Empirical Software Engineering (2021) 26: 99
https://doi.org/10.1007/s10664-021-09989-x

E-APR: Mapping the effectiveness of automated program repair techniques

Aldeida Aleti[1] · **Matias Martinez**[2]

Accepted: 27 May 2021 / Published online: 13 July 2021

Abstract
Automated Program Repair (APR) is a fast growing area with numerous new techniques being developed to tackle one of the most challenging software engineering problems. APR techniques have shown promising results, giving us hope that one day it will be possible for software to repair itself. In this paper, we focus on the problem of objective performance evaluation of APR techniques. We introduce a new approach, Explaining Automated Program Repair (E-APR), which identifies features of buggy programs that explain why a particular instance is difficult for an APR technique. E-APR is used to examine the diversity and quality of the buggy programs used by most researchers, and analyse the strengths and weaknesses of existing APR techniques. E-APR visualises an instance space of buggy programs, with each buggy program represented as a point in the space. The instance space is constructed to reveal areas of hard and easy buggy programs, and enables the strengths and weaknesses of APR techniques to be identified.

Keywords Automated program repair · Software features

1 Introduction

Software can not be seen or touched, but it has a physical existence. With software embedded into many devices today, software failures have caused not only inconveniences but also tragedies, such as the deaths of patients due to massive overdose caused by an avoidable error in a radiation therapy machine (Kaner et al. 2008). A more recent case is Google's self-driving cars (controlled by software), which experienced 272 failures in less than a year. These failures would have resulted in at least 13 crushes killing their human drivers if they

Communicated by: Christoph Treude

✉ Aldeida Aleti
aldeida.aleti@monash.edu

Matias Martinez
matias.martinez@uphf.fr

[1] Faculty of Information Technology, Monash University, Melbourne, Australia

[2] Université Polytechnique Hauts-de-France, Valenciennes, France

had not intervened (Harris 2016). Software failures are also the cause of massive economical losses, costing the global economy $41 billion annually (Software 2013). Repairing software faults, however, is becoming an extremely difficult and expensive task – constituting up to 90% of the software expenses (Le Goues et al. 2013) – due to the increasing complexity and size of software systems. A modern car, for example, has 100 million lines of code, and this number is expected to increase to 200-300 millions in the near future (Charette 2009). Hence the critical task of software repair must be automated.

Automated Program Repair (APR) has been identified as the grand challenge in software engineering research (Mark Harman 2018). Many APR methods have shown promising results in fixing bugs with minimal, or even no human intervention (Le Goues et al. 2012a; Le Goues et al. 2012b; Martinez and Monperrus 2015; Xuan et al. 2017). Despite many studies introducing various APR techniques (APRTs), much remains to be learned, however, about what makes a particular technique work well (or not) for a specific software system (Anand et al. 2013). The effectiveness of APRTs is likely to be problem dependent, which calls for an analysis of the software characteristics that impact their effectiveness in order to help practitioners select the most appropriate technique for their software system.

In addition, results claiming the superior performance of an APRT over other techniques on a selected set of software systems may not generalise to untested systems. It is likely that there are software systems where an APRT excels because it is exploiting some particular characteristics of the buggy program. Thus, an understanding of conditions under which an APRT can be expected to succeed or fail is essential, however, this is rarely included in published studies. The aim of this paper is to address the issue of objective assessment of APRTs, and we achieve this by answering the following research questions:

RQ1 What impacts the effectiveness of APRTs? - Research introducing new APRTs or experimental studies investigating the performance of different techniques usually is based on a carefully selected set of buggy programs. These works offer little insight into the characteristics of the buggy programs and how they are correlated with the effectiveness of APRTs. The overwhelming majority of published work in APR only describes the benefits of the newly introduced technique, while just a few mention the limitations or present negative results.

Certain limitations of APRTs have previously been discussed in the literature, such as the issue with patch overfitting (a patch generated by a tool that, while being valid according to the correctness oracle, they are still incorrect and potentially introduce new bugs that can not be captured by the correctness oracle). On the other hand, negative results in terms of why certain techniques can not repair certain bugs have not been investigated in the literature so far. In this paper, we aim to find out if particular features of a buggy program correlate with the effectiveness of APRTs, thus providing insights on why some techniques might be more or less suited to certain software and bug instances. We achieve these kind of insights by proposing a new method for analysing the effectiveness of APRTs.

RQ2 Are APR datasets significantly different? - Most research in APR uses well-known datasets, such Defects4J, which can result in the techniques to be tailored towards solving particular problems, and as a result not generalise well for other problems. In this paper, we aim to show how different these datasets are in terms of the features that have an impact on the effectiveness of existing APRTs. This allows

us to understand if existing benchmarks are sufficiently diverse for stress-testing the effectiveness of APR techniques.

RQ3 How can we select the most suitable APR technique? The final aim of this research is to investigate the effectiveness of Machine Learning techniques for APRT selection. We investigate different multi-label classification techniques and report their effectiveness in terms of recall, precision and f1-score.

To answer these research questions, we introduce a new approach which characterises both strengths and weaknesses of existing APR techniques. E-APR is inspired from earlier work on instance space analysis in the area of machine learning (Muñoz et al. 2018) and search based software testing (SBST) (Oliveira et al. 2018, 2019). These approaches are concerned with the problem of objective performance evaluation of different algorithms used in machine learning (Muñoz et al. 2018) and SBST (Oliveira et al. 2018, 2019), and the impact of the choice of problem instances. The methodology used in these studies extend the Rice's framework (1976) with the aim of gaining insights into why some algorithms might be more or less suited to certain problem instances.

E-APR extends the methodology from Oliveira et al. (2018, 2019) and Muñoz et al. (2018) to the automated program repair problem. E-APR allows for a more objective assessment of existing APR techniques, and helps in understanding why certain APR techniques cannot generate plausible patches for certain bugs. We apply our framework on a large study of 2,141 bugs from 130 projects, and 23,551 repair attempts. E-APR uses software and bug features to characterise the buggy program instances, and learns which features have an impact on the effectiveness of APRTs. For human programmers, software repair is challenging because fixing bugs is a difficult task. While there are bugs that can be trivially fixed, many of us can remember a bug that took hours, if not days and weeks to be understood and fixed (Eisenstadt 1997). The approach we devise gives insights into how an APR technique can be selected to automatically fix bugs.

2 The E-APR Framework

The E-APR framework has two main goals:

- to help designers of APRTs gain insight into why some techniques might be more or less suited to repair certain buggy programs, thus devising new and better techniques that address any challenging areas, and
- to help software developers select the most effective APRT for their software system.

E-APR provides a way for objective assessment of the overall effectiveness of an APR technique. It is based on previous work on instance space analysis and algorithm selection in the area of Search-Based Software Testing (Oliveira et al. 2018, 2019), machine learning (Muñoz et al. 2018), and optimisation (Smith-Miles and Tan 2012). The concept of instance space analysis was first introduced by Smith-Miles in her seminal work looking at the strengths and weaknesses of optimisation problems, and forms the foundation of the E-APR approach. Understanding the effectiveness of an APR technique is critical for selecting the most suitable technique for a particular buggy program, thus avoiding trial and error application of APR techniques.

An overview of the E-APR framework is presented in Figure 1. E-APR starts with a set of buggy programs $p \in P$ and a portfolio of APRTs $t \in T$. The performance of APRTs is

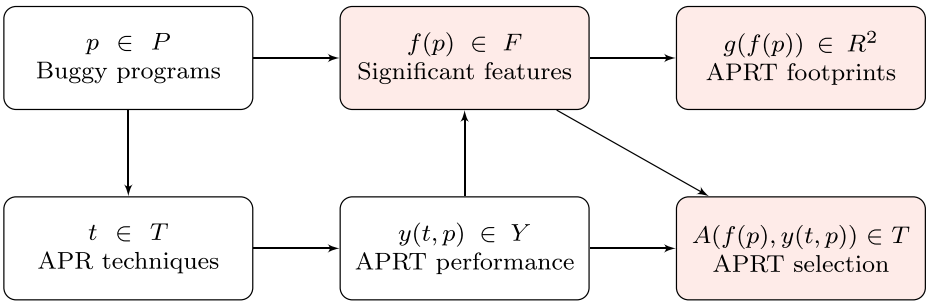

Fig. 1 An overview of E-APR

measured for each buggy program as $y(t, p)$, which indicates whether a plausible patch has been found for that program. The first step of E-APR is to identify the significant features of buggy programs ($f(p) \in F$) that have an impact on how easy or hard they are for a particular APRT. Next, E-APR constructs the APRT footprints ($g(f(P))) \in R^2$ which indicate the area of strength for each APRT. Finally, E-APR applies machine learning techniques on the most significant features to learn a model that can be used for APRT selection for future application.

2.1 Buggy Programs

Buggy Programs, defined in Figure 1 as $p \in P$ are software instances used by researchers to evaluate automated program repair techniques. Most of the APRTs for Java use Defects4J (Just et al. 2014).

Durieux et al. (2019) is one of the few that uses 5 peer-reviewed Java bug benchmarks: Bears (Madeiral et al. 2019), Bugs.jar (Saha et al. 2018), IntroClassJava and QuixBugs (Lin et al. 2017) and Defects4J (Just et al. 2014). Our analysis is based on the experimental data generated by Durieux et al. (2016b), which is available at github.com/program-repair/RepairThemAll_experiment.

In total, we consider 2,141 bugs from 130 projects, and 23,551 repair attempts. A repair attempt is the execution of an APRT on a buggy program. The execution of all repair attempts on the 5 benchmarks by the 11 APRTs took 314 days (Durieux et al. 2019). The patches considered in this study are *plausible patches*. These patches produce: a) the failing test cases (that exposed the bug) pass, and b) the remaining test cases continue to pass. Those patches are also known as *plausible* patches (Qi et al. 2015). Previous work have shown that a test-suite adequate patch can produce passing all tests but they are yet incorrect. Those are *overfitting* patches (Smith et al. 2015) and can arise due to the weakness of the test-suite used for synthesising the patches. Overfitting detection is not yet mature (i.e., not capable of detecting all overfitting patches) and thus adopting such techniques could introduce some bias in this work, hence we consider *all* patches generated by the repair tools executed by RepairThemAll. This means that we did not filter out the outputs generated by APRTs.

The source of the bugs in the bug benchmark are diverse: Defects4J and Bugs.jar contains real bugs extracted from software repositories, Bears contains real bugs collected from breaking builds on Travis platforms, IntroClassJava contains buggy subjects from students, and QuixBugs contains buggy implementation of well-known algorithms (such as merge-sort).

2.2 APR Techniques

APR techniques are defined in Figure 1 as $t \in T$. In this paper we focus on one family of repair approaches: *test-suite based repair approaches* (Le Goues et al. 2012a). Approaches from this family aim at repairing bugs exposed by at least one failing test case. The main idea of these approaches is to use failed test cases to localise potential faults and then apply mutations to the source code until the program satisfies all unit test cases. The mutations that are applied to the program code can range from small changes like modification, addition or removal of a single code line (Le Goues et al. 2012a) to complex edit operations (Martinez and Monperrus 2015; Kim et al. 2013), which are mined from software repositories and used to fix a fault in a different context.

In this paper we employ 11 repair tools for Java programs similar to the study by Durieux et al. (2019). These tools can be classified into *semantics-based repair tools* (Nopol (Xuan et al. 2017) and DynaMoth (Durieux and Monperrus 2016a)), *a metaprogramming-based tool* (NPEFix (Durieux et al. 2017)), and *generate-and-validate* (ARJA (Yuan and Banzhaf 2018), Cardumen (Martinez M and Monperrus 2018), jGenProg (Martinez M and Monperrus 2016), GenProg-A (Yuan and Banzhaf 2018), jKali (Martinez et al. 2017a), Kali-A (Yuan and Banzhaf 2018), jMutRepair (Martinez M and Monperrus 2016), and RSRepair-A (Yuan and Banzhaf 2018)).

jGenProg and GenProg-A are two Java implementations of GenProg (Le Goues et al. 2012a). Both techniques use a generate-and-validate method to produce patches using a genetic programming approach. The search space consists of patches that are formed through combinations of removing code, and inserting and replacing code from elsewhere in the program under repair (Martinez M and Monperrus 2016).

Cardumen (Martinez M and Monperrus 2018) synthesises patches using the existing code as a basis, by taking code elements from elsewhere in the program and replacing the variables. Each potential patch is filtered based on location and type compatibility, and the remaining patches are prioritised based on how frequently the selected variables occur together.

jKali and Kali-A (Qi et al. 2015) are different implementations of Kali in Java. They attempt to come up with candidate patches by removing or skipping statements. Neither jKali nor Kali-A is a 'repair' program, instead, they are more useful in identifying weak test suites and under-specified bugs (Martinez et al. 2017b). Since Kali simply removes or skips code, if a patch is found, it is a strong indication that the functionally of the removed code is not specified in the test-suite. In addition, if Kali finds a test-suite adequate patch, so can jGenProg or Nopol (Martinez et al. 2017b), the patches found by Kali, however, rarely work beyond the given test-suite.

jMutRepair (Martinez M and Monperrus 2016) performs an exhaustive search of the code and applies the following three types mutation operators on suspicious if conditions. The relational mutation operator with the following values (==,!=,$\leq$, $\geq$, $<$, $>$), the logical mutation operator (AND, OR), and the *Unary* mutation operator which applies negation and positivation.

Nopol (Xuan et al. 2017) focuses on repairing IF conditions, which are amongst the most error-prone elements of Java programs, and many one-change commits simply update an IF condition. Nopol has three main steps. First, it locates a fix location for a potential patch using "angelic fix localisation". This process also involved finding "angelic values", which

are assigned values that can be used at the fix location to make all failing tests pass. Next, Nopol collects runtime data from a test execution, including a snapshot of the program state at candidate fix locations. Then, Nopol translates the angelic values and available variables at the fix location into a Satisfiability Modulo Theorem problem, and attempts to find a solution, which is then translated into a patch.

RSRepair-A (Qi et al. 2014) is a Java implementation of the RSRepair program repair tool written for C programs. RSRepair uses a generate-and-validate technique to prepare patches. It takes inspiration from the GenProg tool, however, instead of using genetic programming as its search method, RSRepair uses random search.

ARJA (Yuan and Banzhaf 2018) uses Genetic Programming to modify and mutate suspicious statements in a program by performing three actions: i) deleting the suspicious statement, ii) replacing the suspicious statement, or iii) inserting extra statements before or after the suspicious statement. ARJA reduces the scope of the search and computation time to speed up the fitness process by applying rules that exclude statements that are not related to the problem (Yuan and Banzhaf 2018).

NPEFix (Durieux et al. 2017) repairs null pointer exceptions at runtime by using two strategies. The first strategy assigns an alternative value (which can be a valid value that is stored in another variable or a random value) for a null dereference. The second strategy skips the execution of the null dereference, by either skipping a single statement or skipping the complete method. All strategies are applicable for any arbitrary objects, including instances of library classes, and instances of domain classes.

In summary, the APR techniques discussed in this section can be broadly categorised based on their high-level repair strategy. For example, jGenProg (Martinez M and Monperrus 2016), ARJA (Yuan and Banzhaf 2018) and RSRepair-A (Qi et al. 2014) use or build upon genetic programming. Other techniques take more unique approaches and are designed to target specific bugs, like NPEFix (Durieux et al. 2017) targeting null pointer exceptions. Other repair tools can only function if code is structured in a certain way, like Nopol (Xuan et al. 2017), which only works when IF conditions are present, and will only find a valid patch if the patch involves changing IF conditions. These observations further support our hypothesise that the performance of each technique will likely be affected by the features of the code. Different repair strategies may favour different code features, and that different bug targeting will definitely perform badly on code with the wrong type of bug.

2.3 APRT Performance Measures

An APRT Performance Measures $y(t, p) \in Y$ takes as input the patches generated by an APRT $t \in T$ for a particular buggy program $p \in P$. There exist various measures of APRT performance focusing on the quality of the patches produced. In this work we consider *test-suite adequate* patches (Le Goues et al. 2012a). We acknowledge that a portion of the patches may be overfitting, i.e., according to the test suite, the buggy program may appear to have been fixed by the patch, however, new errors may have been introduced. The problem of filtering correct patches (e.g., (Yu et al. 2019; Xiong et al. 2018; Xin and Reiss 2017; Le et al. 2019)) is currently being addressed by many researchers, who are looking at ways for automating or semi-automating this process, since manually inspecting all generated patches by automated program repair techniques is not practical. The APRT performance measures in E-APR ($y(t, p) \in Y$) can easily be extended to new measures, such as patch correctness.

2.4 Significant Features

A critical step of E-APR is identifying features of buggy program instances $f(p) \in F$ that have an impact on the effectiveness of APR techniques. Features are problem dependent and must be chosen such that the varying complexities of the buggy program instances are exposed, any known structural properties of the software systems are captured, and any known advantages and limitations of the different APRTs are related to features.

For the purpose of this work, an APR technique is effective if it can generate a plausible patch for a buggy software system. While much is known and reported on features that correlate with software quality, we must consider that there may be other unknown features that have an impact on the effectiveness of APR techniques. In addition, it is possible that not all known features are useful for our goal of separating the hard and easy software instances. The candidate set of features may contain redundancy, with features measuring aspects of a buggy program that are either similar or not relevant to expose the hardness of the APR task itself. Thus, a small set of relevant features must be selected.

Learning significant features has two steps: first we define how to measure the quality of a particular set of features, and second, we apply a Genetic Algorithm to select the set that maximises this measure. A subset of features is considered of high quality if they result in an instance space – as defined by the 2-dimensional projection of the subset of features – with buggy programs that show similar performance of APRTs closer to each other. The best subset of features is the one that can best discriminate between easy and hard buggy program instances for APR techniques.

E-APR aims at identifying features that are able to create a clear separation of the buggy program instances, such that we can clearly see the different clusters of buggy programs where each APRT is effective. We employ principal component analysis (PCA) (Jolliffe 2011) to locate significant features. PCA learns a linear combinations of the buggy program features. The first PC is the linear combination of the variables which explain the maximum amount of variance in the dataset. Each subsequent PC is orthogonal to all previously calculated PCs and captures a maximum variance under these conditions. In our work, the subset of variables that have large coefficients and therefore contribute significantly to the variance of each PC, are identified as the significant features which are selected to explain bugs.

Given $|F|$ software features, we can have at most $|F|$ components which are estimated in decreasing order of the variance (measured through the eigenvalue of each PC) they explain in the dataset. We analyse for each PC the features that are found significant. This shows which dimensions are the main drivers of APR technique effectiveness and help explain why this is the case. In PCA, usually only the first few components are regarded as important. We retain the first 2 components, which makes visualising the footprints of the algorithms much easier.

E-APR uses a genetic algorithm (Aleti et al. 2014) to search the space of possible subsets of k features, with the classification accuracy on an out-of-sample test set used as the fitness function to guide the search for the optimal subset. The instance space is generated in iterations, until an optimal subset of features is found (Muñoz et al. 2018). The genetic algorithm performs the following steps to select the features and generate the instance space:

1. a set of buggy program features is selected;
2. an instance space is generated using the selected features and PCA to reduce the dimensionality;
3. the fitness of the set of features is evaluated ;
4. if the features are not adequate, go back to step 1.

Once the best set of features features is identified, E-APR creates a 2-D instance space that helps inspect the relationships between problem instances, their features and objectively assess APRT performance. 2D visualisation has been found to be effective in visualising footprints (Oliveira et al. 2018, 2019; Muñoz et al. 2018), hence we follow a similar approach as previous work. Similar approaches have been proposed in the literature for feature subset selection for machine learning (Bengio and Chapados 2003), optimisation (Smith-Miles et al. 2014), and search-based software testing tasks (Oliveira et al. 2018). Certainly, other feature selection methods proposed in the literature (Guyon and Elisseeff 2003) would also be suitable for the task at hand.

2.5 APRT Footprints

The idea of algorithm footprints was first introduced by Smith-Miles and Tan (2012) and aims to determine the relative performance of different algorithms across various classes of instances. In the original paper (Smith-Miles and Tan 2012), the authors focused on optimisation problems. Rather than reporting algorithm performance averaged across a chosen set of benchmark instances, the authors develop metrics for an algorithm's performance generalised across a diverse set of instances. E-APR extends these ideas to Automated Program Repair techniques and aims to measure APRT footprint, which gives an indication of the area of strength of these algorithms.

Once the significant features have been identified, they are used to analyse and visualise the footprints of the APR techniques. In order to facilitate the visualisation of the footprints, similar to previous work (Smith-Miles and Tan 2012; Oliveira et al. 2018, 2019), we utilise the 2-D instance space created using PCA as a dimensionality reduction technique, and project the instances to two dimensions, while making sure that we retain as much information as possible. PCA rotates the data to a new coordinate system $\mathbf{R}^k$, with axes defined by linear combinations of the selected F^* features, where $k = |F^*|$. The k new axes are the eigenvectors of the $k \times k$ covariance matrix.

We retain the two principal eigenvectors which correspond to the two largest eigenvalues of the covariance matrix. The instance space is then projected on this two-dimensional space. We use the variance explained in the data by the two principal components as a measure of the loss in information due to dimensionality reduction. Following a similar approach to previous work on dimensionality reduction (Smith-Miles et al. 2014), we accept the new two dimensional instance space as adequate if most of the variance in the data is explained by the two principal axes. The two principal components z_1 and z_2 are then used to visualise the footprints of the APR technique (APRT).

If our goal was only to make performance predictions on the best APR tool for repairing a particular software system, we could use machine learning algorithms to identify the relationship between software features and APR performance. Machine learning on its own does not allow for explanations as to why a particular APRT works well. Our goal in this paper is much broader than only making prediction, as we aim to visualise the footprints of the different APR approaches and provide insights into the workings of these methods.

Next, we calculate the relative size of APRT footprints by estimating the area of the hull covering the software instances where the technique is expected to perform well. This is a metric of the relative goodness of the APRT across the software instance space. Formally, given the convex hull $H(S)$ of an area defined by points $S = \{(x_i, y_i)\}, \forall i = 1, ..., n$, the

area $A(H(S))$ is given by

$$A(H(S)) = \frac{1}{2}\sum_{j=1}^{k}(x_j y_{j+1} - y_j x_{j+1}) + (x_k y_1 - y_k x_1), \quad (1)$$

where the subset $\{(x_j, y_j), \forall j = 1, ..., k\}, k \leq n$ defines the extreme points of $H(S)$. Using (1), we compare the relative size of the footprint of each APRT to determine which APRT has the largest footprint and explore the degree of overlap of the footprints.

2.6 APRT Selection

In the final step, E-APR predicts, based on the most significant software features, the most effective APR technique for repairing particular buggy programs. E-APR uses the most significant features as an input to machine learning algorithms to learn the relationship between the instance features and APR method performance. For this purpose, we can use a variety of machine learning algorithms, such as decision trees, or support vector machines for binary labels (effective/ineffective), or statistical prediction methods, such as regression algorithms or neural networks for continuous labels (e.g., time complexity of the approach).

In this work, we investigate four machine learning approaches for multi-label classification (Madjarov et al. 2012). These methods are support vector machine (SVM) (Boser et al. 1992), a random forest classifier (RFC) (Prabhu and Varma 2014), a decision tree (DT) (Quinlan 1996) and a multi-layer perceptron (MLP) (Ruck et al. 1990). We now briefly describe those techniques.

Support Vector Machine (SVM) is a supervised learning model with associated learning algorithms that analyse data for classification and regression analysis. For classification, SVM aims at finding a hyper-plane in the feature space, which separates the training data into two classes while maximising the margin (in the feature space) between this hyper-plane and the two classes (Vapnik 1995).

Decision Tree (DT) uses observations about an item (represented in the branches) to learn an item's target value (represented in the leaves). Classification trees are those trees where the target variable can take a discrete set of value, leaves represent class labels and branches represent conjunctions of features that lead to those class labels.

Random Forest Classifier (RFC) is an ensemble learning method for classification that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes.

Multi-Layer Perceptron (MLP) consists of a feed-forward artificial neural network which is a system of interconnected neurons representing a nonlinear mapping between an input vector and an output vector. MLP is used for classification by assigning output nodes to represent each class. MLPs are typically trained using a supervised learning technique called back-propagation.

At the end of this process, E-APR produces a model that can be used for algorithms selection in automated program repair. This model can be retrained and extended with more APR tools and features.

3 Experimental Design

We implement the E-APR framework described in Section 2, and conduct a set of experiments and analysis to answer the research questions stated in Section 1. In this section, we describe: the automated program repair techniques, the benchmark of buggy programs, and the set of software features.

3.1 Buggy Program Features

Features are problem dependent and must be chosen so that the varying complexities of the problem instances are exposed, any known structural properties of the buggy programs are captured, and any known advantages and limitations of the different program repair techniques are related to features. The most common measures and metrics used to characterise features of a software system are extracted from code.

Among others, we use object-oriented code metrics based on measurement theory and expertise of experienced software developers (Chidamber and Kemerer 1994b). These metrics are also mapped to the Quality Model for Object-Oriented Design (El-Wakil et al. 2004), which is a comprehensive model that establishes a clearly defined and empirically validated model to assess object-oriented design quality attributes such as understandability and reusability, and relates them through mathematical formulas with structural object-oriented design properties such as encapsulation and coupling. The set of code metrics, presented in Table 1, includes simple metrics, which count the number of methods or lines of code, to more complex metrics that measure the interaction between methods and the depth of inheritance tree. As in this paper we focus on Java APR, we also include Java-Specific method features, which are presented in Table 2.

In addition to code features widely used by software practitioners and researchers, we also consider a set of Observation-based features (Yu et al. 2019). Those features were manually crafted for targeting different open challenges from automated program repair's field such as prediction of source code transformations on buggy code (Yu et al. 2019) and detection of incorrect patches (Ye et al. 2019).

Observation-based features capture different characteristics of a buggy program. Initially, Defects4J (Just et al. 2014) was considered as a starting point, which is a dataset of real Java bugs and the corresponding human-written patches, widely used in evaluations of automated program repair tools (Durieux et al. 2019). We recorded the following information for each code element in the buggy code affected by the patch and in the patched code: *a*) the characteristics of the elements (e.g., the type of a variable is primitive), and *b*) the relation of such elements with respect to the rest of the buggy and patched file, respectively. Finally, the designers defined a set of features from those observations. For example, from Listing 1, it was observed that the buggy statement references to a variable ($p1$) which has compatible type and similar name to another variable ($p2$) in scope. From this observation, they created a feature named "HVSN" (Has Variable with Similar Name).

Observation-based features were included in the set of buggy program features because they allow us to capture the characteristics of code elements related to bug that: *a)* can be repaired by a tool, and *b)* cannot be repaired by any tool.

Thus, our approach could predict whether a buggy program can be repaired (or not) by a particular repair tool, and to determine which is the most adequate repair tool to face the bug. A simple example to illustrate the intention behind the adoption of such features: the buggy version of bug Chart-11 from Listing 1 has the feature $HVSN$ with a $true$ value and it is successfully repaired by Cardumen (Martinez M and Monperrus 2018) but neither by

Table 1 Object-oriented features

WMC	Weighted methods per class is a measure of complexity in a class (Chidamber and Kemerer 1994a).
DIT	Depth of inheritance is the depth of inheritance of the class (i.e. number of ancestors in direct lineage) (Chidamber and Kemerer 1994a).
NOC	Number of children is an indication of the scope of properties. It counts the sub-classes that inherit the methods of the parent class (Chidamber and Kemerer 1994a).
CBO	Coupling between object classes is a count of the number of other classes to which the current class is coupled (Chidamber and Kemerer 1994a).
RFC	Response for a class measures the interaction of the class' methods with other methods (Chidamber and Kemerer 1994a).
LCOM	Lack of cohesion in methods. This metric counts the sets of methods in a class that are not related through the sharing of some of the class's fields (Chidamber and Kemerer 1994a).
CA	Afferent coupling is a measure of how many other classes use the specific class.
CE	Efferent couplings. This is a measure of how many other classes are called within the given class.
LCOM3	Lack of cohesion in methods. This metric is defined as the number of connected components in the call graph.
NPM	Number of public methods for a class
LOC	Lines of code. As the name indicates, this measure counts the lines of code in a class. We take the average lines of code per class in a buggy program.
DAM	Data access metric. This metric is the ratio of the number of private and protected attributes to the total number of attributes declared in the class.
MOA	Measure of aggregation. This is the percentage of data declaration in the system whose types are of user defined classes (i.e., data types other than system defined classes such as integers, real numbers etc).
MFA	Measure of functional abstraction is the ratio of the number of methods inherited by a class to the total number of methods accessible by members in the class.
CAM	Cohesion among methods of class computes the relatedness among methods of a class based upon the parameter list of the methods.
IC	Inheritance coupling calculates the number of parent classes to which a given class is coupled.
CBM	Coupling between methods measures the total number of new/redefined methods to which all the inherited methods are coupled.
AMC	Average method complexity measures the average method size (the number of java binary codes in the method) for each class.

for instance Arja nor GenProg (Durieux et al. 2019). Thus, our intuition is that other bugs having that feature could be repaired by Cardumen.

Observation-based features are grouped into three categories: *1)* features related to the *Usage* of code elements, for example, the feature OUIA indicates if a statement references a local variable that has not been referenced in other statements before it, *2)* features related to the *Syntax* of code elements, for example, the feature HVSN (Has Variable with Similar Name) indicates whether, given a statement that references a variable, there exist other vari-

Table 2 Java specific method features

AC	Abstract methods count is the number of abstract methods in a class.
ASMC	Abstract static methods count is the number of static methods in a class.
DAMC	Default abstract methods count.
DASMC	Default abstract static methods count.
DMC	Default methods.
DSM	Default static methods count.
GMC	General methods count
GSMC	General static methods count
MC	Methods count.
PriAMC	Private abstract methods count.
PriASMC	Private abstract static methods count.
PMC	Private methods count.
PSMC	Private static methods count.
ProAMC	Protected abstract methods count.
ProASMC	Protected abstract static methods count.
ProMC	Protected methods count.
ProSMC	Protected static methods count.
PubAMC	Public abstract methods count.
PubASMC	Public abstract static methods count.
PubMC	Public methods count.
PubSMC	Public static methods count.
SMC	Static methods count
Observation-based Features (see complete list at (Yu et al. 2019))	
Usage	Related to usage of e.g. variables and invocations
Syntax	Related to syntax of e.g. variable's identifiers
Types	Related to types of e.g. variables, and parameters.

ables in the same scope that have a similar identifier name with that variable; *3)* features related to the *Types* of code elements, for example, the feature VTSV indicates whether, given a statement that references a variable, there exist other variables in the same scope that are type compatible with that variable.

In total, we have 146 Observation-based features. The complete list is available in our (Appendix 2020). These features can be computed using the open-source tool Coming (Martinez and Monperrus 2019), which is available online at https://github.com/SpoonLabs/coming.

```
@@ -272,7 +272,7 @@ public static boolean equal(
        GeneralPath p1, GeneralPath p2)

        PathIterator iterator1 = p1.getPathIterator(null);
-       PathIterator iterator2 = p1.getPathIterator(null);
+       PathIterator iterator2 = p2.getPathIterator(null);
```

Listing 1 Human-written for bug Chart-11 from Defects4J

3.2 E-APR Input Data

For each buggy program, we first create a vector where each dimension corresponds to a particular feature. We add to that vector an additional dimension per each APRT considered in this experiment: its value is '1' if the corresponding APRT produced a plausible patch and a '0' otherwise. Table 3 shows an example of the features extracted from 4 buggy programs. Each row has the values of the features extracted for a program, and it is a vector of features. From the second to the fifth column, it shows the values corresponding to 4 object-oriented features (wmc, dit, npc and cbo). The last two columns indicate whether the buggy program could be repaired by two approaches (Kali and Arja).

To create a vector with features for each buggy program, we compute the Object-oriented and Java-Specific method features, which are calculated at the *class-level*. Then we calculate the average value of these features over all classes for each buggy program. Next, we compute the Observation-based features. Instead of considering all statements from the buggy program, we focus on a subset of them: those that, with a given probability, could have the bug. Note that, for predicting which is the most suitable tool given a program bug (Section 4.3), our approach does not know in which statement(s) the bug is located or the human patch. For this reason we apply fault localisation to filter the statements. To retrieve those statements, we compute the suspicious value of each statement using GZoltar tool (Campos et al. 2012), which uses the Ochiai formula (Abreu et al. 2007) to compute the suspiciousness value. GZoltar is the most prominent fault localisation tool used by the Java repair systems considered in this study. For each buggy program, we select the 100 most suspicious statements returned by GZoltar. We consider 100 as it is a common cut-off value used in program repair experiment, e.g., see analysis from (Long and Rinard 2016a). If a patch we obtain from our dataset is applied in a statement not included in the mentioned list of suspicious statements returned by the fault localisation tool, we include that statement in the list, with the goal of also analysing it. We found 75 bugs having, at least, one patch of such case. Next, we compute the Observation-based features for each of those statements. Finally, we compute the average of the features that characterise the suspicious statements.

4 Results

We present the results for each research question, and aim to provide insights into why the different APR techniques work. First we present the most significant features that impact

Table 3 A snapshot of the dataset

Buggy program	wmc	dit	noc	cbo	Kali	Arja
Jackrabbit	9.37	0.78	0.23	12.51	1	0
Accumulo	11.94	0.81	0.22	13.23	1	0
Flink	8.43	0.75	0.31	10.79	1	1
Wicket	8.84	0.58	0.41	11.01	0	1

All buggy programs in this example are from project Bugs.jar

APRT effectiveness. Second, we investigate the diversity of exiting buggy datasets used for APR. Next, we investigate the differences between exiting APRTs by analysing their strengths and weaknesses using the most significant features. Finally, we present the results from the Machine Learning algorithms used for APRT selection.

4.1 RQ1. What impacts the effectiveness of existing APRTs?

We performed feature learning on the total list of features (described in Section 3.1) that were extracted from **1,282** buggy programs. The aim is to select the best set of features that highlights the strengths and weaknesses of the APR techniques. To account for the randomness in the results, each trial of feature learning was run 10 times on each buggy program for each approach, using different random seeds, and the mean was considered. Out of the 146 features that were part of the study, E-APR identified the following 9 optimal features which best capture the difficulty in generating patches for APR:

(F1) **MOA:** Measure of Aggregation.
(F2) **CAM:** Cohesion Among Methods
(F3) **AMC:** Average Method Complexity
(F4) **PMC:** Private Method Count
(F5) **AECSL:** Atomic Expression Comparison Same Left indicates the number of statements with a binary expression that have more than an atomic expression (e.g., variable access). This feature belongs to *Syntax* category.
(F6) **SPTWNG:** Similar Primitive Type With Normal Guard indicates the number of statements that contain a variable (local or global) that is also used in another statement contained inside a guard (i.e., an If condition). This feature belongs to *Usage* category.
(F7) **CVNI:** Compatible Variable Not Included is the number of local primitive type variables within the scope of a statement that involves primitive variables that are not part of that statement. This feature belongs to *Usage* category.
(F8) **VCTC:** Variable Compatible Type in Condition measures the number of variables within an If condition that are compatible with another variable in the scope. This feature belongs to *Type* category.
(F9) **PUIA:** Primitive Used In Assignment measures the number of primitive variables in assignments. This feature belongs to *Type* category.

Using these features we were able to define the footprints of the techniques with with the highest topological preservation of 87% (explained variance). In essence, we can conclude the following.

> **RQ1:** The most significant features that have an impact on the effectiveness of APR techniques are the Object-Oriented Features: MOA, CAM, AMC, PMC, and the Observation-based feature: AECSL, SPTWNG, CVNI, VCTC, and PUIA.

To visualise the results in a 2-D instance space, we apply PCA as a dimensionality reduction technique on the optimal subset of features. Two new axes were created, which are linear combinations of the selected set of most significant features. The coordinate system

that defines the new instance space is defined as:

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} 0.38 & -0.02 \\ -0.16 & 0.19 \\ 0.37 & -0.04 \\ -0.06 & 0.36 \\ 0.08 & 0.28 \\ 0.17 & 0.22 \\ 0.07 & 0.31 \\ -0.34 & 0.01 \\ 0.12 & 0.16 \end{bmatrix}^T \begin{bmatrix} \text{MOA} \\ \text{AECSL} \\ \text{PMC} \\ \text{SPTWNG} \\ \text{AMC} \\ \text{CVNI} \\ \text{VCTC} \\ \text{CAM} \\ \text{PUIA} \end{bmatrix} \tag{2}$$

The new coordinates (depicted in (2)) are a combination of the 9 features. CAM, PMC and MOA have the highest contribution on z_1, and SPTWNG, AMC and VCTC contribute the most to z_2. CVNI, AECSL, and PUIA contribute equally to both coordinates.

We plot the footprints of the 11 APRTs in Figure 2. Each point in the 2-D instance space represents a buggy program. If an APR technique produced a patch for a particular program, it is considered Effective, otherwise, we label it as Ineffective. Each graph in Figure 2 represents the footprint of one of the techniques that we study in this paper. The x-axis and y-axis are the two principal components z_1 and z_2, defined in (2).

A visual inspection of the footprints shows that while some techniques appear more similar than others (for example, jKali is more similar to jMutRepair than NPEFix), each technique has its unique strengths.

All APRTs apart form NPEFix repaired bugs located at the top-right of the instance space. These are bugs from Defects4J benchmark (see Figure 4), which confirms a long held hypothesis that APRTs are being perfected to repair bugs from this dataset.

4.1.1 Footprints Size

Table 4 shows the area size of the APRT footprints, measured using (1). The size of the footprint is an indication of the overall performance of the APRT. The larger the footprint, the more diverse bugs an APRT can repair.

While most techniques have relatively similar footprint size, jGenProg is the winner. The footprint size is not based on the number of programs that a technique was able to repair. Instead, the effectiveness of an APRT is measured in terms of the diversity of the features of these programs and their spread in the instance space. An APRT that can repair more diverse bugs is considered to be more effective.

4.1.2 Significant Software Features

Figure 3 depicts the feature footprints, which shows how the buggy program instances score in terms of the most significant features.

MOA and PMC The cluster of buggy programs where mostly jMutRepair, RSRepair, Nopol, GenProgA and Arja are effective has a lower measure of aggregation (MOA) and private methods count (PMC). MOA (as defined in Table 1) is the percentage of data declaration in the system whose types are of user defined classes, as opposed to those of system defined classes, such as integers, real numbers etc. It indicates that, compared to other approaches, it is easier for jMutRepair, RSRepair, Nopol, GenProgA and Arja to repair bugs originating from buggy programs that have fewer user declared types and lower number of private

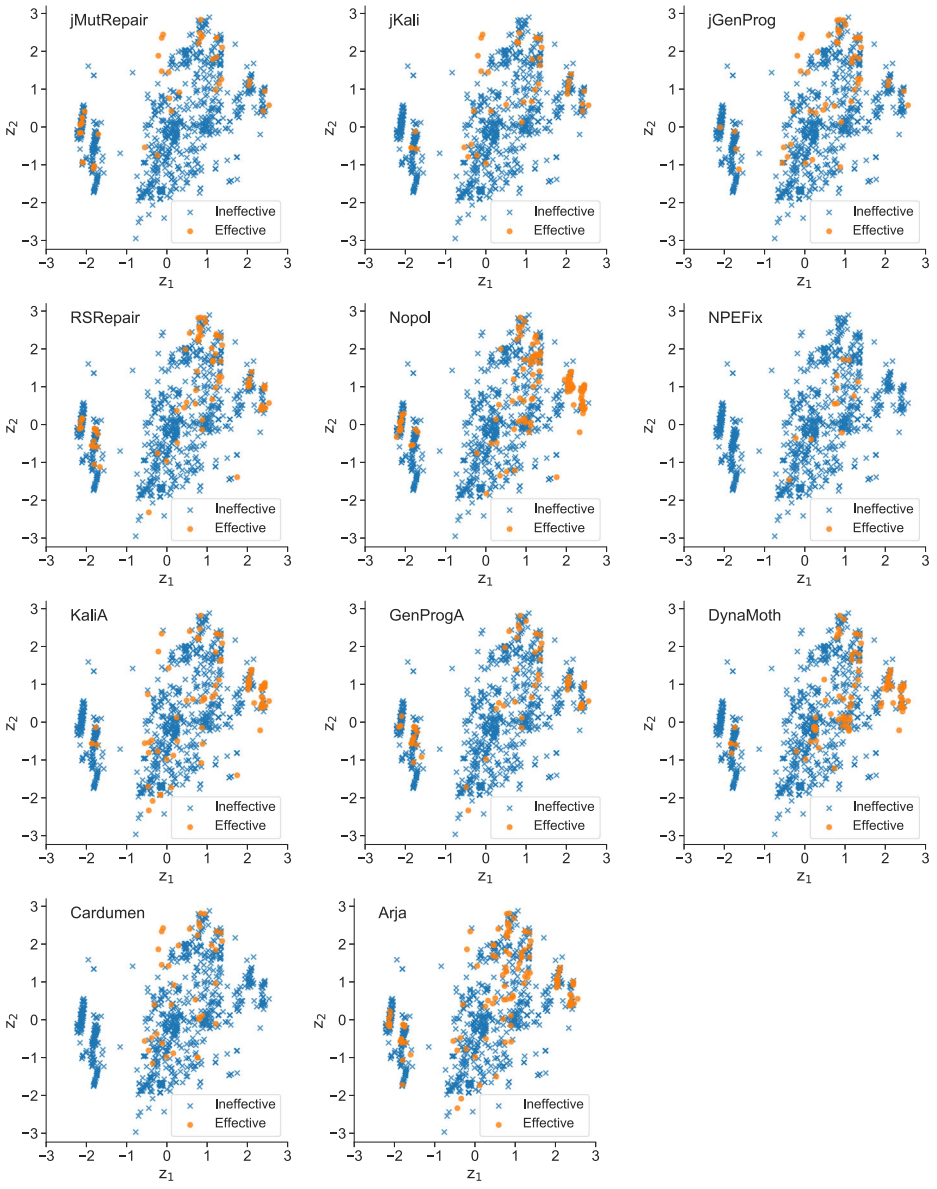

Fig. 2 APR technique footprints. Each point is a buggy class, and is labelled as Effective, if the technique was able to generate a plausible patch for it

methods. jMutRepair, RSRepair, GenProgA and Arja are from the class of generate-and-validate APR techniques, with GenProgA and RSRepair being variations of GenProg. These tools make use of mutation operators to generate patches, which in general can be an effective way to fix bugs, but it proves ineffective in programs with many private methods and user defined types. This indicates that more sophisticated operators are required to fix such programs.

Table 4 Performance differences between the APRTs

APRT	Footprint size	APRT	Footprint size
jMutRepair	0.223	jKali	0.215
jGenProg	**0.388**	RSRepair	0.006
Nopol	0.236	NPEFix	0
KaliA	0.052	GenProgA	0.004
DynaMoth	0.169	Cardumen	0.257
Arja	0.016		

CAM The third most significant feature is cohesion among methods in a class (CAM), which is a measure of class cohesion. The cluster of buggy programs where mostly jMutRepair, RSRepair, Nopol, GenProgA and Arja are effective is high in terms of CAM. High class cohesion is a desirable property and has previously been linked with high software quality. As mentioned above, MutRepair, RSRepair, GenProgA and Arja use mutation to generate new patches for buggy programs, which is quite simple and works well with highly cohesive programs where related program elements are in the same place (in this case, class). Mutation applies random changes in code, and is less likely to introduce new bugs if classes are highly cohesive. On the other hand, DynaMoth is effective at repairing bugs

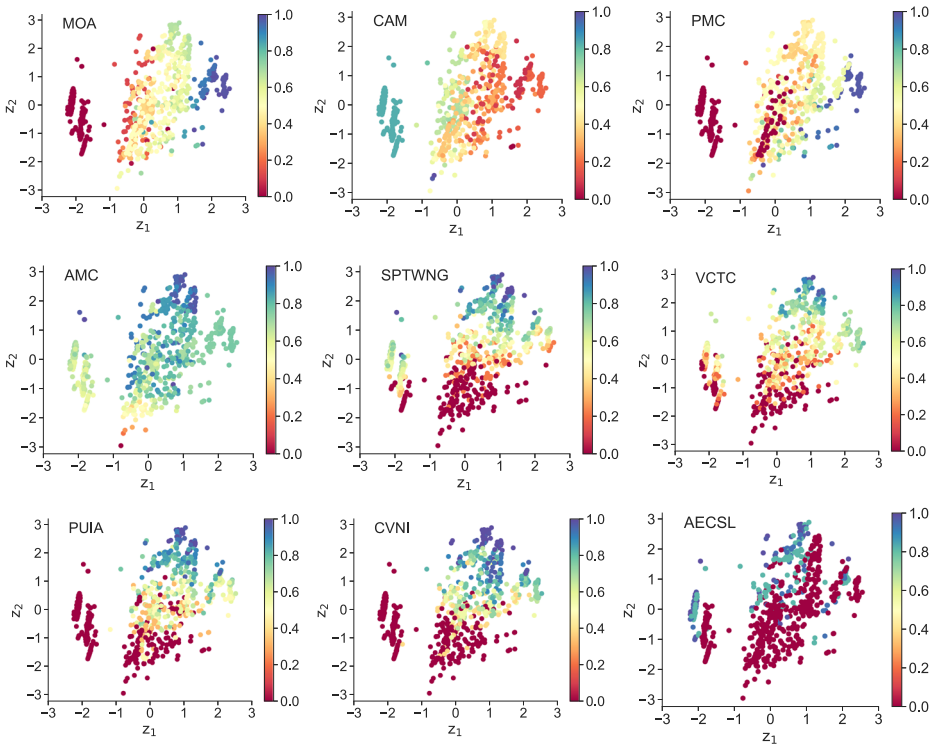

Fig. 3 Feature footprints. The values of features have been normalised between 0 and 1, and the colour scheme is used to represent the values of features

from programs with low cohesion. DynaMoth is a semantic-based APR tool, which performs a dynamic synthesis of patches for repairing conditional bugs. The tool specifically addresses the issue of complex method calls and low cohesion, which explains its superior performance in buggy programs with low CAM.

AMC Average Method Complexity is relatively high in the upper right part of the instance space, where most APRTs are able to generate plausible patches. AMC is defined as the average method size (the number of java binary codes in the method) for each class, indicating that APRTs are usually more effective with longer methods.

Observation-based features These metrics capture different characteristics of the buggy parts of the programs. Out of the 146 features, E-APR identified 5 significant Observation-based – SPTWNG, VCTC, PUIA, CVNI and AECSL – whose footprints we show in Figure 3. Four of these five features – SPTWNG, VCTC, PUIA, CVNI – have very similar footprints.

SPTWNG (Similar Primitive Type With Normal Guard) indicates the number of statements that contain a variable (local or global) that is also used in another statement contained inside a guard (i.e., an If condition). VCTC (Variable Compatible Type in Condition) measures the number of variables within an If condition that are compatible with another variable in the scope. PUIA (Primitive Used In Assignment) measures the number of primitive variables in assignments. CVNI (Compatible Variable Not Included) is the number of local primitive type variables within the scope of a statement that involves primitive variables that are not part of that statement. Finally, AECSL (Atomic Expression Comparison Same Left) indicates the number of statements with a binary expression that have more than an atomic expression (e.g., variable access). Programs with a high value of these features are more likely to be repaired by most techniques, while jMutRepair, Arja, KaliA, Nopol and RSRepair can generate plausible patches even for programs with low feature values. Since these features measure properties of the potential buggy locations, it makes sense that programs with such high feature values are more likely to be repaired.

In summary, the effectiveness of APRTs is impacted by software features, which makes these methods problem dependent, and as such, no technique can be considered the best in all cases. We observe different strengths and weaknesses of existing APRTs, which calls for methods that make it possible to select the most suitable technique given a buggy program with particular features.

4.2 RQ2. Are existing APR datasets significantly different?

The 2-D instance space that was constructed to analyse the effectiveness of APRTs, also allows us to analyse the location of the different benchmarks, which reveals how diverse they are. The dataset footprint presented in Figure 4 shows the reduced feature space with instances labelled according to the dataset they belong to. Each point is a bug from a particular dataset.

The features that were eventually found as significant and used to create this instance space, are the ones that have a good linear relationship with algorithm performance. For some APRTs, the choice of features may be better than for others, however, our approach chooses a common feature set that performs well on average across all algorithms.

We observe that there is a distinctive cluster on the left of Figure 4 composed of only bugs from IntroClassJava. It is clear that this dataset is significantly different from the other datasets. Further away from this cluster, is the footprint of Defects4J, which is on

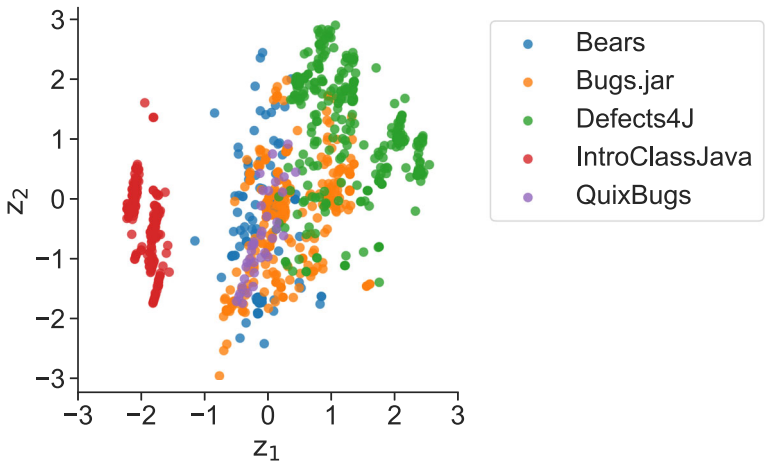

Fig. 4 Benchmark footprint. Each point corresponds to a bug from a particular benchmark dataset

the rightmost side of the graph. This indicates that Defects4J is significantly different from IntroClassJava.

On the other hand, the footprints of Bears, Bugs.jar and QuixBugs overlap to a greater extent. They are spread between IntroClassJava and Defects4J and have a higher spread than the other datasets. Bugs.jar covers a larger are and encompasses that one from Quixbugs.

Bugs.jar contains some bugs obtained from the same software as the other datasets (e.g., Apache, Commons, Math), thus the bugs are eventually the same. QuixBugs is a set of buggy implementation of well known algorithms (e.g., Quixsort), and each buggy program in this dataset is a single class. The others datasets are real buggy programs, composed of several classes.

In summary, the answer to the second research question is as follows:

> **RQ2:** IntroClassJava and most of bugs from Defects4J are significantly different from the other benchmark datasets. Meanwhile, Bugs.jar has the most diverse bugs; its footprint encompasses that of QuixBugs, the majority of Bear's footprint and a portion of Defects4J's footprint.

The dataset footprint also helps us understand if a dataset is biased, that is if it doesn't fill the possible instance space, and lies within the 'footprint' (area of strength) of one APRT only, and doesn't give other algorithms a chance to show their strength. We particularly observe that the footprint of Defects4J lies within the area of strength of most APRTs apart from NPEFix, whose footprint is shown in Figure 2. What this means is that if the performances of APRTs are compared solely on this dataset, the evaluation can be biased and demonstrate only the strengths of these approaches. The footprints of Bugs.jar, Quixbugs and Bears lie within the area where most APRTs apart from NPEFix are not able to generate plausible patches, indicating that these datasets are quite challenging for these approaches and exhibit less bias.

Our finding from this research task can inform researchers who develop new APRTs in the selection of the bug benchmark to test their technique. It wouldn't be sufficient to test a

new APRT on just one dataset, and a technique that works for Defects4J may not produce good results when repairing IntroClassJava.

4.3 RQ3. How can we select the most suitable APRT?

To answer this question, the E-APR framework uses multi-label classification algorithms to predict the most suitable APRT to repair buggy programs with particular features. We use 10-fold cross validation to evaluate the performance of four notable Machine Learning techniques: Support Vector Machine (SVM), Random Forest Classifier (RFC), Decision Tree (DT), and Multi-Layer Perceptron (MLP).

We use the scikit-learn Python implementation of these approaches and employed MLSMOTE (Charte et al. 2015) to address the class imbalance problem. The performance of the two approaches is evaluated in terms of precision, recall, and f1-score. Precision is the fraction of instances that are correctly predicted, calculated as:

$$P = \frac{TP}{TP + FP} \tag{3}$$

where *TP* is the true positives and *FP* is the false positives. Recall measures how accurately the model is able to identify the relevant data.

$$R = \frac{TP}{TP + FN} \tag{4}$$

where FN is false negatives. F1-Score is the harmonic mean of P and R, computed as follows:

$$F1 = 2\frac{P \cdot R}{P + R} \tag{5}$$

Results are shown in Table 5.

Table 5 The performance of Support Vector Machine (SVM), Random Forest Classifier (RFC), Decision Tree (DT) and Multi-Layer Perceptron (MLP) classifier in terms of precision (P), recall (R) and f1-score (F1)

	SVM			RFC			DT			MLP		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Arja	0.83	0.76	0.80	0.89	**0.87**	0.88	0.96	0.84	0.90	0.96	0.86	**0.91**
Cardumen	0.77	0.71	0.74	**0.86**	**0.86**	**0.86**	0.86	0.68	0.76	0.85	0.59	0.70
DynaMoth	0.68	0.78	0.72	0.93	0.81	0.87	0.83	0.81	0.82	**0.94**	**0.89**	**0.91**
GenProgA	0.72	**0.93**	0.81	**0.86**	0.83	**0.85**	0.79	0.81	0.80	0.85	0.59	0.70
KaliA	**0.92**	0.84	0.88	0.87	**0.91**	**0.89**	0.88	0.79	0.83	0.88	0.79	0.83
NPEFix	0.47	0.82	0.60	**0.82**	0.82	**0.82**	0.56	**0.83**	0.67	0.00	0.00	0.00
Nopol	0.61	**0.93**	0.74	**0.92**	0.73	**0.81**	0.83	0.76	0.79	0.71	0.45	0.56
RSRepair	**0.90**	0.81	**0.85**	0.86	0.75	0.80	0.82	**0.87**	0.85	0.85	0.74	0.79
jGenProg	0.65	0.42	0.51	**0.80**	0.77	0.78	0.81	0.81	0.81	0.79	**0.83**	0.81
jKali	0.87	0.77	0.81	**0.93**	0.88	**0.90**	0.86	**0.91**	0.89	0.89	0.87	0.88
jMutRepair	0.77	0.48	0.59	**0.85**	0.52	0.65	0.75	0.50	0.60	0.84	**0.67**	**0.74**
micro avg	0.76	0.76	0.76	**0.88**	**0.81**	**0.84**	0.84	0.79	0.82	0.87	0.74	0.80
macro avg	0.74	0.75	0.73	**0.88**	**0.81**	**0.84**	0.81	0.78	0.79	0.78	0.67	0.72
weighted avg	0.78	0.76	0.76	**0.88**	**0.81**	**0.84**	0.84	0.79	0.81	0.85	0.74	0.79

The results indicate that, while all ML algorithms perform well in the task of APRT selection, the performance of RFC is clearly better than SVM. As a comparison, the work from Le et al. (2015) presents a similar task (predict whether a bug can be repaired by a genetic-programming based repair approach i.e., GenProg (Le Goues et al. 2012a)) and reports a precision of 72%.

> **R3:** The Random Forest Classifier is the best performing Machine Learning technique for APRT selection, and can predict the most suitable technique with **88%** precision, **81%** recall and **84%** f1-score.

Given the high performance of E-APR for predicting the most suitable APRT, it is of high-priority for us to integrate this approach to existing repair infrastructures such as RepairThemAll (Durieux et al. 2019) or Repairnator (Monperrus et al. 2019). For example, RepairThemAll has 11 automated repair techniques, but it does not offer any capabilities or guidelines in terms of which technique to select. Integrating E-APR with RepairThemAll would make it possible for users to select the most suitable APRT on the fly. Repairnator, on the other hand, is a software bot that automatically repairs broken Travis builds. Given a buggy program that produces a build to fail, Repairnator executes different repair approaches (including jGenProg, Nopol, among others) one by one, and the execution order is hard-coded. By incorporating E-APR, Repairnator could first execute E-APR to obtain the most suitable repair approaches for the buggy program, and execute them accordingly. In the future, we will investigate the effectiveness of integrating our approach with automated program repair infrastructure, such as RepairThemAll and Repairnator.

The overhead of using E-APR to select the most suitable APRT within existing APR infrastructures is minimal. APRT first must extract the code features by doing static analysis of the buggy program (it takes a few seconds to extract the nine feature we have identified as significant). Then, based on the extracted features, APRT uses the trained model to perform the prediction in few milliseconds.

5 Discussion and Threats to Validity

5.1 Features selection

A threat to the validity of this study is the selection of the considered features. Our approach E-APR considers 3 sets of features, each of them selected with a clear purpose: one aims at capturing object-oriented features, the second aims at capturing features specific to Java, and the third aims at capturing features related to the bug fixing activity (Observation-based features). Thus, we consider that the set of features is diverse enough to capture the characteristics of the program under analysis.

In this work, we complement extensively used features (e.g., Object-oriented features (Chidamber and Kemerer 1994b)) with a novel set of features (Observation-based features) that aim at characterising buggy programs. As the latter features are novel, there is a risk they do not precisely characterise buggy programs. However, a recent work (Yu et al. 2019) used Observation-based features to successfully predict source code transformations applied on buggy program. For this reason, we consider that Observation-based features can be used to predict the most suitable APR tool to apply to a buggy program.

5.2 Correctness of Patches

A threat to the validity is the correctness of patches. In our experiment from Section 4 we consider all generated patches (plausible) rather than focusing only on correct patches. One of the reasons is the availability of data on patch correctness. The number of correct patches generated by APRTs is much lower than the number of plausible patches. For instance, the recent manual evaluation done by Tian et al. (2020) of the patches we have used in this paper (more than 67,000 patches from RepairThemAll (Durieux et al. 2019)) found only 900 correct patches, from 20 different bugs (14 from Defects4J, 5 from Quixbugs and 1 from Bugs.jar). We reproduce our experiment, available in our (Appendix 2020), by considering the bugs that could be repaired by at least one correct patch. We found that the dataset consisting of 20 bugs that were correctly repaired is not sufficient for the machine learning algorithms used to identify significant features and create the algorithm footprints.

While we think that considering correct patches is an important next step, and a priority for our future work, the results with plausible patches provide some important insights into how APRTs work and how effective they are. Current APRTs find it challenging to even produce plausible patches, and E-APR helps understand why this is the case, and what kind of weaknesses future research into APRT should focus on. Moreover, recent studies shows that a plausible patch, even being overfitting and not adequate for repairing a bug, could give developer a valuable piece of information. For instance, Ginelli et al. (2020) studied code-removal patches, which works on manual machine patch analyses (e.g., Qi et al. 2015) labelled most of them as overfitting patches. They found that in 95.8% of the cases having an overfitting code-removal patch, it reveals different kinds of problems affecting the test suites that are relevant for the developers. Thus, they show this type of overfitting patches is useful: it exposes a particular weakness of the test suites.

5.3 Selection of Repair Tools

During the last years, several repair tools have been presented to repair Java bugs. Two previous works have tried to execute the tools (i.e., the materialization of repair approaches) on real bugs: (Durieux et al. 2019) could executed 11 repairs tools, and (Liu et al. 2020) executed 16 tools. Both papers list the reasons about why other repair approaches and tools could not be executed.

In this paper we consider the execution data from 11 tools. The main reason is that those tools were executed on 5 different bug benchmarks and generated patches are publicly available (Durieux et al. 2019). Other tools have exclusively focused on Defects4J, and we were not able to generate results for other datasets we consider in this study. For example, (Liu et al. 2020) evaluated 16 repair approaches only on Defects4J. We have included in our (Appendix 2020) initial results of an experiment done by considering the patches of that experiment, which includes, in addition to 10 repair tools considered in our paper (all except NPEfix), another 6: ACS (Xiong et al. 2017), Avatar (Liu et al. 2019a), FixMiner (Koyuncu et al. 2020), kPar (Liu et al. 2019), SimFix (Jiang et al. 2018), TBar (Liu et al. 2019b). From those initial results, we could not draw conclusive results. Our conjecture is the experiment has not enough diverse data: a single dataset evaluated (Defects4J), which contains bugs extracted from only 5 projects (Commons Math, Commons Lang, Joda Time, jFree Chart, and Closure).

We prioritised in this paper having a larger dataset, and found that the techniques we consider are diverse enough to demonstrate the capabilities of the proposed technique. We

consider that this point (i.e., the selection of evaluated tools) does not invalidate the novelty of our technique.

5.4 Failure Information

Some repair techniques are designed to fix specific bugs, and their effectiveness can be limited by the nature of the bug that is addressed (Monperrus 2018; Gazzola et al. 2019). For instance, NPEFix repairs null pointer exceptions and is unlikely to be useful in other cases. In this paper, we decided to focus on the features that allow to *characterise* the buggy program under repair, without considering, for instance, the type of failure. Our approach, however, can easily be extended to include failure information. For instance, an extension could include a new set of features that characterise the failure, for example null pointer exceptions, stack overflow, array index error.

E-APR does not consider failure information since most of the bugs considered in this study do not produce a failure, but an *incorrect output*. The incorrect output is exposed by the failing test case via *assertions*. For instance, by inspecting Defects4J Dissection (Sobreira et al. 2018) we found that 304 out of 395 (77%) bugs from Defects4J are due to incorrect output. We explored the meta-data of bugs using http://program-repair.org/defects4j-dissection (Sobreira et al. 2018) and found that 275 bugs are due to an *Assertion-FailedError* (for example, Chart-7: `junit.framework.AssertionFailedError: expected:<1> but was:<3>`) or *unit.framework.ComparisonFailure* (e.g. `junit.framework.ComparisonFailure: expected:<String[[]]> but was:<String[;]>`). Both exceptions are thrown by the testing framework after detecting the incorrect output.

5.5 Integration with Repair Infrastructures

To our knowledge, repair infrastructures such as Repairnator or RepairThemAll do not have the ability to predict, given a buggy program taken as input, with is the most suitable APR tool to generate a test-suite adequate patch for it. Repairnator calls APR tools in a fixed order, independently of the characteristics of the program under analysis. On the contrary, the user of RepairThemAll must decide the APR to be call. In both cases, our approach E-APR could be integrated to both of them. For Repairnator, E-APR could determine the order of the APR tools to be called with the goal of calling first the tools that are most suitable for a given buggy program. Similarly, for RepairThemAll, E-APR could suggest the user the APR tool to be invoked.

6 Related Work on the Effectiveness of APR Techniques

Researchers working in the area of APR have acknowledged that evaluating the quality of patches produced by APR techniques is crucial (Martinez et al. 2017a; Smith et al. 2015). To this end, Qi et al. (2015) studied the plausible generated by GenProg (Le Goues et al. 2012b) for C programs, and classified them as plausible (passing all tests), *overfitting* (plausible and incorrect) and *correct* (plausible, and do not have latent defects and do not introduce new defects or vulnerabilities (Long and Rinard 2016a)). They found that most of the reported patches were *overfitting*. Long and Rinard (2016a) analysed the patch search space of two repair tools, SPR (Long and Rinard 2015) and Prophet (Long and Rinard

2016b), and found that overfitting patches are typically orders of magnitude more abundant than correct patches.

Other works have studied the ability of APR techniques to repair buggy Java programs. For example, Martinez et al. (2017a) manually studied the correctness of patches produced by three APR techniquess over defects from Defects4J benchmark. They found that only a small number of bugs (9/47) could be correctly repaired.

Liu et al. (2020) executed 16 repairs tools on Defects4J and manually analyzed the generated patches following the procedure defined in that work. They found that the percentage of patches correctness varies between the tools at is belong of the 37% for 15/16 tools.

Ye et al. (2019) studied the repairability of bugs from QuixBugs (Lin et al. 2017), a dataset of 40 small buggy programs (between 9 and 69 LOC). They found that 15 bugs could be repaired by Nopol (Xuan et al. 2017) and approaches from Astor (Martinez M and Monperrus 2016), which generated in total 64 plausible patches. However, they found that 33 of them were incorrect.

The presence of overfitting patches has motivated researchers to investigate the amount of the overfitting patches (e.g., (Yu et al. 2019; Wang et al. 2020)), detect overfitting patches (e.g., DiffTGen (Xin and Reiss 2017), PatchSim (Xiong et al. 2018), Static code feature via learning (Wang et al. 2020), ODS (Ye et al. 2019)), and to avoid generating such patches (e.g., UnsatGuided (Yu et al. 2019), CapGen (Wen et al. 2018), Anti-pattern (Tan et al. 2016)). Empirical studies also have studied overfitting patches in detail. For example, Liu et al. (2020) conducted an large-scale empirical study which analyzed the correctness of patches generated by 16 repairs tools (10 mentioned in Section 2.2 (all except NPEFix) and ACS (Xiong et al. 2017), Avatar (Liu et al. 2019a), FixMiner (Koyuncu et al. 2020), kPar (Liu et al. 2019), SimFix (Jiang et al. 2018), TBar (Liu et al. 2019b)). One of their main findings is that many plausible patches are related to wrong locations of the patches. As previously found by Liu et al. (2019), the accuracy of fault localization tool has a direct and substantial impact on the performance of APR tools.

Our work extends existing research in analysing the effectiveness of APR techniques by examining what software features impacts the repairability of a software system. We characterise a software system using code features (e.g., depth of inheritance tree and method cohesion) and determine the most significant features that have impact on whether an APR technique can generate a patch.

There has also been some research in characterising patches generated by APR techniques to investigate how these patches differ from the ones generated by human programmers.

Wang et al. (2019) compared the difference between 177 correct patches for Defects4J bugs generated by APR techniques and the patches written by developers. To characterise the bugs, the authors considered 6 metrics: a) Patch size, b) Number of chunks c) Number of modified files, d) Number of modified methods e) Line coverage, and f) Branch coverage. They found that automatically generated patches are on average syntactically different compared to the patches generated by developers. Patches generated by APR techniques are usually longer, have a higher number of chunks, and have a higher line and branch coverage.

Similarly, Smith et al. (2015) studied the quality of patches generated by two C program repair approaches (GenProg and TprAutoRepair). The authors used two metrics that were *dynamically* computed (i.e., by running the program under repair): a) number of passing and failing test cases, and b) test suite coverage.

Both Wang et al. (2019) and Smith et al. (2015) focus on analysing the kind of patches generated by APR techniques. The aim of these works is to understand how good the patches are, and how they are different from developer-generated patches. Our work, instead, aims at understanding what kind of software systems and bugs APR techniques are able to repair. This will help explain how and why they work, and as a result, make it possible to select the right technique given a new buggy software system.

In their research, Smith et al. (2015) state that "Automatic repair should be used in the appropriate contexts" and that "Our results suggest that more work is needed to fully understand and characterise test suite quality beyond coverage metrics alone". The E-APR framework addresses these two research challenges by investigating 146 features, and building a machine learning model that enables the selection of the most suitable APR technique for a given buggy program.

Another related work is the one by Motwani et al. (2018) which investigates correlations between the effectiveness of APR techniques and different aspects of bugs, such as bug importance and bug complexity. Results were analysed at course-grained level, with the findings showing weak to moderate correlation between bug importance and the ability of the APR technique to produce a patch. The results also show that APR techniques are effective in repairing easy bugs - as measured by the number of files and lines that have to be changed to fix the bug - while struggling with more complex bugs. This study makes an important step towards understanding where APR techniques work. In this paper, we take this research one step further by providing a more detailed analysis of the effectiveness of different APR techniques. The framework we propose allows us to examine the effectiveness of individual techniques in a visual and numerical way. We measure the footprints of the different APR techniques and whether their results overlap. This helps us understand the strengths and weaknesses of individual techniques, and their similarities in a more fine-grained way.

Le et al. (2015) present a work that has a similar goal to ours: they build an oracle that can predict whether fixing a failure should be delegated to a genetic-programming-based automated repair technique. The authors first extract features from an early stage of running a repair tool. Then, they pass the values of these features to learn a discriminative model capable of predicting whether continuing a genetic programming search will lead to a repair within a desired time limit.

Beyond the similarities, there are notable differences between our work and the work by Le et al. (2015).

First, Le et al. (2015) focus on genetic-programming-based automated repair technique, while our approach is independent of the type of repair technique. For instance, it considers genetic-programming-based technique (Arja and Genprog), semantic-based techniques (Nopol) and exhaustive methods (jMutRepair). Le et al. (2015) consider 27 features, 18 of which are related to genetic-programming. We use 3 sets of features (in total more than 200 features) that are independent of any repair technique and aim to describe the buggy program under repair. Le et al. (2015) analyse the early stage of a genetic-programming-based technique to extract 18 features. This means that it could be necessary to modify a repair approach to extract those features or to monitor the execution logs. E-APR considers the features extracted from a buggy program and trains the prediction model using the output from previous executions (i.e., a bug was patched or not by a technique). Le et al. (2015) considers one dataset of bugs (ManyBugs (Le Goues et al. 2015) with 105 bugs) and one repair tool (GenProg), while we consider 11 repair tools and 5 datasets (1282 bugs).

Lin et al. (2020) studied the non-repairability factors of various APR techniques. They analysed 11,818 execution logs from 27 Java tools, and found that 25.7% of them contained unexpected exceptions that prevent those tools to find a patch.

7 Conclusion

In this paper, we introduced E-APR, which is a novel framework for assessing strengths and weaknesses of APR techniques for Automated Program Repair (APR). We identified nine significant software features that have an impact on APRT effectiveness. These features were then used to provide explanations on an APR technique's effectiveness across a range of buggy programs. We introduced a method for visualising APRT footprints, which reveal strengths and weaknesses of the APR techniques in fixing buggy programs.

We conducted an analysis of 11 different APR techniques applied to 2,141 bugs from 130 projects, constituting in total 23,551 repair attempts. Our approach effectively identified APRT footprints and the features that impact the effectiveness of an automated program technique. Using the most significant features, we applied two machine learning approaches that learns the relationship between software features and APRT effectiveness. Random Forest Classifier showed the best performance, with 88% precision, 81% recall and 84% f1-score.

Acknowledgements The authors would like to acknowledge Prof. Kate Smith-Miles and her team working on matilda.unimelb.edu.au. The methodology on Instance Space Analysis constitutes the foundations of this work. Matilda was used to create the instance spaces presented in Figures 2 and 3.

References

Abreu R, Zoeteweij P, van Gemund AJC (2007) On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007), pp. 89–98

Aleti A, Moser I, Meedeniya I, Grunske L (2014) Choosing the appropriate forecasting model for predictive parameter control. Evolutionary computation 22(2):319–349

Anand S, Burke EK, Chen TY, Clark J, Cohen MB, Grieskamp W, Harman M, Harrold MJ, Mcminn P (2013) An orchestrated survey of methodologies for automated software test case generation. Journal of Systems Software 86(8):1978–2001

Appendix (2020) Appendix e-apr. https://github.com/UPHF/eapr

Bengio Y, Chapados N (2003) Extensions to metric-based model selection. J Mach Learn Res 3(Mar):1209–1227

Boser BE, Guyon IM, Vapnik VN (1992) A training algorithm for optimal margin classifiers. In: Proceedings of the Fifth Annual Workshop on Computational Learning Theory, ser. COLT '92. New York, NY, USA: Association for Computing Machinery, p. 144–152. [Online]. Available: https://doi.org/10.1145/130385.130401

Campos J, Riboira A, Perez A, Abreu R (2012) Gzoltar: an eclipse plug-in for testing and debugging. In: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp 378–381

Charette RN (2009) This Car Runs on Code. [Online; accessed 10-December-2018]. [Online]. Available: https://spectrum.ieee.org/transportation/systems/this-car-runs-on-code

Charte F, Rivera AJ, del Jesus MJ, Herrera F (2015) Mlsmote: approaching imbalanced multilabel learning through synthetic instance generation. Knowl-Based Syst 89:385–397

Chidamber S, Kemerer C (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. Software Engineering, IEEE Transactions on 20(6):476–493

Durieux T, Cornu B, Seinturier L, Monperrus M (2017) Dynamic Patch Generation for Null Pointer Exceptions Using Metaprogramming. In: Proceedings of the 24th IEEE International Conference on Software Analysis. Evolution and reengineering (SANER '17). IEEE, pp 349–358

Durieux T, Madeiral F, Martinez M, Abreu R (2019) Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts, in Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, pp. 302–313

Durieux T, Monperrus M (2016) Dynamoth: Dynamic Code Synthesis for Automatic Program Repair, in International Workshop on Automation of Software Test. ACM, pp 85–91

Durieux T, Monperrus M (2016) Introclassjava: A Benchmark of 297 Small and Buggy Java Programs, University of Lille, University of Lille, Tech. Rep #hal-01272126

Eisenstadt M (1997) My hairiest bug war stories. Commun ACM 40(4):30–37

El-Wakil M, El-Bastawisi A, Boshra M, Fahmy A (2004) Object-oriented design quality models a survey and comparison. In: 2nd International Conference on Informatics and Systems, pp 1–11

Gazzola L, Micucci D, Mariani L (2019) Automatic Software Repair: A Survey. IEEE Transactions on Software Engineering 45(1):34–67

Ginelli D, Martinez M, Mariani L, Monperrus M (2020) A comprehensive study of code-removal patches in automated program repair," arXiv, Tech. Rep. 2012.06264, [Online]. Available: 2012.06264

Guyon I, Elisseeff A (2003) An introduction to variable and feature selection. Journal of machine learning research 3(Mar):1157–1182

Harris M (2016) Google reports self-driving car mistakes: 272 failures and 13 near misses, [Online; accessed 10-December-2018]. [Online]. Available: https://www.theguardian.com/technology/2016/jan/12/google-self-driving-cars-mistakes-data-reports

Jiang J, Xiong Y, Zhang H, Gao Q, Chen X (2018) Shaping Program Repair Space with Existing Patches and Similar Code, in Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '18). ACM, pp 298–309

Jolliffe I (2011) Principal component analysis. Springer

Just R, Jalali D, Ernst MD (2014) Defects4j: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs, in Proceedings of the 23rd International Symposium on Software Testing and Analysis. ACM, pp. 437–440

Kaner C, Bach J, Pettichord B (2008) Lessons learned in software testing. John Wiley & Sons

Kim D, Nam J, Song J, Kim S (2013) Automatic patch generation learned from human-written patches. In: International Conference on Software Engineering. IEEE Press, pp 802–811

Koyuncu A, Liu K, Bissyandé T, Kim D, Klein J, Monperrus M, Le Traon Y (2020) Fixminer: Mining relevant fix patterns for automated program repair. Empir Softw Eng 25(3):1980–2024. [Online]. Available: https://doi.org/10.1007/s10664-019-09780-z

Le X-BD, Bao L, Lo D, Xia X, Li S, Pasareanu C (2019) On reliability of patch correctness assessment. In: Proceedings of the 41st International Conference on Software engineering, ser. ICSE '19. IEEE Press, p. 524–535. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00064

Le Goues C, Nguyen T, Forrest S, Weimer W (2012a) Genprog: a generic method for automatic software repair. Software Engineering, IEEE Transactions on 38(1):54–72

Le Goues C, Dewey-Vogt M, Forrest S, Weimer W (2012b) A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: International Conference on Software engineering ser. ICSE IEEE Press. pp. 3–13

Le Goues C, Forrest S, Weimer W (2013) Current challenges in automatic software repair. Softw Qual J 21(3):421–443

Le Goues C, Holtschulte N, Smith EK, Brun Y, Devanbu P, Forrest S, Weimer W (2015) The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. IEEE Trans Softw Eng 41(12):1236–1256

Le XD, Le TB, Lo D (2015) Should fixing these failures be delegated to automated program repair? In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), pp 427–437

Lin D, Koppel J, Chen A, Solar-Lezama A (2017) Quixbugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge, in ACM SIGPLAN International Conference on Systems, Programming, Languages, and applications: Software for Humanity. ACM, pp 55–56

Lin B, Wang S, Wen M, Zhang Z, Wu H, Qin Y, Mao X (2020) Understanding the non-repairability factors of automated program repair techniques, p 10

Liu K, Koyuncu A, Bissyandé TF, Kim D, Klein J, Le Traon Y (2019) You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In: 2019 12th IEEE Conference on Software Testing Validation and Verification (ICST), pp 102–113

Liu K, Koyuncu A, Kim D, Bissyandé TF (2019) AVATAR: Fixing semantic bugs with fix patterns of static analysis violations," in Proceedings of the 26th. IEEE International Conference on Software Analysis, Evolution, and Reengineering. IEEE, pp 456–467

Liu K, Koyuncu A, Kim D, Bissyandé TF (2019) Tbar: Revisiting template-based automated program repair. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, p. 31–42. [Online]. Available: https://doi.org/10.1145/3293882.3330577

Liu K, Wang S, Koyuncu A, Kim K, Bissyandé TF, Kim D, Wu P, Klein J, Mao X, Traon YL (2020) On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, p. 615–627. [Online]. Available: https://doi.org/10.1145/3377811.3380338

Long F, Rinard M (2015) Staged program repair with condition synthesis. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, pp. 166–178. [Online]. Available: https://doi.org/10.1145/2786805.2786811

Long F, Rinard M (2016) An analysis of the search spaces for generate and validate patch generation systems. In: Proceedings of the 38th International Conference on Software Engineering, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, p. 702–713. [Online]. Available: https://doi.org/10.1145/2884781.2884872

Long F, Rinard M (2016) Automatic patch generation by learning correct code. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ser. POPL '16. New York, NY, USA: Association for Computing Machinery, p. 298–312. [Online]. Available: https://doi.org/10.1145/2837614.2837617

Madeiral F, Urli S, Maia M, Monperrus M (2019) Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In: Proceedings of the 26th, IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19), pp 468–478. Hangzhou, China: IEEE

Madjarov G, Kocev D, Gjorgjevikj D, Džeroski S (2012) An extensive experimental comparison of methods for multi-label learning. Pattern recognition 45(9):3084–3104

Mark Harman PO (2018) From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In: IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 1–23

Martinez M, Durieux T, Sommerard R, Xuan J, Monperrus M (2017) Automatic Repair of Real Bugs in Java:, A Large-scale Experiment on the Defects4J Dataset. Empir Softw Eng 22(4):1936–1964

Martinez M, Durieux T, Sommerard R, Xuan J, Monperrus M (2017) Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. Empir Softw Eng 22(4):1936–1964

Martinez M, Monperrus M (2016) ASTOR: A Program Repair Library For Java. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, Demonstration Track. ACM, pp 441–444

Martinez M, Monperrus M (2018) Ultra-Large Repair Search Space with Automatically Mined Templates: the Cardumen Mode of Astor. In: Colanzi TE, McMinn P (eds) International Symposium on Search-Based Software Engineering. Lecture Notes in Computer Science, vol 11036, Springer, pp. 65–86

Martinez M, Monperrus M (2015) Mining software repair models for reasoning on the search space of automated program fixing. Empir Softw Eng 20(1):176–205

Martinez M, Monperrus M (2019) Coming: A tool for mining change pattern instances from git commits," in Proceedings of the 41st International Conference on Software Engineering: Companion proceedings, ser. ICSE '19. IEEE Press, p. 79–82. [Online]. Available: https://doi.org/10.1109/ICSE-Companion.2019.00043

Monperrus M (2018) Automatic Software Repair: a Bibliography. ACM Comput Surv 51(1):17:1–17:24. [Online]. Available: https://doi.org/10.1145/3105906

Monperrus M, Urli S, Durieux T, Martinez M, Baudry B, Seinturier L (2019) Repairnator patches programs automatically. Ubiquity, vol. 2019

Motwani M, Sankaranarayanan S, Just R, Brun Y (2018) Do automated program repair techniques repair hard and important bugs? Empir Softw Eng 23(5):2901–2947

Muñoz MA, Villanova L, Baatar D, Smith-Miles K (2018) Instance spaces for machine learning classification. Mach Learn 107(1):109–147

Oliveira C, Aleti A, Grunske L, Smith-Miles K (2018) Mapping the effectiveness of automated test suite generation techniques. IEEE Trans Reliab 67(3):771–785

Oliveira C, Aleti A, Li Y-F, Abdelrazek M (2019) Footprints of fitness functions in search-based software testing. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1399–1407

Prabhu Y, Varma M (2014) Fastxml: A fast, accurate and stable tree-classifier for extreme multi-label learning. In: Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining, pp 263–272

Qi Z, Long F, Achour S, Rinard M (2015) An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '15),. ACM, vol 2015, pp 24–36

Qi Y, Mao X, Lei Y, Dai Z, Wang C (2014) The Strength of Random Search on Automated Program Repair. In: Proceedings of the 36th International Conference on Software Engineering. ACM, pp 254–265

Quinlan JR (1996) Learning decision tree classifiers. ACM Computing Surveys (CSUR) 28(1):71–72

Rice JR et al (1976) The algorithm selection problem. Advances in computers 15(65-118):5

Ruck DW, Rogers SK, Kabrisky M, Oxley ME, Suter BW (1990) The multilayer perceptron as an approximation to a bayes optimal discriminant function. IEEE transactions on neural networks 1(4):296–298

Saha RK, Lyu Y, Lam W, Yoshida H, Prasad MR (2018) Bugs.jar: A Large-scale, Diverse Dataset of Real-world Java Bugs, in International Conference on Mining Software Repositories. ACM, pp 10–13

Smith EK, Barr ET, Le Goues C, Brun Y (2015) Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair, in Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15). ACM, pp 532–543

Smith-Miles K, Baatar D, Wreford B, Lewis R (2014) Towards objective measures of algorithm performance across instance space. Computers & Operations Research 45:12–24

Smith-Miles K, Tan TT (2012) Measuring algorithm footprints in instance space. In: 2012 IEEE Congress on Evolutionary Computation. IEEE, pp. 1–8

Sobreira V, Durieux T, Madeiral F, Monperrus M, Maia MA (2018) Dissection of a Bug dataset: Anatomy of 395 Patches from Defects4J. In: Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '18),. Campobasso, Italy: IEEE, pp 130–140

Software RW (2013) University of Cambridge Study: Failure to Adopt Reverse Debugging Costs Global Economy $ 41 Billion Annually, [Online; accessed 10-December-2018]. [Online]. Available:. https://www.roguewave.com/company/news/2013/university-of-cambridge-reverse-debugging-study

Tan SH, Yoshida H, Prasad MR, Roychoudhury A (2016) Anti-patterns in search-based program repair. In: ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp 727–738

Tian H, Liu K, Kaboré AK, Koyuncu A, Li L, Klein J, Bissyandé TF (2020) Evaluating representation learning of code changes for predicting patch correctness in program repair, Inproceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering ACM

Vapnik VN (1995) The nature of statistical learning theory. Berlin heidelberg: Springer-Verlag

Wang S, Wen M, Chen L, Yi X, Mao X (2019) How different is it between machine-generated and developer-provided patches? an empirical study on the correct patches generated by automated program repair techniques

Wang S, Wen M, Lin B, Wu H, Qin Y, Zou D, Mao X, Jin H (2020) Automated patch correctness assessment: How far are we?. 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 968–980

Wen M, Chen J, Wu R, Hao D, Cheung S-C (2018) Context-Aware Patch generation for better automated program repair. In: International conference on software engineering. ACM, pp 1–11

Xin Q, Reiss S (2017) Identifying test-suite-overfitted patches through test case generation. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, p. 226–236. [Online]. Available: https://doi.org/10.1145/3092703.3092718

Xiong Y, Liu X, Zeng M, Zhang L, Huang G (2018) Identifying patch correctness in test-based program repair. In: Proceedings of the 40th International Conference on Software Engineering, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, p. 789–799. [Online]. Available: https://doi.org/10.1145/3180155.3180182

Xiong Y, Wang J, Yan R, Zhang J, Han S, Huang G, Zhang L (2017) Precise Condition Synthesis for Program Repair, in Proceedings of the 39th International Conference on Software Engineering (ICSE '17). IEEE Press, pp 416–426

Xuan J, Martinez M, Demarco F, Clement M, Marcote SRL, Durieux T, Berre DL, Monperrus M (2017) Nopol: Automatic repair of conditional statement bugs in java programs. IEEE, Transactions Software Engineering 43(1):34–55

Ye H, Martinez M, Durieux T, Monperrus M (2019) A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark, in International Workshop on Intelligent Bug Fixing (co-located with SANER). IEEE, pp 1–10

Ye H, Gu J, Martinez M, Durieux T, Monperrus M (2019) Automated classification of overfitting patches with statically extracted code features. arXiv, Tech. Rep. 1910.12057, [Online]. Available: 1910.12057

Yu Z, Martinez M, Bissyandé TF, Monperrus M (2019) Learning the relation between code features and code transforms with structured prediction," arXiv, Tech. Rep. 1907.09282, [Online]. Available: 1907.09282
Yu Z, Martinez M, Danglot B, Durieux T, Monperrus M (2019) Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system. Empir Softw Eng 24(1):33–67
Yuan Y, Banzhaf W (2018) ARJA: Automated Repair Of Java Programs via Multi-Objective Genetic Programming, IEEE Transactions on Software Engineering, vol PP