs with their basic features are shown in Table II, with the last one being P_0. From the available test cases, we used three testing criteria to show the applicability of the approach, namely *random* testing (denoted as τ_{RAN}), *statement coverage*-based testing (τ_{COV}), and *robustness* testing (τ_{ROB}). While for the first two criteria several test cases were available from the repository, for the third one we wrote test cases from the TSL specification[15].

TABLE II: Software programs considered in this study.

Program	Language	Size (LoC)	Functions	Avg Cycl. Complexity	Number of Faults
Offline Learning					
Grep (v3)	C	10068	146	14.49	18
Flex (v3)	C	10459	162	9.78	17
Make (v1)	C	35545	268	10.56	19
Online Learning					
Space	C	6199	136	3.87	33

C. Offline Learning

To build the knowledge base, we carried out testing sessions on the three selected programs. For each program,

[14]SIR repository: http://sir.unl.edu/portal/index.php

[15]TSL specification are written according to the category-partition method; we worked on *choices* so as to include only error cases (i.e., test cases from the invalid partitions), then generating test cases from them with the TSL tool

TABLE III: Comparison between classifiers.

Algorithm	Grep			Flex			Make		
	PD	PF	Bal	PD	PF	Bal	PD	PF	Bal
Naive Bayes (NB)	97.72 %	13.21%	90.52%	97.46 %	6.5 %	95.06 %	93.1 %	15.72 %	87.86 %
Bayes Net (BNet)	98.49 %	20.64 %	85.36 %	97.34 %	4.75 %	96.15 %	96.28 %	32.91 %	76.58%
Decision Trees (DT)	99.49 %	11.74 %	91.69%	98.72 %	17.09 %	87.88 %	98.53 %	45.49 %	67.81%
Logistic (Log)	98.76 %	23.64 %	91.36 %	98.32 %	19.55 %	86.12%	98.15 %	44.43 %	68.55%

we hypothesized a budget of test cases allocated to them proportionally to the number of originally available test cases in the repository; this led to 600, 500 and 800 test cases for *Grep*, *Flex*, and *Make*, respectively[16]. The execution of test cases led to detect and remove all the faults (except one in the *Make* program); since fault location is known and we tracked their activation, the debug has been immediate; at each exposed failure, the corresponding fault has been removed[17]. The cumulative number of faults detected for each testing session is reported in Figure 1. During these sessions, dynamic

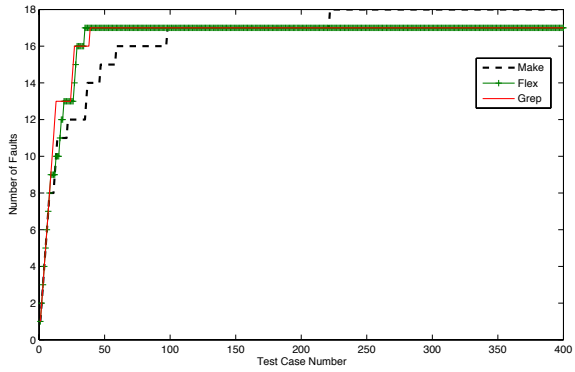

Fig. 1: Cumulative number of failures for testing sessions

metrics introduced in Section III-B have been collected for each technique along with the outcome of each test case. These data are used to train and test classifiers. To choose the program acting as predictor, we tested the *basic cross-program prediction*. As first step, the distance is computed among the programs used in the offline learning and the program that will go under test, i.e., *Space*. Based on their normalized metrics (see Table I), the closest software program to space is *Grep*. The three Euclidean distances turned out to be: $d_{0,Grep} = 2.7687$, $d_{0,Flex} = 3.0439$, $d_{0,Make} = 3.6740$.
As second step, the four classifiers are cross-validated on the *Grep* dataset; results are in Table III and IV. Note that the former Table reports the average results (over 100 executions) of cross-validation for each dataset, not only *Grep*. Performance of classifiers differ in the three datasets among each other (e.g., the highest *Balance* value with *Grep*, by Decision Tree, is the lowest one with *Make*). However, the average high performance (especially in the *recall* attribute that is never under 90%) suggests that the selected dynamic metrics (e.g., MTTF(t), LTTF(t), NT(t)) my be good predictors for test cases outcome. By running an *InfoGain* attribute ranking algorithm, in which attribute are ranked by their *information gain* [19], we observed that the top five metrics useful for prediction are: $LTTF_{\tau_j,i}$, NT_i, $\overline{MTTF}_{\tau_j,i}$, F_i, $MTTF_i$.
The latter Table (IV) reports, in each cell, the result of the *Wilcoxon* test for the *Balance* measure. We first consider the row with more favourable comparisons (i.e., *Decision Tree*); from this we chose the ones statistically equivalent, thus *Bayesian Net* and *Logistic Regression*. From cross-validation results on the *Grep* dataset from Table III, it is clear that all the classifiers give good performance, but statistically the *Naive Bayes* classifier is slightly worse, and thus it is discarded. The remaining three validated classifiers are used in the successive cross-prediction step, to choose only one of them. Using *Grep* as training, and *Flex* (with 500 samples), and *Make* (with 800 samples) as test set, we obtained the values in Table V. Based on results, the chosen pair at the end of this phase is $< Grep, Decision\ Tree >$. The last step of the analysis is performed with the online learning step applied to *Space*.

TABLE IV: Wilcoxon Test at α=0.05. B=statistically better; W=statistically worse; 0=statistically equivalent

Algorithms	DT	BNet	Log	NB
DT	**-**	**0**	**0**	**B**
BNet	**0**	**-**	**0**	**0**
Log	**0**	**0**	**-**	**0**
NB	**W**	**0**	**0**	**-**

TABLE V: Cross-program Prediction

From P_i to P_j	DT	BNet	Log
Grep to Flex	81.93	79.73	76.70
Grep to Make	75.73	76.90	48.88
Weighted Average	**78.11**	**77.99**	**59.58**

D. Online Learning

To start the online phase, the learnt model is applied first at the beginning, in order to provide the initial weights to techniques, and then at each iteration of the online algorithm, to adapt weights. The online algorithm will terminate when the testing budget is exhausted. For testing *Space*, we assume a set of 500 test cases as budget.
The first step is to determine the number of test cases to perform at the first iteration (i.e., NT^0) (cf. with Equation 3); given a maximum tolerable error of $\epsilon = 0.1$, and a significance level of $\delta = 0.01$, $NT^0 = 46$. Then, dynamic metrics from

[16]The selection of test cases from the available ones may bias result; this is minimized by randomizing selection and choosing a number of test cases proportional to the available ones.

[17]Note that in the following, the number of "exposed failures" and of "detected faults" will be the same, since for each exposed failure there will be exactly one fault detected and removed. Namely, a test case is repeated until it succeeds; a failure caused by a test case from which more faults would be removed is counted like repeated executions of that test case until it succeeds, and one fault at each execution is removed

Grep regarding the first NT^0 test cases are considered, in order to predict the $outcome_0(0)$, ..., $outcome_0(t^* = NT^0)$ variables. Finally, the number of test cases for each criterion ($N^0_{\tau_j}$) is derived from Equation 1. When the history-based initial allocation is considered, by using *Grep* as training set and the Decision tree classifier, we have: $NT^0_{\tau_{RAN}} = 24$, $NT^0_{\tau_{COV}} = 14$, and $NT^0_{\tau_{ROB}} = 8$[18]. Starting the testing session with this allocation, the importance sampling algorithm terminated in 12 iterations; in each iteration it updates probabilities according to Equation 2. Figure 2 shows the evolution of the assignment of test cases by the algorithm to the various criteria. Note that the assignments take the impact of offline learning into account; Figure 3 depicts the corresponding evolution of α. It indicates that the prediction from *Grep* to *Space* is working well, because the *Balance* is always around 0.7, and thus the impact of offline learning is noticeable in this case. The procedure starts with 24,

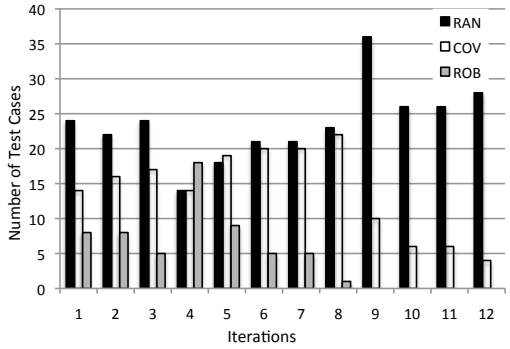

Fig. 2: Assignment of test cases to the testing criteria

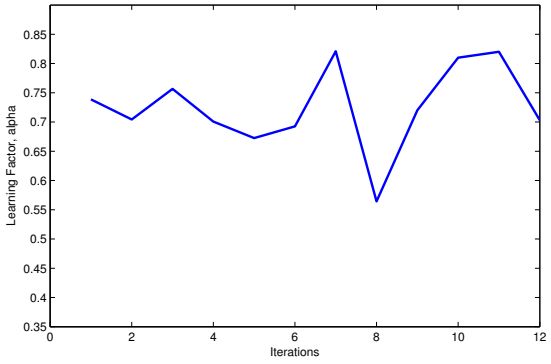

Fig. 3: Evolution of the learning factor

14, and 8 test cases over 46; then, as faults are found by the techniques, the assignment changes according to the importance sampling. Up to the iteration 8, the coverage-based and random testing perform well, and thus more test cases are assigned to them (except in iteration 4, where a high detection rate of robustness testing in the previous iterations caused the algorithm to assign 18 test cases to it). The algorithm ends up with much more test cases suggested for random testing, and with robustness testing having no test case assigned at iteration 9, due to its lower performance after iteration 4. The number of assigned test cases for each iteration is 46 up to iteration 8; then, when robustness testing receives no assignment, the number per iteration becomes 32[19]. The final result of this experiment is reported in Figures 4 and 5, which show the number of failures exposed *vs.* the number of test cases. It synthesizes the performance of *On-Off* learning compared to the others. Figure 4 compares *On-Off* learning

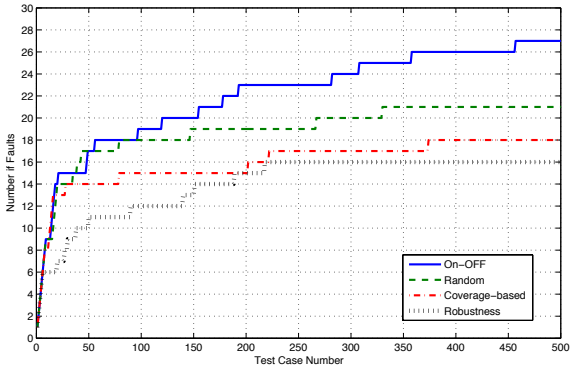

Fig. 4: Cumulative number of faults: On-Off approach *vs.* Random, Coverage-based, Robustness testing

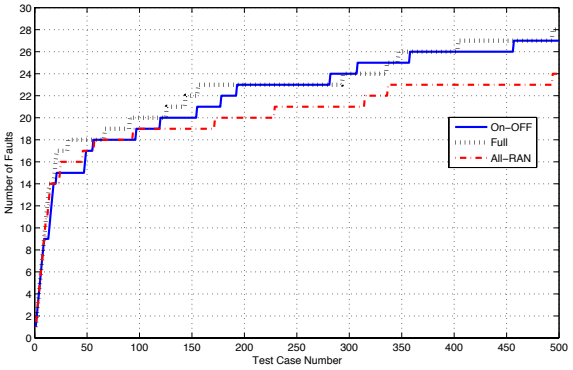

Fig. 5: Cumulative number of faults: On-Off approach *vs.* Full selection, and random selection of criterion (All-RAN)

against single applications of each testing technique. The superiority of combining techniques versus applying single techniques is confirmed. This is especially beneficial after the initial phase of testing (in our case, after about 40 % of testing time), since the detection power of single techniques starts decreasing significantly. Of the other techniques, random testing performed better, as predicted by its past performance in the three selected programs. Testing sessions in Figure 5 compare the *On-Off* method with the totally random approach (i.e., a testing criterion is selected randomly at each test case execution), and with the *full* method (for which a total of $500 \times 3 = 1500$ test cases are executed). *On-Off* method outperforms the random selection of criteria and is very close

[18]The percentage of $outcomes_{0,\tau_i}$ variables denoting a failure-exposing test case are: 0.5294, 0.2941, and 0.1764 for, respectively, random testing, coverage-based testing, and robustness testing.

[19]Iteration 12 ends with a total of 496 test cases; thus iteration 13 has only 4 test cases, and it is not reported in Figure

to the *full* method; in some cases it is even better[20], and ends with only one fault less than the *full* approach, with one-third of resources. This is due to the focusing of test selection towards criteria more effective at that time.

E. Pure Online and Offline Learning

We finally assess the strategy with respect to the availability of past information. In the previous experiment, the prediction from past data showed good performance; however this may not always be the case (e.g., due to the heterogeneity of the programs, or to the unavailability of data). Thus we consider the cases in which no past data are used (pure online learning) and in which only past data are used (i.e., pure offline learning, where the initial history-based assignment, 24, 14, 8, is kept for all the session). The pure online learning scheme leads to an initial *uniform* allocation of test cases, in contrast to the history-based scheme, and to not include the smoothing factor α in the probabilities update formula, Eq. 1. With the uniform allocation, we have: $NT^0_{\tau COV} = NT^0_{\tau ROB} = 15$ and $NT^0_{\tau RAN} = 16$[21]. The testing session conducted with this setting led to the result of Figure 6 and 7. The former applies only to online learning, since offline learning has a constant assignment of test cases. It highlights that the assignment has a more fluctuant trend than history-based scheme; this means that the allocation of test cases is more sensitive to the performance of the previous iteration. Due to the absence of the smoothing factor, the robustness testing is soon left out from the assignment, and it does not receive resources after 92 test cases. Similarly, the coverage-based method is stopped after about 200 test cases[22]. Such exclusions also determine the behaviour of Figure 7, where it is clear that the pure online learning approach gets soon saturated, since it does not give any possibility to the other techniques after just 200 test cases. The performance is high at the beginning, which apparently suggests that the history can actually be neglected. However, observing the behaviour of the pure offline learning approach, it can be noted that it starts much worse in terms of number of faults found, but toward the end of the session some more faults are detected, because coverage-based and robustness testing can still provide their contribution. Performance is worse than pure online learning but more linear. Of course without online adaptation such an approach cannot achieve performance of *On-Off* learning strategy.

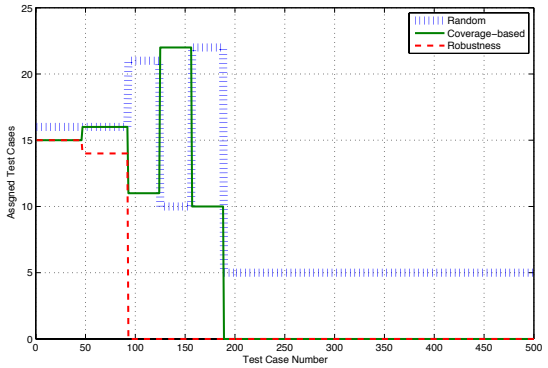

Fig. 6: Assignment of test cases to the testing criteria

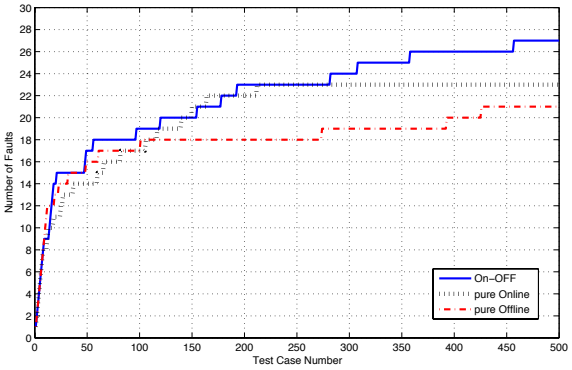

Fig. 7: Cumulative number of faults

V. Related Work

In this Section a brief overview of related work is given. Although combining testing techniques has been clearly shown to be advantageous [1], [2], [3], the problem of how to best select and combine techniques is still an open issue. Machine learning methods are very promising due to their ability to cope with uncertainty that characterizes software testing problems. Dietterich et *al.* identified in 2008 software testing as one of the most challenging domains for machine learning over the successive ten years [20]. At the same time, Namin and Sridharan [9] recently recognize that, despite the potential of learning methods, their usage (especially referring to Bayesian methods) in software testing is still in its early stages. Despite this "early stage", some relevant examples of learning methods applied to software testing appeared in the literature. In [21], authors propose to adopt decision trees to improve test suites in category-partition testing in what they called the MELAB (MachinE Learning based refinement of BlAck-box test specification) process. The same author then described some other experience in applying machine learning in testing-related problems [8], e.g., in debugging/fault localization, in fault prediction for aiding risk-driven testing, and in automatic test oracles identification. Although they are all related to testing, these do not address the problem of testing criteria selection. Online learning methods found more application for techniques selection. Our online learning phase is inspired to the work by Sridharan and Namin [18], which utilizes importance sampling for prioritizing operators in mutation testing. Their approach is aimed at finding the most important set of operators in terms of percentage of mutants left alive. Similarly, our approach aims at finding the

[20]Note that it cannot be assumed that the full method is the best ideal approach, since, the selected test cases per criterion are always a subset of all the available ones, and may be not the best test cases

[21]The spare test case is allocated to the historical best criterion

[22]Note that in this case iterations are many more than 12; the first two have 46 test cases; then, after robustness testing exclusion, they are composed of 32 test cases, and when also the coverage-based approach is excluded, they have only 5 test cases, for a total of 67 iterations.

most important testing criteria. In our case, we augmented the amount of available information by considering historical data. An approach using adaptation in test cases selection is proposed in the works by Cai about adaptive testing [22]. In adaptive testing, the software is modelled as controlled Markov chains (CMCs) and the control theory is used to act on test case selection given the feedback from the field. A further application of learning methods to testing is found in [23], where an adaptive sampling mechanism to identify feasible paths in a control flow graph with high traversing probability is presented. These are good examples of learning methods for various problems of software testing; our idea is to complement offline and online learning to combine history-based decision making and adaptive learning for testing technique selection. Besides machine learning, the problem of testing technique selection has been addressed in a different manner. For instance, Vegas and Basili [24] categorize in a characterization schema all the information, collected in interviews with experts, necessary to select techniques. Their final result is a storage scheme for future techniques selection. Even though their contribution does not provide methods for actually perform the selection, that information could be used for improving our offline machine learning approach. A qualitative selection procedure is also suggested in [25], in which the use of mapping matrix is proposed to analyze V&V techniques and their ability of finding defects. However, the work simply describes a way to organize the information about defects and testing techniques, but they just outline a flow of high-level steps with few quantitative support.

VI. Conclusion

This paper presented a method for testing techniques combination. The approach adopts methods typically used for classification/prediction problems along with a Bayesian-based method to iteratively direct the attention towards more likely failure-exposing techniques. Highly accurate predictions are obtained for testing technique performance using the defined testing session features and machine learning algorithms. These are then used in the adaptive algorithm for technique selection achieving clear improvements in fault detection effectiveness. It is worth noting that results are obtained with one experiment on medium-size C programs; repetitions of the experiment are therefore necessary to confirm the obtained results, as well as the usage of larger (and diverse) programs as case-studies, which may potentially exhibit different results. This will be a main research direction in the near future. Moreover, in the future we plan to extend the approach by accounting for the severity of failures as one additional *dynamic* metric, so as to characterize a technique also with respect to the number of more (or less) severe failures it is able to expose.

Acknowledgements

This work has been partially supported by the European Commission in the context of the FP7 project "ICEBERG", Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 324356, and by the TENACE PRIN Project (n. 20103P34XC) funded by the Italian Ministry of Education, University and Research.

References

[1] B. Littlewood, P. Popov, L. Strigini, N. Shryane, Modelling the Effects of Combining Diverse Software Fault Detection Techniques. IEEE Transactions on Software Engineering. 26(12), 1157–1167, 2000.

[2] S. Wagner, Software Quality Economics for Combining Defect-Detection Techniques. Proc. of NET.OBJECT DAYS, 559–574, 2005.

[3] R.W. Selby, Combining Software Testing Strategies: An Empirical Evaluation. Proc. of IEEE Workshop on Software Testing, 82–90, 2006.

[4] D. Zhang, J. J. P. Tsai. Machine learning and software engineering. Software Quality Journal, 11(2):87–119, 2003.

[5] C. Catal, B.M. Diri, A systematic review of software fault prediction studies. Elsevier, Expert Systems with Applications. 36 (4), 2009.

[6] L.C. Briand, Y. Labiche, X. Liu, Using Machine Learning to Support Debugging with Tarantula. Proc. IEEE International Symposium on Software Reliability Engineering, 2007.

[7] D. Zhang, Applying Machine Learning Algorithms In Software Development. Proc. of the 2000 Monterey Workshop on Modeling Software System Structures in a Fastly Moving Scenario, 275–291, 2000.

[8] L. Briand. Novel applications of machine learning in software engineering. Intl. Conference on Quality Software, 3–10, 2008. Keynote Speaker.

[9] A.S. Namin, M. Sridharan, Bayesian reasoning for software testing. Proc. of the FSE/SDP workshop on Future of software engineering research (FoSER '10), 2010.

[10] C.M. Bishop, Pattern Recognition and Machine Learning. Springer-Verlag New York, 2008.

[11] T. Menzies, J. Greenwald, A. Frank, Data Mining Static Code Attributes to Learn Defect Predictors. IEEE Trans. on Sw. Eng. 33 (1) (2007) 2–13.

[12] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, B. Murphy, Cross-project Defect Prediction–A Large Scale Experiment on Data vs. Domain vs. Process. Proc. 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2009, pp. 91–100.

[13] E. Witten, M. Frank, M. Hall, Data Mining: Practical Machine Learning Tools and Techniques, 3rd ed., Elsevier, 2011.

[14] T. Ostrand, E. Weyuker, R. Bell, Predicting the Location and Number of Faults in Large Software Systems. IEEE Trans. on Software Engineering 31 (4) (2005) 340–355.

[15] T. Menzies, A. Dekhtyar, J. Distefano, J. Greenwald, Problems with Precision: A Response to Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors', IEEE Trans. on Software Engineering 33 (9) (2007) 637–640.

[16] D. J. Sheskin, Handbook of Parametric and Nonparametric Statistical Procedures, 4th ed., Chapman & Hall/CRC, 2007.

[17] D. Fox, Adapting the Sample Size in Particle Filters through KLD-Sampling, Intl. Journal of Robotics Research, 22(12), 985–1004, 2003.

[18] M. Sridharan, A.S. Namin, Prioritizing mutation operators based on probabilistic sampling. In ISSRE, November 1–4, 2010.

[19] M. Hall, G. Holmes, Benchmarking Attribute Selection Techniques for Discrete Class Data Mining. IEEE Trans. on Knowledge and Data Engineering 15 (6) (2003) 1437–1447.

[20] T. G. Dietterich, P. Domingos, L. Getoor, S. Muggleton, P. Tadepalli. Structured Machine Learning: The Next Ten Years. Machine Learning, 73:3–23, 2008.

[21] L.C. Briand, Y. Labiche, Z. Bawar, Using Machine Learning to Refine Black-box Test Specifications and Test Suites. Proc. IEEE International Conference on Quality Software (QSIC), 2008.

[22] K. Cai, B. Gu, H. Hu, Y. Li, Adaptive software testing with fixed-memory feedback. J. Syst. Softw. 80, 8(2007), 1328–1348.

[23] N. Baskiotis, M. Sebag, M.C. Gaudel, S. Gouraud, A Machine Learning Approach for Statistical Software Testing. Proc. International Joint Conference on Artificial Intelligence, 2007.

[24] S. Vegas, V. Basili, A Characterization Schema for Software Testing Techniques. Empirical Software Engineering. 10, 437–466, 2005.

[25] M.A. Wojcicki, P. Strooper, An Iterative Empirical Strategy for the Systematic Selection of a Combination of Verification and Validation Technologies. Proc. of the 5th Intl. Workshop on Software Quality, 2007.