Information and Software Technology 106 (2019) 142–160

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Automatically identifying code features for software defect prediction: Using AST N-grams

Thomas Shippey[a], David Bowes[b], Tracy Hall[c,*]

[a] University of Hertfordshire, United Kingdom
[b] University of Central Lancashire, United Kingdom
[c] Lancaster University, United Kingdom

ABSTRACT

Context: Identifying defects in code early is important. A wide range of static code metrics have been evaluated as potential defect indicators. Most of these metrics offer only high level insights and focus on particular pre-selected features of the code. None of the currently used metrics clearly performs best in defect prediction.

Objective: We use Abstract Syntax Tree (AST) n-grams to identify features of defective Java code that improve defect prediction performance.

Method: Our approach is bottom-up and does not rely on pre-selecting any specific features of code. We use non-parametric testing to determine relationships between AST n-grams and faults in both open source and commercial systems. We build defect prediction models using three machine learning techniques.

Results: We show that AST n-grams are very significantly related to faults in some systems, with very large effect sizes. The occurrence of some frequently occurring AST n-grams in a method can mean that the method is up to three times more likely to contain a fault. AST n-grams can have a large effect on the performance of defect prediction models.

Conclusions: We suggest that AST n-grams offer developers a promising approach to identifying potentially defective code.

1. Introduction

The aim of this paper is to automatically identify features of faulty Java code and use these features to improve defect prediction performance. Our approach is based on analysing the Abstract Syntax Tree (AST) for a piece of code. AST n-grams are sets of Java AST nodes. These AST n-grams define the low level programming constructs that have been used in a piece of code and the order in which these are used. We analysed the code to identify AST n-grams in nine open source systems and two commercial telecommunication Java systems. We report many AST n-grams that are significantly associated with faults[1] across all eleven systems. We show that including AST n-grams in defect prediction models improves predictive performance.

Traditionally studies have focused on investigating which static features of code are associated with defects [31]. These previous approaches are top-down, focusing on a particular pre-selected set of code features, for example features associated with coupling or size. Many other features of code that may be fault-prone are not considered in such top-down approaches. The performance of these traditional defect prediction models seems to have reached a performance ceiling [49]. In response to this ceiling a new bottom-up approach to identifying the defective features of code is emerging. This approach automatically learns the defective features of code by analysing the semantics of the code via the Abstract Syntax Tree. Wang et al. [71] built promising defect prediction models based on a subset of AST nodes using neural networks. Pradel and Sen [55] also used a neural network to build good defect prediction models using a sub-set of AST nodes (those based on identifiers and literals). We extend both of these previous studies by analysing the full set of AST nodes in relation to defects, rather than only the limited sub-set of features previously investigated. We report important new code features related to defects. We also go further by reporting our results at method rather than class level and evaluating our approach on an extended set of projects including two closed source projects. We identify features of code not used in defect prediction previously and which have large effects on the performance of defect prediction models.

Our approach starts by serialising a Java method's AST. For each method, a serialisation is created by using a pre-order traversal of the AST, with the ordering being determined by the sequence in which the

* Corresponding author.
E-mail addresses: t.shippey@herts.ac.uk (T. Shippey), dbowes@uclan.ac.uk (D. Bowes), t.hall3@lancaster.ac.uk (T. Hall).

[1] We use the IEEE definition [35] of a fault being a reported defect, where a defect is a mistake in code made by a developer which may result in a failure of the program to execute as planned.

https://doi.org/10.1016/j.infsof.2018.10.001
Received 20 February 2018; Received in revised form 4 September 2018; Accepted 2 October 2018
Available online 4 October 2018
0950-5849/© 2018 Elsevier B.V. All rights reserved.

nodes are visited. AST n-grams are then extracted from the method serialisation. These AST n-grams are an n-gram[2] of the serialised AST, where an n-gram unit is a node of the AST. These AST n-grams capture the low level building blocks that have been used in the code and, so, provide comprehensive fine grained insight into the features of that code.

We investigate and quantify the relationship between AST n-grams and faults, by answering the following research questions:

Research Question 1: Are any AST n-grams significantly associated with faulty code?

Research Question 2: What is the effect size of AST n-grams significantly associated with faulty code?

Research Question 3: Does the inclusion of AST n-grams that are significantly associated with faults in defect prediction models improve the performance of these models?

We answered the first two research questions by analysing five different systems and, to guard against overfitting, we added six systems for the third research question. In total there are nine open source systems and two commercial telecommunication systems. Using the Java AST, we extracted the AST n-grams from each system and used the SZZ [67] algorithm to identify faulty methods. We then used non-parametric tests to identify significant relationships between AST n-grams and faults in Java methods. We calculated the effect size of the significant relationships found. Finally, we performed defect prediction on all systems, creating models with the AST n-grams significantly associated with faults, and models without. We compared these models to determine if there was a significant difference between the performance. As a baseline, we also compared our models, which are built with all the possible AST nodes, with the reduced set used by Wang et al. [71]. Our approach differs slightly from Wang et al. [71] as our analysis focuses on the type of AST nodes rather than the contents of those nodes. Wang et al. [71] provide node content analysis for a small set of nodes (names of methods and variables). Our analysis de-emphasises node contents as Wang et al. [71] reported that their node content analysis produced project and developer-specific findings are not generalisable.

Our results make three contributions:

Contribution 1. We present an automatic code analysis technique that comprehensively and objectively serialises all low level code constructs used in a software system. Specifically we introduce the concept of a method serialisation and n-gram of this serialisation called an AST n-gram. This analysis technique allows researchers and practitioners to better understand the structure of the code in individual systems.

Contribution 2. We present important new evidence on fault-prone code constructs. We identify relatively common code structures which can make a method four times as likely to be fault-prone. We identify two code structures which involve identifiers which are fault-prone across all five systems we investigate. This new evidence of fault-prone code structures provides researchers and practitioners with new information with which to strengthen existing defect reduction approaches.

Contribution 3. We show that the inclusion of AST n-grams in within-project software defect prediction models significantly improves the performance of models built using source code metrics. Performance improves when AST n-grams significantly associated with faults are added to the models. This improvement can be up to 4.6 times that of the model constructed with just source code metrics. This means that we could find up to 4.6 times more defects using AST n-grams. Our findings can improve the effectiveness of defect prediction and also in the future could be integrated into developer IDE's (Integrated Development Environments) to reduce faults being initially introduced into code, or efficiently direct testing.

The rest of this paper is structured as follows: Section 2 describes related work. Section 3 outlines how we conducted our investigation and Section 4 presents results. Section 6 we highlight related work and we note the potential threats to validity in Section 5. In Section 7 we discuss the implications of our findings. Finally we conclude in Section 8.

2. Background

Software defect prediction uses machine learning to determine potentially defective areas in software code. The predictions make it possible for the developer to focus on areas of the software system before release, reducing the time and effort of finding defects by other means. Software defect prediction relies on three main components; dependent variables, independent variables and a model. Dependent variables are the defect data for the particular piece of code (i.e. is it defective or not), which can be binary, or continuous. Independent variables are the metrics which can describe the software code, how it has changed or who changed it. Independent variables come in two forms, software code metrics; those that can be derived from the software code itself, and process metrics; metrics that measure the change of software code or software practices over time. The model contains the rule(s) or algorithm(s) that predict the dependent variable from the independent variables. These rules can be as simple as the number of independent variables in the model, or be as complicated as decision trees[3] and regression[4] techniques. To determine the effectiveness of the model, the variables are split into test and training sets. Where the training set is used to create a model and that model is then used on the test set to predict potential defects. These predictions are then investigated to determine if they are correct or not by certain performance measures.

Previous work on features of code in relation to defects is focused on defining and evaluating source code metrics (SCM) that measure particular code features. Examples of source code metrics include - lines of code, object oriented metrics and McCabe's complexity metrics. Various studies have measured source code using such metrics and looked at how the code features measured relate to defects [6,33,41,46,51,73,78].

Lines of code (LOC) is a simple measure that has been commonly used to indicate where defects are. For example, Fenton and Ohlsson [22] analysed pre and post release defects of a large communications system. They found that LOC was good at ranking defective methods. LOC has been used in many other studies [8,30,38,76,78] and has been reported to be good at predicting defective code [31]. However LOC measures only one coarse grained feature of code and so provides limited insight into potential sources of defects.

Chidamber and Kemerer (CK) developed six Object Oriented (OO) metrics to measure the object oriented features of code (e.g. Coupling, Depth of Inheritance Trees and Weighted Methods per class). These metrics have been successfully used in studies to identify defective code [4,11,15,19,39]. Although, compared to LOC, the CK metrics do measure some finer grained features of code, and also identify more of those features, they still identify only a fixed subset of possible code features.

McCabe's cyclomatic complexity metric [44] focuses on identifying branching structures in code and measuring the number of logical paths though the code. Other forms of cyclomatic complexity have been proposed [23,81]. Cyclomatic Complexity is another commonly used metric in defect prediction studies with mixed success (e.g. studies [47,48,69]). However, again, when used in defect prediction Cyclomatic Complexity focuses only on a small set of pre-determined code features likely to be related to defects.

Most of the traditional SCMs (above) have been extensively used in previous defect prediction studies [31]. Most of these metrics suffer from being very coarse grained and with capability to measure only a small sub-set of code features. Gray et al. [29] suggest that the coarse grained nature of such metrics prevents machine learning techniques

[2] An n-gram is a term we have taken from computational linguistics describing a contiguous subset of a larger sequence, normally a sequence of text or speech. In this study we have used the 92 Oracle Java nodes.

[3] A decision tree algorithm is one that creates a graph of decisions based on the chance of an event happening.

[4] Regression analysis seeks to determine best fit of independent value(s) based on a dependent value(s).

from effectively differentiating between defective and non-defective methods: if one method has the same metric values as another (say in terms of LOC), but they have not been labelled the same in terms of their defectiveness, this will hinder the learning algorithm's ability to learn. Gray et al. [29] identifies many methods in the NASA datasets[5] which have identical values across a range of metrics but different defectiveness labels. This suggests that the current commonly used set of metrics is not sufficient to differentiate methods for defect prediction.

Complexity and size are code features commonly used in defect prediction [47,48,69]. Despite much effort in identifying and evaluating such features of code, there is no static code feature which consistently identifies problematic code across systems [31,48]. Code features that indicate defects are usually system-specific [80]. Combinations of features have, so far, performed most promisingly in defect prediction. For example, Shivaji et al. [66] used combinations of static code metrics, object oriented metrics, churn metrics and textual features while Bird et al. [11] used combinations of developer contribution network metrics. Unfortunately, collecting data for such combinations is difficult, time consuming and costly. Furthermore, the ability of such combinations to identify defects, relies on the performance of each single feature included in the combination. Therefore, it is important to be able to identify features indicative of defective code and to develop associated code analysis techniques to identify these features.

Defect predictions are usually reported in studies at the package, class or file level of granularity (e.g. [4,52]). Hata et al., report that predictions at this relatively high level of granularity are not necessarily useful to developers [34] and that predictions at lower levels of granularity are likely to be most useful to developers. Such low granularity predictions (e.g. at method level) present developers with fewer lines of code in which to locate the predicted defect. Locating the predicted defect is often via manual inspection and so the fewer the lines of code to be inspected the less developer time is wasted searching for the defect. Giger et al. [26] report good predictive performance at method level using both change metrics and source code metrics. However achieving good predictive performance at the method level is not easy. Indeed Pascarella et al. [54] replicated [26] with a release-based performance evaluation strategy but reported poor predictive performance at method level. Despite a growing preference for method-level defect prediction, it remains an open challenge to build defect prediction models at method level [54]. We are amongst the few studies reporting defect predictions at method level and the performances that we report are competitive to studies reporting at higher levels of granularity.

3. Methodology

To identify AST n-grams of Java code which are associated with faulty methods we collected data about which methods were faulty. We also needed to know which set of Java AST n-grams each method contained. We used statistical techniques to identify which AST n-grams are significantly associated with faulty and with non faulty methods of code. We finally include the AST n-grams that are significantly associated with faulty methods of code to defect prediction models and compare performance of those models, to models formed with the reduced AST n-gram set proposed by Wang et al. [71].

3.1. Open source and commercial datasets

The open source Java systems analysed in this study were chosen because they have already been extensively used in defect prediction studies [31]. Although we collected fault data ourselves, we chose Eclipse.JDT.core 3.0 because the faults for this system had previously been mapped between the bug tracking system and the version control system [10,40,67]. Using a system which had been analysed for faults previously allowed us to validate our own technique for locating code that was faulty. We also analysed major releases of ArgoUML 0.20 and AspectJ 1.7.0 because they were Java solutions to different problems and had also been previously studied [18,57,72]. We also collected fault data from two commercial telecommunications systems. The code, together with raw bug tracking and version control data was provided to us by a large international telecommunications company based in the UK. The contextual information for each system can be found in Table 1. During the software defect prediction phase of this work, we added six more systems (shown in Table 1). We added these extra systems because we wanted to be sure that the defect prediction results we had for the first five systems would be sustained in other open source systems. These systems were chosen as our previous work had shown them to have sufficient defects to perform defect prediction [65].

Table 1
The 11 systems analysed in this paper. The T2and T1release number is the revision number before the systems were put into production.

System	Release	KLOC	Total Methods
EJDT	3.0	292	13,885
ArgoUML	0.20	273	12,330
AspectJ	1.7.0	353	21,980
T1	-	52	3914
T2	-	36	4896
JMRI	2.4	550	19,861
SocialSDK	1.1.8.2015	69	10,183
GenoViz	5.4	193	8489
JBoss Reddeer	1.2	38	6475
K Framework	3.6	39	5297
JMOL	6	225	2269

3.2. Identifying which methods are faulty

For each of the systems we compiled a dataset of faulty methods. We found which methods were faulty at the time of release by finding the fault insertion and fix points. To identify faulty methods we used the SZZ approach as it has been used in many previous studies [16,24,40,41,74,79]. SZZ is a fault linking algorithm described by Śliwerski, Zimmermann, and Zeller [67]. SZZ was based on work by Cubranic and Murphy [16] and Fischer et al. [24], who inferred links between Bugzilla defect reports with CVS commit messages. The SZZ algorithm matches the fault fix described in a bug tracking system with the corresponding commit in a version control system that 'removed' the fault. By backtracking through the version control records, it is possible to identify earlier code changes which ended up being 'fixed'. It is assumed that the earlier code changes inserted the fault. The method of code is therefore labeled as faulty between the time the fault was inserted and the time it was fixed. Using this technique it is possible to identify for a particular snapshot of the code base, which methods are faulty and which are not. If there are multiple changes multiple times in the past, we assume that the fault inducing change is the one immediately before the defect report. The method is marked as defective if the version snapshot lies between the fault inducing commit and the fault fixing commit and there is no change between these two commits. The tool we have created tracks individual lines throughout the history of the project to determine which methods are defective at a particular time. More details about our tool can be found in [65].

There will be defective methods which have not yet been reported. It is therefore important to carry out the fault mapping after sufficient time has passed for users to report most faults. It is unlikely that all defects will be reported and therefore there will be false negatives. Kim et al. [42] suggests that as long as the number of false negatives and false positives is less than 20% in total, defect prediction can be carried out [42]. This is an important point, early work by Zimmermann et al. [79] only managed to map about 50% of faults reported in the

[5] http://nasa-softwaredefectdatasets.wikispaces.com/

Table 2
Checking the bug-links for false positives. Zimmermann has similar precision but lower recall. Precision is the proportion of correctly classified bug-links from all those bug-links classified ($TP/(TP + FP)$) and recall is the proportion of bug-links correctly classified from all possible correct bug-links ($TP/(TP + FN)$).

	This Paper	Zimmermann et al. [79]
True Positive	727	483
False Positives	5	2
Total Positives	732	485
Total Negatives	151	398
Recall	80%	53%
Precision	99%	100%

Table 3
Table to show the fault density of each of the datasets. N.B. Tables have been presented in order of percentage faulty.

System	Version	Total Methods	Faulty Methods	% Faulty
T2	198,468	4896	612	12.5
T1	198,468	3914	360	9.2
EJDT	3.0	13,885	589	4.24
ArgoUML	0.20	12,330	42	0.34
AspectJ	1.7.0	21,980	19	0.09
JMOL	6	2269	294	12.96
GenoViz	5.4	8489	827	9.63
K Framework	3.6	5297	421	7.95
SocialSDK	1.1.8.2015	10,183	754	7.40
JMRI	2.4	19,861	1385	6.97
JBoss Reddeer	1.2	6475	416	6.42

bug tracking systems to changes in the code base. Later Bird et al. [10] improved the mapping by removing some of the constraints that Zimmermann had introduced, for example the requirement to have matching bug IDs in a predefined format. The implementation of SZZ used in this paper was improved slightly from the original. It has a higher weighting for those numbers found in commit messages that are in the bug database and takes into account the “Fix for” prefix. The implementation was verified by manually checking ALL bug links found for EJDT 3.0. Table 2 shows that the implementation used in this study has 80% recall and 99% precision.

Alencar da Costa et al. [17] recently evaluated the SZZ variants used in studies and our variant falls into Alencar da Costa et al., s L-SZZ category. This is because not only does our tool use annotation graphs to achieve line mappings and is aware of meta changes but also identifies the largest bug introducing change. Approaches in the L-SZZ category are currently most mature in identifying bug introducing changes.

Table 3 shows the defectiveness of each of the systems studied in our experiment. The levels of fault-proneness varies across all systems. T2has the highest fault density with over 12% defective methods, followed by the second telecommunications system T1with around 9%. AspectJ has only 19 methods faulty out of a potential 21,980 and so has a very low fault-proneness of 0.09%.

3.3. Extracting the Java AST n-grams

For each of the systems investigated, each file in the project was compiled using the Oracle JDK which builds an AST. Each method in a class was turned into a method sequence using a modified version of the standard Oracle Java pretty printer[6]. A method sequence is a list of AST nodes in order of when they are visited by the pretty printer, which is a pre-order depth first traversal. The pretty printer uses a visitor pattern to transforms the source code by applying styling rules (e.g. appropriate indents and spacing), which can make it easier for people to view. We

[6] PrettyPrinter.java is found in tools.jar of the Oracle JDK

```
public void cloneForMethod() {
    int x = 10;
    for(int i = 0; i < x; i++) {
        System.out.println(x*i);
    }
}
```

Fig. 1. The code that is transformed in Fig. 2 using the Pretty Printer.

modified the pretty printer so when it visits a node on the AST, it will store that node in a sequence. This means that the order in which code constructs are visited is maintained. For example, we would transform the piece of Java source code in Fig. 1 into the method sequence in Fig. 2. This method was used to transform all methods in the five systems to create a database of method sequences. Table 4 shows how many methods were transformed into method sequences and the average length of these method sequences. Method sequence lengths vary across the system, with EJDT 3.0 having the longest average by around 20 nodes. The commercial systems have both a lower average sequence length and maximum sequence length. This is because the company has a policy of keeping both classes and methods as short as possible.

From our method sequences, we can extract n-grams, which we call *“AST n-grams”*. We extract from each method AST n-grams which are 1-gram, 2-gram and so on, up until the maximum length of n-grams. For example, we want to extract the AST n-grams from an example sequence (M) `[A; A; C; D; E]` where the maximum AST n-gram length is three. In total the set CS contains 11 AST n-grams. Fig. 3 shows the AST n-grams in set CS. For this study we have set the maximum AST n-gram length to seven. This is because as an AST n-gram length gets longer, there is an exponential increase in the number of AST n-grams available. The limit of seven nodes prevented potential computational problems when we came to analysing the results. We do not extract the contents of the AST n-grams as our analysis is at the type level of granularity rather than the instance level of granularity. This means that our results are less influenced by the particular coding idiosyncrasies of individual developers in individual projects. Our aim is to present results that are more likely to be generalisable.

3.4. Analysing the AST n-grams

To find which AST n-grams are related to faults we compare the ratio of n-grams in non faulty code to the number of n-grams in faulty code. In this study we compare the number of instances of an AST n-gram. An n-gram is marked as faulty if it appears in a faulty method at the chosen snapshot. As the distribution of AST n-grams is non-normal we used the non-parametric Fisher’s exact test to determine if an AST n-gram was significantly associated with a fault. Fisher’s exact test determines if there are non-random associations between two categorical variables. In our case, the classifications are the presence or absence of a n-gram and faulty or not. To clarify our statistical analysis, we provide a worked example using EJDT 3.0. In EJDT 3.0 there is a 2.65% chance of any AST n-gram being defective. This is determined by dividing the number of faulty AST n-gram instances over all AST n-gram instances. The contingency table (Table 5) is for the AST n-gram METHOD_INVOCATION MEMBER_SELECT. This AST n-gram is the start of a method call. Our null hypothesis is that METHOD_INVOCATION MEMBER_SELECT appears in the same proportion of faulty n-grams as non-faulty n-grams. Our alternate hypothesis is that the AST n-gram appears in a greater proportion of faulty n-grams than non-faulty n-grams. When the Fisher’s exact test is applied to the contingency table in Table 5 we get a p-value of 0.0. This is below our α of 0.001. This means there is evidence to reject the null hypothesis. We would conclude that the start of a method call in EJDT 3.0 is significantly associated with faults. We set our α to 0.001 as we wanted to reduce the amount of false positives when analysing the potential significant n-grams.

```
METHOD; MODIFIERS; PRIMITIVE_TYPE; BLOCK; VARIABLE; MODIFIERS;
   PRIMITIVE_TYPE; INT_LITERAL; FOR_LOOP; VARIABLE; MODIFIERS;
   PRIMITIVE_TYPE; INT_LITERAL; LESS_THAN; IDENTIFIER; IDENTIFIER;
   EXPRESSION_STATEMENT; POSTFIX_INCREMENT; IDENTIFIER; BLOCK;
   EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT;
   MEMBER_SELECT; IDENTIFIER; MULTIPLY; IDENTIFIER; IDENTIFIER;
```

Fig. 2. The code from Fig. 1 transformed into a node sequence.

Table 4
Table to show the sequence statistics for each of the systems analysed. The minimum sequence length for all systems was three. A sequence of length three is found for zero argument empty constructors e.g. `public Foo()` N.B. Tables have been presented in order of percentage faulty.

System	Version	Total Sequences	Avg Sequence Length	Sequence σ	Max Sequence Length
T1	198,468	3914	19.73	21.88	351
T2	198,468	4896	24.00	25.87	386
EJDT	3.0	13,885	55.47	129.64	4802
ArgoUML	0.20	12,330	36.04	68.61	2989
AspectJ	1.7.0	21,980	37.79	75.18	2644

```
CS = {(A), (A; A), (A; A; C), (A; C), (A; C; D), (C), (C; D),
   (C; D; E), (D), (D; E), (E)}
```

Fig. 3. The possible set of 11 AST n-grams (with a maximum n-gram of three) taken from an example method sequence `A; A; C; D; E`.

Table 5
Contingency Table for AST n-gram METHOD_INVOCATION MEMBER_SELECT in EJDT 3.0.

	Faulty	Non-Faulty	Total
N-gram	32,493	7750	40,243
(%)	80.74	19.26	-
No N-gram	230,681	4,311,566	4,542,247
(%)	0.05	95.5	-
Total	263,174	4,319,316	4,582,490

We have calculated the effect sizes for the AST n-grams significantly associated with faults using an odds-ratio [14]. The odds-ratio will quantify how strongly the presence of an AST n-gram will be associated with a fault. The odds-ratio has a range of zero to infinity. If the odds-ratio is less than one, then the n-gram may be more associated with non-faults. When the odds-ratio is one then that means that the AST n-gram does not have any effect on the fault proneness of the code. The greater the amount away from one, the greater the effect that the AST n-gram has on the association with faults.

3.5. Defect prediction using the AST n-grams

3.5.1. Building the defect prediction models

To answer RQ3, we carried out defect prediction using the AST n-grams. To show that the significant AST n-grams were having an effect, we created basic defect prediction models without the AST n-grams. These models were created with the default set of static code metrics calculated by the program JHawk[7]. These metrics include lines of code (LOC), variable declarations and Halstead metrics. In total there are 27 method level metrics and the full list of software code metrics used can be found in Appendix A. We then added the AST n-grams significantly associated with faults, up until a maximum of 200 n-grams. We chose a maximum 200 AST n-grams due to computational constraints. Adding attributes to models has an exponential computational time costs when building the model. These increased costs vary from learner to learner with high costs associated with, for example, Random Forest learners but minimal cost increases associated with Naïve Bayes.

In total, we had 11 metric datasets per classifier for each system, one metric dataset with just JHawk metrics, and then 10 more metric datasets with AST n-grams significantly associated with faults as additional attributes. When we added the AST n-grams as attributes to the models, we used the binary presence of an n-gram, not the number of n-grams in a method. We used the binary value rather than total as binary values are standard in defect prediction. Our approach of adding AST n-grams to a base of existing source code metrics increases the information available to the defect prediction model. Increasing information diversity has been previously shown to improve predictive performance (e.g. [31]) and incrementally adding information to the model is being increasingly used in defect prediction research (e.g. [12]).

3.5.2. Training the models using AST n-grams

To determine which AST n-grams to add as attributes to our initial baseline model, we calculated the top 200 most occurring AST n-grams significantly associated with faults at the 99.9% level only from methods in the training set for each run and fold. We calculated the significant AST n-grams in only each training set to make sure that when testing our models they did not have any prior knowledge of the significant AST n-grams. For each run and fold, we created 10 models per classifier to compare with the original model. From zero up to 50 n-grams, we added an additional 10 significant n-grams to the metric dataset and trained the model on this new dataset, after 50 we increased the additional number of AST n-grams by 25 until we reached 200. So the first model would be trained using the initial JHawk metrics only, then second model uses the JHawk metrics plus the top ten most occurring AST n-grams significantly associated with faults in the training set and the third model would have the top 20 and so on, until 50 AST n-grams. After 50 n-grams, the models would be trained with an additional 25 AST n-grams, so the next model after 50 is 75 n-grams, then 100, then 125 and so on. Until the last comparison model had the top 200 most frequently occurring AST n-grams that were significantly associated with faults in that particular training set. Our analysis suggests that generally the more n-grams provided the more defect prediction performance improves. However this is not always the case and identifying the particular

[7] We used JHawk version 5.0 to conduct our study

Table 6
The number of significant n-grams in each of the 11 comparison metric datasets.

		Number of Significant N-grams										
Model #	JHawkMetrics	10	20	30	40	50	75	100	125	150	175	200
0	✓	×	×	×	×	×	×	×	×	×	×	×
1	✓	✓	×	×	×	×	×	×	×	×	×	×
2	✓	✓	✓	×	×	×	×	×	×	×	×	×
3	✓	✓	✓	✓	×	×	×	×	×	×	×	×
4	✓	✓	✓	✓	✓	×	×	×	×	×	×	×
5	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×
6	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×
7	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×
8	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×
9	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×
10	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×
11	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

combinations of n-grams that work best for particular data sets/projects improves performance most. Song et al. [68] also notes the need to tailor features to projects to achieve good predictive performance. Table 6 shows how many AST significant n-grams are in each of the 11 models.

3.5.3. Cross validation scheme

Each dataset was split into ten stratified folds to perform cross validation. Each fold was held out in turn to produce a test set and the other folds were used to produce the training set. Using stratified cross validation ensures that there are instances of the defective class in each test set, thus reducing the likelihood of classification uncertainty. The classifiers were trained using the training sets and the test set then used to evaluate the model. This experiment was repeated 100 times for each classifier and system dataset. We chose 100, instead of the more common 10 times, as Mende [45] reports that using 10 experiment repeats results in an unreliable final performance figure. This meant that we determined the top 200 AST n-grams that are significantly associated with faults in 1000 training sets and computed 33,000 models per system[8].

3.5.4. Classifier selection

We created our models using three classifiers: Naïve Bayes, J48[9], and Random Forest. Bayes classifiers are simple probabilistic classifiers based on applying Bayes' theorem. Naïve Bayes is the most popular Bayes classifier. Naïve Bayes is called naive since every feature (module) is assumed to be fully independent. Naïve Bayes will produce models based on the combined probabilities of a dependent variable being associated with the different independent variables. J48 and Random Forest use decision tree learning. Decision tree learning uses a decision tree as a predictive model to map observations to a target value. Decision trees that only take a finite set of values are called classification trees. The J48 classifier builds decision trees based on the information gain of attributes. At a node in the tree, the J48 algorithm will chose the attribute of the data that most effectively splits the set into subsets enriched in one class or another. The split is based upon the attribute with the highest normalised information gain. This then repeats on the smaller subsets until the tree is built. Random Forest is an ensemble technique which aggregates the predictions made by a collection of decision trees. Each of the trees is said to be randomised as they train on a subset of available features. The mode classification across all individual classifiers is taken as the final prediction for each test vector (method). We chose these three classifiers because they are popular modelling classifiers according to Hall et al. [31]. We built the models using a Java implementation of Weka[10] using the default options for each classifier.

[8] 100 runs * 10 folds * 3 classifiers * 11 datasets = 33,000 models

[9] J48 is the Weka implementation of the C4.5 algorithm.

[10] Weka version 3.7.12

3.5.5. Predictive performance

We calculated each model's performance using four different measures: precision, recall, f-measure and Matthew's correlation coefficient (MCC). See Table 7 for definitions of these measures. Precision, recall and f-measure were chosen as they are very commonly published in defect prediction papers, and have a range of 0 to 1, with 0 being no better than random prediction and 1 being perfect prediction. MCC was chosen because it is easy to understand and includes all four components of the confusion matrix. MCC has a range of -1 to 1, with 1 being perfect prediction and -1 being total disagreement. An MCC score of 0 means the performance is no better than a random prediction.

We will statistically compare what effect the addition of AST n-grams has on our models. Our hypothesis is that models trained using AST n-grams will perform better than ones that do not contain AST n-grams significantly associated with faults. We used the Wilcoxon signed-rank test to compare the differences in performance measurements from the 100 runs of the models with the significant AST n-grams to those without. We have used the Wilcoxon signed-rank test as our data does not follow the normal distribution and is paired. Our data is paired as we have calculated the performance measurement for each run and fold (and the methods in each run and fold are always the same). We calculated the effect size of each test using Cliff's delta. We again have used Cliff's delta as our data is not normally distributed. We calculated the effect size to show the level of impact the AST n-grams had on the performance of our models. Cliffs delta(d) gives a score between -1 and 1, with ± 1 being the largest effect size. If d is less than 0.147 the effect is negligible, d above 0.147 and lower than 0.33 the effect is small, bigger than 0.33 and lower than 0.474 the effect size is medium and d value over 0.474 is considered large [60].

4. Results

In total there were 306,924 different AST n-grams found across the five systems we used to perform research questions 1 and 2. In all five of these systems there are AST n-grams significantly associated with faults. Our results will highlight which AST n-grams are significant across the five systems and which AST n-grams appear most in individual systems. We will highlight the AST n-grams which have the biggest effect sizes. Finally, along with the addition of six further projects we will show that our defect prediction performance improves when AST n-grams are added.

4.1. RQ1- Are Any AST n-grams significantly associated with faulty code?

There are 6411 AST n-grams[11] significantly associated with faults in at least one of the systems at the 99.9% level. Of these AST n-grams, 95%

[11] We refer to the number of AST n-gram types rather than the number of AST n-gram instances throughout the Results Section

Table 7
Performance measures used in this study.

Measure	Defined as	Description
Recall (*R*)	$\frac{TP}{TP+FN}$	Proportion of defective units correctly classified
Precision (*P*)	$\frac{TP}{TP+FP}$	Proportion of units correctly predicted as defective
F-measure	$\frac{2 \times R \times P}{R+P}$	Harmonic Mean of precision and recall
Matthews Correlation Coefficient (MCC)	$\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$	A correlation coefficient between the observed and predicted binary classifications. Also known as the ϕ coefficient

```
try {
    String n = bundle.getString(name[i]);
    if (n != null && n.length() > 0) {
        result[i] = n;
    } else {
        result[i] = name[i];
    }
    } catch (MissingResourceException e) {
        result[i] = name[i];
    }
```

Fig. 4. The red portion of the text is the AST n-gram VARIABLE; MODIFIERS; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER. MODIFIERS; IDENTIFIER; METHOD_INVOCATION MEMBER_SELECT; IDENTIFIER; will be part of the same code, but does not have the `VARIABLE` kind, which starts the line. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Table 8
Table to show the total number of AST n-grams and AST n-grams significantly associated with faulty methods for each system ($\alpha = 0.01$). N.B. Tables have been presented in order of percentage faulty.

System	Unique AST n-grams	Sig AST n-grams	%
T2	29,147	919	3.15
T1	18,351	359	1.96
EJDT 3.0	178,780	4745	2.65
ArgoUML	90,499	328	0.36
AspectJ	165,005	448	0.27

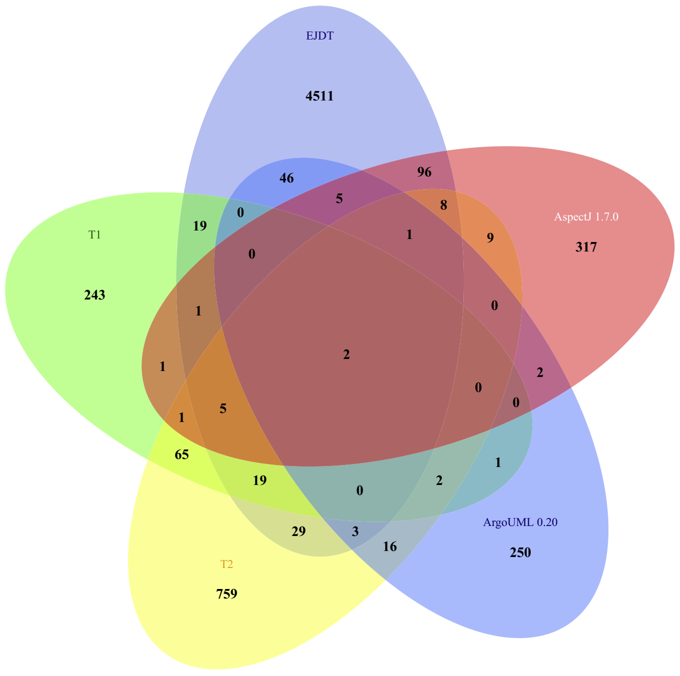

Fig. 5. A Venn diagram to show the distribution of fault-prone AST n-grams between the systems. In total 2 AST n-grams are significantly associated with faults in all five systems.

are significantly associated with faults in only one of the systems. The two commercial systems share around 7% of the same AST n-grams significantly associated with faults. Fig. 5 shows that only two AST n-grams are significantly associated with faulty methods across all five systems. Had the association been random we would have expected $0.03515 \times 0.0196 \times 0.0265 \times 0.0036 \times 0.0027 = 0.00007219945$ AST n-grams to be significantly associated with faults in all systems. The two AST n-grams which are significantly associated with faults in all five systems is greater than would be expected by chance ($0.000312\% > 0.00000000031\%$). These two n-grams are: `VARIABLE; MODIFIERS; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT IDENTIFIER` and `MODIFIERS; IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER`. Example code of these AST n-grams is found in Fig. 4, shown in red. Both of these n-grams are examples of a method call and are related to one another. This shows that methods with method calls could be more likely to be faulty across systems. Both of the n-grams have an odds ratio of 1.95, meaning that a method has nearly double chance of being faulty when it has a method call, compared to methods without a method call. Methods that do not contain method calls are less likely to be faulty across systems. This makes sense as these methods are likely to be methods such as getters, setters or interface methods and will be less fault prone.

Table 8 shows that EJDT has the most AST n-grams significantly associated with faults (4,745) however, T2has the largest by percentage (3.15%). ArgoUML has the least number of AST n-grams associated with faults (328). In EJDT more than 12% of the AST n-grams are significantly associated with faults. There does not seem to be a relationship between the number of unique AST n-grams and the number of AST n-grams that are significantly associated with faults.

Table 9 shows the top five most fault-prone AST n-grams in each system. Identifiers and method calls appear often in the 25 AST n-grams shown. These n-grams include `IDENTIFIER` and `METHOD_INVOCATION`. Some of these AST n-grams have very low odds-ratio, indicating that they may not impact the overall faultiness of a method very much. However, some of the ratios are still quite large. For example, `IDENTIFIER METHOD_INVOCATION` has an odds ratio of 1.56, meaning it could impact the faultiness of a method around 56% more than a method without this n-gram.

In each of the systems there are AST n-grams that are significantly associated with faults and that appear only in faulty methods. Table B.16 shows the n-grams which appear most often but only in faulty methods. All of the n-grams are around six or seven nodes in length, but appear infrequently. In the two commercial systems, the n-grams tend to appear around 20 times or less, in ArgoUML and AspectJ, the n-grams appear less than 10 times. This suggests possible differences between commercial and open source systems.

Table 9
Table showing the five most faulty AST n-grams in each system.

AST n-gram	Total N-grams	Faulty N-grams	% Faulty	Odds-Ratio
T2			3.15	
IDENTIFIER	39,371	7361	18.70	1.09
IDENTIFIER; IDENTIFIER	10,958	2233	20.38	1.21
IDENTIFIER; METHOD_INVOCATION	4439	1101	24.80	1.56
VARIABLE; MODIFIERS; IDENTIFIER	5296	1057	19.96	1.18
METHOD_INVOCATION; IDENTIFIER	4676	915	19.57	1.15
T1			1.96	
IDENTIFIER; IDENTIFIER	6541	991	15.15	1.14
IDENTIFIER; IDENTIFIER; IDENTIFIER	1762	298	16.91	1.30
IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER	1758	293	16.67	1.27
IDENTIFIER; EXPRESSION_STATEMENT;	1531	275	17.96	1.40
IDENTIFIER; VARIABLE; MODIFIERS; IDENTIFIER; VARIABLE; MODIFIERS; IDENTIFIER	1265	236	18.66	1.46
EJDT 3.0			2.65	
MEMBER_SELECT	90,007	21,440	23.82	1.31
MEMBER_SELECT; IDENTIFIER	71,051	16,267	22.89	1.24
EXPRESSION_STATEMENT	38,603	7816	20.25	1.06
PARENTHESIZED	29,105	6403	22.00	1.18
IF; PARENTHESIZED	19,325	4390	22.72	1.23
ArgoUML 0.20			0.36	
METHOD_INVOCATION	54,926	505	0.92	1.19
MEMBER_SELECT; IDENTIFIER	34,309	328	0.96	1.24
EXPRESSION_STATEMENT	22,257	219	0.98	1.27
EXPRESSION_STATEMENT; METHOD_INVOCATION	15,166	172	1.13	1.47
EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT	10,808	114	1.05	1.37
AspectJ 1.7.0			0.27	
MEMBER_SELECT	92,366	630	0.68	1.27
METHOD_INVOCATION	83,991	550	0.65	1.22
MEMBER_SELECT; IDENTIFIER	73,475	527	0.72	1.34
METHOD_INVOCATION; MEMBER_SELECT	64,647	492	0.76	1.42
METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER	51,380	395	0.77	1.43

4.2. RQ2 - What Is the effect size of AST n-grams significantly associated with faulty code?

A relatively high number of AST n-grams have a very large effect on the fault-proneness of methods. For example, in EJDT the AST n-gram `BREAK; CASE; INT_LITERAL;EXPRESSION_STATEMENT; METHOD_INVOCATION; IDENTIFIER` appears only in faulty methods, so will have an odds-ratio of infinity. This AST n-gram will be linked to a piece of code in a switch statement. Over 1000 other AST n-grams appear only in faulty methods in EJDT. Most of these exclusively faulty AST n-grams occur infrequently with those in EJDT appearing on average around 15 times. The same phenomenon is also present in ArgoUML, AspectJ and the two commercial systems. In these systems the mean number of times an exclusively faulty n-gram occurs is lower than EJDT at around 5, 2, 6 and 7 respectfully. Such infrequently occurring fault-prone AST n-grams contribute relatively little to the overall faultiness of a system.

However, many AST n-grams with a relatively large effect on fault-proneness do appear more frequently. Such AST n-grams are likely to have more impact on the overall faultiness of a system. Table 10 show fault-prone AST n-grams that appear more than two standard deviations away from the mean number of faulty n-grams in all five of the systems studied in this paper. For example, those n-grams chosen for EJDT will be only those n-grams which have over 1032 faulty n-grams $(84.43 + (2 \times 473.91))$. These n-grams will have a greater impact on the overall faultiness of a system.

Figs. 8, 6 and 7 show source code examples of AST n-grams with the highest effect size for the open source systems. The underlined red text highlights which source code is covered by the AST n-gram. The methods that have been chosen for these code examples are methods that have been identified as faulty[12]. Fig. 9 shows an example method from EJDT to highlight the AST n-gram with the largest effect size for each of the commercial telecommunications systems. We are unable to publish source code from these systems for commercial reasons.

Fig. 8 highlights an example of AST n-gram with the highest effect size in EJDT. This n-gram is a long message chain, with four different object requests. In this case too, the result of this long message chain has been used as a variable in a method call. Long message chains have been identified in the past as a problem [25,32]. Figs. 6 and 7 also show examples of message chains as these AST n-grams had the highest effect sizes for these two systems.

4.3. RQ3: Does the inclusion of AST n-grams significantly associated with faults in software defect prediction models help improve their predictive performance?

Yes. The inclusion of AST n-grams significantly associated with faults can result in significant improvements on a models predictive performance. In some cases, these increases are very large. Fig. 10 shows how MCC performance changes as AST n-grams significantly associated with faults are added to the 11 systems studied[13]. In all 11 systems MCC has improved due to the inclusion of AST n-grams and in nine of the 11 systems, these improvements are seen across all three classifiers. The biggest increase in median MCC performance can be seen in EJDT, where J48 has risen by around 0.39, from 0.19 to around 0.58 (213%). In percentage terms, the biggest median increase was in T1 using logistic regression, where MCC increased by just over 3,900% (0.0007 to 0.27) from the baseline model. The biggest increase over all performance measures was seen in Reddeer, where the median precision of the logistic regression model median went from 0 to 0.52 (a percentage increase could not be calculated). The biggest median percentage increase across

[12] It is important to note that the red text that is highlighted may not be the cause of the fault in the method

[13] Line plots for the three other performance measures are found in the Appendix

Table 10
Table to show the AST n-grams with the greatest odds ratios and also appear in over two standard deviations from the mean in all five systems.

AST n-gram	Total N-grams	Defective N-grams	% Defective	Odds Ratio
ArgoUML 0.20				
EXPRESSION_STATEMENT; METHOD_INVOCATION	15,166	172	1.15	1.47
METHOD_INVOCATION; IDENTIFIER	9876	108	1.11	1.42
EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT	10,808	114	1.07	1.37
EXPRESSION_STATEMENT	22,257	219	0.99	1.27
MEMBER_SELECT; IDENTIFIER	34,309	328	0.97	1.24
AspectJ 1.7				
IF; PARENTHESIZED	16,008	142	0.89	1.65
IF	16,008	142	0.89	1.65
IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT	22,812	201	0.88	1.64
IDENTIFIER; METHOD_INVOCATION	27,377	233	0.85	1.59
IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT; IDENTIFIER	18,678	158	0.85	1.57
EJDT				
MEMBER_SELECT; MEMBER_SELECT; MEMBER_SELECT; MEMBER_SELECT	2479	1093	44.09	3.29
MEMBER_SELECT; MEMBER_SELECT; MEMBER_SELECT	3658	1533	41.91	3.01
CHAR_LITERAL	3090	1188	38.45	2.61
MEMBER_SELECT; IDENTIFIER; MEMBER_SELECT; IDENTIFIER	7010	2274	32.44	2.01
ASSIGNMENT; MEMBER_SELECT; IDENTIFIER	4598	1487	32.34	2
T2				
NEW_CLASS	2,567	664	25.87	1.65
IDENTIFIER; METHOD_INVOCATION	4,439	1,101	24.8	1.56
IDENTIFIER; VARIABLE; MODIFIERS; IDENTIFIER	2,986	651	21.8	1.32
IDENTIFIER; IDENTIFIER	10,958	2,233	20.38	1.21
IDENTIFIER; VARIABLE; MODIFIERS	3,751	753	20.07	1.19
T1				
NEW_CLASS; IDENTIFIER	968	220	22.73	1.88
NEW_CLASS	1079	228	21.13	1.71
IDENTIFIER; VARIABLE; MODIFIERS; IDENTIFIER; VARIABLE; MODIFIERS; IDENTIFIER	1265	236	18.66	1.46
IDENTIFIER; IDENTIFIER; EXPRESSION_STATEMENT	1095	203	18.54	1.45
IDENTIFIER; VARIABLE; MODIFIERS; IDENTIFIER; VARIABLE; MODIFIERS	1297	236	18.2	1.42

```
        if ((row != -1) && (c.size() > row)) {
            c.remove(row);
            Model.getCoreHelper().setTaggedValues(tab.getTarget(),
                c);
            model.fireTableChanged(new TableModelEvent(model));
        }
    }
```

Fig. 6. The red portion of text is an example of the AST n-gram EXPRESSION_STATEMENT; METHOD_INVOCATION; MEMBER_SELECT. This method was faulty in the ArgoUML system in class TabTaggedValues.java. This class is a table view of a UML models elements tagged values. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

```
SourceTypeBinding t = (SourceTypeBinding) i.next();
ContextToken tok = CompilationAndWeavingContext.enteringPhase(
                CompilationAndWeavingContext.
                    WEAVING_INTERTYPE_DECLARATIONS, t.sourceName);
weaveInterTypeDeclarations(t);
}
```

Fig. 7. The red portion of text is an example of the AST n-gram IDENTIFIER; METHOD_INVOCATION; MEMBER_SELECT. This method was faulty in the AspectJ system in class AjLookupEnvironment. This class overrides the default EJDT LookupEnvironment. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

```
this.contents[localContentsOffset++] = (byte) nameIndex;
descriptorIndex =
  constantPool.literalIndex(
    codeStream.methodDeclaration.binding.declaringClass.signature())
      ;
this.contents[localContentsOffset++] = (byte) (descriptorIndex >>
   8);
```

Fig. 8. The red portion of text is an example of the AST n-gram MEMBER_SELECT; MEMBER_SELECT; MEMBER_SELECT; MEMBER_SELECT. This is a part of a longer method that was faulty in the EJDT 3.0 system in class ClassFile.java. This class represents a class file wrapper on bytes. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

```
public Expression getExpression() {
    if (expression == null) {
        setExpression(new SimpleName(getAST()));
        }
        return expression;
}
```

Fig. 9. The red portion of text is an example of the AST n-gram NEW_CLASS; IDENTIFIER which has the largest effect size for T2and T1. This method was taken from a method in EJDT 3.0. We are unable to show code from the T2or the T1system. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

all performance measures was for the recall in T1 using logistic regression. The median increased by just below 12,000% (0.002 to 0.22).

In the majority of cases, MCC increases as a greater number of AST n-grams are added. For example, in EJDT, the addition of 10 AST n-grams significantly associated with faults significantly improves the MCC median from 0.19, when using no AST n-grams, to 0.37 (a 97% increase)[14] When 20 AST n-grams are added, the MCC median increases again by around 0.1 (3%), these increases continue and when 200 AST n-grams significantly associated with faults are added, the median MCC has risen just over 212% (0.40) on the original baseline model MCC median (0.19). Table 11 shows that on average over all the systems, adding 10 AST n-grams improves MCC by around 47%, 50 n-grams by around 131% and 200 n-grams 238%. These improvements are also seen in recall (58%, 322% and 574%), precision (24%, 29% and 49%) and f-measure (49%, 222% and 391%).

Despite the poor performance of the models in some of the systems, most notably ArgoUML and AspectJ, AST n-grams can make a significant impact. For example, there is an increase with ArgoUML and Logistic, where MCC raises from around 0.03 to around 0.10 when 75 AST n-grams are added, then stays steady until 200. This could suggest that there was an AST n-gram that has a large effect added in the 50–75 bracket. Similarity, systems K, Reddeer, SocialSDK and T1 have an MCC which is around 0 using the baseline metrics and the Logistic model. The inclusion of n-grams, allows the logistic models performance increase to around 0.20. AspectJ with J48 sees an increase in MCC from 0 to around 0.20 by the time 75 AST n-grams have been added. AspectJ had on average only 69 significant n-grams in each run and fold, so we were unable to get results beyond a maximum of 75 n-grams for this system. The poor performance of AspectJ and ArgoUML is probably due to the very low proportion of defective methods (see Table 3) in the systems, which is a known problem [3].

Fig. 12 plots the median effect size of the change between the defect prediction performance measure MCC with no n-grams and the different numbers of significant AST n-grams. The highest effect sizes are found in EJDT, JMRI and SocialSDK, where Cliff's D reacts the maximum 1. In JMOL it reaches this maximum after 30 AST n-grams using both Naive Bayes and logistic regression. Generally, the more AST n-grams that are added to the model, the greater the effect on the models. In all of the systems, the AST n-grams have eventually a large effect on at least one

[14] Significant at the 99% level with Cliffs D effect size of 0.97.

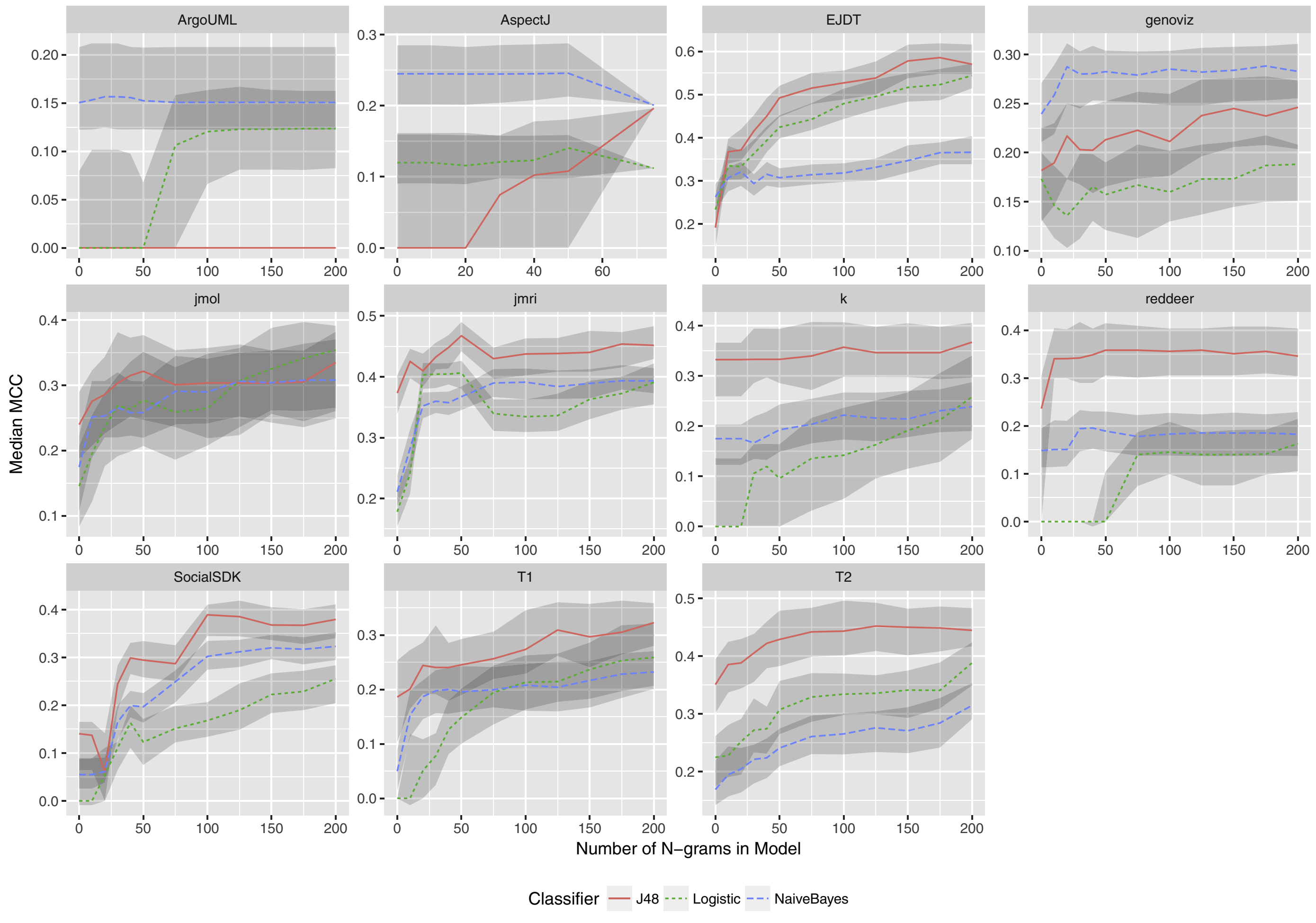

Fig. 10. A line plot to show the change in MCC across all the 11 systems when different levels of N-grams are added to the baseline models.

Table 11
Average changes in the performance measures when AST n-grams significantly associated with defects are added to the base line models.

N-grams	Recall	%	Precision	%	F-measure	%	MCC	%
10	0.03	58.00	0.04	23.82	0.03	49.33	0.03	47.37
20	0.07	124.93	0.04	22.87	0.05	92.42	0.05	61.67
30	0.10	178.86	0.08	34.34	0.08	129.90	0.07	91.39
40	0.11	263.94	0.07	33.82	0.09	189.63	0.09	122.74
50	0.12	321.94	0.06	28.54	0.10	221.62	0.09	131.32
75	0.13	404.48	0.09	38.53	0.12	284.77	0.11	172.33
100	0.14	449.52	0.10	39.65	0.13	311.65	0.12	189.34
125	0.15	456.57	0.10	41.72	0.13	321.04	0.12	196.69
150	0.15	494.20	0.11	45.38	0.14	344.82	0.13	211.35
175	0.16	545.04	0.11	46.57	0.15	372.26	0.13	224.83
200	0.17	573.66	0.12	49.31	0.15	390.58	0.14	237.68

Table 12
The average Cliff's D effect score for all the systems when 200 AST n-grams significantly associated with faults are added to the base line models ($d < 0.147$ Negligible, $d < 0.33$ = small, $d < 0.474$ = medium, $d > 0.474$ large [59]).

Measure	J48	Logistic	Naive Bayes	All
Recall	0.6450	0.8975	0.8492	0.7973
Precision	0.3373	0.6115	0.3483	0.4324
F-measure	0.6678	0.8862	0.7347	0.7629
MCC	0.6267	0.8118	0.6952	0.7112

of the classifiers. In six of the systems, the AST n-grams have a large effect on all of the classifiers. The AST n-grams have at least a small effect on MCC in all of the classifiers in nine of the 11 systems, AspectJ and ArgoUML being the only two systems to miss out. Similar effect sizes are seen for the other performance measures[15]. Table 12 shows that at the when 200 AST n-grams significantly associated with faults are added to the models, the performance measure they have the most effect on is recall, with an average median effect size of 0.80, whilst the lowest effect is on precision, with an average median effect size of 0.43. If we look at all the different n-gram levels, then recall is still has the largest average effect sizes with a Cliffs D value of 0.56, whilst AST n-grams only have a small effect on the precision (0.25). 200 AST n-grams have the largest effect on the logistic classifier performance, with an average effect size of around 0.80. Naive Bayes and J48 have an average large effect size of 0.66 and 0.57 respectfully when 200 AST n-grams are added. Over all the AST n-grams, the average median effect sizes for the classifiers come down to 0.51, 0.45 and 0.40 respectfully, which is around the large effect threshold.

4.3.1. *Comparing the performance of the full set to reduced set*

Fig. 11 shows that using the reduced set that Wang et al. [71] proposed does not achieve the same results as the full set. Firstly, for most of the systems, the number of AST n-grams found in the systems is lower than that for models built with the full set of available nodes. In most cases, the median MCC performance across systems does not rise as dramatically as with the full available AST nodes, or does not rise at all. For example, J48's median MCC in EJDT only rises around 0.04 from the baseline of 0.19 to a 50 AST n-grams MCC of 0.23. This 22% rise is significant (at the 99% level), however this has only a small effect size (0.31). This rise is 0.26 (114%) less than what is achieved using the full sets 50 AST n-grams significantly associated with faults and 0.35 less (156%) less than the full sets 200 AST n-grams (both significant at the 99% level). Overall, median MCC is 0.07 (38%) higher using the full set, compared the Wang et al. [71] reduced set across all the systems and classifiers. Recall is on average 0.08 (45%) higher using the full set, precision 0.08 (22%) higher and f-measure 0.08 (39%) higher. Table 13 shows the difference[16] in performance measures across all of the AST n-gram levels. MCC and recall perform much better in the full set of AST nodes, by performing on average up to 472% and 279% better than the reduced set respectfully. On average, precision is less impacted by the reduction of the AST nodes, where there is minimal difference in the performance of our models built with the two different AST node sets between 0–50 n-grams and then an increase of up to 64% between 50 and 150 AST n-grams significantly associated with faults.

Fig. 13 shows a line plot of the effect of reducing the number of available AST nodes on the performance measure MCC across the systems, where a score of 1 means that the full set is performing better than the reduced set, 0 there is no effect of using the full set or the reduced set and -1 means the reduced set is performing better than the full set. The effect of using the full set over the reduced set is normally greater. In EJDT, T2, AspectJ and SocialSDK for all classifiers the effect on the performance of MCC when using the full set of AST nodes is very large compared to using the reduced set. In some instances, the there is a small effect in the favour of using the reduced set. For example, in GenoViz, logistic regression never goes above zero and stays at an average of -0.15. Overall, the effect on the performance of the model is greater if we use the full set compared to the reduced. Table 14 shows the median effect sizes over the 11 systems for each of the four effect sizes and classifiers. There is on average, a large effect on the change in recall (0.67) and f-measure (0.51), a medium effect on the change in MCC (0.41) and a negligible effect in the change in precision (0.12).

5. Threats to validity

There are threats to validity in the research presented in this paper which fall into the categories of internal, external and construct validity.

5.1. *Internal validity*

An internal threat is the small number of faults reported for both ArgoUML and AspectJ. These small datasets create the problem that it may be difficult to generate statistically significant results. It is always difficult to generate significant results with small sets of data. A small number of faults is typical of many software systems and we feel it is a particular strength that our technique is able to find significant results in the systems despite a small number of faults. The technique we show does find significant results and also finds some of the same significant AST n-grams as those found in the systems with larger numbers of reported faults.

[15] The line plots for the other performance measures are found in the appendix

[16] The difference is the full set minus the reduced set

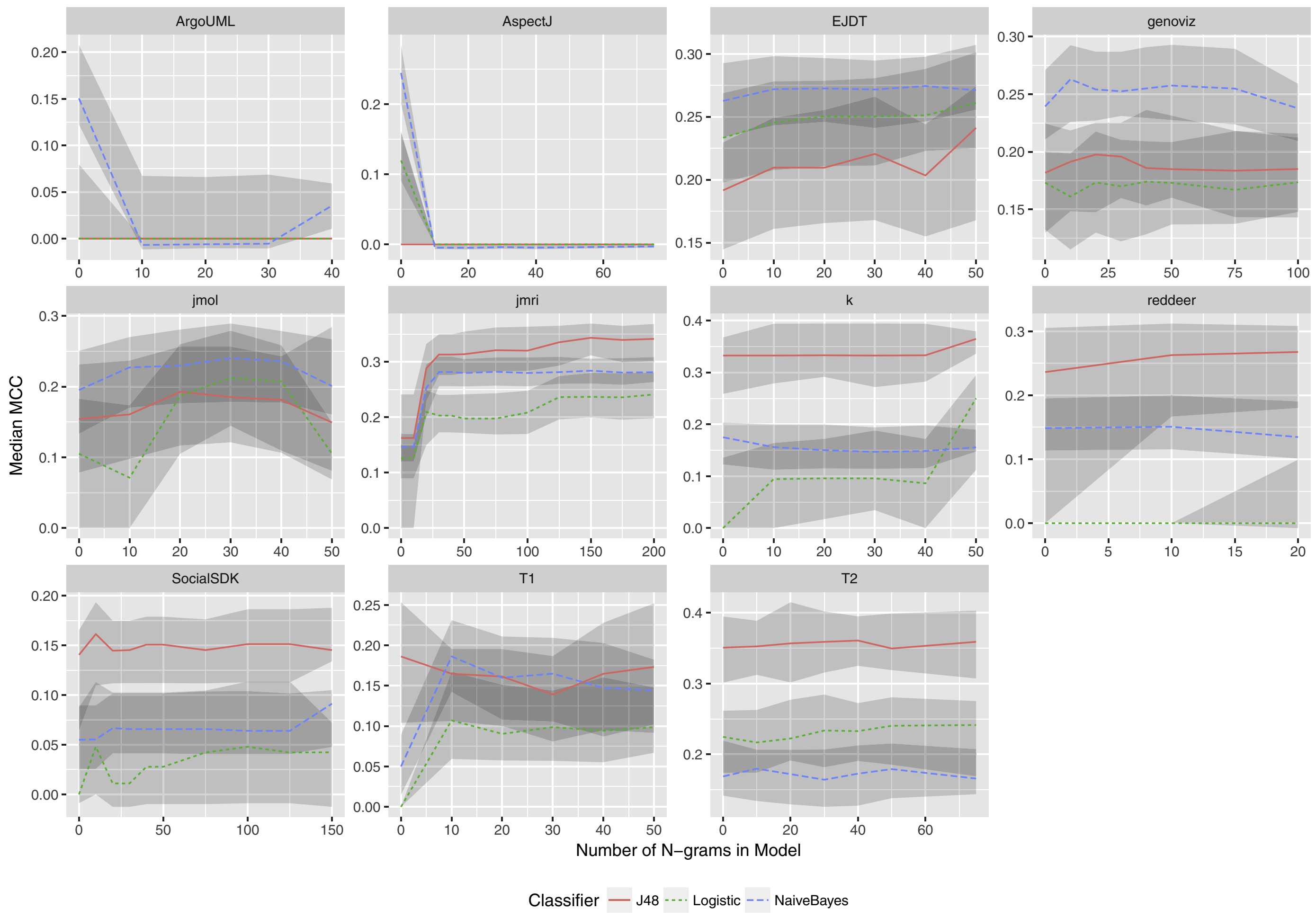

Fig. 11. A line plot to show the change in MCC across all the 11 systems when different levels of N-grams are added to the baseline models when using the reduced Wang et al. [71] AST node set.

Table 13
Median changes between the models built using the full set of AST nodes vs the reduced Wang et al. [71] set (Full set - reduced set).

	Recall		Precision		F-measure		MCC	
N-grams	Change	%	Change	%	Change	%	Change	%
10	0.01	0.00	0.02	0.00	0.02	0.00	0.02	6.62
20	0.02	4.82	0.02	0.00	0.02	5.32	0.03	12.55
30	0.04	20.68	0.03	1.59	0.05	14.17	0.05	21.04
40	0.06	45.80	0.02	0.00	0.06	16.58	0.05	34.24
50	0.10	69.96	-0.00	-0.47	0.07	25.85	0.07	35.20
75	0.08	22.71	0.07	2.76	0.10	18.32	0.10	33.58
100	0.09	194.72	0.05	11.11	0.08	159.20	0.08	91.31
125	0.22	452.86	0.16	66.44	0.27	323.81	0.23	223.04
150	0.20	472.41	0.14	64.02	0.26	259.29	0.20	279.47

Table 14
The average Cliff's D effect score for all the systems between the changes in using the full set and Wang et al. [71]'s reduced set of AST nodes ($d < 0.147$ Negligible, $d < 0.33$ = small, $d < 0.474$ = medium, $d > 0.474$ large [59]).

column	J48	Logistic	Naive Bayes	All
Recall	0.4874	0.6533	0.8727	0.6711
Precision	0.1748	0.0891	0.1016	0.1218
F-measure	0.5141	0.5646	0.4461	0.5082
MCC	0.4948	0.3187	0.4103	0.4080

The processing power available to us meant that we had to limit the maximum length of an AST n-gram. Limiting the length of an AST n-gram to seven means that we have not investigated possible significant AST n-grams that are over this length. There could be frequently occurring longer significant AST n-grams that are significantly associated with faulty code. With greater processing power, longer AST n-grams could be analysed.

The statistical evaluations used could be a threat to the results published in this paper. We have used significance testing which has its critics [20]. We have tried to alleviate the problems that occur with statistical testing by having a large alpha value (0.001) and by using non-parametric tests. Also, we do not say that the AST n-grams are

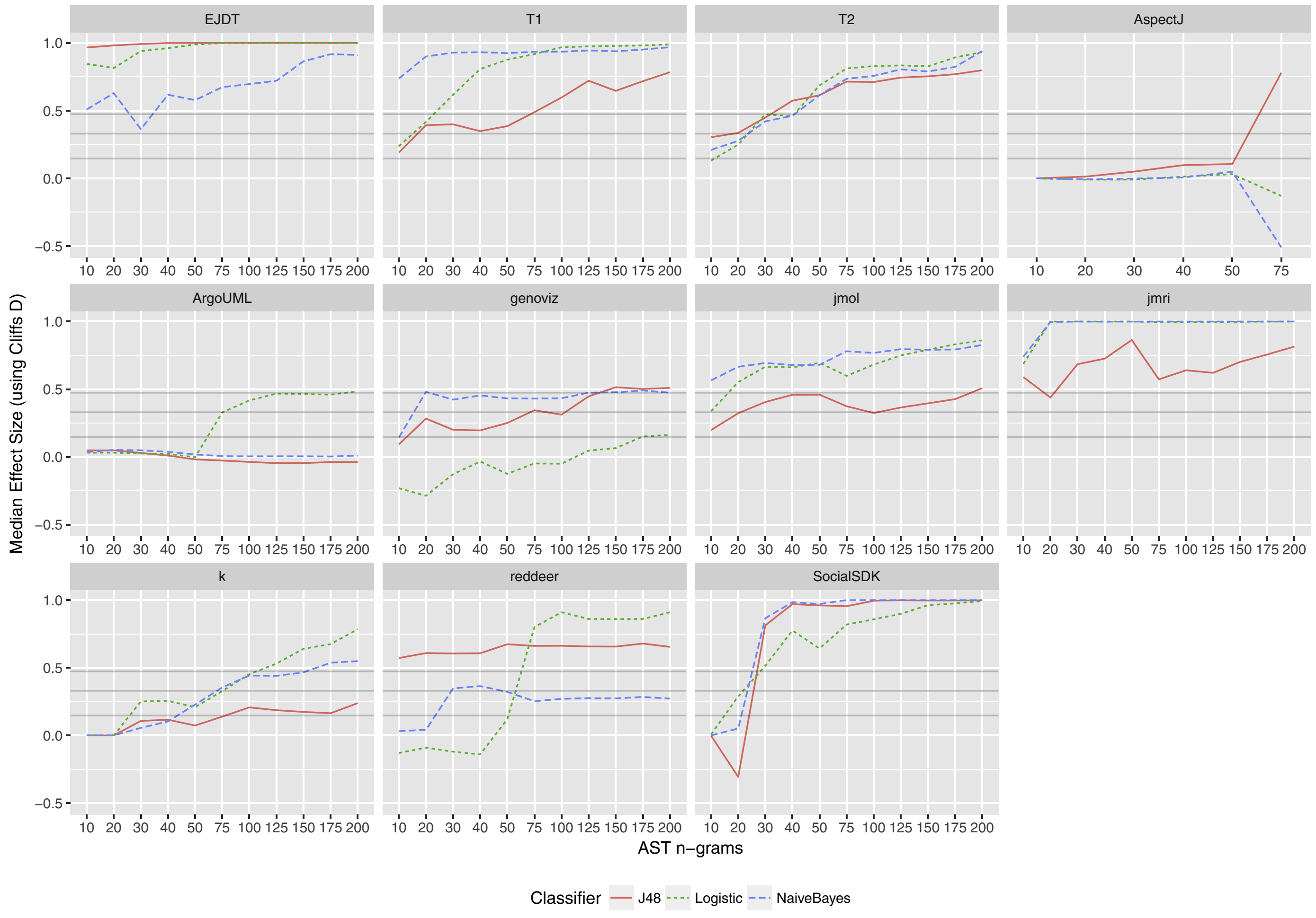

Fig. 12. A line plot to show the level of effect on the performance measure MCC when AST n-grams are added to our models. The three grey straight lines are the effect size indicators (d < 0.147 Negligible, d < 0.33 = small, d < 0.474 = medium, d > 0.474 large [59]).

definitely associated with faults, just that there is evidence to suggest that the faulty AST n-grams appear in a greater number of faults than that could be given by chance. There could be some AST n-grams that have been designated as significantly associated with faults that do not have an overall effect. We have tried to overcome this threat by including the effect sizes of all the AST n-grams with sufficient evidence to reject our null hypothesis. We do not assume that the AST n-gram is the root cause of any fault that has occurred in any of the systems, just that they appear to be in a higher number of faulty methods than could have happened by chance.

We have not controlled for programming confounds which result in different AST n-grams being extracted from the source code. For example, we have not controlled for the developer experience of the developers from either the open source or commercial projects or the coding guidelines put in place by the company. At the communications company, there is a guideline that all methods and classes must be kept as short as possible, this could have impacted on the number of n-grams we could have extracted from their code.

5.2. External validity

An external threat is the number of systems chosen to analyse. The technique shown in this paper may only work for these systems and may not translate to smaller or larger systems. Our technique works on systems with a high or low number of identified faults and both commercial and open source systems. We have shown that the AST n-grams significantly associated with faults in all of the systems can have an impact on the performance of defect prediction models. The technique itself could be easily applied to other Java systems. The commercial systems may not be representative to other open source or commercial systems, however it is very difficult to analyse this as it is extremely difficult to gain access to commercial fault data.

Another external threat is that we have only extracted AST n-grams from the Java programming language. We have not extracted code constructs from other programming languages such as C/C+ + or Python. There may not be AST n-grams significantly associated with faults nor might they help improve software defect prediction performance when added to the baseline models in these languages. The technique could be applied to other programming languages with the use of specific AST parsers and this is future work.

5.3. Construct validity

Repository mining to find faulty code in systems is an inexact science. Latent defects may not have been discovered and may lie dormant in the code base. Faults may also have been fixed but not reported properly in the commit message. SZZ relies on good commit messages and the technique may have not been able to find all the

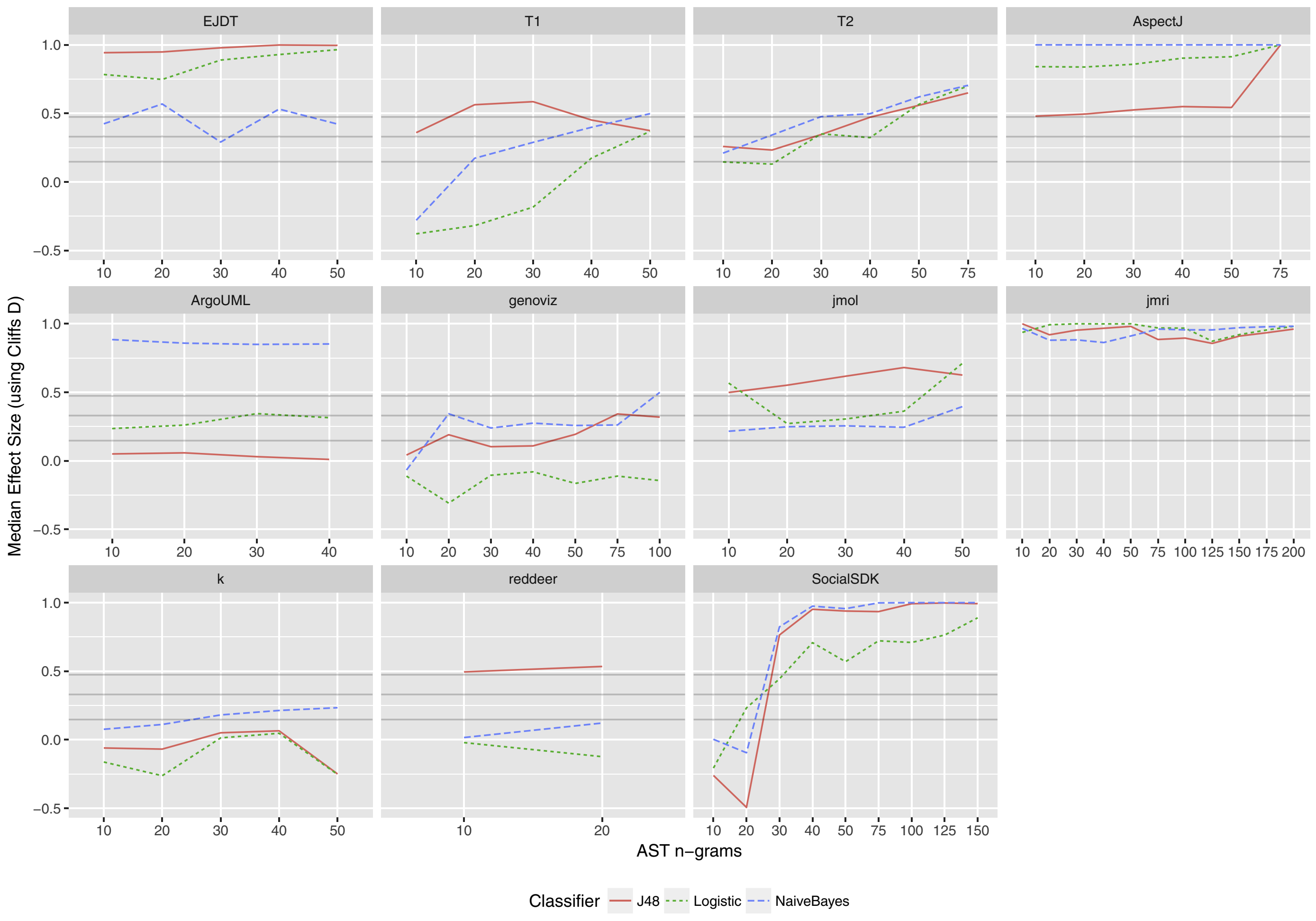

Fig. 13. A line plot to show the level of effect on the change of the performance measure MCC between the full AST node set vs Wang et al. [71]'s reduced set. The three grey straight lines are the effect size indicators (d < 0.147 Negligible, d < 0.33 = small, d < 0.474 = medium, d > 0.474 large [59]).

possible fault fix points. We have performed a manual inspection to investigate if our implementation of SZZ is accurate. As we report, our implementation is able to achieve both high recall and high precision. There also may be changes in a reported fix, that were not actually part of the fix, which could lead to certain methods being misclassified as faulty. This could introduce a lot of noise into our datasets and could lead to inaccurate defect prediction. At present, there is no way of consistently determining in the commit messages which changes were part of the fix and which changes were not.

The use of Git may have meant some of the defect links or insertion points may not have the correct dates. This is because Git is a distributed versioning system. This means that each developer using Git to develop the system has its own local repository that it will control. Defect fixes or insertion points could have been made locally and not pushed to the central repository for a while after these changes have taken place. The insertion point or defect fix would be the point the developer's local repository was committed to the main central repository. Although this will mean that the insertion points and defect fix may have happened sooner, these fixes would not have been available to build the distributed system until they were committed to the central repository. Therefore, for the main build of the system, where a developer commits their local changes to the central repository would be where the defect fix or insertion point is for the main build of system.

The technique described in this paper has focused on fault granularity at method level. This association means that there are AST n-grams within a faulty method, but are not actually the cause of the fault. This could skew the results in favour of AST n-grams that are in frequently occurring methods. The statistical test chosen helps to mitigate this limitation as the AST n-gram has to be in significantly more of the defective methods than it is expected to. Also, an AST n-gram could be missed as it is not located within a method. This AST n-gram could be a code structure outside a method, such as in the fields of a Java class.

6. Related work

There is a huge body of related work in relation to software defect prediction. Various different metrics have been identified and examined to see if they have positive effects on various models. In this section we will compare our work to the less tradition source code metrics, software patterns and also to work being carried out that involves the use of the AST.

6.1. Source code metrics

Some defect prediction studies have used less traditional SCMs, most have been based on analysing the text of the code. One such approach is the use of metrics based on a lexical analysis of text in the code [75]. This lexical analysis uses identifiers as indicators of a method being defective, but has problems due to the differences between spoken languages. Binkley et al. [9], use a language processing based defect prediction measure called a QALP score [9]. The QALP score measures the similarity between a method's comments and its source code using a vector space model. The results showed that the QALP score is helpful in predicting defects in files. Mizuno et al. [50] used spam filtering techniques to create a defect detection technique. This technique treated source code as text files and used the text mining techniques used in spam filtering to identify problematic patterns in the text. The technique was able to classify more than 75% of methods correctly. Abebe et al. [1] improved the detection of defects by using lexicon bad smells (LBS) in conjunction with software structure metrics. Lexicon bad smells are poor quality use of identifier names in software code. Poor quality lexicon for identifiers has been shown to be associated with the introduction of faults [13]. Examples of lexicon bad smells are short terms used as identifiers (e.g abbreviation or acronym) and meaningless terms (e.g. foo and bar). Shivaji et al. [66] go further and investigate code features using a modified version of the bag-of-words approach (BOW+ [63]) to identify a sub-set of code features based on operators such as !=, + + and &&. Overall these previous approaches have shown promise in terms of defect prediction but have not been fully exploited. Our approach goes further and analyses the comprehensive set of low level code features based on the abstract syntax tree.

Complexity and size are code features commonly used in defect prediction [47,48,69]. Despite much effort in identifying and evaluating such features of code, there is no static code feature which consistently identifies problematic code across systems [31,48]. Code features that indicate defects are usually system-specific [80]. Combinations of features have, so far, performed most promisingly in defect prediction. For example, Shivaji et al. [66] used combinations of static code metrics, object oriented metrics, churn metrics and textual features while Bird et al. [11] used combinations of developer contribution network metrics. Unfortunately, collecting data for such combinations is difficult, time consuming and costly. Furthermore, the ability of such combinations to identify defects, relies on the performance of each single feature included in the combination. Therefore, it is important to be able to identify features indicative of defective code and to develop associated code analysis techniques to identify these features.

6.2. Coding patterns

Software patterns in the past have been used to warn programmers against bad coding practices. Fowler and Beck [25] coined the phrase “software code smell” to indicate a pattern in the code that could relate to a deeper problem within a software system. These smells are not indicators of problems on their own, but could point to an underlying problem within the system. Fowler and Beck [25] presented 22 code smells and suggested that they are a hint to decide where and when the software code should be refactored. Some of the ‘code smells’ have been shown to be a problem in software engineering and are a cause of faults within a software system [32]. Code clones have been shown to be sometimes bad [36] or sometimes not so bad [28,43,56,64] at identifying defects in a software system. Zhang et al. [77] investigated all the 22 code smells to see if there was evidence that they actually caused trouble within software systems. They found four papers that investigated the association between code smells and defects. Large classes and long methods have been significantly associated with software faults [76]. This has also been seen in defect prediction studies that have used LOC to show that the greater the LOC, the more chance a defect is to appear. Shotgun surgery was also significantly associated with software faults, but data classes, refused bequest and feature envy were all shown to not be significantly associated with software faults [32,76].

The AST n-gram technique could have a potential advantage over the code smell approach. The AST n-gram technique does not rely on a human definition of a defective code structure and will produce an unbiased list of all the possible coding constructs that have the potential to introduce defects into a software system. Gil and Maman [27] introduced 27 micro-patterns that are based on a variety of programming practices in Java. Destefanis et al. [21] showed that the presence of anti-micro-patterns caused an increment of the fault-proneness in classes in Eclipse. Micro-patterns are based at class level, whilst our AST n-gram approach is based at method level.

6.3. The use of the abstract syntax tree (AST)

The use of the AST has featured in previous work on code analysis. ASTs have been used extensively in code clone detection techniques. A code clone is where two pieces of code in a software system are identical or similar. A lexer breaks down the text of the source code into a sequence of tokens [61] and then these tokens of two pieces of code are compared to find the clones. This technique was used to make two prominent code cloning tools - CCFinder [37] and CP-Miner [62]. The AST is used for clone detection by processing the trees with either tree-matching methods or structural metrics [62]. The AST has been used in many techniques [7,58,70] which have reported good precision and recall [61] showing that this approach is effective at differentiating features of code.

Previous work identifying code constructs have also used ASTs. Nguyen et al. [53] examine changes at the AST level. Allamanis and Sutton [2] mine for code idioms using the AST in combination with nonparametric Bayesian probabilistic tree substitution grammars. The approaches above are different to ours. Although Allamanis and Sutton [2] use AST code snippets, they prune the potential list of AST trees using natural language processing. We did not use this method as we did not want to bias the discovery of potential defective AST n-grams.

Wang et al. [71] used the AST to extract semantic features of source code. They leveraged a deep belief network to automatically learn semantic features from tokens that were extracted from the AST. The authors show that by using semantic features taken from the AST they are able to improve software defect prediction both within project and cross project compared to using the original metric datasets. We do not determine semantic features from the AST, but use the AST to create a sequence of each method in a software system. Wang et al. [71] extracted information from only three different types of nodes from the AST - method/class invocations, declaration nodes and control flow nodes, we have included all possible nodes available in the AST. This was to avoid biasing the discovery of AST n-grams significantly associated with faults. We have performed defect prediction at the method level compared to the file level in Wang et al. [71]’s approach. In addition, our final results show that our approach achieves better defect prediction results when we include a greater number of potential AST nodes.

Pradel and Sen [55] use deep learning to develop a framework for code analysis at the AST level. They particularly investigate the relationship between defects and identifiers and literals in code. Pradel and Sen’s name-based bug detection yields 132 programming mistakes in real-world JavaScript code.

7. Discussion

Our technique identifies all the code features of a software system, creating an impartial list of coding constructs, called AST n-grams, used in the development of five different software systems. Our findings contribute important new information on the nature of the code used in systems and on the faults in that code. Previous work uses only a small number of selected metrics as independent variables to defect prediction models. Our work offers future researchers a comprehensive set of metrics covering all of the features of code used in a system.

7.1. Fault-Prone code constructs

Some fault-prone AST n-grams which appear frequently in code are related to coding constructs previously identified as potentially problematic.

The AST n-grams: METHOD_INVOCATION; MEMBER_SELECT and MEMBER_SELECT; IDENTIFIER are significantly associated with faults. These AST n-grams relate to the message chain code smell. The message chain code smell was identified as a problem by Fowler and Beck [25] and has also been reported to have a small but significant effect on faults [32]. A *message chain* will introduce coupling. High coupling makes the software harder to maintain as changes undertaken in one part of the chain, will impact on other parts of the chain. The other parts of the chain could become defective if they are not modified to reflect the change. The longer the chain, the more changes are needed and therefore there is a higher chance of a defect occurring. Our results show that kinds associated with message chains could effect the chance of that method being defective. Our findings add further evidence confirming the problem of *message chains* within code.

The IDENTIFIER kind occurs in many of the frequently occurring AST n-grams that are significantly associated with fault-prone code. Identifiers are frequently changed and could easily be coded incorrectly. Research into lexicon bad smells has focused on the bad use of identifiers and the language of identifiers within code [1,5,13]. Lexicon bad smells are where developers have used short terms (e.g abbreviation or acronym) or meaningless terms (e.g. foo and bar) as identifiers. Poor quality lexicon for identifiers has been shown to be associated with the introduction of faults [13]. Lexicon bad smells have been able to improve defect prediction when used alongside traditional source code metrics [1]. Arnaoudova et al. [5] found that identifier terms with high entropy had a greater chance of being faulty. Our results suggest that there is a relationship between identifiers in code and defects.

Our results also show that there are many fault-prone coding constructs which have not been identified previously. Our results could explain the current ceiling of defect predictors [48]. Over-dependence on commonly used single metrics such as cyclomatic complexity or using popular object oriented metrics as independent variables in defect prediction will restrict predictive performance to only a sub-set of defects. There are many more coding structures that could be exploited as independent variables in defect prediction. Using many different code features as independent variables in defect prediction has been shown to perform well [31,66]. Our results provide future researchers and practitioners with a potentially powerful new set of independent variables to use in defect prediction.

We also plan to investigate whether the performance of our models using AST n-grams is improved when metrics in addition to basic static code metrics are used. In particular it would be useful to establish whether the addition of churn metrics to the base model improves performance. Churn metrics have been shown to be helpful to the performance of defect prediction models and so we also plan to build our models based only on the analysis of churned code, i.e. only the code involved in changes. Varying the base model on which AST n-grams are added offers further possibilities of increasing predictive performance.

7.2. The effect that AST N-Grams have on the fault-Proneness of methods

Our results identify many AST n-grams that occur frequently in systems and which have a significant effect on fault-proneness. Many of these AST n-grams occur in over a thousand methods. The effect that these AST n-grams have on fault-proneness varies up to over 300% in the case of some regularly occurring EJDT AST n-grams. Focusing defect reduction activities on these AST n-grams is likely to significantly reduce the number of defects in systems.

There are many infrequently occurring AST n-grams which appear to be highly fault-prone in each of the five systems. Some AST n-grams always appear faulty. Where such AST n-grams occur developers must be highly suspicious of these methods. Each of these AST n-grams occurs very rarely (sometimes only in one or two methods). Their effect on overall system defectiveness may be minimal. However the cumulative number of these AST n-grams is likely to have a significant effect on overall system fault-proneness. Identifying these AST n-grams and ensuring code efficacy could prove cost effective for developers.

Our results also suggest possible differences between commercial and open source systems. Different AST n-grams relate to faults differently between systems. In the two commercial systems, we discussed AST n-grams that were always faulty with the developers. The developers described that they were having problems with methods that created anonymous classes within them. This could explain the number of AST n-grams significantly associated with faults containing the kind NEW_CLASS. Our AST n-gram analysis helped confirm to the developers that the problems that they were experiencing were impacting of faults. Whereas in EJDT, not only do AST n-grams that are always faulty appear more often, with one AST n-gram appearing over 200 times and only in faulty methods, but the AST n-grams in EJDT 3.0 appear to be linked to switch statements. Our results suggest that it is very important that the project context of faults is understood as the faulty features of projects are likely to vary.

8. Conclusion

We demonstrate AST n-grams as a promising new approach to identifying defects in code. Our approach is based on a comprehensive analysis of the low level features of Java code via the abstract syntax tree. Our approach allows us to analyse many more low level features of code than conventional fault analysis studies. Previous studies are usually based on analysing a single code feature or a sub-set of code features in relation to faults. Such previous approaches have the limitation that they are only ever able to identify the sub-set of defects that relate to the code features analysed.

We have shown that there are many AST n-grams of Java code that are significantly associated with faulty code. Our AST n-grams range from one to a maximum of seven kinds in length, showing that problematic AST n-grams can be small. The AST n-grams significantly associated with faults tend to be local to specific systems with very few occurring across systems. In terms of answering our research questions:

Research Question 1: Are any AST n-grams significantly associated with faulty code? Our results identify individual AST n-grams that are significantly associated with faults. AST n-grams that we found significantly associated with faults have resonance with findings in previous fault analysis studies. For example we report that the AST n-grams: METHOD_INVOCATION; MEMBER_SELECT and MEMBER_SELECT; IDENTIFIER are significantly associated with faults. These AST n-grams relate to the Message Chain code smell which has been previously reported to have a small but significant effect on faults [32]. Our results show that IDENTIFIER occurs in many of the frequently occurring AST n-grams that are significantly associated with fault-prone code. It seems that it is very easy for developers to get identifiers wrong. This finding, again, relates to previous findings on the textual analysis of code in relation to faults [75]. Our results show that there are hundreds of new unexplored AST n-grams that have not been analysed before in defect prediction.

Research Question 2 - What is the effect size of AST n-grams significantly associated with faulty code? There are many AST n-grams in each of the five systems we investigated which have a very large effect on faults. The AST n-grams which have the largest effect on faults tend to occur infrequently in systems (e.g. in less than 10 times in a system). When these AST n-grams do appear, many appear only in faulty methods, the effect size of such AST n-grams is very large. Individually these AST n-grams are likely to have only a minimal effect on overall system defectiveness. However we also report some frequently occurring AST n-grams that have large effect sizes. Many of these AST n-grams occur over 1000 times and have effect sizes of over 300% making a method three times as likely to contain a defect than one that does not have that particular AST n-gram. Identifying AST n-grams in individual systems with high effect sizes could likely be a particularly cost-effective way for developers to reduce defects in their systems.

Research Question 3: Do Significant AST n-grams help Improve Defect Prediction Performance? The inclusion of the most common AST n-grams significantly associated with faults can help improve the performance of our defect prediction models. For some models, adding AST n-grams significantly associated with faults had a very large effect on the performance of the models. When we added more AST n-grams to the models, the better the defect prediction models performed. However, sometimes this was not the case, and it seems that it was dependent on the significant AST n-grams that were used. For example, for some models we could get some significant increases when we added 20 significant n-grams, but these increases diminished when we added any more. We compared the performance of our models, with models created using Wang et al. [71]'s reduced set. Whether you were using the full set, or the reduced set performance in the defect prediction results can be improved. Our full set, performed on average better than using the smaller set, sometimes with large effects. In small instances, the smaller set of available AST nodes performed better than the full set, but still improved the defect prediction results. Further work is needed to help improve our defect prediction by implementing a better strategy of selecting which AST n-grams to use.

There are various potential uses for AST n-grams which could contribute to the reduction of defects in code. AST n-grams could be used as independent variables in defect prediction models. The best sub-set of AST n-grams could be identified for individual systems. AST n-grams significantly associated with faults could be integrated into an IDE to form warnings to developers. These warnings could identify methods high in fault-prone code structures. The developers could then change the code or make special effort to ensure it is performing correctly. The top 1% of AST n-grams significantly associated with faults could be used as a starting point for such warnings. AST n-gram information could also be used in testing. Most of the AST n-grams significantly associated with faults are located in a relatively small number of methods. Directing test effort to these methods could effectively and efficiently reduce defects.

Future work will also include analysing why the technique works better on some systems than others. Whilst this is not uncommon in defect prediction studies, further work planned involves analysis into why some systems experience larger improvements or decreases in performance. Our results show that the AST n-grams can have a significant improvement on the performance of models that are built with just source code metrics, however other metrics have proven very successful at improving defect prediction performance. Future work will include the comparison to metrics such as process metrics.

The results of our study are important. We provide new evidence and information on coding structures which are good predictive markers for faults. Companies spend large budgets on finding and fixing defects. Using our findings to reliably identify even a few extra defects early in development could save significant resources.

Acknowledgments

This work was partly funded by a grant from the UK's Engineering and Physical Sciences Research Council under grant number: EP/L011751/1. We would like to thank our collaborator for allowing us to use their source code, defect repository and version control systems.

Appendix A. JHawk Metrics Used

Table A1
The JHawk metrics extracted for each method in each of the five systems studied in this paper.

Metric	Description
Cyclomatic Complexity	McCabes cyclomatic Complexity for the method.
Number of Arguments	Number of Arguments
Number of Comments	The number of Comments associated with the method.
Number of Comment Lines	The number of Comment Lines associated with the method.
Variable Declarations	The number of variables declared in the method.
Variable References	The number of variables referenced in the method.
Number of statements	The number of statements in the method.
Number of expressions	The number of expressions in the method.
Max depth of nesting	The maximum depth of nesting in the method.
Halstead length	The Halstead length of the metric.
Halstead vocabulary	The Halstead vocabulary of the method.
Halstead volume	The Halstead volume of the method.
Halstead difficulty	The Halstead difficulty of the method.
Halstead effort	The Halstead effort of the method.
Halstead bugs	The Halstead prediction of the number of bugs in the method.
Total depth of nesting	The total depth of nesting in the method.
Number of casts	The number of class casts in the method.
Number of loops	The number of loops (for, while) in the method.
Number of operators	The total number of operators in the method.
Number of operands	The total number of operands in the method.
Class References	The classes referenced in the method.
External methods	The external methods called by the method.
Local methods	The local methods called by the method.
Exceptions referenced	The exceptions referenced by the method.
Exceptions thrown	The exceptions thrown by the method.
Modifiers	The modifiers (static, public etc) applied to the signature of the method.
Lines of Code	The number of lines of code in the method.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.infsof.2018.10.001.

References

[1] S. Abebe, V. Arnaoudova, P. Tonella, G. Antoniol, Y. Gueheneuc, Can lexicon bad smells improve fault prediction? in: Reverse Engineering (WCRE), 2012 19th Working Conference on, 2012, pp. 235–244.
[2] M. Allamanis, C. Sutton, Mining idioms from source code, in: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, in: FSE 2014, ACM, New York, NY, USA, 2014, pp. 472–483.
[3] E. Arisholm, L.C. Briand, Predicting fault-prone components in a java legacy system, in: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering, ACM, 2006, pp. 8–17.
[4] E. Arisholm, L.C. Briand, E.B. Johannessen, A systematic and comprehensive investigation of methods to build and evaluate fault prediction models, J. Syst. Softw. 83 (1) (2010) 2–17.
[5] V. Arnaoudova, L.M. Eshkevari, R. Oliveto, Y.-G. Guéhéneuc, G. Antoniol, Physical and conceptual identifier dispersion: Measures and relation to fault proneness, in: Proceedings of the International Conference on Software Maintenance (ICSM) - ERA Track, 2010, pp. 1–5.
[6] V.R. Basili, L.C. Briand, W.L. Melo, A validation of object-oriented design metrics as quality indicators, Software Engineering, IEEE Transactions on 22 (10) (1996) 751–761.
[7] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, ICSM, 1998.
[8] R.M. Bell, T.J. Ostrand, E.J. Weyuker, Looking for bugs in all the right places, in: Proceedings of the 2006 International Symposium on Software Testing and Analysis, in: ISSTA '06, ACM, New York, NY, USA, 2006, pp. 61–72.
[9] D. Binkley, H. Feild, D. Lawrie, M. Pighin, Increasing diversity: natural language measures for software fault prediction, J. Syst. Softw. 82 (11) (2009) 1793–1803.

[10] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, P. Devanbu, Fair and balanced?: bias in bug-fix datasets, in: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, in: ESEC/FSE '09, ACM, New York, NY, USA, 2009, pp. 121–130.
[11] C. Bird, N. Nagappan, H. Gall, B. Murphy, P. Devanbu, Putting it all together: Using socio-technical networks to predict failures, in: Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on, IEEE, 2009, pp. 109–119.
[12] D. Bowes, T. Hall, M. Harman, Y. Jia, F. Sarro, F. Wu, Mutation-aware fault prediction, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ACM, 2016, pp. 330–341.
[13] S. Butler, M. Wermelinger, Y. Yu, H. Sharp, Relating identifier naming flaws and code quality: An empirical study, in: Reverse Engineering, 2009. WCRE '09. 16th Working Conference on, 2009, pp. 31–35.
[14] J. Cornfield, A method of estimating comparative rates from clinical data. applications to cancer of the lung, breast, and cervix, J. Natl. Cancer Inst. 11 (6) (1951) 1269–1275.
[15] A.E.C. Cruz, K. Ochimizu, Towards logistic regression models for predicting fault--prone code across software projects, in: Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on, IEEE, 2009, pp. 460–463.
[16] D. Cubranic, G. Murphy, Hipikat: recommending pertinent software development artifacts, in: Software Engineering, 2003. Proceedings. 25th International Conference on, 2003, pp. 408–418.
[17] D.A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, A.E. Hassan, A framework for evaluating the results of the szz approach for identifying bug-introducing changes, IEEE Trans. Softw. Eng. 43 (7) (2017) 641–657.
[18] M. D'Ambros, M. Lanza, R. Robbes, On the relationship between change coupling and software defects, in: Reverse Engineering, 2009. WCRE'09. 16th Working Conference on, IEEE, 2009, pp. 135–144.
[19] M. D'Ambros, M. Lanza, R. Robbes, An extensive comparison of bug prediction approaches, in: Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on, IEEE, 2010, pp. 31–41.
[20] J. Demšar, On the appropriateness of statistical tests in machine learning, Workshop on Evaluation Methods for Machine Learning in conjunction with ICML, 2008.
[21] G. Destefanis, R. Tonelli, G. Concas, M. Marchesi, An analysis of anti-micro-patterns effects on fault-proneness in large java systems, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing, ACM, 2012, pp. 1251–1253.
[22] N.E. Fenton, N. Ohlsson, Quantitative analysis of faults and failures in a complex software system, IEEE Trans. Softw. Eng. 26 (8) (2000) 797–814.
[23] N.E. Fenton, S.L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, 2nd, PWS Publishing Co., Boston, MA, USA, 1998.
[24] M. Fischer, M. Pinzger, H. Gall, Populating a release history database from version control and bug tracking systems, in: Proceedings of the International Conference on Software Maintenance, in: ICSM '03, IEEE Computer Society, Washington, DC, USA, 2003. 23–
[25] M. Fowler, K. Beck, Refactoring: Improving the Design of Existing Code, Addison-Wesley, Boston, MA, USA, 1999.
[26] E. Giger, M. D'Ambros, M. Pinzger, H.C. Gall, Method-level bug prediction, in: Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, in: ESEM '12, ACM, New York, NY, USA, 2012, pp. 171–180.
[27] J.Y. Gil, I. Maman, Micro patterns in java code, in: ACM SIGPLAN Notices, 40, ACM, 2005, pp. 97–116.
[28] N. Göde, R. Koschke, Frequency and risks of changes to clones, in: Proceeding of the 33rd International Conference on Software Engineering, in: ICSE '11, ACM, New York, NY, USA, 2011, pp. 311–320. 196.
[29] D. Gray, D. Bowes, N. Davey, Y. Sun, B. Christianson, The misuse of the nasa metrics data program data sets for automated software defect prediction, in: Evaluation Assessment in Software Engineering (EASE 2011), 15th Annual Conference on, 2011, pp. 96–103.
[30] T. Gyimothy, R. Ferenc, I. Siket, Empirical validation of object-oriented metrics on open source software for fault prediction, Softw. Eng., IEEE Trans. 31 (10) (2005) 897–910. LOC.
[31] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, Softw. Eng., IEEE Trans. 38 (6) (2012) 1276–1304.
[32] T. Hall, M. Zhang, D. Bowes, Y. Sun, Some code smells have a significant but small effect on faults, ACM Trans. Softw. Eng. Methodol. (TOSEM) (2014).
[33] A.E. Hassan, Predicting faults using the complexity of code changes, in: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, 2009, pp. 78–88.
[34] H. Hata, O. Mizuno, T. Kikuno, Bug prediction based on fine-grained module histories, in: Software Engineering (ICSE), 2012 34th International Conference on, IEEE, 2012, pp. 200–210.
[35] IEEE, IEEE Standard classification for software anomalies, IEEE Std 1044–2009 (Revision of IEEE Std 1044–1993) (2010) 1–23.
[36] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, Do code clones matter? ICSE, 2009.
[37] T. Kamiya, S. Kusumoto, K. Inoue, Ccfinder: a multilinguistic token-based code clone detection system for large scale source code, IEEE Trans. Softw. Eng. 28 (2002) 654–670.
[38] T.M. Khoshgoftaar, N. Seliya, Comparative assessment of software quality classification techniques: an empirical case study, Empirical Softw. Engg. 9 (3) (2004) 229–257.
[39] T.M. Khoshgoftaar, K. Gao, N. Seliya, Attribute selection and imbalanced data: Problems in software defect prediction, in: Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on, 1, accept, 2010, pp. 137–144.
[40] S. Kim, T. Zimmermann, K. Pan, E.J.J. Whitehead, Automatic identification of bug-introducing changes, in: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, in: ASE '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 81–90.
[41] S. Kim, T. Zimmermann, E. Whitehead, A. Zeller, Predicting faults from cached history, in: Software Engineering, 2007. ICSE 2007. 29th International Conference on, 2007, pp. 489–498.
[42] S. Kim, H. Zhang, R. Wu, L. Gong, Dealing with noise in defect prediction, in: Software Engineering (ICSE), 2011 33rd International Conference on, 2011, pp. 481–490.
[43] J. Krinke, Is cloned code more stable than non-cloned code? in: SCAM, 2008, pp. 57–66.
[44] T. McCabe, A complexity measure, Softw. Eng., IEEE Trans. 2 (4) (1976) 308–320, doi:10.1109/TSE.1976.233837.
[45] T. Mende, On the evaluation of defect prediction models, The 15th CREST Open Workshop, 2011.
[46] T. Mende, R. Koschke, Revisiting the evaluation of defect prediction models, in: Proceedings of the 5th International Conference on Predictor Models in Software Engineering, ACM, 2009, p. 7.
[47] T. Mende, R. Koschke, Effort-aware defect prediction models, in: Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on, 2010, pp. 107–116.
[48] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, Softw. Eng., IEEE Trans. 33 (1) (2007) 2–13.
[49] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, T. Zimmermann, Local versus global lessons for defect prediction and effort estimation, Softw. Eng., IEEE Trans. 39 (6) (2013) 822–834.
[50] O. Mizuno, S. Ikami, S. Nakaichi, T. Kikuno, Spam filter based approach for finding fault-prone software modules, in: Proceedings of the Fourth International Workshop on Mining Software Repositories, in: MSR '07, IEEE Computer Society, Washington, DC, USA, 2007. 4–
[51] N. Nagappan, T. Ball, Use of relative code churn measures to predict system defect density, in: Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, IEEE, 2005, pp. 284–292.
[52] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, B. Murphy, Change bursts as defect predictors, in: Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on, accept, 2010, pp. 309–318.
[53] H.A. Nguyen, A.T. Nguyen, T.T. Nguyen, T. Nguyen, H. Rajan, A study of repetitiveness of code changes in software evolution, in: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, 2013, pp. 180–190.
[54] L. Pascarella, F. Palomba, A. Bacchelli, Re-evaluating method-level bug prediction, in: 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20–23, 2018, 2018, pp. 592–601, doi:10.1109/SANER.2018.8330264.
[55] M. Pradel, K. Sen, Deep Learning to Find Bugs, Technical Report, TU Darmstadt, 2017.
[56] F. Rahman, C. Bird, P. Devanbu, Clones: What is that smell? in: Proc. 7th IEEE Working Conf. Mining Software Repositories (MSR), 2010, pp. 72–81.
[57] J. Ratzinger, T. Sigmund, H.C. Gall, On the relation of refactorings and software defect prediction, in: Proceedings of the 2008 international working conference on Mining software repositories, ACM, 2008, pp. 35–38.
[58] A. Raza, G. Vogel, E. Plödereder, Bauhaus– a tool suite for program analysis and reverse engineering, in: In Reliable Software Technologies, Ada Europe 2006 (LNCS 4006, 2006, p. 71.
[59] J. Romano, J.D. Kromrey, J. Coraggio, J. Skowronek, Appropriate statistics for ordinal level data: Should we really be using *t*-test and cohen?sd for evaluating group differences on the nsse and other surveys, in: annual meeting of the Florida Association of Institutional Research, 2006, pp. 1–33.
[60] J. Romano, J.D. Kromrey, J. Coraggio, J. Skowronek, L. Devine, Exploring methods for evaluating group differences on the nsse and other surveys: Are the *t*-test and cohen's d indices the most appropriate choices, annual meeting of the Southern Association for Institutional Research, 2006.
[61] C.K. Roy, J.R. Cordy, A survey on software clone detection research, School of Computing TR, Queen's University 115 (2007).
[62] C.K. Roy, J.R. Cordy, R. Koschke, Comparison and evaluation of code clone detection techniques and tools: a qualitative approach, Sci. Comput. Program. 74 (2009) 470–495.
[63] S. Scott, S. Matwin, Feature engineering for text classification, in: ICML, 99, 1999, pp. 379–388.
[64] G. Selim, L. Barbour, W. Shang, B. Adams, A. Hassan, Y. Zou, Studying the impact of clones on software defects, in: Reverse Engineering (WCRE), 2010 17th Working Conference on, 2010, pp. 13–21. 396.
[65] T. Shippey, T. Hall, S. Counsell, D. Bowes, So you need more method level datasets for your software defect prediction?: Voilà!, in: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2016, Ciudad Real, Spain, September 8–9, 2016, 2016, pp. 12:1–12:6, doi:10.1145/2961111.2962620.
[66] S. Shivaji, E.J. Whitehead, R. Akella, K. Sunghun, Reducing features to improve bug prediction, in: Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on, accept, 2009, pp. 600–604.
[67] J. Śliwerski, T. Zimmermann, A. Zeller, When do changes induce fixes? SIGSOFT Softw. Eng. Notes 30 (4) (2005) 1–5.

[68] Q. Song, Z. Jia, M. Shepperd, S. Ying, J. Liu, A general software defect-proneness prediction framework, Softw. Eng., IEEE Trans. 37 (3) (2011) 356–370.

[69] B. Turhan, T. Menzies, A. Bener, J. Di Stefano, On the relative value of cross-company and within-company data for defect prediction, Empirical Softw. Eng. 14 (5) (2009) 540–578.

[70] V. Wahler, D. Seipel, J.W.V. Gudenberg, G. Fischer, Clone detection in source code by frequent itemset techniques, in: In SCAM, 2004, pp. 128–135.

[71] S. Wang, T. Liu, L. Tan, Automatically learning semantic features for defect prediction, in: Proceedings of the 38th International Conference on Software Engineering, in: ICSE '16, ACM, New York, NY, USA, 2016, pp. 297–308, doi:10.1145/2884781.2884804.

[72] A. Wasylkowski, A. Zeller, C. Lindig, Detecting object usage anomalies, in: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ACM, 2007, pp. 35–44.

[73] E.J. Weyuker, T.J. Ostrand, R.M. Bell, Comparing the effectiveness of several modeling methods for fault prediction, Empirical Softw. Eng. 15 (3) (2010) 277–295.

[74] C.C. Williams, J. Spacco, Szz revisited: verifying when changes induce fixes, in: DEFECTS, 2008, pp. 32–36.

[75] A. Zeller, T. Zimmermann, C. Bird, Failure is a four-letter word: a parody in empirical research, PROMISE, 2011.

[76] H. Zhang, An investigation of the relationships between lines of code and defects, in: Software Maintenance, 2009. ICSM 2009. IEEE International Conference on, 2009, pp. 274–283.

[77] M. Zhang, T. Hall, N. Baddoo, Code bad smells: a review of current knowledge, J. Soft. Maint. Evol. 23 (2011).

[78] Y. Zhou, B. Xu, H. Leung, On the ability of complexity metrics to predict fault-prone classes in object-oriented systems, J. Syst. Softw. 83 (4) (2010) 660–674. LOC.

[79] T. Zimmermann, R. Premraj, A. Zeller, Predicting defects for eclipse, in: Proceedings of the Third International Workshop on Predictor Models in Software Engineering, 2007.

[80] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, B. Murphy, Cross-project defect prediction: A large scale experiment on data vs. domain vs. process, in: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, in: ESEC/FSE '09, ACM, New York, NY, USA, 2009, pp. 91–100.

[81] H. Zuse, Software Complexity: Measures and Methods, Walter de Gruyter & Co., Hawthorne, NJ, USA, 1990.