Test Smells for Flaky Test Prediction

Pranay Reddy Juturu
pjuturu@stevens.edu
Stevens Institute of Technology
Hoboken, New Jersey, USA

Ashay Pable
apable@stevens.edu
Stevens Institute of Technology
Hoboken, New Jersey, USA

Diya Sanghvi
dsanghv1@stevens.edu
Stevens Institute of Technology
Hoboken, New Jersey, USA

ABSTRACT

Flaky tests are tests that provide variable results even when run under identical conditions. These tests can be a blocker to the development process since they produce false-positive and false-negative results and increase testing time and cost. To ensure the reliability and correctness of the software testing process, flaky tests must be identified and eliminated. Test smells, on the other hand, signal possible problems with the test code, such as bad design or a lack of maintenance. Researchers have been investigating the use of test smells as a sign of flaky tests in recent years, as they may identify underlying flaws in the test code that can contribute to flakiness. We explore the various sorts of test smells that have been detected, as well as the methodologies used to identify them, as well as methods for predicting flaky tests using test smells. We also assess the efficacy of various approaches and identify gaps in current research that need to be addressed. Our approach emphasizes the potential of using test smells as an early warning system for flaky tests, as well as the need for additional research in this area to improve the accuracy and reliability of flaky test prediction.

CCS CONCEPTS

• **Software Engineering → AI for SE**;

KEYWORDS

software quality, mining software repositories

1 INTRODUCTION

Binary classification is a vital activity in software development since it allows engineers to determine whether or not a program is showing a specific behavior. Regression testing is a crucial step in software development, as it helps to ensure that software is delivered continuously with quality and minimal failures after changes to the production code. During this phase, developers rely on the test results to determine whether a program has a bug resulting from recent changes. However, the presence of flaky tests can make this evaluation unreliable. Flaky tests are a type of test with an intermittent behavior that alternates between passing and failing when executed in the same codebase, without any changes. This non-deterministic behavior frustrates developers, as it makes it challenging to identify and fix the root cause of the problem. Additionally, flaky tests are difficult to debug and can cause delays in the release cycles, halting the development process. Flaky tests can be a significant challenge in software development and identifying them is essential for ensuring the reliability and accuracy of test results. Dynamic and static approaches can be used to identify flaky tests, with each approach having its advantages and disadvantages. Dynamic approaches involve re-executing test cases a fixed number of times, which can be expensive and error-prone. It can also be difficult to determine how many executions are enough to identify flakiness accurately. Static approaches, on the other hand, do not require code re-execution and rely on machine learning methods to predict flakiness likelihood based on various features obtained from the code. Recently, an alternative approach for predicting flaky tests has been proposed based on identifying test smells. Test smells are associated with potential design problems in the test code, and their presence may impact software quality and lead to test flakiness. The alternative approach uses a set of predictors composed only of metrics collected statically, such as the size of the test case, the number of smells in the test code, and binary features related to the presence or absence of 19 test smells. The study found that this approach had better performance than the vocabulary-based model for cross-project prediction, achieving an F-measure of 0.83 with Random Forest. We are extending the proposed solution by considering five new classifiers to predict the flaky tests. The inputs to the model will be the existence of test smells like assertion Roulette, and conditional Test Logic along with a list of some others (19 in total) with test flakiness as a boolean.

- Static approaches do not need the code to be executed again. Models built using static features have many advantages and are less costly.
- Pinto et al. [4]built a set of predictors considering that some patterns within the test code may be employed to identify flaky tests automatically.
- The authors came to the conclusion that the vocabulary-based strategy performs poorly when used across projects because it is context-sensitive and prone to overfitting.
- Considering this result, an alternative approach for flaky test prediction based on test smells is used. Test smells are associated with potential design problems in the test code.
- Test smells are a deviation from how tests should be created, arranged, and interacted with one another. That deviation can indicate issues with test design and negatively impact test performance.
- An open-source test smell detection tool, tsDetect is used. For each test case, this tool requires the identification of the corresponding production code to detect the test smells.

2 RELATED WORK

We have come across some papers that compare various methods to predict flaky tests. First, 'An Evaluation of Machine Learning Methods for Predicting Flaky Tests'[1] utilizes Naive

Study	Year	Method	Category	ML algo	Training Size	Result
An Evaluation of Machine Learning Methods for Predic	2020	ML	Identify flaky tests	Naive-Bayes, Support Vector Machines, and Random Forests	~2500	Precision: >90%, Recall: < 10%
FlakeFlagger: Predicting Flakiness Without Rerunning	2021	Mixed	Identify flaky tests	FlakeFlagger(own model), and a hybrid of vocabulary based approach and flakeflagger	21734	Precision 95%, Recall 74%
DeFlaker: Automatically Detecting Flaky Tests	2018	Traditional	Identify flaky tests	DeFlaker (own software) (used techniques like code coverage)	96 Java Projects	Recall: 95.5%
On the use of test smells for prediction of flaky tests	2021	ML	Identify flaky tests	Random Forest, Decision Tree, KNN, LR, LDA, Perceptron, SVM, Naive Bayes	2854	Precision: 83%, Recall: 83%

Figure 1: Related work review

Bayes, Support Vector Machines (SVM), and Random Forests (RF) models and showed RF performed better when it comes to precision (> 90%) but provided very low recall (< 10%) as compared to NB (i.e., precision < 70% and recall >30%). They extracted test cases from multiple open-source projects and used the dataset. Second, 'FlakeFlagger: Predicting Flakiness Without Rerunning Tests'[2] proposes a FlakeFlagger model that achieved a 60% average precision on 24 Java projects while the highest being 90% on spring-boot and going as low as 0% on others. Other methods compared in the paper include the DeFlake model, a vocabulary-based approach, and a combined approach between the flake-flagger and a vocabulary-based approach that yielded the highest average precision of 66%. One of the limitations in the paper appears to come from the fact that the authors had to supersample the data using SMOTE which some ML researchers are not very accepting of, Besides the 0% accuracy clearly shows that the model may not be useful as a general algorithm. Third, 'DeFlaker: Automatically Detecting Flaky Tests'[3] introduces a software called deflaker, since this is an old paper, they do not use ML and try to use traditional software engineering techniques to solve the problem. However, they manage to achieve good accuracy on a given set of data, which might change if tested on a different dataset. They ran the software on 96 Java projects and claim to have a recall of 95%. This high accuracy gave DeFlaker a place in the task and is used as a base software for comparison to many other papers like the one above mentioned. They utilize a lot of techniques and use the result to analyze and create a list of all flaky tests like AST builder, code coverage recorder, test outcome monitor, etc. All these techniques lead to pretty impressive results on their tested projects. However, such traditional processes may fail on a project out of their set and may not be general solutions to the problem at hand. Fourth, is our base paper 'On the use of test smells for prediction of flaky tests'[5], which utilizes test smells for the prediction of flaky tests with the help of ML models like Random Forest, Decision Tree, KNN, LR, LDA, Perceptron, SVM, Naive Bayes, xgboost, bagging, stack, voting, NN. They claim to have achieved 83% Precision and 83% Recall on a Random Forest algorithm which was the highest they could achieve. Their dataset is extracted data from existing open-source projects.

3 STUDY DESIGN

Our study aims to determine if test smells can be used to predict the existence of flaky tests.

To find the test smells in the code, the authors employed a program called tsDetect[6]. Assertion roulette, conditional test logic, and mystery guest are just a few of the code patterns the program flags as being indicative of test smells. After that, the authors examined the test smells and took pertinent characteristics from the code. They used a feature selection technique known as mutual information to choose the most crucial attributes. The features with the highest mutual information score are chosen using this method, which assesses the dependencies between the features and the target variable (flaky tests). After feature selection, the authors used one-hot encoding to convert the data into a numerical format appropriate for machine learning techniques. We have considered five more classifiers XGBoost, Bagging, Stack, voting, and NN. The list of parameters as input to the model includes[6]:

- Lines of code
- Smell Count
- Assertion Roulette
- Conditional Test Logic
- Constructor Initialization
- Default Test
- Duplicate Assert
- Eager Test
- Empty Test
- General Fixture
- Ignored Test
- Lazy Test
- Magic Number Test
- Mystery Guest
- Print Statement
- Redundant Assertion
- Resource Optimism
- Sensitive Equality
- Sleepy Test
- Unknown Test
- Verbose Test

After studying the data we decided to drop empty columns as well as rows that contain some empty entries for categorical columns because these empty entries do not provide any information to the model. To increase the dataset's quality and guarantee that we are only using pertinent data to train our models, empty columns and rows should be removed. The columns 'Author Email', 'Author Name', 'Committer Email', 'Committer Name', 'Commit Message', 'Commit SHA', 'file path', and 'Line' is removed. These columns either have no bearing on our goal or include categorical items with blank. Once this was done we decided to drop columns with zero relevance to the target column Klass which is Binary making it a binary classification. The columns that have zero relevance to the target columns are 'App', 'Build time in minutes', 'TimeStamp', and 'Version'. We have used random forest, decision trees, Naive Bayes, KNN, LDA, NN, voting, stack, bagging, xgboost, and Logistic Regression models. Grid search with cross-validation is used to adjust the hyperparameters for each model and the model with the best validation set performance is chosen. The random forest provides the best results out of the tested models. We use k-fold cross-validation, where we divide the data into k subsets and use each subset as a validation set in turn, to guarantee that our results are not overfitting the training data. This procedure is repeated k times, and the average performance over all iterations is reported. The evaluation metrics we used are Accuracy, Precision, Recall, F1 score, Matthews correlation coefficient, and Area Under the Curve. The performance of each of the models

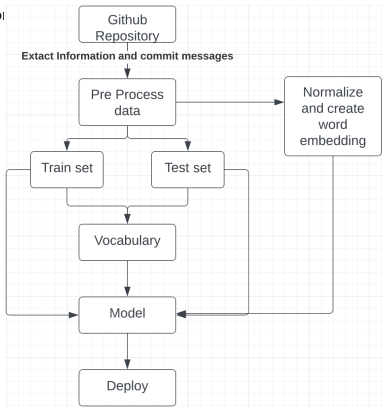

Figure 2: Flowchart

is expressed and compared in the further sections of this paper. From studying the data we also concluded that the dataset only identifies if a test is flaky with a binary representation without the inclusion of any magnitude to provide the extent of flakiness, which might produce some limitations and could be scope for future experimentation. For embedding of text inputs we are using CountVectorizer and TfidfVectorizer from the sklearn.feature_extraction.text class.

Our dataset consists of data coming from a variety of famous GitHub projects and extracting information, like commit messages, analyzing tests, LOC, etc.

Our workflow will consist of information extraction from Github repositories, specifically the commit messages, and analyzing tests to identify test smells. This information is then used to feed data into our classifier once the text from the commit messages is tokenized and embedded. This is then fed into the trained model to classify the data point into flaky or not flaky.

4 EXPERIMENTAL RESULTS

The authors considered seven classifiers in their study. In our experiment, we considered those seven classifiers along with five other classifiers Xgboost, NN, Stack, Bagging, and Voting. We observed that the stack performs similarly to the decision tree in the first scenario. The accuracies range from 79% to 83% with xgboost being the worst performer among the five. In the inter-project scenario, the stack classifier performed better than the previously reported classifier. In the intra-project scenario, nn and xgboost performed better than LR with an accuracy of 77%.

RQ1 – How accurately can we predict test flakiness based on test smells in the test cases?

By first training and then evaluating the classifiers, the prediction model was developed. Every classifier had a reasonable performance archive except the Naive Bayes classifier with an accuracy of 65%. Random Forest classifier had higher accuracy and precision of 83.6%. The collected results demonstrate that test smell-based models, with precision values ranging from 75% to 83%, perform reasonably well in predicting test flakiness. The trained classifiers were tested using the flaky

step	classifier	acc	precision	recall	f1	mcc	auc	VP	FN
training	randomForest	0.836331	0.836912	0.836331	0.836402	0.672862	0.905892	NaN	NaN
training	decisionTree	0.830935	0.831005	0.830935	0.830959	0.661362	0.855146	NaN	NaN
training	naiveBayes	0.652878	0.738687	0.652878	0.610184	0.368766	0.783951	NaN	NaN
training	smo	0.751799	0.752165	0.751799	0.751188	0.502338	0.829965	NaN	NaN
training	knn	0.812950	0.812908	0.812950	0.812918	0.625101	0.812445	NaN	NaN
training	logisticRegression	0.793165	0.793934	0.793165	0.792616	0.585713	0.873613	NaN	NaN
training	perceptron	0.776978	0.777729	0.776978	0.776340	0.553177	0.864558	NaN	NaN
training	lda	0.782374	0.783548	0.782374	0.781608	0.564326	0.861758	NaN	NaN
training	xgboost	0.798561	0.800734	0.798561	0.798613	0.599171	0.897271	NaN	NaN
training	bagging	0.825540	0.825936	0.825540	0.825607	0.651015	0.897479	NaN	NaN
training	stack	0.830935	0.831419	0.830935	0.831005	0.661935	0.913326	NaN	NaN
training	voting	0.827338	0.827513	0.827338	0.827383	0.654315	0.912821	NaN	NaN
training	nn	0.814748	0.819576	0.814748	0.814618	0.634520	0.893415	NaN	N

Figure 3: Training result parameters

step	classifier	acc	precision	recall	f1	mcc	auc	VP	FN
testing-intra-projects	randomForest	0.685714	NaN	0.685714	NaN	NaN	NaN	24	11
testing-intra-projects	decisionTree	0.657143	NaN	0.657143	NaN	NaN	NaN	23	12
testing-intra-projects	naiveBayes	0.571429	NaN	0.571429	NaN	NaN	NaN	20	15
testing-intra-projects	smo	0.657143	NaN	0.657143	NaN	NaN	NaN	23	12
testing-intra-projects	knn	0.514286	NaN	0.514286	NaN	NaN	NaN	18	17
testing-intra-projects	logisticRegression	0.742857	NaN	0.742857	NaN	NaN	NaN	26	9
testing-intra-projects	perceptron	0.714286	NaN	0.714286	NaN	NaN	NaN	25	10
testing-intra-projects	lda	0.657143	NaN	0.657143	NaN	NaN	NaN	23	12
testing-intra-projects	xgboost	0.771429	NaN	0.771429	NaN	NaN	NaN	27	8
testing-intra-projects	bagging	0.685714	NaN	0.685714	NaN	NaN	NaN	24	11
testing-intra-projects	stack	0.714286	NaN	0.714286	NaN	NaN	NaN	25	10
testing-intra-projects	voting	0.714286	NaN	0.714286	NaN	NaN	NaN	25	10
testing-intra-projects	nn	0.771429	NaN	0.771429	NaN	NaN	NaN	27	8

Figure 4: Intra-Project

tests included in the idFlakies dataset to confirm the model's performance in the cross-project environment. The newly added classifiers performed similarly to the existing ones with accuracy ranging from 79% to 83%.

In the intra-project scenario, the performance of all the classifiers is dropped. In a previous study, Logistic regression attained the highest score accurately identifying 26 out of 9 flaky tests. The newly added xgboost and nn classifiers had an accuracy of 77% better than any other classifier. In an inter-project scenario, the classifier's performance declined more sharply. With recall values ranging from 48% to 55%, the classifiers do not differ significantly from one another, with Naive-Bayes reaching a value of 14% by accurately identifying 17 out of 103 flaky tests. In the extended study, we found that the stack classifier has a recall of 56.6%. The results collected demonstrate that the smells can be utilized to predict flakiness. But, in the inter-project scenario, performance suffers significantly. The findings demonstrate that the performance of the smell-based models is equivalent to, and occasionally even superior to vocabulary model. The classifier's performance ranges from 11% to 56%. This led to the conclusion that smells are reliable indicators of flakiness.

RQ2 – Which attributes are the most strongly associated with test flakiness prediction?

To identify associations between attributes and flakiness, we used sklearn.feature_selection.mutual_info_classify method of scikit-learn that allows us to select features and in the future experiment by eliminating the least relevant features to optimize the model. The function calculates Mutual Information, which is the measure of the mutual dependence between two random variables. We use MI over Correlation between each

MSR

step	classifier	acc	precision	recall	f1	mcc	auc	VP	FN
testing-inter-projects	randomForest	0.541667	NaN	0.541667	NaN	NaN	NaN	65	55
testing-inter-projects	decisionTree	0.550000	NaN	0.550000	NaN	NaN	NaN	66	54
testing-inter-projects	naiveBayes	0.141667	NaN	0.141667	NaN	NaN	NaN	17	103
testing-inter-projects	smo	0.550000	NaN	0.550000	NaN	NaN	NaN	66	54
testing-inter-projects	knn	0.508333	NaN	0.508333	NaN	NaN	NaN	61	59
testing-inter-projects	logisticRegression	0.475000	NaN	0.475000	NaN	NaN	NaN	57	63
testing-inter-projects	perceptron	0.475000	NaN	0.475000	NaN	NaN	NaN	57	63
testing-inter-projects	lda	0.475000	NaN	0.475000	NaN	NaN	NaN	57	63
testing-inter-projects	xgboost	0.500000	NaN	0.500000	NaN	NaN	NaN	60	60
testing-inter-projects	bagging	0.500000	NaN	0.500000	NaN	NaN	NaN	60	60
testing-inter-projects	stack	0.566667	NaN	0.566667	NaN	NaN	NaN	68	52
testing-inter-projects	voting	0.533333	NaN	0.533333	NaN	NaN	NaN	64	56
testing-inter-projects	nn	0.541667	NaN	0.541667	NaN	NaN	NaN	65	5

Figure 5: Inter-Project

	position	token	information_gain	total_ocurences	total_flaky_occurences	total_nonflaky_occurences
0	0	loc	2.544574e-01	2777	1377	1400
1	1	assertionRoulette	8.323976e-02	1389	968	421
2	2	smellsCount	2.705301e-02	2655	1356	1299
3	3	sleepyTest	1.948161e-02	112	105	7
4	4	generalFixture	1.600704e-02	267	61	206
5	5	duplicateAssert	1.552284e-02	376	269	107
6	6	constructorInitialization	1.094276e-02	68	63	5
7	7	printStatement	1.056366e-02	58	55	3
8	8	sensitiveEquality	5.852377e-03	129	95	34
9	9	lazyTest	5.490260e-03	1788	817	971
10	10	resourceOptimism	4.252596e-03	75	17	58
11	11	conditionalTestLogic	4.217346e-03	356	219	137
12	12	unknownTest	2.110196e-03	544	234	310
13	13	verboseTest	1.771423e-03	7	7	0
14	14	magicNumberTest	1.108724e-03	411	227	184
15	15	mysteryGuest	5.519103e-04	124	71	53
16	16	eagerTest	2.573414e-04	970	496	474
17	17	redundantAssertion	9.909595e-08	8	4	4
18	18	defaultTest	0.000000e+00	0	0	0
19	19	emptyTest	0.000000e+00	0	0	0
20	20	ignoredTest	0.000000e+00	0	0	0

Figure 6: Features

attribute as it is more versatile and can capture non-linear relationships, while correlation is limited to linear relationships. **The equation for MI between a feature X and a target variable y is:**

$$MI(X, y) = \sum p(x, y) * log(p(x, y)/(p(x) * p(y)))$$

While PCA Is a great Dimensionality tool, its usage as a feature selection is controversial due to the information loss it causes, as well as the reduced flexibility to manually select attributes to eliminate. We identified that the most relevant feature in the dataset was loc (lines of Code) followed by assertion roulette (test smell), while the least relevant feature turned out to be redundant assertion (which is a test smell). Whereas, IgnoredTest, emptyTest, and defaultTest have no relation with an absolute zero score, thus indicating the need to drop the three columns. The dataset only identifies if a test is flaky with a binary representation without the inclusion of any magnitude to provide the extent of flakiness, which might produce some limitations and could be scope for future experimentation.

RQ3 – How does the test smell-based approach compare with the existing vocabulary-based approach?
In the vocabulary-based approach, the values of VP (True Positive) and FN (False Negative) for the classification metrics are

classifier	acc	precision	recall	f1	mcc	auc	VP	FN
randomForest	0.971223	0.971579	0.971223	0.971199	0.942660	0.989448	NaN	NaN
decisionTree	0.928058	0.928324	0.928058	0.928083	0.856181	0.928241	NaN	NaN
naiveBayes	0.951439	0.951548	0.951439	0.951416	0.902767	0.950959	NaN	NaN
smo	0.967626	0.967975	0.967626	0.967599	0.935440	0.992377	NaN	NaN
knn	0.929856	0.930938	0.929856	0.929889	0.860675	0.930736	NaN	NaN
logisticRegression	0.967626	0.967975	0.967626	0.967599	0.935440	0.994179	NaN	NaN
perceptron	0.965827	0.965952	0.965827	0.965811	0.931625	0.991794	NaN	NaN
lda	0.866906	0.871221	0.866906	0.866858	0.738208	0.876601	NaN	NaN

Figure 7: Vocabulary-based approach

NaN (Not a Number). This is due to the dataset used for evaluation in the vocabulary-based approach only containing flaky tests and excluding non-flaky tests. As a result, this dataset lacks True Negatives (TN) and False Negatives (FN), making it impossible to calculate VP and FN.

Precision, recall, and F1-score for the vocabulary-based technique cannot be determined in the absence of TN and FN data. The accuracy and AUC numbers for this technique are the only ones that the authors have reported. We trained the classifiers with the training and testing dataset using the vocabulary-based approach. The vocabulary-based strategy performs better than the smell-based approach: the best F1 score for vocabulary-based models is 97% (Random Forest), while the score for the smell-based approach is 83%. (Random Forest). The disparity is greater when MCC is analyzed. The best outcome for the smell-based technique was 0.66, and the best result for the vocabulary-based approach was 0.94. This score takes into consideration true and false positives, as well as negatives. The cross-project validation results, however, demonstrate that the test smell-based strategy yields superior outcomes. The test smell-based strategy yielded 74% of recall (LR) in the intra-project context, while the vocabulary-based approach only managed to reach 57%. (KNN).

Using the training and testing datasets, the performance of the vocabulary-based models is superior to that of the test smell-based models. Yet, the smell-based approach achieves noticeably higher outcomes in the intra-project and inter-project contexts in the cross-project validation scenario.

5 EXAMPLE

Here we take the example of the Hadoop source code by running our model on it.

The following table shows the confusion matrix of our model on the data.

	Positive	Negative
True	1247	130
False	126	1274

The attributes are generated by TsDetect and here we show the cases of each of the conditions where the model fails and succeeds. We also show the confusion matrix when running on the validation set. Figure 8 shows the test case isOriginalTGTReturnsCorrectValues() belongs to the TestSecurityUtil class of the Hadoop source code. Figures 9 show the case

```java
@Test
public void isOriginalTGTReturnsCorrectValues() {
  assertTrue(SecurityUtil.isTGSPrincipal
      (new KerberosPrincipal("krbtgt/foo@foo")));
  assertTrue(SecurityUtil.isTGSPrincipal
      (new KerberosPrincipal("krbtgt/foo.bar.bat@foo.bar.bat")));
  assertFalse(SecurityUtil.isTGSPrincipal
      (null));
  assertFalse(SecurityUtil.isTGSPrincipal
      (new KerberosPrincipal("blah")));
  assertFalse(SecurityUtil.isTGSPrincipal
      (new KerberosPrincipal("krbtgt/hello")));
  assertFalse(SecurityUtil.isTGSPrincipal
      (new KerberosPrincipal("krbtgt/foo@FOO")));
}
```

Figure 8: False Negative TestCase

```java
/** Corrupt a block and ensure metrics reflects it */
@Test
public void testCorruptBlock() throws Exception {
  // Create a file with single block with two replicas
  final Path file = getTestPath("testCorruptBlock");
  final short replicaCount = 2;
  createFile(file, 100, replicaCount);
  DFSTestUtil.waitForReplication(fs, file, replicaCount, 15000);

  // Disable the heartbeats, so that no corrupted replica
  // can be fixed
  for (DataNode dn : cluster.getDataNodes()) {
    DataNodeTestUtils.setHeartbeatsDisabledForTests(dn, true);
  }

  verifyZeroMetrics();
  verifyAggregatedMetricsTally();
  BlockManagerTestUtil.stopRedundancyThread(bm);
  // Corrupt first replica of the block
  LocatedBlock block = NameNodeAdapter.getBlockLocations(
      cluster.getNameNode(), file.toString(), 0, 1).get(0);
  cluster.getNamesystem().writeLock();
  try {
    bm.findAndMarkBlockAsCorrupt(block.getBlock(), block.getLocations()[0],
        "STORAGE_ID", "TEST");
  } finally {
    cluster.getNamesystem().writeUnlock();
  }

  BlockManagerTestUtil.updateState(bm);
  MetricsRecordBuilder  rb = waitForDnMetricValue(NS_METRICS,
      "CorruptBlocks", 1L, 500);
  // Verify aggregated blocks metrics
  assertGauge("LowRedundancyBlocks", 1L, rb);
  assertGauge("PendingReplicationBlocks", 0L, rb);
  assertGauge("PendingReconstructionBlocks", 0L, rb);
  // Verify replicated blocks metrics
  assertGauge("LowRedundancyReplicatedBlocks", 1L, rb);
  assertGauge("CorruptReplicatedBlocks", 1L, rb);
  assertGauge("HighestPriorityLowRedundancyReplicatedBlocks", 1L, rb);
```

Figure 9: False Positive TestCase

loc	20.0
smellsCount	5.0
assertionRoulette	1.0
conditionalTestLogic	0.0
constructorInitialization	0.0
defaultTest	0.0
duplicateAssert	0.0
eagerTest	1.0
emptyTest	0.0
generalFixture	0.0
ignoredTest	0.0
lazyTest	1.0
magicNumberTest	0.0
mysteryGuest	1.0
printStatement	0.0
redundantAssertion	0.0
resourceOptimism	1.0
sensitiveEquality	0.0
sleepyTest	0.0
unknownTest	0.0
verboseTest	0.0

Figure 10: True negative Parameters

where the model fairly recognized it as a flaky test while it wasn't one belonging to the False Positive part of the population. Figures 10 and 13 show the cases where our model was successfully able to identify if the tests were flaky. Figure 12 shows the case where our model identified the test as flaky successfully.

loc	20.0
smellsCount	5.0
assertionRoulette	1.0
conditionalTestLogic	0.0
constructorInitialization	0.0
defaultTest	0.0
duplicateAssert	0.0
eagerTest	1.0
emptyTest	0.0
generalFixture	0.0
ignoredTest	0.0
lazyTest	1.0
magicNumberTest	0.0
mysteryGuest	1.0
printStatement	0.0
redundantAssertion	0.0
resourceOptimism	1.0
sensitiveEquality	0.0
sleepyTest	0.0
unknownTest	0.0
verboseTest	0.0

Figure 11: True negative TestCase

```java
@Test
public void testShellCommandTimerLeak() throws Exception {
  String quickCommand[] = new String[] {"/bin/sleep", "100"};

  int timersBefore = countTimerThreads();
  System.err.println("before: " + timersBefore);

  for (int i = 0; i < 10; i++) {
    Shell.ShellCommandExecutor shexec = new Shell.ShellCommandExecutor(
        quickCommand, null, null, 1);
    try {
      shexec.execute();
      fail("Bad command should throw exception");
    } catch (Exception e) {
      // expected
    }
  }
  Thread.sleep(1000);
  int timersAfter = countTimerThreads();
  System.err.println("after: " + timersAfter);
  assertEquals(timersBefore, timersAfter);
}
```

Figure 12: True positive TestCase

loc	18.0
smellsCount	4.0
assertionRoulette	1.0
conditionalTestLogic	1.0
constructorInitialization	0.0
defaultTest	0.0
duplicateAssert	0.0
eagerTest	0.0
emptyTest	0.0
generalFixture	0.0
ignoredTest	0.0
lazyTest	1.0
magicNumberTest	0.0
mysteryGuest	0.0
printStatement	0.0
redundantAssertion	0.0
resourceOptimism	0.0
sensitiveEquality	0.0
sleepyTest	1.0
unknownTest	0.0
verboseTest	0.0

Figure 13: True Positive Parameters

6 DISCUSSION

The findings of our study show that using test smells to predict flaky tests can be useful. All of the investigated classifiers (Random Forest, Decision Tree, Logistic Regression, KNN, XGBoost, Bagging, Stack, NN, and Voting) achieved a fair level of accuracy in identifying flaky tests, according to our research. The Random forest classifier performed the best, followed closely by the decision tree, bagging, and stack classifiers, with Voting and KNN coming third and fourth. The naive Bayes classifier performs the worst with an accuracy of just 65%. Furthermore, our findings indicate that the use of test smells is particularly effective in predicting flaky tests for projects in the same domain. In the inter-project scenario, the stack performs better than previously reported logistic regression with an accuracy of 56%. In the intra-project scenario, Xgboost and NN perform better than LR with an accuracy of 77% Intra-project testing was substantially more accurate than inter-project testing, implying that using test smells within a project rather than across various projects may be more helpful in identifying flaky tests. This finding is consistent with prior research that found flakiness to be more prevalent within a project rather

than across multiple projects.

6.1 Implications for Practitioners:

The findings of this study show that among the five additional classifiers stack predict test smells identical to the Decision tree classifier and very close to the best-performing classifier Random forest. As a result, practitioners should think about incorporating any of these classifiers into their test smell prediction tasks. An implication of this study is that when predicting test smells, practitioners should not rely entirely on a single classifier. They could instead explore utilizing numerous classifiers and comparing their predictions to achieve improved accuracy and resilience. The study's findings are valuable for practitioners involved in software testing and quality assurance. Practitioners can increase the accuracy and reliability of their test smell prediction jobs by combining multiple classifiers and exploiting their particular strengths, which can lead to more efficient and successful software development processes.

6.2 Implications for Researchers:

The newly included classifiers, XGBoost, Bagging, Stacking, NN, and Voting, did not outperform the best classifier, Random Forest. This shows that adding more complicated classifiers may not result in a meaningful improvement in the performance of test smell prediction models. As a result, more work could be invested in creating more effective feature selection strategies while also investigating other areas of prediction models such as hyperparameter tuning and ensemble techniques. Additionally, researchers could look into combining different classifiers with the random forest to potentially improve the model's performance. Furthermore, future research could look into the use of deep learning techniques to predict test smells.

6.3 Implications for Tool builders:

The results of this study imply that tool builders should think about testing their tools with several classifiers. Using different classifiers provides for a more thorough examination of the tool's usefulness and can provide additional insights into the tool's performance. Furthermore, tool builders should consider the trade-off between classifier performance and the computational resources required to run them. They should continue to investigate new machine-learning techniques and algorithms. While the new classifiers evaluated in this study performed similarly to the Random Forest, other algorithms may outperform it and provide more accurate test smell predictions. As a result, tool developers must stay up to date on breakthroughs in the field of machine learning and evaluate new techniques as they become available.

7 THREATS TO VALIDITY

Construct Validity: The degree to which a concept is operationalized (i.e., how it is measured) accurately reflects the intended construct is known as its construct validity. The relation between test smells and the flaky test is the construct that interests this study's researchers. To operationalize this concept, a variety of test smells, including the assertion roulette and conditional test logic were considered. There is a potential threat to the identification of the flaky tests.
The most widely used metrics in the machine learning (ML) community were adopted to evaluate the classifiers in order to reduce this threat, which can aid in improving the study's generalizability and reliability. However, the authors utilized a program called tsDetect to find test smells in the code during the pre-processing stage of the test code. The production class, a vital component of the codebase that the tests are testing, has occasionally been missed by tsDetect. As a result, there were instances where test smells could not be extracted from the code, which could jeopardize the study's findings.

Internal Validity: The degree to which a research study accurately ascertains the link between the independent variable and the dependent variable is referred to as internal validity. The existence of confounding variables, which are variables that can impact the relationship between the independent and dependent variables, is one factor that could endanger internal validity. Confounding variables in this study may include things like the size and complexity of the codebase, the level of experience of the developers, and the particular programming language employed. The authors employed statistical techniques like logistic regression and decision trees to account for the effects of these confounding variables and identify the association between test smells and flaky tests in order to address this possible danger.

External Validity: The generalizability of research findings may be constrained by factors that pose a threat to external validity. Four open-source projects were utilized by the authors to gather data for their study, although other software projects in different domains or with various characteristics might not be comparable to these projects. The results may not apply to projects created in other programming languages, for instance, since all of the projects utilized in the study were written in Java. The study's projects were small to medium-sized, and the authors also pointed out that they might not accurately represent the features of larger software projects because of their size. The authors emphasized the limits of their study and suggested that future research should investigate whether or not their findings apply to other software projects and domains in order to address this possible threat.

Conclusion Validity: The degree to which the inferences made from the data accurately reflect the underlying relationships between the variables under investigation is referred to as conclusion validity. The authors of the research thoroughly analyzed their data and assessed how well their classifier models performed using the relevant statistical methods. They also talked about some of the study's possible drawbacks, namely the use of a single dataset and the scant number of projects examined. They did not, however, point out any particular

problems or worries regarding the inferences made from the data.

8 FUTURE WORK

A promising solution to the problem of finding and anticipating flaky tests in software testing is presented by the authors. More study in this area is still needed. The current study focused on a particular set of test smells, but there might be additional clues that might be used to spot and foretell problematic testing. To increase the accuracy of shaky test prediction models, future studies could include other test smells or other elements, such as code complexity or ambient elements. Despite the study's encouraging predictions for flaky tests, it is crucial to comprehend how employing test smells impacts the entire software testing process. Future studies could look into how using test smells affects testing effectiveness, efficiency, and overall software product quality. The tsDetect tool was used in the investigation to find test smells in the codebase. Nonetheless, this tool's accuracy and dependability could yet be enhanced. Future studies could concentrate on creating more advanced tools or enhancing currently available technologies more correctly and effectively detect test scents. The study does not address the fundamental reasons for flakiness, even if it offers a means to recognize and anticipate flaky tests. Future studies could look at the underlying factors that contribute to flaky testing and devise methods to stop them from happening in the first place.

9 CONCLUSION

According to the study, a number of test smells, such as assertion roulette and conditional test logic, were strongly linked to flaky tests. Based on these smells, the classifiers created using machine learning approaches were highly accurate in predicting flaky tests. The accuracy of the test smell detection tool and the small size and scope of the dataset utilized in the study are two major risks to the validity of the study, the authors point out. The authors contend that additional study is required to both explore additional variables that can affect test flakiness in software testing and to confirm the efficacy of test smells as predictors of flaky tests. Overall, the study offers insightful information about the use of test smells as a viable method for enhancing the accuracy and effectiveness of software testing.

REFERENCES

[1] Azeem Ahmad. 2020. An evaluation of machine learning methods for predicting flaky tests. In *27th Asia-Pacific Software Engineering Conference (APSEC 2020) Singapore (virtual), December 1, 2020.*

[2] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1572–1584. https://doi.org/10.1109/ICSE43902.2021.00140 ISSN: 1558-1225.

[3] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 433–444. https://doi.org/10.1145/3180155.3180164

[4] Bruno Camara, Marco Silva, Andre Endo, and Silvia Vergilio. 2021. On the Use of Test Smells for Prediction of Flaky Tests. In *Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing* (Joinville, Brazil) *(SAST '21)*. Association for Computing Machinery, New York, NY, USA, 46–54. https://doi.org/10.1145/3482909.3482916

[5] B. H. P. Camara, M. A. G. Silva, A. T. Endo, and S. R. Vergilio. 2021. On the use of test smells for prediction of flaky tests. In *Brazilian Symposium on Systematic and Automated Software Testing*. 46–54. https://doi.org/10.1145/3482909.3482916 arXiv:2108.11781 [cs].

[6] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. TsDetect: An Open Source Test Smells Detection Tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1650–1654. https://doi.org/10.1145/3368089.3417921