Data-Augmented Software Diagnosis

Amir Elmishali[1] and **Roni Stern**[1] and **Meir Kalech**[1]
[1]Ben Gurion University of the Negev
e-mail: amir9979@gmail.com, roni.stern@gmail.com, kalech@bgu.ac.il

Abstract

The task of software diagnosis algorithms is to identify which software components are faulty, based on the observed behavior of the system. Software diagnosis algorithms have been studied in the Artificial Intelligence community, using a model-based and spectrum-based approaches. In this work we show how software fault prediction algorithms, which have been studied in the software engineering literature, can be used to improve software diagnosis. Software fault prediction algorithms predict which software components is likely to contain faults using machine learning techniques. The resulting data-augmented diagnosis algorithm we propose is able to overcome of key problems in software diagnosis algorithms: ranking diagnoses and distinguishing between diagnoses with high probability and low probability. This allows to significantly reduce the outputted list of diagnoses. We demonstrate the efficiency of the proposed approach empirically on both synthetic bugs and bugs extracted from the Eclipse open source project. Results show that the accuracy of the found diagnoses is substantially improved when using the proposed combination of software fault prediction and software diagnosis algorithms.

1 Introduction

Software is prevalent in practically all fields of life, and its complexity is growing. Unfortunately, software failures are common and their impact can be very costly. As a result, there is a growing need for automated tools to identify software failures and isolate the faulty software components, such as classes and functions, that have caused the failure. We focus on the latter task, of *isolating faults in software components*, and refer to this task as software diagnosis.

Model-based diagnosis (MBD) is an approach to automated diagnosis that uses a model of the diagnosed system to infer possible *diagnoses*, i.e., possible explanations of the observed system failure. While MBD was successfully applied to a range of domains [1; 2; 3; 4], it has not been applied successfully yet to software. The reason for this is that in software development, there is usually no formal model of the developed software. To this end, a scalable software diagnosis algorithm called Barinel has been proposed [5]. Barinel is a combination of MBD and Spectrum Fault Localization (SFL). SFL considers traces of executions, and finds diagnoses by considering the correlation between execution traces and which executions have failed. While very scalable, Barinel suffers from one key disadvantage: it can return a very large set of possible diagnoses for the software developer to choose from. To handle this disadvantage, Abreu et al. [5] proposed a Bayesian approach to compute a likelihood score for each diagosis. Then, diagnoses are prioritize according to their likelihood scores.

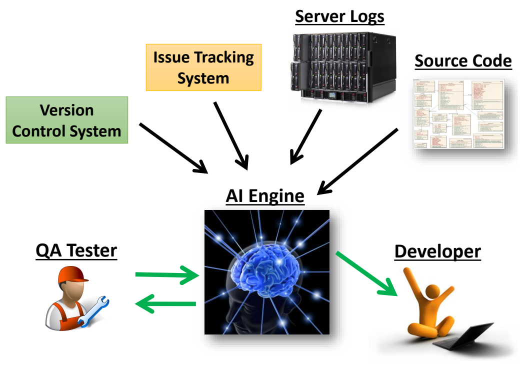
Thanks to the open source movement and current software engineering tools such as version control and issue tracking systems, there is much more information about a diagnosed system than revealed by the traces of performed tests. For example, version control systems store all revisions of every source files, and it is quite common that a bug occurs in a source file that was recently revised. Barinel is agnostic to this data. We propose a data-driven approach to better prioritize the set of diagnoses returned by Barinel.

In particular, we use methods from the software engineering literature to learn from collected data how to predict which software components are expected to be faulty. These predictions are then integrated into Barinel to better prioritize the diagnoses it outputs and provide more accurate estimates of each diagnosis likelihood.

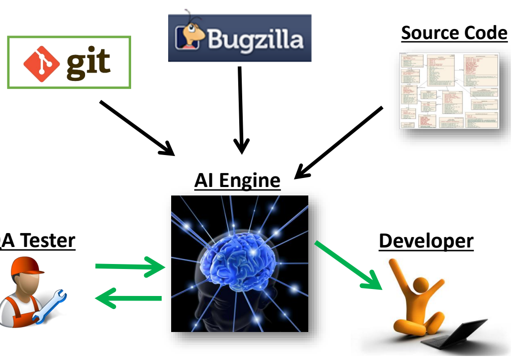
The resulting data-augmented diagnosis algorithm is part of a broader software troubleshooting paradigm that we call *Learn, Diagnose, and Plan (LDP)*. In this paradigm, illustrated in Figure 1(a), the troubleshooting algorithm learns which source files are likely to fail from past faults, previous source code revisions, and other sources. When a test fails, a data-augmented diagnosis algorithm considers the observed failed and passed tests to suggest likely diagnoses leveraging the knowledge learned from past data. If further tests are necessary to determine which software component caused the failure, such test are planned automatically, taking into consideration the diagnoses found. This process continues until a sufficiently accurate diagnoses is found.

In this work we implemented this paradigm and simulated its execution on a popular open source software project – the Eclipse CDT. Information from the Git version control and the Bugzilla issue tracking systems was used, as illustrated in Figure 1(b) and explained in the experimental results.

Results show a huge advantage of using our data-augmented diagnoser over Barinel with uniform priors for both finding more accurate diagnoses and for better selecting tests for troubleshooting. Moreover, to demonstrate the potential benefit of our data-augmented approach we also



(a) Learn, Diagnos, and Plan Paradigm



(b) Our current implementation

Figure 1: The learn, diagnose, and plan paradigm and our implementation.

experimented with a synthetic fault prediction model that is correctly identifies the faulty component. As expected, using the synthetic fault prediction model is better than using the learned fault prediction model, thus suggesting room for further improvements in future work. To our knowledge, this is the first work to integrate successfully a data-driven approach into software diagnosis.

2 Model-Based Diagnosis for Software

The input to classical MBD algorithms is a tuple $\langle SD, COMPS, OBS\rangle$, where SD is a formal description of the diagnosed system's behavior, $COMPS$ is the set of components in the system that may be faulty, and OBS is a set of observations. A diagnosis problem arises when SD and OBS are inconsistent with the assumption that all the components in $COMPS$ are healthy. The output of an MBD algorithm is a set of *diagnoses*.

Definition 1 (Diagnosis)**.** *A set of components* $\Delta \subseteq COMPS$ *is a* diagnosis *if*

$$\bigwedge_{C\in\Delta} (\neg h(C)) \wedge \bigwedge_{C'\notin\Delta} (h(C')) \wedge SD \wedge OBS$$

is consistent, i.e., if assuming that the components in Δ *are faulty, then* SD *is consistent with* OBS*.*

The set of components ($COMPS$) in software diagnoses can be, for example, the set of classes, or all functions, or even a component per line of code. Low level granularity of components, e.g., setting each line of code as a component, will result in very focused diagnoses (e.g., pointing on the exact line of code that was faulty). Focusing the diagnoses in such way comes at a price of an increase in the computational effort. Automatically choosing the most suitable level of granularity is a topic for future work.

Observations (OBS) in software diagnosis are observed executions of tests. Every observed test t is labeled as "passed" or "failed", denoted by $passed(t)$ and $failed(t)$, respectively. This labeling is done manually by the tester or automatically in case of automated tests (e.g., failed assertions).

There are two main approaches for applying MBD to software diagnosis, each defining SD somewhat differently. The first approach requires SD to be a logical model of the correct functionality of every software component [6]. This approach allows using logical reasoning techniques to infer diagnoses. The main drawbacks of this approach is that it does not scale well and modeling the behavior of software component is often infeasible.

2.1 SFL for Software Diagnosis

An alternative approach to software diagnosis has been proposed by Abreu et al. (5; 7), based on *spectrum-based fault localization (SFL)*. In this SFL-based approach, there is no need for a logical model of the correct functionality of every software component in the system. Instead, the *traces* of the observed tests are considered.

Definition 2 (Trace)**.** *A* trace *of a test* t*, denoted by* $trace(t)$*, is the sequence of components involved in executing* t*.*

Traces of tests can be collected in practice with common software profilers (e.g., Java's JVMTI). Recent work showed how test traces can be collected with low overhead [8]. Also, many implemented applications maintain a log with some form of this information.

In the SFL-based approach, SD is implicitly defined in SFL by the assumption that a test will pass if all the components in its trace are not faulty. Let $h(C)$ denote the health predicate for a component C, i.e., $h(C)$ is true if C is not faulty. Then we can formally define SD in the SFL-based approach with the following set of Horn clauses:

$$\forall test \quad (\bigwedge_{C\in trace(test)} h(C)) \rightarrow passed(test)$$

Thus, if a test failed then we can infer that at least one of the components in its trace is faulty. In fact, a trace of a failed test is a *conflict*.

Definition 3 (Conflict)**.** *A set of components* $\Gamma \subseteq COMPS$ *is a conflict if* $\bigwedge_{C\in\Gamma} h(C) \wedge SD \wedge OBS$ *is inconsistent.*

Many MBD algorithms use conflicts to direct the search towards diagnoses, exploiting the fact that a diagnosis must be a hitting set of all the conflicts [9; 10; 11]. Intuitively, since at least one component in every conflict is faulty, only a hitting set of all conflicts can explain the unexpected observation (failed test).

Barinel is a recently proposed software MBD algorithm [5] based on exactly this concept: considering traces of tests with failed outcome as conflicts and returning their hitting sets as diagnoses. With a fast hitting set algorithm, such as the STACATTO hitting set algorithm proposed by Abreu et al. [12], Barinel can scale well to large systems. The main drawback of using Barinel is that it often outputs a large set of diagnoses, thus providing weaker guidance to the programmer that is assigned to solve the observed bug.

2.2 Prioritizing Diagnoses

To address this problem, Barinel computes a *score* for every diagnosis it returns, estimating the likelihood that it is true. This serves as a way to prioritize the large set of diagnoses returned by Barinel.

The exact details of how this score is compute is given by Abreu et al. [5]. For the purpose of this paper, it is important to note that the score computation used by Barinel is Bayesian: it computes for a given diagnosis the posterior probability that it is correct given the observed passes and failed tests. As a Bayesian approach, Barinel also requires some assumption about the *prior probability* of each component to be faulty. Prior works using Barinel has set these priors uniformly to all components. In this work, we propose a data-driven way to set these priors more intelligently and demonstrate experimentally that this has a huge impact of the overall performance of the resulting diagnoser.

3 Data-Augmented Software Diagnosis

The prior probabilities used by Barinel represent the a-priori probability of a component to be faulty, without considering any observed system behavior. Fortunately, there is a line of work on *software fault prediction* in the software engineering literature that deals exactly with this question: which software components is more likely to have a bug. We propose to use these software fault predictions as priors to be used by Barinel. First, we provide some background on software fault prediction.

3.1 Software Fault Prediction

Fault prediction in software is a classification problem. Given a software component, the goal is to determine its class – healthy or faulty. Supervised machine learning algorithms are commonly used these days to solve classification problems. They work as follows. As input, they are given a set of *instances*, in our case these are software components, and their correct labeling, i.e., the correct class for each instance. They output a *classification model*, which maps an instance to a class.

Learning algorithm extract *features* from a given instance, and try to learn from the given labeled instances the relation between the features of an instance and its class. Key to the success of machine learning algorithms is the choice of *features* used. Many possible features were proposed in the literature for software fault prediction.

Radjenovic et al. [13] surveyed the features used by existing software prediction algorithms and categorizes them into three families. **Traditional.** These features are traditional software complexity metrics, such as number of lines of code, McCabe [14] and Halstead [15] complexity measures.

Object Oriented. These features are software complexity metrics that are specifically designed for object oriented programs. This includes metrics like cohesion and coupling levels and depth of inheritance.

Process. These features are computed from the software change history. They try to capture the dynamics of the software development process, considering metrics such as lines added and deleted in the previous version and the age of the software component.

It is not clear from the literature which combination of features yields the most accurate fault predictions. In a preliminary set of experiments we found that the combination of features that performed best is a combination of 68 features from the features listed by Radjenovic et al. [13] worked best. This list of features included the McCabe [14] and Halstead [15] complexity measures, several object oriented measures such as the number of methods overriding a superclass, number of public methods, number of other classes referenced, and is the class abstract, and several process features such as the age of the source file, the number of revisions made to it in the last release, the number of developers contributed to its development, and the number of lines changed since the latest version.

As shown in the experimental results section, the resulting fault prediction model was accurate enough so that the overall data-augmented software diagnoser be more effective than Barinel with uniform priors. However, we are not sure that a better combination of features cannot be found, and this can be a topic for future work. The main novelty of our work is in integrating a software fault prediction model with the Barinel.

3.2 Integrating the Fault Prediction Model

The software fault prediction model generated as described above is a classifier, accepting as input a software component and outputting a binary prediction: is the component predicted to be faulty or not. Barinel, however, requires a real number that estimates the prior probability of each component to be faulty.

To obtain this estimated prior from the fault prediction model, we rely on the fact that most prediction models also output a confidence score, indicating the model's confidence about the classified class. Let $conf(C)$ denote this confidence for component C. We use $conf(C)$ for Barinel's prior if C is classified as faulty, and $1 - conf(C)$ otherwise.

4 Experimental Results

To demonstrate the benefits of the proposed data-augmented approach, we implemented it and evaluated it as follows.

4.1 Experimental Setup

As a benchmark, we used the source files, tests, and bugs reported for the Eclipse CDT open source software project (`eclipse.org/cdt`). Eclipse CDT is a popular open source Integrated Development Environment (IDE) for C/C++. The first release dates back to December 2003 and the latest release we consider, labeled `CDT 8.2.0`, was released in June 2013. It consists of 8,502 source code files and have had more than 10,129 bugs reported so far (for all releases). In addition, there are 3,493 automated tests coded using the JUnit unit testing framework.

Determining Faulty Files

Eclipse CDT is developed using the Git version control system and the Bugzilla issue tracking system. Git maintains all versions of each source file in a repository. This enables computing process metrics for every version of every source file. Similarly, Bugzilla is used to maintain all reported bugs. Some source file versions are marked in the Git repository as versions in which a specific bug was fixed. The Git repository for Eclipse CDT contained matching versions of source files for 6,730 out of 10,129 bugs reported as fixed in Bugzilla. We performed our experiments on these 6,730 bugs.

For both learning and testing a fault prediction model, we require a mapping between reported bug and the source files that were faulty and caused it. One possible assumption is that *every* source file revision that is marked as fixing bug X is a faulty file that caused X. We call this the "All files" assumption. The "All files" assumption may overestimate the number of faulty files as some of these files may have been modified due to other reasons, not related to the bug. Even if all changes in a revision are related to fixing a bug, it still does not mean that all these files are faulty. For example, properties files and XML configuration files. As a crude heuristic to overcome this, we also experiment with an alternative assumption that we call the "Most modified" assumption. In the "Most modified" assumption, for a given bug X we only consider a single source file as faulty from all the files associated with bug X, We chose from these source file the one in which the revision made to that source file was the most extensive. The extensiveness of the revision is measured by the number of lines added, updated, and deleted to the source file in this revision. Below we present experiments for both "All files" and "Most modified" assumptions. Śliwerski et al. [16] proposed a more elaborate method to heuristically identify the source files that are caused the bug, when analyzing a similar data set.

Training and Testing Set

The sources files and reported bugs from 5 releases, 8.0.0–8.1.1, were used to train the model of our data-augmented diagnoser, and the source files and reported bugs from release 8.1.2 were used to evaluate it.

4.2 Comparing Fault Prediction Accuracy

As a preliminary, we evaluated the quality of the fault prediction models used by our data-augmented diagnoser on our Eclipse CDT benchmark.

All files	Precision	Recall	F-Measure	AUC
Random Forest	0.56	0.09	0.16	0.84
J48	0.44	0.17	0.25	0.61
Naive Bayes	0.27	0.31	0.29	0.80
Most modified	Precision	Recall	F-Measure	AUC
Random Forest	0.44	0.04	0.08	0.76
J48	0.15	0.03	0.05	0.55
Naive Bayes	0.08	0.31	0.12	0.715

Table 1: Faulty prediction performance.

We used the Weka software package (`www.cs.waikato.ac.nz/ml/weka`) to experiment with several learning algorithms and compared the resulting fault prediction models. Specifically, we evaluated the following learning algorithms: Random Forest, J48 (Weka's implementation of a decision tree learning algorithm), and Naive Bayes. Table 1 shows the precision, recall, F-measure, and AUC of the fault prediction models generated by each of these learning algorithms. These are standard metrics for evaluating classifiers. In brief, *precision* is the ratio of faulty files among all files identified by the evaluated model as faulty. *Recall* is the number of faulty files identified as such by the evaluated model divided by the total number of faulty files. *F-measure* is a known combination of precision and recall. The AUC metric addresses the known tradeoff between recall and precision, where high recall often comes at the price of low precision. This tradeoff can be controlled by setting different sensitivity thresholds to the evaluated model. AUC is the area under the curve plotting the accuracy as a function of the recall (every point is a different threshold value).

All metrics range between zero and one (where one is optimal) and are standard metrics in machine learning and information retrieval. The unfamiliar reader can find more details in Machine Learning books, e.g. Mitchell's classical book [17].

The results for both "All files" and "Most modified" assumptions show that the Random Forest classifier obtained the overall best results. This corresponds to many recent works. Thus, in the results reported henceforth, we only used the model generated by the Random Forest classifier in our data-augmented diagnoser. The precision and especially recall results are fairly low. This is understandable, as most files are healthy, and thus the training set is very imbalanced. This is a known inhibitor to performance of standard learning algorithms. We have experimented with several known methods to handle this imbalanced setting, such as SMOTE and random under sampling, but these did not produce substantially better results. However, as we show below, even this imperfect prediction model is able to improve the existing data-agnostic software diagnosis algorithm. Note that we also experimented with other popular learning algorithms such as Support Vector Machine (SVM) and Artificial Neural Network (ANN), but their results were worse than those shown in Table 1.

Next, we evaluate the performance of our data-augmented diagnoser in two diagnostic tasks: finding diagnoses and guiding test generation.

4.3 Diagnosis Task

First, we compared the data-agnostic diagnoser with the proposed data-augmented diagnoser in the task of finding accurate diagnoses. The input is a set of tests, with their traces and outcomes and the output is a set of diagnoses, each diagnosis having a score that estimates its correctness. This score was computed by Barinel as desribed earlier in the paper, where the data-agnostic diagnoser uses uniform priors and the proposed data-augmented diagnoser uses the predicted fault probabilities from the learned model.

	Most modified		All files	
Diagnoser	Precision	Recall	Precision	Recall
Data-agnostic	0.72	0.27	0.55	0.26
Data-augmented	0.90	0.32	0.73	0.35
Syn. (0.6,0.01)	0.97	0.39	0.96	0.45
Syn. (0.6,0.1)	0.84	0.35	0.89	0.42
Syn. (0.6,0.2)	0.77	0.34	0.83	0.39
Syn. (0.6,0.3)	0.73	0.33	0.78	0.37
Syn. (0.6,0.4)	0.69	0.32	0.74	0.36

Table 2: Comparison of diagnosis accuracy.

To compare the set of diagnoses returned by the different diagnosers, we computed the *weighted average* of their precision and recall. This was computed as follows. First, the precision and recall for every diagnoses was computed. Then, we averaged these values, weighted by the score given to the diagnoses by Barinel. This enables aggregating the precision and recall of a set of diagnoses while incorporating which diagnoses are regarded as more likely according to Barinel's. For brevity, we will refer to this weighted average precision and weighted average recall as simply precision and recall.

Table 2 shows the precision and recall results of the data-agnostic diagnoser and our data-augmented diagnoser, for both "Most modified" and "All files" assumptions. Each result in the table is an average over the precision and recall obtained for 50 problem instances. A problem instance consists of (1) a bug from one of the bugs reported for release 8.1.2. of Eclipse CDT, and (2) a set of 25 tests, chosen randomly, while ensuring that at least one tests would pass through the faulty files.

Both precision and recall of the data-augmented and data-agnostic diagnosers support the main hypothesis of this work: a data-augmented diagnoser can yield substantially better diagnoses that a data-agnostic diagnoser. For example, the precision of the data-augmented diagnoser under the "Most modified" assumption is 0.9 while that of the data-agnostic diagnoser is only 0.72. The superior performance of the data-augmented diagnoser is shown for both "Most modified" and "All files" assumptions. Another observation that can be made from the results in Table 2 is that while the precision of the data-augmented diagnoses is very high and is substantially better than that of the data-agnostic diagnoser, the improvement in recall is relatively more modest. This can be explained by the precision and recall results of the learned model, shown in Table 1 and discussed earlier. There too, the recall results was far worse than the precision results (recall that we are using the model learned by the Random Forest learning algorithm). It is possible that learning a model with higher recall may result in higher recall for the resulting diagnoses. We explore the impact of learning more accurate fault prediction model next.

Synthetic Priors

The data-augmented diagnoser is based on the priors generated by the learned fault prediction model. Building better fault prediction models is an active field of study [13] and thus future fault prediction models may be more accurate than the ones used by our data-augmented diagnoser. To evaluate the benefit of a more accurate fault prediction model on our data-augmented diagnoser, we created a *synthetic fault prediction model*, in which faulty source files get P_f probability and healthy source files get P_h, where P_f and P_h are parameters. Setting $P_h = P_f$ would cause the data-augmented diagnoser to behave in a uniform distribution exactly like the data-agnostic diagnoser, setting the same prior probability for all source files to be faulty. By contrast, setting $P_h = 0$ and $P_f = 1$ represent an optimal fault prediction model, that exactly predicts which files are faulty and which are healthy.

The lines marked "Syn. (X,Y)" in Table 2 mark the performance of the data-augmented diagnoser when using this synthetic fault prediction model, where $X = P_f$ and $Y = P_h$. Note that we experimented with many values of P_f and P_h, and presented above a representative subset of these results.

As expected, setting lowering the value of P_h results in more better diagnoses being found. Setting a very low P_h value improves the precision significantly up to almost perfect precision (0.97 and 0.96 for the "Most modified" and "All files", respectively). The recall results, while also improving as we lower P_h, do not reach a very high value. For $P_h = 0.01$, the obtained recall is almost 0.39 and 0.45 for the "Most modified" and "All files", respectively.

A possible explanation for these low recall results lays in the fact that all the evaluated diagnosers use the Barinel diagnosis algorithm with different fault priors. Barinel uses these priors only to prioritize diagnoses, but Barinel considers as diagnoses hitting sets of faulty traces. Thus, if two faulty components are used in the same trace, only one of them will be detected even if both have very high likelihood of being faulty according to the fault prediction model.

Considering More Tests

Next, we investigate the impact of adding more tests to the accuracy of the returned diagnoses.

Figure 2 shows the precision and recall results (Figures 2 (a) and (b), respectively), as a function of the number of observed tests. We compared the different diagnosers, given 25, 40, 70, 100, and 130 observed tests.

The results show two interesting trends in both precision and recall. First, as expected, the data-agnostic diagnoser performs worse than the data-augmented diagnoser, which in terms performs worse than the diagnoser using a synthetic fault prediction model, with $P_h = 0.01$. This supports our main hypothesis — that data-augmented diagnosers can be better than a data-agnostic diagnoser. Also, the better performance of Syn. (0.6, 0.01) demonstrates that future research on improving the fault prediction model will results in a better diagnoser.

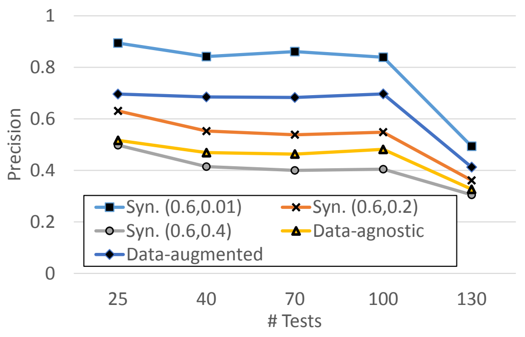
The second trend is that adding more tests reduces the precision and recall of the returned diagnoses. This, at first glance, seem counter-intuitive, as we would expect more tests to allow finding more accurate diagnoses and thus higher recall and precision. This non-intuitive results can be explained by how tests were chosen. As explained above, the observed tests were chosen randomly, only verifying that at least one test passes through each faulty source file. Adding randomly selected tests adds noise to the diagnoser. By contrast, intelligent methods to choose which tests to add can improve the accuracy of the diagnoses [18]. This is explored in the next section. Another reason for the degraded performance when adding more tests is that more tests may pass through more fault source files, in addition to those from the specific reported bug used to generate the problem instance in the first place. Thus, adding more tests increases the amount of faulty source files to detect.

4.4 Troubleshooting Task

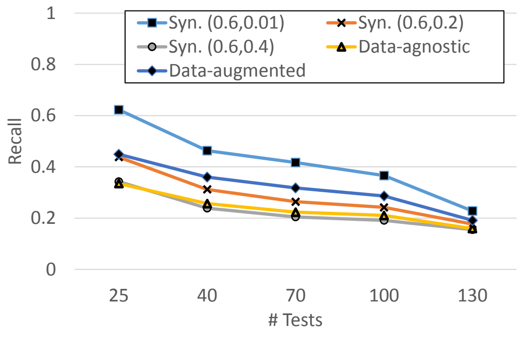
Efficient diagnosers are key components of *troubleshooting algorithms*. Troubleshooting algorithms choose which tests to perform to find the most accurate diagnosis. Zamir et al. [18] proposed several troubleshootings algorithms specifically designed to work with Barinel for troubleshooting software bugs. In the below preliminary study, we evaluated the impact of our data-augmented diagnoser on the overall performance of troubleshooting algorithms. Specifically, we implemented the so-called *highest probability* (HP) troubleshooting algorithm, in which tests are chosen in the following manner. HP chooses a test that is expected to pass through the source file having the highest probability of being faulty, given the diagnoses probabilities.

We run the HP troubleshooting algorithm with each of the diagnosers mentioned above (all rows in Table 2). We compared the HP troubleshooting algorithm using different diagnosers by counting the number of tests were required to reach a diagnoses of score higher than 0.7.

Table 3 shows the average number of tests performed by the HP troubleshooting algorithm until it halts (with a suitable diagnosis). The results show the same over-arching



(a) Precision results



(b) Recall results

Figure 2: Diagnosis accuracy as a function of # tests given to the diagnoser.

Algorithm	Most modified	All files
Data-agnostic	20.24	18.06
Data-augmented	10.80	15.45
Syn. (0.6,0.01)	3.94	14.91
Syn. (0.6,0.1)	15.44	17.83
Syn. (0.6,0.2)	19.78	18.99
Syn. (0.6,0.3)	20.90	19.24
Syn. (0.6,0.4)	20.74	19.18

Table 3: Avg. additional tests for troubleshooting.

theme: the data-augmented diagnoser is much better than the data-agnostic diagnoser for this troubleshooting task. Also, using the synthetic fault prediction model can result in even further improvement, thus suggesting future work for improving the learned fault prediction model.

5 Conclusion, and Future Work

We presented a method for using information about the diagnosed system to improve Barinel, a scalable, effective, software diagnosis algorithm [7]. In particular, we incorporated a software fault prediction model into Barinel. The resulting data-augmented diagnoser is shown to outperform Barinel without such a fault prediction model. This was verified experimentally using a real source code system (Eclipse CDT), real reported bugs and information from the software's source control repository. Results also suggests that future work on improving the learned fault prediction model will result in an improved diagnosis accuracy. In addition, it is worthwhile to incorporate the proposed data-augmented diagnosis methods with other proposed improvements of the based SFL-based software diagnosis, as those proposed by Hofer et al. [19; 20].

References

[1] Brian C. Williams and P. Pandurang Nayak. A model-based approach to reactive self-configuring systems. In *Conference on Artificial Intelligence (AAAI)*, pages 971–978, 1996.

[2] Alexander Feldman, Helena Vicente de Castro, Arjan van Gemund, and Gregory Provan. Model-based diagnostic decision-support system for satellites. In *IEEE Aerospace Conference*, pages 1–14. IEEE, 2013.

[3] Peter Struss and Chris Price. Model-based systems in the automotive industry. *AI magazine*, 24(4):17–34, 2003.

[4] Dietmar Jannach and Thomas Schmitz. Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach. *Automated Software Engineering*, 1:1–40, 2014.

[5] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Simultaneous debugging of software faults. *Journal of Systems and Software*, 84(4):573–586, 2011.

[6] Franz Wotawa and Mihai Nica. Program debugging using constraints – is it feasible? *Quality Software, International Conference on*, 0:236–243, 2011.

[7] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *Automated Software Engineering (ASE)*, pages 88–99. IEEE, 2009.

[8] Alexandre Perez, Rui Abreu, and André Riboira. A dynamic code coverage approach to maximize fault localization efficiency. *Journal of Systems and Software*, 2014.

[9] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artif. Intell.*, 32(1):97–130, 1987.

[10] Brian C. Williams and Robert J. Ragno. Conflict-directed A* and its role in model-based embedded systems. *Discrete Appl. Math.*, 155(12):1562–1595, 2007.

[11] Roni Stern, Meir Kalech, Alexander Feldman, and Gregory M. Provan. Exploring the duality in conflict-directed model-based diagnosis. In *AAAI*, 2012.

[12] Rui Abreu and Arjan JC van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In *SARA*, volume 9, pages 2–9, 2009.

[13] Danijel Radjenovic, Marjan Hericko, Richard Torkar, and Ales Zivkovic. Software fault prediction metrics: A systematic literature review. *Information & Software Technology*, 55(8):1397–1418, 2013.

[14] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[15] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[16] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[17] Tom Mitchell. *Machine learning*. McGraw Hill, 1997.

[18] Tom Zamir, Roni Stern, and Meir Kalech. Using model-based diagnosis to improve software testing. In *AAAI Conference on Artificial Intelligence*, 2014.

[19] Birgit Hofer, Franz Wotawa, and Rui Abreu. Ai for the win: Improving spectrum-based fault localization. *ACM SIGSOFT Software Engineering Notes*, 37:1–8, 2012.

[20] Birgit Hofer and Franz Wotawa. Spectrum enhanced dynamic slicing for better fault localization. In *ECAI*, pages 420–425, 2012.