

**UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

RAFAEL RAMPIM SORATTO

**VOCABULÁRIO DE TESTES INSTÁVEIS ENTRE LINGUAGENS DE
PROGRAMAÇÃO**

CAMPO MOURÃO, PR, BRASIL

2025

RAFAEL RAMPIM SORATTO

**VOCABULÁRIO DE TESTES INSTÁVEIS ENTRE LINGUAGENS DE
PROGRAMAÇÃO**

Vocabulary of flaky tests across programming languages

Dissertação de Mestrado apresentado como requisito para obtenção do título de Mestre em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Marco Aurélio Graciotto Silva

CAMPO MOURÃO, PR, BRASIL

2025



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)

Esta licença permite compartilhamento, remixe, adaptação e criação a partir do trabalho, mesmo para fins comerciais, desde que sejam atribuídos créditos ao(s) autor(es). Conteúdos elaborados por terceiros, citados e referenciados nesta obra não são cobertos pela licença.



Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Campo Mourão



RAFAEL RAMPIM SORATTO

VOCABULÁRIO DE TESTES INSTÁVEIS ENTRE LINGUAGENS DE PROGRAMAÇÃO

Trabalho de pesquisa de mestrado apresentado como requisito para obtenção do título de Mestre Em Ciência Da Computação da Universidade Tecnológica Federal do Paraná (UTFPR). Área de concentração: Metodologia E Técnicas Da Computação.

Data de aprovação: 04 de Abril de 2025

Dr. Marco Aurelio Graciotto Silva, Doutorado - Universidade Tecnológica Federal do Paraná

Dr. Andre Takeshi Endo, Doutorado - Universidade Federal de São Carlos (Ufscar)

Dr. Igor Scaliante Wiese, Doutorado - Universidade Tecnológica Federal do Paraná

Documento gerado pelo Sistema Acadêmico da UTFPR a partir dos dados da Ata de Defesa em 16/05/2025.

Dedico este trabalho à minha família, pelos momentos de apoio durante a realização deste trabalho.

AGRADECIMENTOS

Certamente estes parágrafos não irão atender a todas as pessoas que fizeram parte dessa importante fase de minha vida. Portanto, desde já peço desculpas àquelas que não estão presentes entre essas palavras, mas elas podem estar certas que fazem parte do meu pensamento e de minha gratidão.

Agradeço ao meu orientador Prof. Dr. Marco Aurélio Graciotto Silva, pela sabedoria com que me guiou nesta trajetória.

Aos meus colegas de sala.

Aos colaboradores da universidade que contribuíram pelo ambiente no qual o trabalho foi desenvolvido.

Gostaria de deixar registrado também, o meu reconhecimento à minha família, pois acredito que sem o apoio deles seria muito difícil vencer esse desafio.

Enfim, a todos os que por algum motivo contribuíram para a realização desta pesquisa.

Agradeço a Fundação Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela bolsa de fomento à pesquisa.

RESUMO

Contexto: O teste de regressão é uma atividade de verificação e validação de sistemas presente na engenharia de software moderna. Nesta atividade, testes podem falhar sem nenhuma alteração de implementação, caracterizando-se como testes instáveis (*flaky test*). Este tipo de instabilidade pode atrasar o lançamento do software e reduzir a confiança dos testes. Uma forma de identificar tais testes instáveis é pela reexecução dos testes, mas isso possui um custo computacional elevado. Uma alternativa à reexecução é a análise estática do código dos casos de teste, identificando padrões relacionados à instabilidade. Nesse contexto, por enquanto observam-se apenas trabalhos de vocabulário que abordam aplicações de linguagens únicas (Java, Python, Javascript). Para conhecer os limites e possibilidades dessa abordagem, é interessante verificar a intersecção de vocabulários instáveis entre diferentes linguagens.

Objetivo: O objetivo deste trabalho foi avaliar a técnica de vocabulário para predição de testes instáveis de aplicações Javascript, Java e Python. **Método:** Para atingir tal objetivo foi construído um conjunto de dados com casos de teste instáveis presentes em projetos de código aberto no Github que utilizam Javascript, e também foram utilizados conjuntos de dados de trabalhos anteriores para Python e Java. A seguir foram criados modelos de classificação, considerando o vocabulário de instabilidade entre diferentes projetos e linguagens. Após avaliar a precisão dos modelos, verificamos as palavras com maior ganho de informação para predição de instabilidade entre linguagens. **Resultados:** Criamos a ferramenta ShakerJS que conseguiu identificar 102 *flaky tests* em 36% de 36 projetos relevantes em JavaScript utilizando o estresse de CPU e memória. Os termos com maior ganho de informação para predição de *flaky tests* neste conjunto foram: **await, async, if e path**. Em JavaScript, semelhantemente ao Java, o modelo Random Forest (RF) teve a melhor precisão (0,96), enquanto o modelo *Decision Tree* (DT) superou ligeiramente o RF em termos de recall (0,77 vs 0,63). Os resultados são melhores quando o modelo é treinado e testado no mesmo escopo de projeto e linguagem. Alguns projetos chegam a alcançar resultados de 100% de precisão e recall. Verificamos que a qualidade dos modelos em relação às falhas intermitentes totalmente desconhecidas é baixa. Verificamos a baixa qualidade dos modelos em relação às falhas intermitentes de diferentes linguagens de programação: JavaScript, Java e Python. O melhor resultado foi para o modelo treinado na linguagem JavaScript e testado na linguagem Java, que apresentou o F1-Score (F1) de **0.63**. Porém, no que diz respeito ao vocabulário instável entre linguagens os resultados são positivos. Entre as linguagens de programação Python e JavaScript podemos verificar termos relacionados com as seguintes causas raízes de *flaky tests*: comunicação assíncrona (**await, 400, data**), concorrência, eventos de interface gráfica do Usuário (**plot, page, button, click e width**, dependência de Tempo (**time**), vazamentos de Recursos (**source, data, args**). Entre as linguagens Java e JavaScript, podemos verificar a intersecção de termos relacionados a concorrência (**manager**), espera assíncrona (**await**) e vazamentos de Recursos (**new, class**). Entre as linguagens Python e Java, podemos verificar termos relacionados com comunicação assíncrona (**waitFor**), concorrência (**get, select**), dependência de Tempo (**date**), vazamentos de recursos (**evaluate, catch**). **Conclusões:** Este trabalho apresenta resultados relevantes para uma identificação mais eficiente de testes instáveis em projetos que utilizam Javascript, Java e Python. Existe uma intersecção entre contextos e causas raízes de instabilidades de diferentes projetos e linguagens. Porém, utilizar somente o conteúdo puro do código-fonte pode reduzir o desempenho dos modelos de predição em um cenário com diferentes projetos e linguagens.

Palavras-chave: teste de software; teste instável; vocabulário; linguagens de programação; predição.

ABSTRACT

Context: Regression testing is a verification and validation activity of systems present in modern software engineering. In this activity, tests may fail without any implementation changes, characterizing them as unstable tests (*flaky test*). This type of instability can delay software release and reduce confidence in the tests. One way to identify such unstable tests is through test re-execution, but this comes with a high computational cost. An alternative to re-execution is static code analysis of the test cases, identifying patterns related to instability. In this context, so far, only vocabulary-based works focusing on single language applications (Java, Python, JavaScript) have been observed. To understand the limits and possibilities of this approach, it is interesting to check the intersection of unstable vocabularies across different languages. **Objective:** The objective of this work was to evaluate the vocabulary technique for predicting unstable tests in JavaScript, Java, and Python applications. **Method:** To achieve this objective, a dataset was constructed with unstable test cases present in open-source projects on GitHub that utilize JavaScript and the dataset from previous works for Python and Java was used. Classification models were then created, considering the instability vocabulary between different projects and languages. After evaluating the models' accuracy, we examined the words with the highest information gain for predicting instability across languages. **Results:** We created the ShakerJS tool, which managed to identify 102 *flaky tests* in 36% of 36 relevant projects in JavaScript using CPU and memory stress. The terms with the highest information gain for predicting *flaky tests* in this set were: **await**, **async**, **if**, and **path**. In JavaScript, similarly to Java, the RF model had the best accuracy (0.96), while the DT model slightly surpassed RF in terms of recall (0.77 vs 0.63). The results are better when the model is trained and tested within the same project scope and language. Some projects achieve results of 100% accuracy and recall. We noted that the quality of the models concerning completely unknown intermittent failures is low. We observed the low quality of the models regarding intermittent failures across different programming languages: JavaScript, Java, and Python. The best result was for the model trained in JavaScript and tested in Java, which presented an F1 of **0.63**. However, regarding the unstable vocabulary across languages, the results are positive. Between the programming languages Python and JavaScript, we can find terms related to the following root causes of *flaky tests*: asynchronous communication (**await**, **400**, **data**), concurrency, graphical user interface events (**plot**, **page**, **button**, **click**, and **width**), time dependency (**time**), and resource leaks (**source**, **data**, **args**). Between the Java and JavaScript languages, we can verify the intersection of terms related to concurrency (**manager**), asynchronous waiting (**await**), and resource leaks (**new**, **class**). Between the Python and Java languages, we can find terms related to asynchronous communication (**waitFor**), concurrency (**get**, **select**), time dependency (**date**), and resource leaks (**evaluate**, **catch**). **Conclusions:** This work presents relevant results for a more efficient identification of unstable tests in projects that utilize JavaScript, Java, and Python. There is an intersection between contexts and root causes of instabilities from different projects and languages. However, using only the pure content of the source code may reduce the performance of prediction models in a scenario with different projects and languages. **Keywords:** software testing; flaky test; vocabulary; programming languages; prediction.

LISTA DE FIGURAS

Figura 1 – Arquitetura de alto nível do <i>DeFlaker</i> , com três fases: antes, durante e após a execução do teste.	28
Figura 2 – Abordagem FlakeFlagger para prever testes provavelmente instáveis, dado um conjunto <i>flaky tests</i> conhecidos.	33
Figura 3 – Processo de construção de um vocabulário instável utilizando diferentes fontes de dados.	41
Figura 4 – Comparação dos Classificadores em JavaScript	63
Figura 5 – Matriz de confusão do Modelo DT	64
Figura 6 – Matriz de confusão do Modelo K Nearest Neighbors (K-NN)	64
Figura 7 – Matriz de confusão do Modelo RF	64
Figura 8 – Matriz de confusão do Modelo Logistic Regression (LR)	64
Figura 9 – Matriz de confusão do Modelo Naive Bayes (NB)	64
Figura 10 – Matriz de confusão do Modelo Linear Discriminant Analysis (LDA)	64
Figura 11 – Interseção do Vocabulário de Testes entre Linguagens (Java, Python e JavaScript)	78
Figura 12 – Intersecção dos top-200 palavras com maior ganho de informação para predição de instabilidade entre linguagens.	82
Figura 13 – Visualização Treemapping dos termos que se repetem no ranking dos top-200 termos com maior ganho de informação para predição de instabilidades entre linguagens (Java, JavaScript, Python).	83

LISTA DE TABELAS

Tabela 1 – Características medidas por PYTEST-CANNIER	31
Tabela 2 – Características das Técnicas Baseadas em Vocabulário	38
Tabela 3 – Casos de testes em JavaScript rotulados por meio da revisão de commits.	47
Tabela 4 – Tabela de repositórios Github e suas respectivas versões (<i>commit</i>) no qual o <i>rerun</i> foi executado.	49
Tabela 5 – Casos de testes em JavaScript rotulados por meio da reexecução.	50
Tabela 6 – Tokens presentes em Flaky Test (FT)	51
Tabela 7 – Métricas do Classificadores em Projetos JavaScript	59
Tabela 8 – <i>Flaky tests</i> e <i>non-flaky tests</i> obtidos pela revisão e reexecução de projetos JavaScript.	61
Tabela 9 – Resultados dos classificadores em JavaScript	62
Tabela 10 – Resultados de Calibração dos Classificadores	62
Tabela 11 – Ganho de informação das palavras para predição de FT no projeto React	66
Tabela 12 – Ganho de informação das palavras para predição de FT no projeto Ant-Design	67
Tabela 13 – Ganho de informação das palavras para predição de FT no projeto React-Native	68
Tabela 14 – Ganho de informação das palavras para predição de FT no projeto React-Router	69
Tabela 15 – Ganho de informação das palavras para predição de FT no projeto Next.js	70
Tabela 16 – Ganho de informação das palavras para predição de FT no projeto Moleculer	70
Tabela 17 – Palavras com maior Information Gain (IG) para FT obtidos no conjunto de execução de projetos relacionados ao React.	71
Tabela 18 – Ganho de Informação por Token na linguagem JavaScript utilizando dados obtidos pela reexecução.	73
Tabela 19 – Ganho de Informação por Token na linguagem JavaScript utilizando dados mistos (revisão + reexecução).	75
Tabela 20 – Características presentes nos Flaky Tests em JavaScript	76

Tabela 21 – Métricas dos modelos em projetos previamente desconhecidos pelo treinamento.	76
Tabela 22 – Métricas dos modelos entre linguagens distintas.	77
Tabela 23 – Ganho de Informação por palavras na linguagem Java	79
Tabela 24 – Ganho de Informação por palavras na linguagem Python	80
Tabela 25 – Ganho de Informação por Token nas linguagens Java, Python e JavaScript (Rerun + Review)	81
Tabela 26 – Resumo dos resultados de Pinto <i>et al.</i> (2020)	84
Tabela 27 – Resumo dos resultados de Camara <i>et al.</i> (2021b)	84
Tabela 28 – Vocabulário Java em um cenário entre projetos	86
Tabela 29 – Vocabulário Java em um cenário em projetos separados	86
Tabela 30 – Contexto do vocabulário entre linguagens	88

LISTAGEM DE CÓDIGOS FONTE

Listagem 1 – Exemplo de flaky test coletado no GitHub do Projeto Angular (commit: 0cdfa01).	21
Listagem 2 – Exemplo de flaky test com utilização de data e hora coletado no GitHub do Projeto Molecular (commit: 99b2f27).	22
Listagem 3 – Exemplo de utilização do ShakerJS integrado ao GitHub Actions. . .	50

LISTA DE ABREVIATURAS E SIGLAS

ATAF	All Tests Are Flaky
AUC	Area Under The Curve
BoW	Bag-of-Words
CUT	Code Under Test
DT	<i>Decision Tree</i>
F1	F1-Score
FN	Falsos Negativos, do inglês <i>False Negative</i>
FT	Flaky Test
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IG	Information Gain
JSON	JavaScript Object Notation
K-NN	K Nearest Neighbors
LDA	Linear Discriminant Analysis
LR	Logistic Regression
MCC	Matthews Correlation Coefficient
ML	Machine Learning
MSR	Mining Software Repositories
NB	Naive Bayes
NFT	Non Flaky Test
NOD	Non Order Dependent Test
OD	Order Dependent Test
PR	Pull Request

RC	Reliability Curve
RF	Random Forest
SVM	Support Vector Machines
TM	Tree Mapping
TS	Test Smells
UI	User Interface
XML	Extensible Markup Language

SUMÁRIO

1	INTRODUÇÃO	15
2	TRABALHOS RELACIONADOS	19
2.1	Caso de teste instável (flaky test)	19
2.2	Técnicas para identificação de <i>flaky tests</i>	23
2.2.1	Técnicas dinâmicas	23
2.2.2	Técnicas mistas	27
2.2.3	Técnicas estáticas	32
2.3	Vocabulário Instável: uma abordagem estática para prever <i>flaky tests</i>	35
2.4	Considerações Finais	38
3	ABORDAGEM	39
3.1	Conhecimento do domínio	41
3.1.1	Conjuntos de dados sobre <i>flaky tests</i> em Java	42
3.1.2	Conjuntos de dados sobre <i>flaky tests</i> em Python	42
3.1.3	Conjuntos de dados iniciais em JavaScript	43
3.1.4	Construção de conjuntos de dados sobre <i>flaky tests</i> em Javascript	45
3.1.5	Resultado da coleta de <i>flaky tests</i> em Javascript	48
3.2	pré-processamento dos casos de teste	50
3.3	Extração de padrões	51
3.3.1	Modelos de classificação	51
3.4	Pós-processamento	54
3.5	Utilização do conhecimento	55
3.6	Cenários propostos para responder às questões de pesquisa	55
4	RESULTADOS	57
4.1	QP1: Com que qualidade podemos prever instabilidades em casos de teste implementados na linguagem JavaScript utilizando vocabulário?	57
4.1.1	Teste dos modelos com dados de um projeto JavaScript obtidos por meio da execução	58
4.1.2	Teste de modelos com dados obtidos por meio da execução e revisão de casos de testes	60

4.2	QP2: quais são os principais termos presentes em códigos de testes associados à instabilidade na linguagem JavaScript?	65
4.3	QP3: Com que qualidade podemos prever instabilidades em casos de teste entre diferentes projetos e linguagens utilizando vocabulário? . . .	74
4.3.1	Teste de projetos previamente desconhecidos pelo modelo	75
4.3.2	Teste de modelos construídos em linguagens distintas	76
4.4	QP4: Quais são os principais termos presentes em códigos de testes associados à instabilidade entre diferentes linguagens?	77
4.5	Discussões	84
4.5.1	Modelos de Classificação	84
4.5.2	Vocabulários	85
4.6	Ameaças à validade	87
4.6.1	Ameaças à validade interna	88
4.6.2	Ameaças à validade externa	89
5	CONCLUSÃO	91
	REFERÊNCIAS	93

1 INTRODUÇÃO

O teste de regressão é uma atividade importante, responsável por garantir que as funcionalidades do software não sejam danificadas com futuras atualizações (Miranda *et al.*, 2021). Durante o processo de implementação, testes automatizados são implementados e executados, garantindo o funcionamento do software em cada atualização. Porém, nem toda falha apresentada no teste de regressão implica necessariamente em falhas no código de produção (Herzig; Nagappan, 2015). As falhas intermitentes ocorrem com *flaky tests*: que passam e, de repente, falham em diferentes execuções. Eles são chamados de *flaky tests* porque possuem resultados distintos na mesma versão do software (Luo *et al.*, 2014a).

Flaky tests causam um impacto negativo no processo de desenvolvimento do software, reduzindo a confiança no teste de regressão e consumindo recursos dos desenvolvedores para diferenciar falhas de alarmes falsos. Por exemplo, em 2016 foram identificados que 1,5% dos testes nos projetos da Google apresentaram instabilidades (Micco, 2016). Com o avanço dos estudos em 2017, observou-se que essa propriedade de instabilidade (*flakiness*) estava relacionado com o atraso das entregas do produto. Mais especificamente, constatou-se que 16% dos 4,2 milhões de testes implementados na Google falharam sem alterações no código, e que eles consumiram de 2% até 16% de todo o recurso computacional disponível na empresa para execução de testes de regressão (Micco, 2016; Micco, 2017). Logo, é necessário identificar e corrigir os *flaky tests* para reduzir custos desnecessários e aumentar a confiança dos testes.

Diante desses desafios, diversas áreas de pesquisa têm se dedicado a solucionar os problemas associados aos *flaky tests*. São elas: investigação sobre as causas de instabilidades em testes (Eck *et al.*, 2019; Lam *et al.*, 2019; Luo *et al.*, 2014a; Romano *et al.*, 2021; Memon; Cohen, 2013); técnicas para identificação de *flaky tests* (Bell *et al.*, 2018; Zhang *et al.*, 2014; Luo *et al.*, 2014a; Eloussi, 2015; Verdecchia *et al.*, 2021); técnicas para correção de *flaky tests* (Shi *et al.*, 2019; Palomba; Zaidman, 2017).

A abordagem mais comum para se identificar *flaky tests* é com a execução repetidas vezes de cada caso de teste, observando-se a manutenção do resultado da execução em cada execução (*rerun*). Nela, existe certo custo computacional significativo associado a cada execução (Luo *et al.*, 2014a; Harman; O’Hearn, 2018; Bell *et al.*, 2018; Eck *et al.*, 2019). Uma limitação do *rerun* é que não existe um número exato de execuções necessárias para encontrar a falha intermitente nem uma única causa raiz de instabilidade em testes. A causa raiz de instabilidades em testes é o fator que leva a resultados divergentes sem que haja alteração no código-fonte da aplicação (Eloussi, 2015; Romano *et al.*, 2021). Essa causa pode ser diversa e variável, por exemplo: espera assíncrona, interação com a interface do usuário (UI), problema de rede, dependência de data e hora, evento de entrada e saída de dados. Também existe o custo para desenvolvedores realizarem a comparação de diferentes execuções com resultados divergentes para identificar a causa raiz. Isto faz com que as técnicas com execução sejam pouco escaláveis (Pontillo *et al.*, 2022). Do ponto de vista dos desenvolvedores da Microsoft, desco-

brir a instabilidade com o *rerun* não é o melhor cenário por demandar tempo da equipe sem nenhum ganho (Herzig; Nagappan, 2015).

Neste sentido, existem trabalhos interessados em reduzir o custo para identificação de *flaky tests* e a dependência da técnica de *rerun*. São exemplos: a ferramenta DEFLAKER que analisa a diferença de cobertura de código entre commits (Bell *et al.*, 2018); abordagens que utilizam aprendizado de máquina revelaram que o vocabulário dos *flaky tests* pode contribuir para identificação de *flaky tests* (Pinto *et al.*, 2020; Camara *et al.*, 2021b; Haben *et al.*, 2021); abordagens que utilizam métricas de cobertura de código e aprendizado de máquina também apresentaram uma acurácia significativa para identificação das falhas intermitentes (Alshammari *et al.*, 2021).

O custo para identificação de instabilidades em testes é um problema que pode ser tratado com técnicas estáticas. Geralmente essas técnicas utilizam informações do processo de teste de regressão para prever e alertar instabilidades. Se por um lado as abordagens estáticas possuem menor custo e precisão, do outro as técnicas que utilizam *rerun* possuem maior custo e precisão. Existem trabalhos que verificam as lacunas que devem ser preenchidas para consolidar a utilização de técnicas estáticas, aumentar a precisão delas e reduzir a dependência do *rerun* para identificar falhas intermitentes (Pontillo *et al.*, 2022).

Existe um ponto comum entre vários *flaky tests* (com relação a causas raiz) e com as linguagens. Foi realizado um estudo em diferentes linguagens de programação (C, Go, Java, JS e Python), sugerindo que a maioria dos problemas poderia ser explicada por uma pequena fração das causas raízes, e poderia ser corrigido por uma fração relativamente pequena de estratégias de correção (Pontillo *et al.*, 2022). Assim, observa-se que existem pontos em comum nas causas raízes e nas correções entre os idiomas (por exemplo, simultaneidade e espera assíncrona são causas comuns de instabilidades na maioria das linguagens) (Barbosa *et al.*, 2022). Também houve pontos que são específicos da linguagem (por exemplo, a liberação inadequada de recursos é uma causa raiz mais presente na linguagem C). Existem diferentes trabalhos sobre vocabulários sugerindo que os termos relacionados à espera assíncrona também são comumente apresentados em diferentes conjuntos (Pinto *et al.*, 2020; Camara *et al.*, 2021b; Haben *et al.*, 2021; Soratto; Silva, 2023). Neste trabalho verificamos se essa intersecção de causas raízes de *flaky tests* entre diferentes linguagens se estende para os vocabulários. Em caso afirmativo, identificar palavras associadas com instabilidades em testes de diferentes linguagens permite aplicação de técnicas baseadas na análise estática do código em um cenário mais amplo. Consequentemente, pode reduzir custos associados ao *rerun*.

Neste trabalho, verificamos as semelhanças e diferenças entre *flaky tests* de diferentes projetos e linguagens utilizando os termos presentes em cada teste instável. São necessários estudos aprofundados sobre as semelhanças e diferenças da predição de instabilidade entre projetos e linguagens para consolidar o uso de técnicas estáticas durante a identificação de *flaky test*. Para verificar os *flaky tests* de diferentes projetos, este trabalho coletou instabilidades reais de projetos escritos na linguagem JavaScript e construiu modelos de classificação de *flaky*

tests. Comparamos os resultados obtidos na linguagem JavaScript, Java e Python para verificar o cenário entre linguagens.

Este trabalho investiga a qualidade na qual podemos prever instabilidades em casos de teste de diferentes projetos e linguagens utilizando vocabulário; os principais identificadores presentes em códigos de testes associados à instabilidade; a relação entre a linguagem e o vocabulário instável; a relação entre o domínio da aplicação e a técnica de vocabulário. Desta forma, as metas abrangem: criação de um conjunto de dados sobre *flaky tests* na linguagem JavaScript; a criação e avaliação de modelos de classificação considerando o vocabulário dos *flaky tests* entre diferentes projetos da linguagem JavaScript; a criação e avaliação de modelos de classificação considerando o vocabulário dos *flaky tests* em diferentes linguagens; a comparação dos resultados em diferentes cenários.

Utilizamos a abordagem de vocabulário instável para identificar *flaky tests* em casos de teste escritos em diferentes linguagens, similarmente a trabalhos anteriores que trataram de software escrito em Java (Pinto *et al.*, 2020; Camara *et al.*, 2021b) e Python (Haben *et al.*, 2021). Foram obtidos bons resultados para linguagem Javascript em um trabalho anterior, porém, com algumas ameaças à validade relacionadas ao conjunto de treinamento (Soratto, 2022). Essas ameaças à validade indicam que alguns termos são específicos da linguagem ou do contexto do projeto. Neste trabalho incrementamos o conjunto de dados em Javascript proposto anteriormente para obter uma variedade de códigos-fonte referentes à *flaky tests* que foram utilizados para criação do modelo de classificação e vocabulário. Revisamos e executamos o pipeline para construção do vocabulário para as linguagens Java e Python. Em seguida, comparamos os resultados de classificação e os termos obtidos para verificar as semelhanças e diferenças dos vocabulários entre linguagens.

Apresentamos contribuições significativas no campo dos testes automatizados ao desenvolver um conjunto de dados inédito de *flaky tests* e não instáveis em JavaScript, identificado através da execução de mais de 40 projetos relevantes no GitHub. Para isso, desenvolvemos uma versão adaptada da ferramenta Shaker para a linguagem JavaScript, chamada ShakerJS. Esta ferramenta pode ser executada localmente ou no GitHub Actions, e é projetada para estressar a execução dos casos de teste e automatizar a rotulagem de *flaky tests*. Descobrimos diferentes palavras possuem bom ganho de informação entre diferentes projetos baseados no React: *fallback*, *expect*, *await*, *waitfor*, *async*, *refresh*, *html*, *string*, *async*, *inner*, *suspense*, *text*, *loading*, *function*, *return*.

Exploramos o uso de algoritmos de aprendizado de máquina para prever instabilidades por meio dos identificadores de programação. Nossos experimentos indicam que esses modelos alcançam alta precisão e recall dentro de projetos e linguagens específicas. No entanto, ao serem aplicados em projetos de diferentes linguagens, como Java e Python, os resultados variaram consideravelmente, com o melhor desempenho observado ao treinar modelos em JavaScript e testá-los em Java (F1-Score (F1) de 0,63).

Além disso, fornecemos visualizações gráficas do impacto de cada palavra nos *flaky tests* de cada projeto ou linguagem. Essa abordagem facilita a visualização das palavras com maior ganho de informação para predição de instabilidades entre projetos e linguagens. Esses achados sugerem que as palavras específicas do projeto e da linguagem impactam a eficácia dos modelos preditivos no cenário intra e entre projetos. O estudo não apenas avança na identificação de *flaky tests*, mas também aponta direções futuras, como a avaliação de contextos semânticos em códigos e a aplicação desse conhecimento em projetos industriais para aumentar a eficácia dos modelos preditivos.

O restante do trabalho está organizado da seguinte forma. No Capítulo 2 são apresentados os conceitos e trabalhos relacionados a *flaky tests* e técnicas para sua identificação. No Capítulo 3 são apresentadas as técnicas de processamento de texto e identificação de palavras para construção de conjuntos de dados para treino e teste. Tais conjuntos de dados são utilizados por algoritmos de classificação para a criação de um vocabulário *flakiness*. Avaliamos a predição de *flaky tests* em relação ao desempenho dos modelos e palavras dos vocabulários entre diferentes linguagens. Os próximos capítulos deste trabalho são organizados da seguinte forma: No Capítulo 2 apresentamos os trabalhos relacionados com técnicas de identificação e categorização de *flaky tests*. No Capítulo 3 apresentamos a abordagem utilizada para identificação estática de *flaky tests* seguindo uma estrutura de mineração de dados e obtenção de vocabulário instável. No Capítulo 4 são apresentados os experimentos e resultados obtidos. No Capítulo 5 são apresentadas as conclusões e perspectivas.

2 TRABALHOS RELACIONADOS

O *flaky test* representa uma variação no resultado do teste em diferentes execuções e sem variação de implementação, conforme detalhado na Seção 2.1. Para solucionar as falhas intermitentes em testes de software, precisamos ter dois processos bem definidos: a identificação e a correção. A identificação consome recursos computacionais para realizar execuções, já a correção consome recursos humanos para identificar a causa raiz da instabilidade e corrigi-la (Herzig; Nagappan, 2015). No sentido de reduzir esses custos, surgem trabalhos propondo dois novos processos mais leves: a predição e prevenção de instabilidades (Pinto *et al.*, 2020; Camara *et al.*, 2021b; Haben *et al.*, 2021). Este trabalho foca na utilização de abordagens de predição para realizar identificação estática de *flaky tests*.

Embora os *flaky tests* tenham sido inicialmente descobertos de maneira acidental durante os testes de regressão, técnicas específicas foram desenvolvidas para identificar essas instabilidades, com o objetivo de reduzir custos e aumentar a eficácia da detecção. Há três abordagens principais para identificar *flaky tests*: dinâmica, estática e mista. A abordagem dinâmica envolve a reexecução dos testes, enquanto a abordagem estática depende de dados presentes nos testes para prever a instabilidade. A abordagem mista combina as duas técnicas.

Nesta dissertação, o foco é na técnica estática baseada no vocabulário dos casos de teste. Assim, os trabalhos relacionados a essa técnica são descritos na Seção 2.3. A partir das características e limitações observadas, sintetizam-se as considerações finais na Seção 2.4, servindo de base para o restante do estudo desta dissertação.

2.1 Caso de teste instável (*flaky test*)

O *flaky test* representa uma variação no resultado do teste em diferentes execuções e sem variação de implementação (Luo *et al.*, 2014a). Este comportamento não determinístico remove parte da confiança nos testes e é geralmente resultado de um cenário não previsto para a execução. A grande diferença entre uma falha comum e a instável é: a falha comum ocorre após alguma alteração no software, quebrando o valor do teste; enquanto a instabilidade quebra o valor somente algumas vezes durante uma sequência de testes, sem nenhuma alteração no código-fonte (Eloussi, 2015).

Um exemplo simples de um *flaky test* relacionado a hora e tempo é um teste que verifica se um recurso está disponível em um determinado horário. Imagine um teste que verifica se uma API, que apenas funciona das 9 horas às 17 horas, retorna um status de “ok” durante esse período. O teste é configurado para ser executado repetidamente, e em algumas execuções, pode falhar porque o horário atual está fora desse intervalo, retornando um erro ao invés do status “ok”. Se o teste for executado em um momento inesperado, como antes das 9 horas ou após as 17 horas, pode gerar resultados inconsistentes, fazendo com que o teste pareça *flaky*.

Portanto, a mesma condição (o horário certo) pode levar a resultados diferentes dependendo do momento em que o teste é realizado.

Outro cenário em que se observam *flaky tests* é quando o teste utiliza integração com elementos, por exemplo API's externas. Isto acontece pois o resultado do serviço externo pode variar por diversos motivos ou até mesmo estar indisponível.

De fato, existem várias razões para que um teste seja *flaky*, de modo que pode ser assumida a hipótese que todos os testes apresentam algum nível de instabilidade. Este conceito é apresentado com a expressão: All Tests Are Flaky (ATAF) por Harman e O'Hearn (2018). Ele indica que todos os testes são instáveis, porém, cada teste possui seu nível de instabilidade. Este nível de instabilidade é chamado de *flakiness*. No caso de teste exemplo apresentado anteriormente para verificar a disponibilidade de uma API, é possível verificar um alto nível de *flakiness*. Isto ocorre pois o serviço externo pode não retornar o resultado esperado por diversos motivos: falha de conexão, fatores ambientais, falha do servidor, entre outros.

Existem fatores associados ao *flakiness*, são eles: versão da linguagem (e.g. Java), sistema operacional, carga do sistema, ordem de execução, *threads* de cálculos, atrasos de retorno de dados, etc. Todos esses elementos podem variar de certa maneira inesperada, ocasionando um *flaky test*. Alguns destes fatores se manifestam comumente em aplicações Web devido ao gerenciamento de comunicações assíncronas complexas, a simultaneidade entre os clientes-servidores e a heterogeneidade dos recursos empregados. Neste cenário é difícil manter o resultado esperado nos testes devido a problemas como gargalos de rede, problemas de memória ou resolução da tela (Morán *et al.*, 2020). De maneira similar aos testes de aplicações Web, em Graphical User Interface (GUI) os casos de teste são sequências de eventos, em vez de entradas simples. O resultado a ser testado pode ser um estado esperado (ou interface gráfica) para a aplicação após uma sequência de eventos. Isto possibilita a simulação de ações realizadas na interface em uma ordem específica. O estado atual da interface depende da entrada de eventos e da execução de funções assíncronas. Neste cenário a instabilidade pode ocorrer por dois motivos: ordem incorreta de eventos ou atraso na execução de funções de *callback* que retornam dados para interface (Morán *et al.*, 2020).

Para exemplificar este contexto da instabilidade em um caso instável real, selecionamos o Pull Request (PR) (#17935) do projeto Angular¹. O Angular é um projeto comumente utilizado em aplicações Web que utilizam a linguagem Javascript. Analisando o único *commit*² deste PR é encontrada a seguinte mensagem do desenvolvedor: “test: menu tests flaky due to missing animation flush”. Neste *commit* são identificados 2 casos de *flaky tests* utilizando como base a alteração (adição da função *tick*) e a mensagem do desenvolvedor: “Dois testes adicionados recentemente para o menu/mdc-menu e o MDC-one do commit inicial parecem perder um ‘tick’ para liberar a animação aberta. Sem o flush, o ‘FakeAsyncTestZone’ relatará um cronô-

¹ <https://github.com/angular/components/pull/17935>

² <https://github.com/angular/components/pull/17935/commits/0cdfa0125e1627bf113472ab60fa27a1db28f9c2>

metro pendente na conclusão do teste.”. De acordo com a documentação³ do Angular, a função *tick* simula a passagem assíncrona de tempo para os temporizadores na função *fakeAsync* do Angular. A fila de micro-tarefas é drenada no início desta função e após qualquer retorno de funções assíncronas terem sido executadas. O número de milissegundos é recebido por parâmetro e utilizado para avançar o temporizador virtual. Funciona de forma semelhante ao *await* sem a necessidade de esperar o tempo ser executado. Neste exemplo, ele serve para finalizar a alteração gráfica realizada pela função *openMenu()* e garantir que o menu está aberto na assertiva da linha de baixo. Note que esta função é inserida antes do *expect* para garantir que a transição de estado da aplicação tenha finalizado após um avanço de tempo com uma duração específica. Ou seja, se essa duração de tempo variar durante a execução do teste, a assertiva ainda poderá falhar.

Listagem 1 – Exemplo de flaky test coletado no GitHub do Projeto Angular (commit: 0cdfa01).

```

1 it('should focus the menu panel if all items are disabled', fakeAsync(() => {
2   const fixture = createComponent(SimpleMenuWithRepeater, [], [FakeIcon]);
3   fixture.componentInstance.items.forEach(item => item.disabled = true);
4   fixture.detectChanges();
5   fixture.componentInstance.trigger.openMenu();
6   fixture.detectChanges();
7   tick(500);

9   expect(document.activeElement)
10     .toBe(overlayContainerElement.querySelector('.mat-mdc-menu-panel'));
11 }));

```

Fonte: Angular (2019), <https://github.com/angular/components/pull/17935>.

No Código-fonte 2 é apresentado um *flaky test* do projeto *Moleculer*⁴, cuja causa refere-se à serialização de dados referentes a data e hora. Um objeto é criado e um de seus atributos é a data atual. Em sequência, é realizada a serialização do objeto e uma assertiva sobre o tamanho do objeto serializado. No entanto, a depender da hora em que o teste é executado, é inserido um sufixo com o fuso horário no atributo de data serializado, alterando o tamanho do objeto. Consequentemente, ocorre a falha inesperada do caso de teste, ou seja, um *flaky test*.

Conforme os exemplos de *flaky tests* apresentados anteriormente, podemos verificar diferentes fatores que influenciam no resultado do teste e consequentemente no seu grau de instabilidade. Portanto, mais de um fator pode ser responsável por resultados intermitentes em casos de teste. Esses exemplos seguem a ideia de que todos testes são instáveis proposta por Harman e O’Hearn (2018), mesmo que este grau de instabilidade seja maior ou menor em alguns casos. Os estudos de Luo *et al.* (2014a) e Eloussi (2015) categorizam as fontes de não-determinismo em *flaky tests*, identificando as seguintes causas comuns:

- **Comunicação Assíncrona:** Testes que dependem de comunicação assíncrona entre diferentes componentes do sistema podem falhar de forma não-determinística dependendo da ordem de chegada das mensagens ou de atrasos imprevisíveis na rede. A

³ <https://angular.dev/api/core/testing/tick>

⁴ <https://github.com/moleculerjs/moleculer>

Listagem 2 – Exemplo de flaky test com utilização de data e hora coletado no GitHub do Projeto Molecular (commit: 99b2f27).

```

1 it("should serialize the event packet", () => {
2   const now = new Date();
3   const obj = {
4     ver: "4",
5     sender: "node-100",
6     id: "8b3c7371-7f0a-4aa2-b734-70ede29e1bbb",
7     event: "user.created",
8     data: {
9       a: 5,
10      b: "Test",
11      c: now
12    },
13    broadcast: true,
14    meta: {},
15    level: 1,
16    needAck: false
17  };
18  const s = serializer.serialize(cloneDeep(obj), P.PACKET_EVENT);
19  expect(s.length).toBe(150);
20  const res = serializer.deserialize(s, P.PACKET_EVENT);
21  expect(res).not.toBe(obj);
22  expect(res).toEqual(obj);
23 });

```

Fonte: Molecular (2022). Disponível em <https://github.com/molecularjs/molecular/pull/1151/commits/99b2f27>.

disponibilidade e a latência da rede são fatores ambientais diretos que afetam esse tipo de instabilidade.

- **Concorrência:** Testes que envolvem múltiplas threads ou processos concorrentes podem apresentar resultados inconsistentes devido a *race conditions* (corridas de dados) ou a ordens de execução não determinísticas. A carga do sistema e a disponibilidade de recursos de processamento (CPU, memória) são fatores ambientais que podem influenciar a ocorrência de *race conditions*.
- **Dependência da Ordem dos Testes:** A ordem na qual os testes são executados pode afetar os resultados de alguns testes, especialmente aqueles que modificam o estado do sistema. Este fator é independente do código em si, mas dependente da ordem da execução de testes no ambiente.
- **Eventos de Interface Gráfica do Usuário (GUI):** Testes que interagem com interfaces gráficas do usuário podem falhar devido à imprevisibilidade da interação do usuário ou a problemas de temporização. Fatores ambientais como o sistema operacional e os drivers podem influenciar significativamente estes resultados.
- **Dependência de Tempo:** Testes que dependem do tempo podem falhar de forma não-determinística devido a pequenas variações no tempo de execução. O relógio do sistema e a carga do sistema são fatores ambientais que afetam a confiabilidade do tempo em testes.
- **Vazamentos de Recursos:** Testes que não liberam adequadamente os recursos (como memória ou descritores de arquivos) podem apresentar comportamentos erráticos em

execuções subsequentes devido à escassez de recursos. A disponibilidade de memória do sistema e a capacidade do sistema operacional de gerenciamento de recursos são fatores ambientais críticos.

Embora *flaky tests* tenham sido inicialmente observados de forma acidental durante testes de regressão, técnicas específicas para identificação dessas instabilidades foram desenvolvidas, buscando reduzir o custo do *rerun*.

2.2 Técnicas para identificação de *flaky tests*

A identificação de *flaky tests* contribui tanto para indústria, permitindo a descoberta de instabilidades em casos de teste de aplicações em desenvolvimento e em produção, quanto para a academia, permitindo aos pesquisadores identificar futuras instabilidades e criar abordagens variadas para isto. Para identificar um teste instável é necessário utilizar informações sobre o resultado da execução do mesmo com resultados divergentes. Quando as informações de execução apresentam resultados diferentes sem alteração do teste executado podemos rotular o teste como instável. Também é possível, com base em elementos do caso de teste, prever que diferentes execuções possam apresentar resultados diferentes e garantir que um teste é instável mesmo sem executar ele. Mesmo que essa predição não seja possível em todos os *flaky tests*, é possível extrair informações de instabilidades já obtidas pela reexecução para prever novas possíveis instabilidades. Portanto, para identificar um *flaky tests* é necessário garantir que o resultado de sua execução pode variar. Existem diversas técnicas para identificação de *flaky tests* (Herzig; Nagappan, 2015; Eloussi, 2015; Romano *et al.*, 2021; Bell *et al.*, 2018; Pinto *et al.*, 2020; Camara *et al.*, 2021b; Camara *et al.*, 2021a; Morán *et al.*, 2020; Cordeiro *et al.*, 2021a; Soratto; Silva, 2023).

Chamaremos de ‘técnicas dinâmicas’ todas as que utilizam a reexecução dos casos de teste; de ‘técnicas estáticas’ todas as que não utilizam a reexecução; e de ‘técnicas mistas’ todas as que reduzem o número de execuções necessárias para identificação do *flaky* utilizando técnicas estáticas.

2.2.1 Técnicas dinâmicas

Uma técnica muito utilizada para identificar *flaky tests* é a reexecução. Ela consiste na repetição da execução do conjunto de casos de teste de uma aplicação. Dessa forma, essa técnica é similar ao que ocorre nos testes de regressão, que inicialmente motivaram a investigação sobre *flaky tests*. A reexecução permite o estudo sobre a correlação (temporal e espacial) em falhas de uma mesma suíte de testes. Por exemplo, vários testes podem depender de algum serviço de rede e quando um teste falha (se o serviço for desativado), todos os outros testes provavelmente também falharão. No entanto, essa técnica de reexecução é pouco eficiente, exigindo uma quan-

tidade elevada de execuções para aumentar a chance de identificar um caso de teste instável (Eloussi, 2015).

No trabalho de Eloussi (2015) foram propostas algumas abordagens para identificação de *flaky tests* com *rerun*:

- **Adiamento de execuções:** Aumentar o tempo para reexecução da suíte de testes pode aumentar a precisão da identificação de *flaky tests*, particularmente para *flaky tests* “em sequência” (testes que tendem a falhar em grupos).
- **Reexecução em uma nova JVM:** Executar testes falhos em uma nova Máquina Virtual Java (JVM) também pode melhorar a precisão da identificação, pois alguns *flaky tests* dependem do estado da JVM. Uma nova JVM fornece um ambiente inicial limpo, reduzindo a probabilidade de falhas falsas.
- **Intersecção de cobertura:** Esta técnica analisa se a cobertura de código do teste falho se cruza com o código alterado recentemente. Se não houver intersecção, a falha é mais provável devido ao próprio teste instável do que a uma modificação recente no código.

Essas melhorias foram avaliadas usando dados de 15 projetos da linguagem Java e 2.715 classes de teste, algumas das quais continham *flaky tests* conhecidos (Eloussi, 2015). Os resultados sugerem que esses aprimoramentos são eficazes para melhorar a precisão da detecção de *flaky tests*, com um custo computacional semelhante ou ligeiramente superior à técnica padrão de reexecução imediata. O aumento de custo é frequentemente compensado pelo tempo reduzido gasto na depuração de falhas de teste não instáveis. Os resultados mostram que as melhorias propostas são aplicáveis, e pode identificar mais *flaky tests* do que a abordagem de executar o teste novamente após uma falha intermitente. Assim, notou-se que a abordagem de *rerun* pode ser adaptada com técnicas variadas para encontrar um número maior de *flaky tests* em projetos.

A técnica proposta por Morán *et al.* (2020), chamada *FlakyLoc*, tem como objetivo localizar automaticamente a causa raiz da instabilidade nos testes de aplicações Web. Essa técnica se baseia na caracterização dos fatores ambientais que influenciam a execução dos testes. São discutidos os desafios enfrentados pelos desenvolvedores ao lidar com *flaky tests* e a dificuldade de identificar e corrigir suas causas. Os ambientes de execução e os dados coletados foram cuidadosamente caracterizados para identificar os fatores ambientais que podem causar *flakiness* (instabilidade) em testes de aplicações web. A técnica se baseia na execução de casos de teste sob diferentes configurações, que incluem:

- **Memória:** Problemas de memória podem ocorrer, especialmente quando várias sessões e navegadores não são fechados corretamente, consumindo a mesma memória;
- **Resolução de Tela:** A resolução da tela é um fator crítico; por exemplo, uma resolução de 800x600 foi identificada como a mais suspeita em relação à *flakiness*;
- **Navegador:** O uso de diferentes navegadores, como Chrome, também pode influenciar a estabilidade dos testes.

- **CPU:** A quantidade de núcleos de CPU disponíveis durante a execução dos testes pode afetar a concorrência e, conseqüentemente, as instabilidades.

A técnica FlakyLoc realiza a reexecução em várias configurações geradas utilizando uma abordagem combinatória, permitindo a identificação de quais combinações de fatores ambientais estavam associadas a falhas. Ela identifica fatores ambientais que podem causar falhas intermitentes em testes web, como a resolução da tela, a versão do navegador e o tráfego de rede, que introduzem instabilidades nos resultados dos testes. A técnica também utiliza uma abordagem baseada em espectro para analisar as execuções dos testes e determinar quais fatores estão associados às falhas observadas. Por exemplo, algumas combinações de configurações com largura de banda de 400KB/s causaram falhas, enquanto outras com a mesma largura de banda mascararam a *flakiness*. Portanto, existem combinações de fatores que podem facilitar a identificação de *flaky tests*. Após identificar a instabilidade, a técnica ainda permite a detecção automática da causa raiz da falha, fornecendo informações valiosas para mitigar e corrigir esses problemas em aplicações reais (Morán *et al.*, 2020).

Entre os vários tipos de *flaky tests*, destacam-se os testes dependentes da ordem. O iD-Flakies foi proposto por Lam *et al.* (2019) para detecção e classificação parcial de *flaky tests*. Para isto, foi desenvolvido um conjunto de dados abrangendo *flaky tests* de projetos de código aberto e um estudo baseado neste conjunto de dados. O iDFlakies automatiza experimentos com projetos Java baseados em Maven. Com essa ferramenta, foi obtido um conjunto de dados contendo 422 *flaky tests*, dos quais 50,5% são dependentes de ordem. Neste estudo, os autores verificaram a prevalência de dois tipos de *flaky tests*: os dependentes de ordem e não.

O IdFlakies possui um plugin Maven para detectar *flaky tests*, classificando-os como dependentes de ordem (OD) ou não dependentes de ordem (NOD). A ferramenta utiliza como entradas uma suíte de testes, uma configuração para ordenar os testes, e o número de execuções da suíte de testes baseado na configuração. A configuração padrão adota a ordenação de métodos de classe aleatórios em 20 rodadas, sendo a que detectou o maior número de *flaky tests* em avaliações. O plugin pode ser integrado a projetos que utilizam Maven para construção e JUnit para execução de testes. Projetos Maven são organizados em módulos, cada qual com seu próprio código e conjunto de testes. Esta ferramenta utiliza um executor de testes personalizado para controlar a ordem de execução dos métodos de teste do JUnit. O processo da ferramenta é dividido em três etapas principais: configuração, execução e classificação. Na etapa de configuração, a ferramenta verifica se todos os testes em um módulo passam. Se algum teste falha, a exploração não prossegue para aquele módulo. Módulos que passam por todos os testes avançam para a execução da suíte de testes conforme a configuração especificada e o número de rodadas. Falhas detectadas em alguma rodada levam à classificação dos testes envolvidos como Order Dependent Test (OD) e Non Order Dependent Test (NOD) (Lam *et al.*, 2019).

flaky tests que dependem da execução prévia de outros casos de teste na ordem de execução do teste são conhecidos como dependentes da ordem. Outro termo para esses *flaky tests*

é ‘vítima’, dado que o caso de teste é instável devido a consequência da execução de casos de teste anteriores (que ‘poluem’ o sistema), afetando seu resultado. A técnica iDFlakies pode detectar *flaky tests* ‘vítimas’, mas não consegue identificar seus poluidores associados. Lam *et al.* (2019) propuseram uma técnica que pode detectar um subconjunto das vítimas de uma suíte de testes e seus poluidores, embora tenha sido projetada principalmente para detectar vítimas.

Embora a técnica IdFlakies possa detectar *flaky tests* vítimas, ela não pode identificar seus poluidores associados. Zhang *et al.* (2014) propuseram uma técnica que pode detectar um subconjunto das vítimas de uma suíte de testes e seus poluidores (embora os autores tenham projetado a técnica principalmente para detectar vítimas). A técnica envolve a execução de todas as permissões de casos de teste de comprimento dois (todas as combinações em ambas as ordens) isoladamente, como em processos separados da Java Virtual Machine ou do interpretador Python. O IdFlakies possui vários parâmetros: o número de *reruns* durante a fase de Configuração, o número de *reruns* durante a fase de Execução, o método de geração das ordens de execução de testes modificadas durante a fase de Execução (por exemplo, embaralhamento) e a porcentagem de falhas adicionais a serem verificadas novamente na fase de Classificação. Dependendo da escolha dos valores para esses parâmetros, o IdFlakies pode exigir um número significativo de execuções de testes e, portanto, impor um custo de tempo proibitivo. Existe uma técnica referida como Pairwise para solucionar o problema. Essa técnica envolve executar todas as permutações de casos de teste de comprimento dois (cada par em ambas as ordens) de forma isolada, como em processos separados de Máquina Virtual Java ou interpretadores Python (Parry *et al.*, 2023).

Inicialmente, Pairwise requer um resultado esperado para cada caso de teste. Esses resultados podem ser obtidos executando cada caso de teste isoladamente para observar seus desfechos sem os possíveis efeitos colaterais de outros casos de teste. Com um resultado esperado para cada caso de teste, Pairwise executa cada permutação de dois casos de teste, garantindo que cada caso tenha a oportunidade de ser tanto o primeiro quanto o segundo a ser executado no par: o candidato a poluidor e a vítima, respectivamente. Para resultados mais confiáveis, Pairwise deve filtrar quaisquer pares onde o candidato a vítima seja um *flaky test* NOD conhecido, por não possuírem um resultado esperado confiável. Para um dado par, se o segundo teste produzir um resultado diferente do esperado, Pairwise o classifica como vítima e classifica o primeiro teste como um de seus poluidores (Parry *et al.*, 2023).

Trabalhos anteriores determinaram que uma dependência de ordem pode envolver mais de dois casos de teste (Shi *et al.* (2019)), embora o estudo empírico de Zhang *et al.* (2014) tenha encontrado que 76% das dependências de ordem envolvem apenas dois casos. Considerando apenas pares de casos de teste, a complexidade de tempo do Pairwise já é quadrática em relação ao tamanho da suíte de testes e, portanto, muito dispendiosa. Considerar permutações mais longas tornaria rapidamente a técnica impraticável (Parry *et al.*, 2023).

O *Shaker* é uma ferramenta projetada para detectar *flaky tests* de forma mais eficiente do que o *rerun* convencional, introduzindo ruído no ambiente de teste (Silva *et al.*, 2020; Cordeiro

et al., 2021b; Silva, 2022). Duas avaliações foram conduzidas: uma avaliação de protótipo utilizando 11 aplicativos Android; e uma avaliação da integração com o GitHub Actions, usando 11 projetos Java (Maven) com mais de 1000 testes ou 1000 estrelas no GitHub. É adicionado ruído ao ambiente de execução utilizando a ferramenta ‘stress-ng’ (King *et al.*, 2013) para criar carga de CPU ou memória durante a execução dos testes. Essa abordagem baseia-se na hipótese de que problemas de concorrência frequentemente causam *flaky tests*. A avaliação da integração mostrou que a adição de ruído aumentou significativamente a probabilidade de detecção de *flaky tests* (1,13 a 6,25 vezes mais que o ReRun padrão). Embora o tempo de execução tenha aumentado em alguns casos, isso foi atribuído à natureza paralela da avaliação e aos processos de estresse concorrentes. Atualmente, o *Shaker* suporta principalmente projetos Java (Maven) e Python (pytest). As configurações de estresse são predefinidas e podem não ser ótimas para todos os projetos ou contextos. O *Shaker* apresenta uma abordagem para detectar *flaky tests*, especialmente aqueles causados por problemas de concorrência. Os resultados demonstram sua superioridade em relação ao *rerun* convencional em termos de eficiência e completude. No entanto, mais pesquisas são necessárias para avaliar sua generalizabilidade em diferentes contextos e plataformas (Cordeiro *et al.*, 2021b).

2.2.2 Técnicas mistas

O *DeFlaker* analisa a diferença entre a cobertura dos casos de teste entre duas versões do código-fonte: a atual e a anterior. Através desta análise estática é obtida uma lista de mudanças na cobertura dos testes. Logo após, os testes da versão atual do código-fonte são executados, gerando relatórios de cobertura de cada caso de teste. Caso exista alguma falha na reexecução, a instabilidade é identificada utilizando a análise estática gerada anteriormente. Essa abordagem permite diferenciar falhas de falhas intermitentes com apenas uma execução, utilizando informações estáticas obtidas previamente (Bell *et al.*, 2018).

As três fases de processamento do *DeFlaker* são apresentadas na Figura 1. Nesta imagem, podemos verificar que a entrada de dados é um repositório de software sob controle de versão (por exemplo, um repositório git). Com base nesta entrada, é possível obter a versão do código-fonte atual e a anterior. A ferramenta chamada de *DeFlaker Coverage Analyzer* é responsável por verificar a diferença de cobertura das duas versões antes de realizar a execução dos testes. Logo em seguida, é realizada a reexecução para coleta e instrumentação da cobertura. Com esse diferencial de cobertura entre duas versões em conjunto da cobertura obtida após as execuções, o *DeFlaker Reporter* gera a saída contendo uma lista de *flaky tests*.

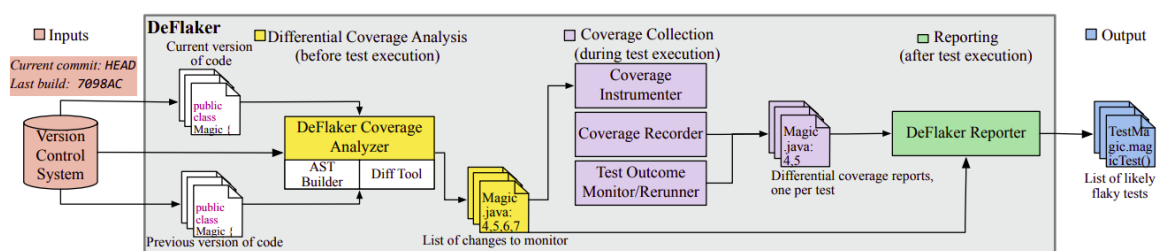
Bell *et al.* (2018) coletaram um total de 96 projetos do GitHub que utilizam TravisCI, têm Java como linguagem principal e utilizam Maven. Escolheram os projetos que tinham pelo menos um teste *flaky* (e uma gama de commits que apresentaram esse comportamento *flaky*). Obtiveram 96 testes *flaky* conhecidos de 26 projetos, consistindo em: (1) 4 projetos (achilles, checkstyle, jackrabbit-oak e toggglz) de um experimento ao vivo nos quais identificaram 5 testes

flaky junto com os *commits* de início e fim que exibiam esse comportamento *flaky*; e (2) 22 outros projetos de código aberto nos quais os desenvolvedores do projeto anteriormente encontraram e corrigiram um total de 91 testes *flaky*. Os 22 projetos com testes *flaky* previamente corrigidos contêm 17 obtidos consultando o GitHub por termos relacionados a testes *flaky* (“in-termit” ou “flak”) e 5 projetos selecionados de um estudo anterior sobre testes *flaky*. A partir dos resultados da consulta ao GitHub, selecionaram 81 testes (de 17 projetos) onde confirmou-se por inspeção manual que a mensagem de *commit* realmente corrigiu um teste *flaky*, e para os quais ainda poderia executar esses testes antigos. Também consultaram um estudo anterior sobre testes *flaky*, selecionando dele 10 testes *flaky* (de 5 projetos) que foram compilados e executados (Bell *et al.*, 2018).

Os *flaky tests* e projetos vêm de várias categorias de domínios (redes, bancos de dados, etc.), oferecendo uma amostra relativamente diversificada de testes *flaky*. Para cada um desses 96 testes *flaky*, Bell *et al.* (2018) identificaram um conjunto preciso de *commits* onde o teste era *flaky* por meio de investigação manual da causa da *flakiness*. Para limitar o tempo dos experimentos, executaram o DeFlaker apenas em 500 *commits* selecionados aleatoriamente a partir dessas faixas. O teste pode ter falhado em qualquer *build* desses *commits* (Bell *et al.*, 2018).

O *DeFlaker* foi avaliado em um conjunto amplo de projetos para encontrar *flaky tests* previamente desconhecidos. Dos 96 projetos de Java de código aberto monitorados, apenas 10 projetos tinham pelo menos um teste que o DeFlaker detectou como um possível *flaky tests*. Foram encontrados 87 *flaky tests* com falhas previamente desconhecidas em 10 projetos, além de 4.846 falhas em versões antigas de 26 projetos com uma baixa taxa de alarmes falsos (1,5%). O *recall* obtido foi maior (95,5% vs. 23%) do que o detector de *flaky tests* padrão do Maven.

Figura 1 – Arquitetura de alto nível do DeFlaker, com três fases: antes, durante e após a execução do teste.



Fonte: Bell *et al.* (2018, p.2).

Na Microsoft, a utilização de dados empíricos para minimizar o número de falsos alarmes de teste foi proposta para reduzir o custo de identificação de instabilidades. Foi utilizado o aprendizado de regras de associação para identificar padrões entre etapas de teste com falha que podem ser usados para classificá-los automaticamente em *flaky* ou *non-flaky*. Este tipo de análise pode ser considerada mista ao utilizar o aprendizado de regras de associação com base no resultado de diferentes execuções dos casos de teste. Ela é valiosa para equipes de produto, pois a inspeção manual de falhas retarda o desenvolvimento de produtos e consome recursos. A extração de regras de associação para analisar etapas de teste individuais ajuda a detectar

padrões exclusivos de alarmes falsos. Com estes padrões de comportamento de teste, foi desenvolvido um modelo de aprendizagem totalmente automático e contínuo para classificar as falhas dos casos de teste como *flaky tests* (Herzig; Nagappan, 2015).

De acordo com Herzig e Nagappan (2015), cada caso de teste de sistema e integração executa uma sequência de etapas ou passos de teste. Todas as quais são necessárias para passar no caso de teste geral. As falhas de teste relatadas aos engenheiros contêm uma lista de etapas de teste que falharam durante a execução. Esses passos de teste são comuns em testes de sistema e integração, pois auxiliam a verificação de diferentes elementos envolvidos no teste. Os passos são utilizados para estruturar as verificações de um caso de teste, facilitando a identificação de falhas ao distribuir a responsabilidade de verificação de pequenos elementos do teste.

A regra de associação sugere que a execução de um caso de teste onde os passos de teste X e Z falham, mas o passo de teste Y passa, deve ser considerada uma *flaky test*. O aprendizado de regras de associação retorna mais do que uma regra de associação simples. Cada regra pode ser tratada como um conjunto separado de condições que, se satisfeitas por um caso de teste, indicam como interpretar o resultado do mesmo. As regras de associação não indicam implicações, mas sim relações probabilísticas que mensuram duas variáveis: *support* e *confidence*. Esses termos são comuns quando se trata de regras de associação em mineração de dados (Podgurski *et al.*, 2003). No trabalho de (Herzig; Nagappan, 2015) o suporte trata-se de um valor entre zero e um que define a proporção de execuções dos casos de teste para os quais todos os antecedentes foram satisfeitos. Um valor de *support* de 0,5 significa que 50% de todas as execuções de casos de teste observadas satisfazem todos os antecedentes. Ou seja, 50% de todas as execuções de casos de teste, os passos de teste X e Z falham enquanto o passo de teste Y passa. A confiança numa regra é definida como o número relativo de execuções de casos de teste observadas para as quais todos os antecedentes e a consequência foram satisfeitas em relação ao número de execuções de casos de teste para os quais todos os antecedentes foram satisfeitos:

$$\text{Confiança}(\{A \rightarrow \{c\}\}) = \frac{\text{Suporte}(A)}{\text{Suporte}(A \cup \{c\})} \quad (1)$$

Onde A representa um conjunto de antecedentes, tais como $\{a_1, \dots, a_n\}$.

Para descobrir padrões entre o comportamento das etapas de teste exclusivo dos *flaky tests*, usamos a aprendizagem de regras de associação (Agrawal *et al.*, 1993) para produzir regras do tipo: $\{a_1, \dots, a_n\} \rightarrow \{c\}$ onde o lado esquerdo da implicação (antecedente) representa uma ou várias condições que precisam de ser satisfeitas para implicar o lado direito (consequente). Essa abordagem mista foi aplicada em testes de sistema e de integração executados durante o desenvolvimento do Windows 8.1 e do Microsoft Dynamics AX. Efetuando mais de 10.000 classificações para cada produto, o modelo apresentou uma precisão média entre 0,85 e 0,90, prevendo entre 34% e 48% de todos os *flaky tests*. Por fim, foi estimado o impacto dos falsos alarmes de teste na velocidade do código que poderiam ter sido evitados usando o modelo de classificação de resultados de testes proposto (Herzig; Nagappan, 2015).

O sistema *Cannier* visa unir os pontos fortes dos métodos baseados em *rerun* e de aprendizado de máquina. Ao fazer isso, ele reduz significativamente os atrasos de tempo causados pela reexecução de testes, ao mesmo tempo que melhora as capacidades de detecção de *flaky tests* em comparação ao uso exclusivo de aprendizado de máquina. Ele integra técnicas de reexecução existentes (como *Rerun*, *iDFlakies* e *Pairwise*) com modelos preditivos de aprendizado de máquina que avaliam a probabilidade de testes serem flaky. O sistema utiliza essas probabilidades para reduzir o número de execuções necessárias, levando a avaliações de teste mais eficientes. O *Cannier* é uma abordagem que combina uma técnica de detecção de *flaky tests* baseada em reexecução com um ou mais modelos de aprendizado de máquina (Parry *et al.*, 2023).

No caso do *rerun* utilizado pelo *iDFlakies*, o problema de classificação de *flaky tests* consiste em distinguir *flaky tests* NOD do restante dos casos de teste. Sendo um problema binário, os *flaky tests* NOD são a classe positiva, enquanto os outros casos de teste são a classe negativa. O modelo de aprendizado de máquina pode fornecer uma probabilidade prevista de pertencimento à classe positiva para cada caso de teste. O *Cannier* atribui um rótulo previsto como positivo a um caso de teste se a probabilidade for superior a um limiar superior, e como negativo se for inferior a um limiar inferior. Isso deixa uma região ambígua entre os dois limites. Ele delega quaisquer casos de teste com probabilidades previstas dentro dessa região ambígua para a técnica baseada em reexecução. Ainda no *Cannier*, utilizando dois modelos de aprendizado de máquina, pode-se reduzir o custo de tempo da técnica *Pairwise*. O primeiro modelo é utilizado para prever a probabilidade de cada caso de teste ser uma vítima, isto é, resolve o problema de classificação de distinguir vítimas de não-vítimas. O segundo modelo faz a mesma previsão, mas para identificar poluidores. Para ambos os modelos, o *Cannier* classifica cada caso de teste acima de um limiar como pertencente à classe positiva (vítima ou poluidor) e os demais como classe negativa (não vítima ou não poluidor) (Parry *et al.*, 2023).

Dessa forma, o *Cannier* produz dois conjuntos não mutuamente exclusivos: um de vítimas, T_V , e outro de poluidores, T_P . Não há razão para que um caso de teste não possa ser tanto uma vítima quanto um poluidor (Wei *et al.*, 2022). Em seguida, aplica o *Pairwise* utilizando apenas os membros de T_P como o primeiro teste de cada par e apenas os membros de T_V como o segundo. Portanto, essa abordagem pode reduzir a complexidade de tempo do *Pairwise* de $O(|T|^2)$, onde T é o conjunto de todos os casos de teste na suíte de testes, para $O(|T_V| \times |T_P|)$, o que é consideravelmente mais rápido mesmo quando T_V e T_P não são significativamente menores que T (Parry *et al.*, 2023).

Os autores realizaram avaliações abrangentes utilizando uma estrutura construída em dois sistemas: *PYTEST-CANNIER* e *CANNIER-FRAMEWORK*. Inicialmente, foram utilizadas uma variedade de pipelines de aprendizado de máquina e um conjunto de 18 métricas estáticas e dinâmicas de casos de teste, conforme apresentadas na Tabela 1.

O *Cannier* é uma abordagem desenvolvida para reduzir o custo de tempo de técnicas de detecção baseadas em reexecução, combinando-as com modelos de aprendizado de máquina.

Tabela 1 – Características medidas por PYTEST-CANNIER

#	Característica	Descrição
1	Contagem de Leituras	Número de operações de leitura no sistema de arquivos.
2	Contagem de Escritas	Número de operações de escrita no sistema de arquivos.
3	Tempo de Execução	Tempo total decorrido para a execução do teste.
4	Tempo de Espera	Tempo aguardando operações de entrada/saída.
5	Trocas de Contexto	Número de trocas de contexto voluntárias.
6	Linhas Cobertas	Número de linhas cobertas durante o teste.
7	Linhas de Código Fonte Cobertas	Linhas cobertas que não são parte dos casos de teste.
8	Mudanças Cobertas	Modificações nas linhas cobertas nos últimos 75 commits.
9	Máximo de Threads	Pico de número de threads concorrentes.
10	Máximo de Processos Filhos	Pico de número de processos filhos concorrentes.
11	Máximo de Memória	Pico de uso de memória durante a execução.
12	Profundidade da AST	Profundidade máxima de aninhamento no código do teste.
13	assertivas	Número de assertivas no código do teste.
14	Módulos Externos	Módulos não padrão utilizados pelo teste.
15	Volume de Halstead	Medida do tamanho da implementação do algoritmo (Peitek <i>et al.</i> (2021), Lundberg <i>et al.</i> (2020), Pontillo <i>et al.</i> (2022))
16	Complexidade Ciclométrica	Número de ramificações no código do teste (Gill e Kemerer (1991), Lundberg <i>et al.</i> (2020), Pontillo <i>et al.</i> (2022)).
17	Linhas de Código de Teste	Total de linhas no código do caso de teste.
18	Manutenibilidade	Facilidade de suporte e modificação do código do teste (Welker (2001)).

Fonte: Parry *et al.* (2023).

Foi realizada uma avaliação inicial da detecção baseada em aprendizado de máquina em um conjunto de dados contendo 89.668 casos de teste de 30 projetos em Python. O desempenho foi avaliado em relação à detecção de *flaky tests* NOD, *flaky tests* (vítimas) e casos de teste poluidores (causadores de instabilidade). Os resultados indicaram que o desempenho dos modelos de aprendizado de máquina foi pouco satisfatório e variável entre projetos (Parry *et al.*, 2023).

Posteriormente, foi investigado o impacto de vetores de características médias na detecção de *flaky tests* baseada em aprendizado de máquina. Identificou-se uma relação positiva entre o tamanho da amostra para produzir os vetores de características médias e o desempenho de detecção do modelo de aprendizado de máquina. No interesse do entendimento do modelo, a técnica SHAP (Lundberg *et al.*, 2020) foi aplicada para quantificar a contribuição de cada característica individual para o valor de saída do modelo. Embora essa técnica revele apenas correlações e não seja apropriada para inferir causalidade, foram feitos achados que suportam tanto a intuição geral dos desenvolvedores quanto resultados da literatura sobre *flaky tests*: são eles: o desempenho de modelos de aprendizado de máquina para detectar casos de teste poluidores; utilizar os valores médios das características dinâmicas dos casos de teste a partir de medições repetidas pode melhorar ligeiramente o desempenho de detecção dos modelos de aprendizado de máquina; e correlações entre várias características de casos de teste e a probabilidade do caso de teste ser instável (Parry *et al.*, 2023).

O *Cannier* alcançou uma redução média de 88% nos custos de tempo associados à reexecução de testes, com apenas uma leve queda na precisão da detecção. A eficácia dos modelos de aprendizado de máquina variou com base no tipo de teste instável (por exemplo, normal,

vítima ou poluidor). O uso de vetores de características agregadas proporcionou um leve aumento no desempenho dos componentes de aprendizado de máquina. Certas características de teste (como tempo de execução, contagem de Leituras, trocas de contexto) foram identificadas como preditores significativos das probabilidades de *flaky tests*, apontando para padrões que poderiam ser aproveitados para uma melhor detecção no futuro. Ele não só aborda questões imediatas relacionadas à detecção de testes *flaky*, mas também fornece conhecimentos mais profundos sobre a interseção entre características dos testes e sua probabilidade de instabilidade. Os autores sugerem direções futuras que podem envolver métodos de inferência causal e a expansão da aplicabilidade da ferramenta com técnicas adicionais (Parry *et al.*, 2023).

2.2.3 Técnicas estáticas

FlakyCat é uma técnica estática para classificar *flaky tests* baseado na causa raiz da instabilidade (Akli *et al.*, 2023). Ela consiste em descobrir a categoria de instabilidade dos *flaky tests*. Ela combina o uso do modelo pré-treinado CodeBERT (Feng *et al.*, 2020) e o Few Shot Learning (Sun *et al.*, 2021) com base em uma rede siamesa. O estudo utilizou dados de estudos anteriores sobre *flaky tests*, acessando dados previamente coletados e classificados (Luo *et al.*, 2014a; Eck *et al.*, 2019; Barbosa *et al.*, 2022; Habchi *et al.*, 2022; Shi *et al.*, 2019). Esses *datasets* forneceram um número considerável de *flaky tests* já categorizados de acordo com as causas raiz de suas falhas. Porém, a quantidade de dados em cada categoria era variável e, muitas vezes, insuficiente para um bom treinamento (Akli *et al.*, 2023).

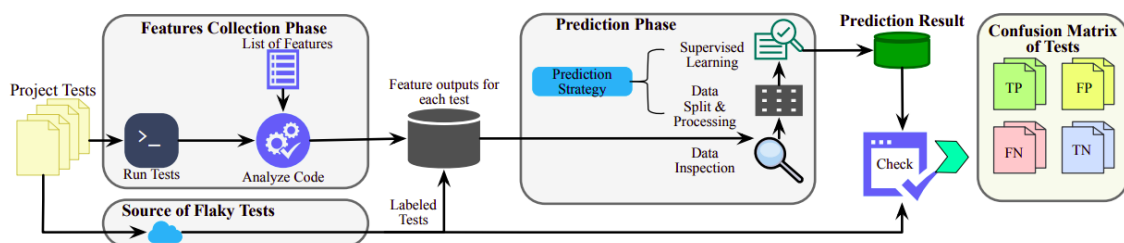
Para compensar a escassez de dados em algumas categorias nos dados existentes e aumentar a robustez do estudo, os autores coletaram um novo conjunto de dados diretamente do GitHub. Eles procuraram *commits* e *pull requests* que continham informações sobre a correção de *flaky tests*, utilizando palavras-chave relacionadas, tais como *flaky*, *intermittent*, *thread*, *concurrency*, *time*, dentre outras. Os autores examinaram as mensagens de *commit* e as alterações de código para classificar manualmente os testes em suas respectivas categorias de instabilidade. Este processo exigiu análise cuidadosa do código e comentários para garantir uma classificação precisa. Foram extraídos os métodos de teste dos arquivos de código-fonte e removidos comentários do código, focando apenas no código executável. Em seguida, utilizou representações vetoriais dos testes usando o CodeBERT (Feng *et al.*, 2020).

O *FlakyCat* foi treinado e avaliado com um conjunto de 451 *flaky tests* de projetos open-source Java. Utilizou as métricas de avaliação padrão para comparar classificadores, incluindo precisão, recall, Matthews Correlation Coefficient (MCC), pontuação F1 e Area Under The Curve (AUC). Essas métricas foram usadas para avaliar o desempenho dos classificadores, incluindo a classificação binária de *flaky tests*. Como seu conjunto de dados é desbalanceado, utilizaram métricas ponderadas para a avaliação. Ele obteve um bom resultado para predição de causas raiz, com f1-score de 73%. Seus resultados indicam uma classificação precisa de *flaky tests* relacionados com: espera assíncrona, dependência de ordem, coleções desordenadas

e relacionadas ao tempo. O melhor F1-score foi para a categoria de espera assíncrona 81%. O pior F1-score foi para flaky tests relacionados com concorrência, com 39%. Isso ocorreu provavelmente porque os casos de teste de concorrência podem ser confundidos com o de espera assíncrona, uma vez que ambos possuem interação com processos (Akli *et al.*, 2023). O Flaky-Cat indica que algumas categorias de instabilidades têm tipos de instruções semelhantes, como a presença de usos de threads em esperas assíncronas e categorias de simultaneidade. Isto revela uma relação entre as declarações do código-fonte e os resultados de predição. As principais limitações são: a dependência de dados suficientes para o treinamento eficaz; a possibilidade de categorizações sobrepostas de *flaky tests*; e a subjetividade na classificação manual e interpretação das sentenças de código. O tamanho reduzido de alguns conjuntos de categorias também afeta o desempenho do modelo (Akli *et al.*, 2023).

Um estudo realizado por Alshammari *et al.* (2021) apresenta limitações na abordagem dinâmica (*rerun*): após executar 10 mil vezes suítes de testes em 24 projetos, alguns testes com *flaky tests* previamente identificados ainda não foram detectados. Então, foi proposta a ferramenta chamada *FlakyFlagger* que recolhe um conjunto de características que descrevem o comportamento de cada teste, e então prevê testes que provavelmente serão *flaky* com base em características comportamentais semelhantes. Com esta abordagem, conforme apresentado na Figura 2, é possível visualizar que o código de testes rotulados como instáveis pode ser útil para predição de futuros *flaky tests* se suas características forem analisadas. No sistema chamado *FlakyFlagger*, após a utilização de técnicas de processamento de texto obtém-se o resultado e a matriz de confusão no final do processo de classificação.

Figura 2 – Abordagem FlakeFlagger para prever testes provavelmente instáveis, dado um conjunto *flaky tests* conhecidos.



Fonte: Alshammari *et al.* (2021, p. 4).

Para avaliar o *FlakyFlagger*, a ferramenta foi aplicada em 23 projetos com *flaky test*, comparando os resultados ao classificador apresentado por Pinto *et al.* (2020). Essa técnica obteve um *recall* semelhante ao classificador (74% vs 72%), mas uma precisão significativamente melhor (60% vs 11%). Conforme os autores, esta melhoria traduz-se numa redução significativa do número de testes com falhas intermitentes mal diagnosticadas (Alshammari *et al.*, 2021).

O sistema chamado *Flast* implementa uma abordagem projetada para prever estaticamente a instabilidade dos testes. Ele aproveita a modelação do espaço vetorial, a pesquisa de semelhanças, a redução da dimensionalidade e a classificação K Nearest Neighbors (K-NN)

para detectar *flakiness*. Para levantar informações sobre a eficiência e a eficácia desta técnica, foi realizada uma avaliação empírica da abordagem. Um total de 13 projetos foram considerados, onde existiam 1.383 testes com falhas e 26.702 testes sem falhas (Verdecchia *et al.*, 2021).

Foi realizada uma comparação quantitativa do *Flast* com os métodos avançados para detectar o caráter instável dos testes no atual estado da arte, são eles: *ReRun* (Micco, 2016), *DeFlaker* (Bell *et al.*, 2018), *iDFlakies* (Lam *et al.*, 2019), *NonDex* (Shi *et al.*, 2016), *Shaker* (Silva, 2022), *Flash* (Dutta *et al.*, 2020), *Association Rules* (Herzig; Nagappan, 2015), *Pattern Search* (Person; Elbaum, 2015), *Bayesian Network* (King *et al.*, 2018). Considerando um conjunto de dados equilibrado que inclui 1.402 testes reais com falhas e outros tantos sem falhas. Os resultados demonstram que, ao ajustar o limiar da abordagem, o *Flast* pode tornar-se mais conservador, de modo a reduzir os falsos positivos, à custa de falhar mais testes potencialmente duvidosos. Ele pode orientar a repetição de testes para impedir a inclusão de novos testes potencialmente instáveis. A eficácia do *Flast* é comparável aos trabalhos no estado da arte, ao mesmo tempo em que proporciona ganhos consideráveis em termos de eficiência (5% da sobrecarga de armazenamento necessária em outras abordagens) ao mesmo tempo em que faz previsões 4 vezes mais rápidas. Porém, o *Flast* não deve ser visto como uma alternativa às soluções dinâmicas existentes. Em vez disso, pode ser combinada com soluções estáticas e dinâmicas em sinergia mútua. Essa abordagem pode ser usada em conjuntos de dados de treinamento pequenos, e ainda não possui preditores mais genéricos que possam ser utilizados em diferentes projetos (Verdecchia *et al.*, 2021).

A técnica chamada *Flakify* funciona como um previsor de ‘caixa preta’ baseado em modelos de linguagem para *flaky tests*. Ao contrário dos métodos existentes, ela só requer o código-fonte dos casos de teste, sem acesso ao código de programa ou definição prévia de recursos. Semelhante ao *FlakyCat* ((Akli *et al.*, 2023)) ele também utiliza o *CodeBERT* ((Feng *et al.*, 2020)), um modelo de linguagem pré-treinado, ajustado para classificar casos de teste como instáveis ou não com base em seu código-fonte (Fatima *et al.*, 2022). Na técnica *Flakify*, o código-fonte de cada caso de teste é pré-processado para remover informações irrelevantes, mantendo apenas as declarações que correspondem a pelo menos um dos oito *Test Smells* (TS) associados à instabilidade de teste. Então, *CodeBERT* é usado para classificar os casos de teste como instáveis ou não com base no código pré-processado (Feng *et al.*, 2020; Fatima *et al.*, 2022).

O *Flakify* foi avaliado em dois conjuntos de dados públicos: um do *FlakeFlagger*, contendo 21.661 casos de teste de 23 projetos Java; e o conjunto de dados *IDoFT*, com 3.862 casos de teste de 312 projetos Java.

O *Flakify* alcançou os seguintes resultados: F1-score de 79% e 73% no conjunto de dados *FlakeFlagger* usando validação cruzada e validação por projeto, respectivamente; F1-score de 98% e 89% no conjunto de dados *IDoFT* usando validação cruzada e validação por projeto, respectivamente. Comparando os resultados do *Flakify* com a técnica *FlakeFlagger*

para o conjunto de dados do FlakeFlagger, o Flakify superou o FlakeFlagger em 10 e 18 pontos percentuais em termos de precisão e recuperação, respectivamente.

O *Flakify* possui as seguintes limitações. O CodeBERT tem um limite de comprimento de 512 tokens, o que pode levar a cortes em casos de teste longos. O pré-processamento ajuda a mitigar esse problema, mas informações relevantes ainda podem ser perdidas. Os conjuntos de dados usados são desequilibrados, com uma menor proporção de casos de teste instáveis. O Flakify é um modelo de caixa preta, o que significa que ele não pode identificar todos os tipos de instabilidade de teste. Finalmente, a precisão da previsão pode ser influenciada por fatores específicos do projeto que não são capturados pela ferramenta.

Portanto, o *Flakify* é um método promissor para prever *flaky tests*. Ele supera as abordagens existentes por exigir apenas o código-fonte do caso de teste. No entanto, existem limitações que devem ser consideradas em aplicações futuras. O artigo sugere a exploração de métodos de pré-processamento adicionais e investigações em conjuntos de dados mais completos (Fatima *et al.*, 2022).

O vocabulário instável é uma técnica estática que revela identificadores textuais fortemente relacionados com instabilidades (Pinto *et al.*, 2020). Modelos de aprendizado de máquina são aplicados para prever *flaky tests* utilizando os termos do código-fonte. Em seguida, um vocabulário instável, contendo os termos com maior ganho de informação para instabilidade, é gerado para auxiliar os desenvolvedores a prevenir *flaky tests* relacionados aos termos conhecidos.

Essa abordagem apresentou bons resultados separadamente para as linguagens Java (Pinto *et al.*, 2020; Camara *et al.*, 2021b), JavaScript (Soratto, 2022; Soratto; Silva, 2023) e Python (Haben *et al.*, 2021). Na próxima seção apresentamos os resultados, perspectivas e limitações destes trabalhos.

2.3 Vocabulário Instável: uma abordagem estática para prever *flaky tests*

No primeiro trabalho sobre o vocabulário instável, foi utilizado um conjunto de dados construído com base em 24 projetos Java, analisados pelo *DeFlaker*. Desses 24 projetos, foram observados 1.403 *flaky tests* e 62.979 *non flaky tests*. Os casos de testes foram transformados em identificadores textuais ou características para utilização do método de predição. Para extrair os termos dos casos de teste, foram utilizadas técnicas de processamento de texto, incluindo: criação de identificadores de palavras (*tokens*), separação de palavras em termos que utilizam *CamelCase* ou similares, lematização, remoção de palavras irrelevantes.

Por meio da reexecução, Pinto *et al.* (2020) encontraram seis projetos com *flaky tests* e um total de 86 *flaky tests* neles. Um total de 55% desses *flaky tests* falharam apenas uma vez, o que significa que o limite de 100 execuções pode ter limitado a observação de *flaky tests* (ou seja, é provável que possamos encontrar mais *flaky tests* com mais execuções). O projeto *alluxio* é um sistema de armazenamento distribuído virtual, enquanto *hector* é uma interface cliente Java de

alto nível para o banco de dados distribuído Cassandra. O *jackrabbit-oak* é uma implementação eficiente de um repositório de conteúdo hierárquico para uso em sites e sistemas de gestão de conteúdo. O *okhttp* é uma biblioteca para gerenciar sessões HTTP de forma eficiente. O *undertow* é uma implementação de servidor web não-bloqueante de alta performance, e o *wro4j* é uma biblioteca para otimizar o tempo de carregamento de páginas web. É importante notar que todos os projetos envolvem entrada/saída (IO), por exemplo, referem-se ao sistema de arquivos ou à rede (Pinto *et al.*, 2020). Este contexto de palavras e projetos pode refletir diretamente no vocabulário e na causa raiz da instabilidade nos projetos, limitando o contexto da técnica.

Quanto aos algoritmos de aprendizagem de máquina utilizados para criar os modelos de classificação, são replicados os classificadores comuns no contexto de engenharia de software, tais como: Random Forest (RF), *Decision Tree* (DT), Naive Bayes (NB), Support Vector Machines (SVM), K-NN, Logistic Regression (LR) (Souza *et al.*, 2014; Pinto *et al.*, 2020; Lampel *et al.*, 2021). o RF teve a melhor precisão (0,99), enquanto *glssvm* teve um desempenho ligeiramente superior ao RF em termos de *recall* (0,92 vs 0,91). Por fim, em relação ao vocabulário de *flaky tests*, foram identificadas palavras como *job*, *table*, *action*, muitas das quais estão associados à execução de tarefas remotamente ou usando uma fila de eventos (Pinto *et al.*, 2020).

A abordagem de vocabulário também foi avaliada no estudo de Camara *et al.* (2021b). Neste trabalho, replicou-se o estudo original, variando a plataforma para execução dos algoritmos de Machine Learning (ML) de Weka para Scikit-learn e adicionando dois novos algoritmos de classificação (LR e Linear Discriminant Analysis (LDA)). Os novos classificadores tiveram resultados semelhantes aos apresentados anteriormente. O algoritmo que apresentou o melhor valor para *recall* foi o LR.

Quando Camara *et al.* (2021b) diferenciam o conjunto de casos de testes de treinamento e teste, em um mesmo projeto de software, o desempenho do modelo de classificação foi baixo. As características com maior ganho de informação neste cenário são distintas das identificadas anteriormente e estão relacionadas com operações de entrada e saída, execução e coordenação de tarefas e palavras-chave da linguagem Java. Isto também ocorre na variação dos projetos que compõe os *datasets* de treinamento e teste. Neste cenário, as características com maior maior ganho de informação estão relacionadas com chamadas assíncronas, diferentemente das identificadas anteriormente por Pinto *et al.* (2020).

As palavras mais comuns no vocabulário original, proposto por Pinto *et al.* (2020), são: ‘job’, ‘table’, ‘id’ e ‘action’. As palavras mais comuns no vocabulário proposto por Camara *et al.* (2021b) são: ‘job’, ‘table’, ‘id’ e ‘service’. Tal semelhança entre os trabalhos anteriores se dá pelo fato de utilizarem conjuntos de dados semelhantes, sobre casos de testes de aplicações implementadas na linguagem Java. No estudo de Camara *et al.* (2021b) são realizados dois experimentos adicionais avaliando o ganho de informação nos cenários entre projetos e separadamente por projetos. O resultado do experimento entre projetos é apresentado na Tabela 28, onde estão presentes as top-20 palavras com maior ganho de informação para predição de instabilidade no conjunto Java. Neste experimento, foram identificadas diversas pala-

avras relacionadas com a espera assíncrona e comunicação com serviços (*await*, *export*, *handler*, *protocol*, *server*, *task*, *taskpayloadbuilder*, *url*). Diferentes características não mostraram grande impacto nos classificadores. O conjunto de palavras também é muito semelhante ao do estudo original: o vocabulário instável da replicação está relacionado com execução, coordenação de tarefas e persistência. As características com maior ganho de informação estão relacionadas a chamadas assíncronas e são distintas daquelas previamente identificadas (Camara *et al.*, 2021b).

Outro estudo de replicação do vocabulário instável foi realizado por Haben *et al.* (2021). Neste estudo, são apresentados os resultados em casos de teste implementados em Python e Java. Em resumo, três variações do trabalho original são propostas:

- Variação da metodologia de avaliação: utiliza a seleção dos conjuntos de treinamento e teste sensível ao tempo para refletir o cenário de uso real;
- Variação da linguagem do código-fonte presente no conjunto de dados: foram coletados 837 *flaky tests* de 9 projetos implementados na linguagem Python para confortar a generalidade dos resultados do vocabulário;
- Variação das *features* extraídas do código-fonte (incluindo elementos do Code Under Test (CUT)): os resultados para identificação de *flaky tests* caí consideravelmente utilizando uma validação mais robusta, mesmo que ainda seja o suficiente para uma predição decente.

Nesta replicação, algumas limitações da abordagem de vocabulário são identificadas em relação as *features* dos classificadores, seleção de dados e conteúdo presente no código-fonte do teste. Uma validação mais robusta diminui consistentemente o desempenho nos resultados relatados do estudo original, mas, o modelo continua capaz de prever decentemente *flaky tests*. As informações contidas no CUT têm um impacto limitado no desempenho dos modelos baseados em vocabulário. Mesmo que o CUT seja considerado a fonte da instabilidade, ele não melhora o desempenho dos classificadores de instabilidade. Os modelos baseados em vocabulário também podem ser usados para prever a instabilidade de testes em Python em um cenário intra-projeto. (Haben *et al.*, 2021).

Assim como em outros trabalhos, uma ameaça à validade é focar em *flaky tests* e não validar certamente os *non-flaky*. Este problema ocorre, pois mesmo aplicando o *rerun*, é impossível provar que o teste nunca falhará devido aos diversos fatores associados à execução. O mesmo problema está presente no *dataset* utilizado em trabalhos anteriores, por exemplo, o DeFlaker (Bell *et al.*, 2018).

Os trabalhos de Camara *et al.* (2021b) e Haben *et al.* (2021) fortalecem a confiança no processo de construção de vocabulários instáveis e ainda pontuam duas limitações: quanto as *features* que serão avaliadas nos modelos, e quanto aos dados utilizados nos processos de treinamento e teste dos modelos.

Na Tabela 2 é apresentado um resumo dos experimentos de vocabulário instável com diferentes linguagens e autores.

Tabela 2 – Características das Técnicas Baseadas em Vocabulário

Linguagem e Autores	Conjunto de Dados Considerados	Representação e Pré-processamento	Técnicas de IA Aplicadas	Forma de Avaliação	Melhores classificadores
Java (Pinto <i>et al.</i> , 2020)	24 projetos com 1403 <i>flaky tests</i> e 62.979 <i>non flaky</i> pela revisão. E 6 projetos com 86 <i>flaky tests</i> pela reexecução	<i>Bag of words</i> e Remoção de <i>stop words</i>	5 Modelos de aprendizado de máquina do Weka (Witten; Frank, 2002). Palavras com maior ganho de informação: <i>job</i> , <i>table</i> , <i>action</i> .	<i>Data split</i> : 80% treinamento 20% teste.	Random Forest
Java (Camara <i>et al.</i> , 2021b)	64 projetos com 256 testes <i>flaky</i>	<i>Bag of words</i> ; limpeza de código	7 Modelos de aprendizado de máquina do Scikit-learn (Pedregosa <i>et al.</i> , 2011)	<i>Data split</i> (80/20)	Random Forest
Java (Camara <i>et al.</i> , 2021b)	64 projetos com 256 testes <i>flaky</i>	<i>Bag of words</i> ; limpeza de código	7 Modelos de aprendizado de máquina do Scikit-learn (Pedregosa <i>et al.</i> , 2011)	Avaliação cruzada entre projetos (<i>cross-project validation</i>)	LDA
Python (Haben <i>et al.</i> , 2021)	9 projetos com 837 testes <i>flaky</i>	<i>Bag of words</i> ; remoção de anotações @ <i>flaky</i>	5 Modelos de aprendizado de máquina	<i>Data split</i> : 80% treinamento 20% teste.	Melhor modelo de classificação (RF) no projeto: <i>Bokeh</i>

Fonte: Autoria própria (2025).

2.4 Considerações Finais

A técnica de identificação estática será utilizada neste trabalho, mais especificamente a construção de vocabulários (Pinto *et al.*, 2020; Camara *et al.*, 2021b; Haben *et al.*, 2021).

É necessário o estudo de modelos generalizáveis de predição em diversos contextos de instabilidades, por exemplo, em cenários intra e inter-projetos (Camara *et al.*, 2021b). Para aumentar a variabilidade de linguagens analisadas, este estudo considerará três linguagens: Java, Python e Javascript. Existe a suposição de que existe um vocabulário instável semelhante entre diferentes linguagens e projetos. No próximo capítulo, será delineada a abordagem de vocabulário para a linguagem JavaScript em um cenário intra e entre projetos. Com os resultados obtidos neste trabalho para a linguagem JavaScript, realizamos comparações com os vocabulários instáveis em Python e Java apresentados neste capítulo.

3 ABORDAGEM

Este trabalho utiliza a abordagem de vocabulário instável para identificar *flaky tests* escritos na linguagem JavaScript e comparar com trabalhos anteriores que trataram de software escrito em Java (Pinto *et al.*, 2020; Camara *et al.*, 2021b) e Python (Haben *et al.*, 2021). O principal objetivo é conhecer as diferenças e semelhanças entre os vocabulários instáveis de diferentes projetos e linguagens. Nesta abordagem, é necessária a utilização de casos de teste com seus respectivos termos para a construção de modelos de classificação. Aplicamos essa abordagem utilizando dados de testes reais coletados de projetos JavaScript. Verificamos as semelhanças e diferenças deste resultado JavaScript com as linguagens Python e Java. As questões de pesquisa abordadas são:

QP1

Com que qualidade podemos prever instabilidades em casos de teste implementados na linguagem JavaScript utilizando vocabulário?

QP2

Quais são os principais termos presentes em códigos de testes associados à instabilidade na linguagem JavaScript?

QP3

Com que qualidade podemos prever instabilidades em casos de teste entre diferentes projetos e linguagens utilizando vocabulário?

QP4

Quais são os principais termos presentes em códigos de testes associados à instabilidade entre diferentes linguagens?

Ao estabelecer um modelo de classificação a partir dos termos utilizados nos casos de teste, podemos avaliar estas questões de pesquisa. A mineração de dados consiste no processo de extração de conhecimento útil e previamente desconhecido em dados, por meio da aplicação de algoritmos que extraem modelos e padrões representativos (Fayyad *et al.*, 1996). Ela é organizada em etapas: conhecimento do domínio, pré-processamento, extração de padrões, pós-processamento e utilização do conhecimento (Rezende, 2005). A abordagem de vocabulário instável segue a estrutura definida pela mineração de dados, aplicada aos casos de testes. Assim, a criação de um vocabulário instável é o resultado de um processo de mineração de dados presentes nos casos de testes de softwares. Seguindo o processo de mineração de dados, utilizamos o seguinte mapeamento das etapas para a construção do vocabulário instável:

1. Conhecimento do domínio: Obtenção de casos de testes rotulados como *flaky* ou *non flaky*;
2. pré-processamento: Representação dos casos de teste, considerando os tokens, em um modelo *bag-of-words*;
3. Extração de padrões: Criação de modelos de classificação para projetos implementados em Javascript, Java e Python;
4. Pós-Processamento: Construção de um vocabulário instável a partir dos resultados do classificador;
5. Utilização do conhecimento: Comparação de vocabulário e modelos de classificação com trabalhos relacionados.

Detalhando mais essas etapas para este estudo, o processo em três grandes grupos de atividades, conforme apresentado na Figura 3. Nesta figura existe um conjunto de dados central, ele contém os casos de testes rotulados no modelo de representação Bag-of-Words (BoW). Ele é uma técnica amplamente utilizada no processamento de linguagem natural para representar textos de forma computacional. Nesse modelo, um documento é considerado como uma coleção (ou “saco”) de palavras, ignorando a ordem sequencial e a gramática, focando apenas na frequência de ocorrência de cada termo.

A implementação típica do BoW envolve etapas como:

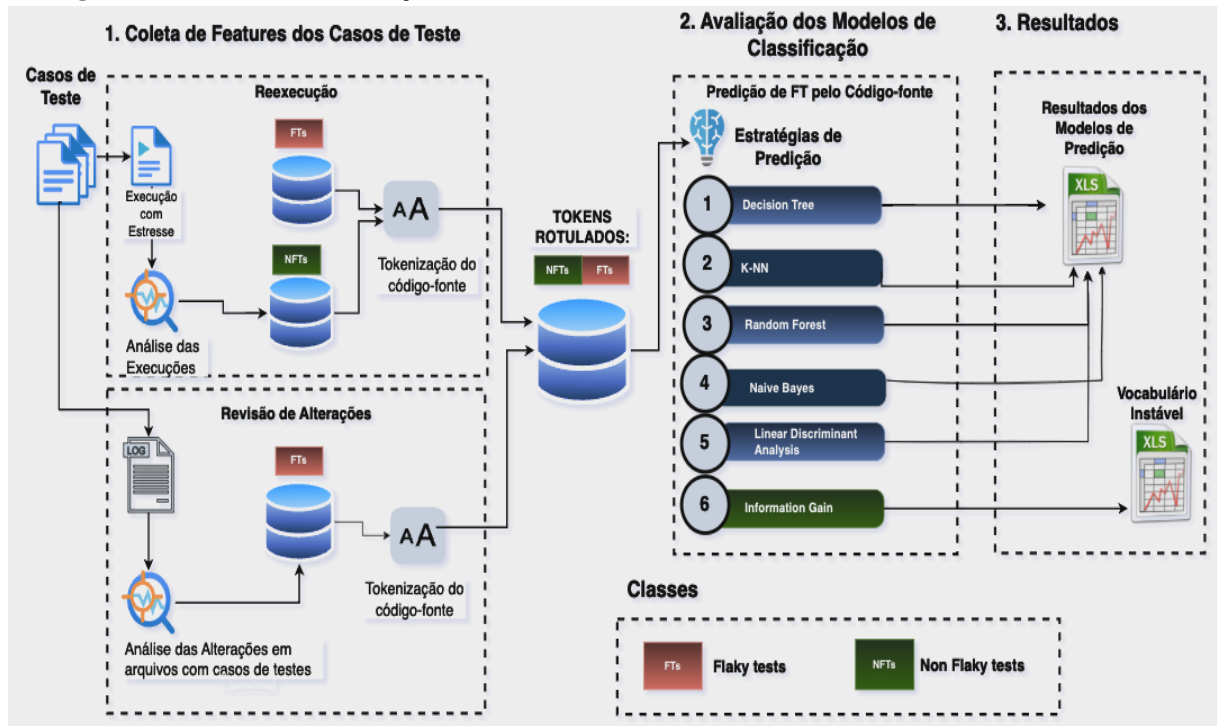
- Tokenização: divisão do texto em palavras ou tokens.
- Construção do vocabulário: criação de um conjunto de todas as palavras distintas presentes na coleção de documentos.
- Representação vetorial: cada documento é convertido em um vetor onde cada posição corresponde a uma palavra do vocabulário, contendo a frequência, a contagem binária (presença ou ausência) ou a ponderação dessa palavra no documento.

O modelo BoW é eficaz na extração de *features* para tarefas como classificação de textos, análise de sentimentos e recuperação de informação (Manning; Schütze, 1999).

Referente ao conhecimento de domínio e pré-processamento, temos o primeiro grupo de atividades, compreendendo a obtenção de dados e extração de características de casos de teste, classificando-se os *flaky tests*. Para as linguagens Java e Python, foram considerados os conjuntos de dados existentes. Para a linguagem JavaScript, foram consideradas duas abordagens para construir o conjunto de dados: a técnica de reexecução com estresse; e a análise dos pedidos de alteração, de modo empírico, complementando conjuntos de dados existentes para essa linguagem. Para ambos os casos, após extraídos e classificados os casos de teste, foram extraídos os termos (*tokens*) e realizado o pré-processamento. A partir deste conjunto de dados rotulados, o segundo grupo de atividades corresponde à extração de padrões, com a criação de modelos de predição de *flaky tests*. O terceiro grupo de atividades refere-se à análise dos resultados, com

o pós-processamento dos dados para estabelecer o vocabulário dos casos de teste instáveis e a utilização desse conhecimento para comparar o vocabulário obtido para cada linguagem.

Figura 3 – Processo de construção de um vocabulário instável utilizando diferentes fontes de dados.



Fonte: Autoria própria (2025).

O restante da organização deste capítulo faz-se conforme a definição apresentada de mineração de dados e detalhada conforme apresentado na Figura 3. O conhecimento do domínio sobre *flaky test* e os dados utilizados são descritos na Seção 3.1. O pré-processamento do conjunto de dados inicial contendo o código-fonte de *flaky tests* JavaScript é descrito na Seção 3.2. Na Seção 3.3 é detalhado todo o processo para extração de características relevantes dos casos de teste e o formato do conjunto de dados proposto. Na Seção 3.4 detalhamos o processo de pós-processamento das informações extraídas. Na seção Seção 3.5 descrevemos como serão utilizados os conhecimentos extraídos para comparar os resultados dos vocabulários em diferentes linguagens.

3.1 Conhecimento do domínio

A construção de um vocabulário de teste instável utiliza dados de projetos reais. Estudos sobre mineração de dados referentes a software, denominados de mineração de repositórios de software (Mining Software Repositories (MSR)), geralmente recuperam os dados de projetos reais, a partir de repositórios de projetos específicos, de portais associados a uma organização

ou de plataformas sociais de desenvolvimento de software. O Github¹ e Gitlab² são exemplos de plataformas amplamente utilizadas pela comunidade de desenvolvimento de software. Considerada a popularidade da plataforma social GitHub (Preston-Werner *et al.*, 2008) e a sua qualidade e frequente uso em estudos similares (Kalliamvakou *et al.*, 2014), ela foi escolhida como fonte de dados para este trabalho.

Entretanto, a coleta de casos de testes do GitHub não é trivial visto que é necessário um critério para seleção de projetos com certa relevância para cada linguagem. Por exemplo, os trabalhos anteriores de vocabulário instável em Java utilizaram um conjunto de dados amplo de casos de testes em diferentes projetos nesta linguagem, inicialmente proposto por Bell *et al.* (2018).

Para construção da nossa base de dados em JavaScript utilizamos duas fontes de dados: o resultado das execuções dos casos de testes de diferentes projetos, e alterações de desenvolvedores em casos de teste instáveis em Javascript, apresentado por Hashemi *et al.* (2022) e Romano *et al.* (2021). Os dados referentes aos *flaky tests* para linguagem Java e Python será replicado dos trabalhos anteriores sobre o vocabulário instável (Pinto *et al.*, 2020; Camara *et al.*, 2021b; Haben *et al.*, 2021).

3.1.1 Conjuntos de dados sobre *flaky tests* em Java

Os trabalhos sobre vocabulário instável em Java de Pinto *et al.* (2020) e Camara *et al.* (2021b) utilizaram o conjunto apresentado por Bell *et al.* (2018) descrito no capítulo anterior. Pinto *et al.* (2020) realizou a coleta do código-fonte dos *flaky tests* rotulados por Bell *et al.* (2018) para aplicação da técnica de vocabulário. Estes dados foram utilizados no trabalho de Camara *et al.* (2021b) e também serão utilizados neste.

Adicionamos este conjunto comprimido no diretório³ de nossos experimentos. Também criamos um arquivo auxiliar chamado *data_view.py* para visualizar esses dados. Ao executá-lo ele informa que temos 1.392 instâncias de testes com o atributo ‘Label’ igual a ‘1’ (*flaky tests*) e 14.661 instâncias de testes com o atributo ‘Label’ igual a 0 (*non flaky tests*) para a linguagem Java.

3.1.2 Conjuntos de dados sobre *flaky tests* em Python

Para o vocabulário na linguagem Python, Haben *et al.* (2021) aplicaram técnicas de MSR para coletar os casos de teste de projetos hospedados no GitHub.

O Python dispõe de um conjunto de frameworks de teste. Para o estudo que originou o conjunto de dados para *flaky tests* em Python, foi considerado o framework Pytest (Pytest,

¹ <https://github.com/>

² <https://about.gitlab.com/>

³ <https://github.com/sorattorafa/flaky-across-languages/tree/main/datasets/java>

2025), que é o equivalente do JUnit para Python e permite que os desenvolvedores escrevam testes para seus programas. O Pytest apresenta uma série de recursos e *plugins*. Em especial, um módulo específico para lidar com *flaky test* pode ser utilizado com o Pytest: o *flaky* (Box, Inc., 2014). Esse módulo permite que os desenvolvedores anotem testes como *flaky* para que sejam automaticamente reexecutados em caso de falha. O desenvolvedor também pode configurar a quantidade máxima de execuções a serem tentadas e o número mínimo de sucesso necessário. Essa anotação pode ser adicionada a uma função de teste ou diretamente à classe de teste, conferindo essa propriedade a todos os testes contidos nela. Dessa forma, essa anotação facilita a identificação de *flaky tests* em projetos Python que utilizam o módulo *flaky*.

Nas experimentações de Haben *et al.* (2021) foram selecionados projetos nos quais havia um número suficiente de testes *flaky* para treinar e testar um modelo, ou seja, 30 testes *flaky*. Isso resultou em um conjunto de dados com 9 projetos e 837 testes marcados pelos desenvolvedores como *flaky* (Haben *et al.*, 2021). Para a seleção de projetos, foi realizada uma mineração no GitHub utilizando a API source-graph, buscando por projetos em Python que contivessem a anotação `@flaky`. Esse processo resultou em 110 projetos, com um total de 1.304 testes marcados como *flaky* (Haben *et al.*, 2021).

Realizamos a extração dos termos na forma de *tokens* para cada caso de teste apresentado por Haben *et al.* (2021). Como parte do pré-processamento, foram removidas as anotações `@flaky`, uma vez que mantê-las poderia enviesar o modelo em direção ao reconhecimento dessa anotação em vez do vocabulário do código. Com a frequência de cada termo do conjunto para cada caso de teste, foi criada a representação em bag of words, similarmente ao feito na criação do conjunto de dados de trabalhos relacionados em Java.

Adicionamos este conjunto apresentado por Haben *et al.* (2021) comprimido no diretório⁴ de nossos experimentos. Também criamos um arquivo auxiliar chamado *data_view.py* para visualizar esses dados. Ao executá-lo ele informa que temos 104 instâncias de testes com o atributo 'Label' igual a '1' (*flaky tests*) e 2.501 instâncias de testes com o atributo 'Label' igual a 0 (*non flaky tests*) para a linguagem Python.

3.1.3 Conjuntos de dados iniciais em JavaScript

Neste trabalho, foram utilizadas duas fontes: os projetos e casos de teste utilizados em trabalhos anteriores; e projetos novos, com identificação de casos de teste instáveis pela reexecução. Consideramos os casos de testes e projetos já conhecidos por trabalhos anteriores de instabilidades em testes implementos com JavaScript por Hashemi *et al.* (2022) e Romano *et al.* (2021) para: (1) obter o código-fonte de *flaky tests* conhecidos; (2) executar os casos de teste em versões (*commits*) que possuem *flakiness* para obter novos *flaky tests*; (3) obter projetos relacionados para execução e coleta de novos *flaky tests*.

⁴ <https://github.com/sorattorafa/flaky-across-languages/tree/main/datasets/python>

A primeira etapa do processo de construção de uma base de dados de *flaky tests* é localizar o código-fonte de instabilidades. Isto pode ser feito de duas maneiras: conforme apresentado por Haben *et al.* (2021) *flaky tests* pode ser obtido pela aplicação de técnicas de MSR buscando rótulos em testes, ou conforme apresentado por Bell *et al.* (2018) reexecutando a bateria de testes diversas vezes em versões que possuem *flakiness*. Com um Flaky Test (FT) identificado pela reexecução, ou por um desenvolvedor no GitHub, é possível obter o código-fonte relacionado à intermitência da falha utilizando um identificador do teste. Geralmente, este identificador é o nome dele dentro de sua bateria de testes.

No trabalho de Romano *et al.* (2021) a maior parte de projetos com instabilidades em testes no JavaScript são de interface (React, Angular, Bootstrap, Next.js). Com esses dados é possível conhecer um pouco sobre as instabilidades existentes no domínio dos projetos JavaScript. Tais projetos são o centro de um ecossistema de outros *frameworks*, por exemplo, o React e o Angular. Portanto, é possível que outros projetos neste ecossistema possuam instabilidades em testes.

Para que os modelos de classificação sejam aplicados durante o processo de entrega de softwares reais, precisamos treiná-los com casos de testes representativos deste cenário. Trabalhos anteriores sobre instabilidades em testes escritos em JavaScript já listaram grande parte dos projetos relevantes neste domínio (Romano *et al.*, 2021; Hashemi *et al.*, 2022; Soratto; Silva, 2023). Utilizamos os projetos já citados anteriormente pelos trabalhos anteriores, complementando com novos projetos relacionados. Por exemplo, os trabalhos anteriores citam o framework Angular, porém existem diversos projetos que utilizam ou complementam a utilização desta biblioteca que podem conter *flaky tests*. Desta forma, é necessário coletar dados do ecossistema de bibliotecas JavaScript de forma mais ampla. Para selecionar esses novos projetos, utilizamos os seguintes critérios:

- Ser projeto de código-fonte aberto e estar disponível no GitHub;
- Ter uma versão estável do software onde os testes executam pelo menos uma vez sem falhas;
- Ter uma suíte de testes com mais de 200 casos de teste;
- Respeitar a diversidade dos projetos.
- Ter mais de 10 contribuidores no projeto

Trabalhos anteriores apresentaram bons resultados em coletas no histórico de projetos de código aberto. Ao todo, foram identificadas 235 instâncias de interações de desenvolvedores relacionadas a instabilidades em testes automáticos de projetos no GitHub por Romano *et al.* (2021) e 452 instâncias apresentadas por Hashemi *et al.* (2022). A maioria dos projetos selecionados por Romano *et al.* (2021) e Hashemi *et al.* (2022) estão conforme os critérios apresentados acima, por utilizarem um filtro semelhante nas palavras-chave dos projetos e no conteúdo das

contribuições coletadas. Utilizaremos os projetos levantados nos trabalhos anteriores como base para selecionar mais projetos relevantes para a linguagem Javascript. Por exemplo, ambos trabalhos citam o projeto React: framework criado pelo Facebook para construção de interfaces ⁵. Existem diversas bibliotecas escritas em JavaScript que se comunicam. Por exemplo, são construídos novos *frameworks* utilizando o React como base, por exemplo, o Next.js ⁶. Ambos projetos possuem mais de 100 mil estrelas, 25 mil *forks*, e 3 mil contribuidores. Semelhantemente, o projeto Angular Components ⁷ complementa a utilização do Angular ⁸, e ambos possuem uma ampla comunidade de contribuidores.

No conjunto de dados apresentado por Romano *et al.* (2021), a coleta de *flaky tests* foi realizada aplicando um filtro de pesquisa com os termos *flaky* e *flakiness* em alterações de código (*commit*) presentes em solicitações de alteração (*pull request*) e relatos de problemas (*issues*) em projetos abertos Java e Javascript. Foram coletados repositórios que continham o nome de diferentes *frameworks* para projetos User Interface (UI) e *web* em suas palavras-chave, por exemplo: ‘react’, ‘angular’, ‘vue’, ‘emberjs’, ‘d3’, ‘svg’, ‘web’, e ‘bootstrap’. Foram identificados mais de 7 mil repositórios com mais de 2 milhões e meio de *commits*. Foi realizado um filtro semelhante ao apresentado por Luo *et al.* (2014b) nos *commits* contendo as *strings* rotulando *flaky tests*: ‘flak*’, ‘intermit*’ e ‘test’. Para especificar ainda mais a busca no contexto de aplicações *web* foram utilizadas palavras-chave de busca com elementos deste cenário: ‘ui’, ‘gui’, ‘window’, ‘display’, ‘click’. Essa coleta de alterações relacionadas ao *flaky test* em diferentes projetos *web* identificou 232 *flaky tests* em 62 projetos.

Adicionamos estes dados no diretório ⁹ de nossos experimentos. Separamos os dados obtidos pelo *rerun* e do *review*. Também criamos um arquivo auxiliar chamado *data_view.py* para visualizar esses dados. Ao executá-lo ele informa que temos 246 instâncias de testes com o atributo ‘is_flaky’ igual a ‘1’ (*flaky tests*) e 6.481 instâncias de testes com o atributo ‘is_flaky’ igual a 0 (*non flaky tests*) para a linguagem JavaScript. Na próxima subseção apresentamos as etapas para coleta destes testes rotulados como instáveis e não instáveis em JavaScript.

3.1.4 Construção de conjuntos de dados sobre *flaky tests* em Javascript

A ênfase desta subseção está em reportar o processo de coleta de dados de *flaky tests*. Registrar informações sobre *flaky tests* é o alicerce para construção de técnicas automáticas de identificação e correção. Utilizando o *rerun* a identificação do código-fonte é automática: o identificador do caso de teste será a referência do código-fonte. Porém, a identificação do código-fonte dos *flaky tests* em conjunto de dados obtidos por contribuições no GitHub não é trivial, necessitando de revisão manual dos códigos afetados e o estudo sobre a suíte de testes

⁵ <https://GitHub.com/facebook/react>

⁶ <https://GitHub.com/vercel/next.js>

⁷ <https://GitHub.com/angular/components>

⁸ <https://GitHub.com/angular/angular>

⁹ <https://github.com/sorattorafa/flaky-across-languages/tree/main/datasets/javascript/data>

de cada projeto. Por exemplo: a identificação da instabilidade pode ser feita pelo ambiente de execução de teste, conteúdo do *commit* ou *pull request*, ou verificando a qual caso de teste certa funcionalidade pertence. Existem ferramentas para automatizar a coleta de informações sobre casos de testes, tais como: filtragem de projetos, coleta de trechos de código a partir de *commits* e processamento léxico do código-fonte (Soratto, 2022).

É possível automatizar a coleta do código-fonte de FT utilizando diferentes abordagens: dinâmicas e estáticas.

Na abordagem estática, os desenvolvedores podem rotular os testes com problemas diretamente no nome do teste. Por exemplo, pode-se adicionar o rótulo '@flaky' no nome de um caso de teste para indicar que ele é instável. Dessa forma, é possível obter o código-fonte relacionado à instabilidade diretamente pelo nome do caso de teste (Soratto, 2022; Soratto; Silva, 2023). Também é possível identificar a instabilidade através do histórico de alterações do projeto, filtrando as modificações que contêm citações de FT. Porém, nada pode ser afirmado em relação aos outros casos de teste que não foram rotulados como FT. Isso ocorre porque eles podem apresentar problemas que ainda não foram revelados e, conseqüentemente, não foram rotulados pelos desenvolvedores.

Mesmo sendo possível classificar um caso de teste como FT pelo rótulo dos desenvolvedores, não é seguro afirmar que todos os outros casos de teste pertencem à classe Non Flaky Test (NFT). Por exemplo, a Seção 3.1.4 apresenta instâncias da classe FT que foram coletadas a partir dos rótulos dos desenvolvedores. Nela, são apresentados quantos *flaky tests* foram rotulados pelos desenvolvedores em cada projeto. No entanto, os modelos de classificação precisam ser treinados com ambas as classes para diferenciar corretamente *flaky tests* de estáveis. Então, é necessário obter instâncias de NFT através da reexecução para garantir a representatividade dos dados entre as classes. Ou seja, na etapa de coleta de dados, a abordagem dinâmica é mais segura para garantir que o modelo seja treinado com dados consistentes. Então, para obter os NFT utilizamos apenas a reexecução. Para obter os FT, utilizamos *flaky tests* rotulados pelos desenvolvedores apresentados na Seção 3.1.4 e também os novos FT descobertos pela técnica de reexecução nos projetos JavaScript selecionados.

Propomos um cenário automático de reexecução, rotulação dos casos de teste como FT e NFT, e extração do código-fonte para casos em JavaScript. Com o resultado das execuções, é possível rotular os casos de teste e extrair informações relevantes, como o código-fonte. Utilizando essa abordagem de reexecução, garantimos que as instâncias classificadas como NFT são de fato estáveis, enquanto as FT são instáveis. Essa preocupação surge porque o FT se manifesta após uma quantidade indeterminada de execuções. Se um caso de teste mantém seu resultado após diversas execuções, é possível afirmar que ele é minimamente estável.

Para aumentar a probabilidade de instabilidades em casos de teste, é possível estressar a CPU e a memória da máquina. A ferramenta Shaker, disponível em ¹⁰, permite a automatização deste processo de reexecução com estresse e armazenamento dos resultados de saída dos testes

¹⁰ <https://GitHub.com/STAR-RG/shaker>

Tabela 3 – Casos de testes em JavaScript rotulados por meio da revisão de commits.

GitHub ID	Name	#FT
styleguidist/react-styleguidist.git	react-styleguidist	1
outline/outline.git	outline	2
apollographql/react-apollo.git	react-apollo	1
react-cosmos/react-cosmos.git	react-cosmos	1
HospitalRun/hospitalrun-frontend.git	hospitalrun-frontend	1
gotify/server.git	server	6
Hacker0x01/react-datepicker.git	react-datepicker	3
twbs/bootstrap.git	bootstrap	1
skbkontur/retail-ui.git	retail-ui	1
thaliproject/postcardapp.git	postcardapp	5
OfficeDev/office-ui-fabric-react.git	office-ui-fabric-react	1
rjsf-team/react-jsonschema-form.git	react-jsonschema-form	1
palantir/blueprint.git	blueprint	3
plotly/plotly.js.git	plotly.js	32
stream-labs/streamlabs-obs.git	streamlabs-obs	2
JetBrains/ring-ui.git	ring-ui	2
NativeScript/nativescript-angular.git	nativescript-angular	1
angular/components.git	components	10
angular/angular.git	angular	6
coralproject/talk.git	talk	1
pinterest/gestalt.git	gestalt	4
uber/baseweb.git	baseweb	7
jhipster/generator-jhipster.git	generator-jhipster	3
nfl/react-helmet.git	react-helmet	1
influxdata/influxdb.git	influxdb	22
enzymejs/enzyme.git	enzyme	1
mobxjs/mobx.git	mobx	2
preactjs/preact.git	preact	17
gatsbyjs/gatsby.git	gatsby	1
vercel/next.js.git	next.js	1
storybookjs/storybook.git	storybook	1
mui-org/material-ui.git	material-ui	3
Total:	32 projetos	144 testes

Fonte: Soratto (2022).

em Python e Java. O resultado de cada execução é salvo no formato Extensible Markup Language (XML), permitindo a extração das características para os classificadores (Cordeiro *et al.*, 2021a).

Adaptamos esta ferramenta para funcionar com casos de teste em JavaScript, criando a versão ‘ShakerJS’, disponível em ¹¹. Esta ferramenta automatiza o processo de coleta de FT e NFT para testes implementados com Jest¹² e Karma¹³. Ela pode ser executada localmente ou em pipelines do GitHub Actions ¹⁴. Utilizamos essa ferramenta para executar os projetos mais relevantes no contexto de interfaces desenvolvidas em JavaScript: React, React Native, Bootstrap, Ant Design, Ant Design Vue, Next.js, Gatsby, React Router, Nativefier, entre outros.

¹¹ <https://GitHub.com/sorattorafa/shaker-js>

¹² <https://jestjs.io/pt-BR/>

¹³ <https://karma-runner.GitHub.io/latest/index.html>

¹⁴ <https://docs.GitHub.com/pt/actions>

Os projetos selecionados, os resultados obtidos, e a comparação dos resultados obtidos entre linguagens e projetos são apresentados na próxima seção.

3.1.5 Resultado da coleta de *flaky tests* em Javascript

Coletamos o código-fonte de *flaky tests* em JavaScript apresentados por trabalhos anteriores (Romano *et al.*, 2021; Hashemi *et al.*, 2022; Soratto, 2022). Identificamos 32 projetos com 144 *flaky tests*, listados na Tabela 8. Recuperamos o código-fonte de todos esses 144 casos de testes e rotulamos eles como instáveis. Para incrementar essa base de *flaky tests* executamos os casos de testes destes 32 projetos pelo menos mil vezes para cada projeto. Também selecionamos novos projetos semelhantes aos selecionados anteriormente. Em geral esses novos projetos são extensões, ou se comunicam com pelo menos um dos 32 projetos citados nos trabalhos anteriores. Todos os projetos selecionados para coleta de dados foram listados na Tabela 4. Os projetos que apresentaram novas instabilidades estão identificados em negrito.

Para criar uma configuração de reexecução padronizada, replicamos a utilização da ferramenta Shaker (Cordeiro *et al.*, 2021a) que foi originalmente construída para estressar execuções de testes escritos em Java e Python. Além de utilizar as configurações disponíveis para estressar os testes em execuções locais e no GitHub Actions, adicionamos a opção para registrar os resultados de saída dos casos de teste executados com Jest e Karma no JavaScript. Chamamos essa extensão de ShakerJS, e disponibilizamos ela em <https://github.com/sorattorafa/shaker-js>. No arquivo *tool_jest.py*¹⁵ existe a classe configurada para executar diversas vezes os testes que utilizam o Jest e salvar os resultados em arquivos ‘.xml’. O mesmo acontece no arquivo *tool_karma.py*¹⁶ para testes que utilizam o Karma. Projetos que utilizam o Karma, geralmente realizam testes integrados com navegadores de uma forma específica. Isto traz a necessidade de usar um gerador de ‘.xml’ diferente daquele utilizado no Jest. A responsabilidade dessas duas classes são: configurar a execução do testes várias vezes e salvar todos os resultados das execuções. Outras ferramentas podem ser desenvolvidas para obter esse histórico de execução em diferentes bibliotecas com este processo desde que o resultado final seja uma lista de arquivos ‘.xml’. Com estes arquivos, o ShakerJS faz a análise das falhas intermitentes no arquivo *print_failures.py*¹⁷, salve elas em um banco de dados na nuvem utilizando o arquivo *post_failures.py*¹⁸.

Com o ShakerJS é possível criar pipelines de execuções dos casos de teste com estresse de CPU e memória que registra automaticamente os *flaky tests* através dos resultados das execuções. Cada execução registra o resultado em uma base de dados. Ao fim de todas as execuções, os resultados intermitentes são revelados em alterações de saídas. Esse pipeline pode ser executado localmente ou em sistemas de integração contínua, como o GitHub Actions. No

¹⁵ https://github.com/sorattorafa/shaker-js/blob/main/shaker/tool_jest.py

¹⁶ https://github.com/sorattorafa/shaker-js/blob/main/shaker/tool_karma.py

¹⁷ https://github.com/sorattorafa/shaker-js/blob/main/shaker/print_failures.py

¹⁸ https://github.com/sorattorafa/shaker-js/blob/main/shaker/post_failures.py

Tabela 4 – Tabela de repositórios Github e suas respectivas versões (*commit*) no qual o *rerun* foi executado.

Git	Commit SHA
facebook/react	45d61cf
facebook/react-native	a1b318c
facebook/metro	a7f8955
facebook/docusaurus	e189978
facebook/hyperion	aa638ae
facebook/memlab	ec905b1
ant-design/ant-design	4f1ca4c
vueComponent/ant-design-vue	9cc7301
twbs/bootstrap	f68a47c
nativifier/nativifier	f22750b
strapi/strapi	6a58621
gatsbyjs/gatsby	cc5a4fc
HospitalRun/hospitalrun-frontend	c45aa10
pinterest/gestalt	712ff12
Hacker0x01/react-datepicker	3463a2d
react-cosmos/react-cosmos	653c15d
uber/baseweb	1d872e5
chartjs/Chart.js	02a7bdb
freeCodeCamp/freeCodeCamp	2965341
lodash/lodash	aa18212
mermaid-js/mermaid	6939cf5
JetBrains/ring-ui	382b812
plotly/plotly.js	011443e
react-boilerplate/react-boilerplate	d19099a
nfl/react-helmet	1b57ddb
react-native-elements/react-native-elements	c87b7ec
styleguidist/react-styleguidist	9886b64
videojs/video.js	caf6d30
airbnb/visx	edcca4
ben-rogeron/twin.macro	5ee856c
react-hook-form/react-hook-form	6fee03c
remix-run/react-router	e6814d5
moleculerjs/moleculer	275eda5
vercel/next.js	99490de
outline/outline	56cae8a
influxdata/influxdb	275eda5
Total: 36 Projetos	36 versões

Fonte: Autoria própria (2025).

Código-fonte 3 apresentamos um exemplo de utilização da ferramenta ShakerJS integrado ao GitHub Actions com seus parâmetros de entrada. Nele, é possível configurar a versão do sistema operacional, a quantidade de vezes que os testes com e sem estresse vão ser executados, o framework de execução dos testes (Jest ou Karma), o comando que será utilizado para executar as atividades de teste e a pasta onde os registros vão ser salvos.

Considerando os projetos selecionados para identificação de FT, listados na Tabela 4, foi realizada a reexecução com auxílio da ferramenta ShakerJS. Conseguimos identificar *flaky tests* em 36% dos projetos utilizando essa ferramenta de estresse dos casos de teste. A quantidade de *flaky tests* por projeto foi apresentada na Tabela 5. Encontramos um total de 102 *flaky tests* em 12 projetos JavaScript. Todos os casos de testes identificados estão disponíveis no re-

Listagem 3 – Exemplo de utilização do ShakerJS integrado ao GitHub Actions.

```

1 name: Flaky Shaker
2 on:
3   push:
4     branches: [ main ]
5   pull_request:
6     branches: [ main ]
7   workflow_dispatch:
8 jobs:
9   build:
10    runs-on: ubuntu-latest
11    steps:
12      - uses: actions/checkout@v3
13      - name: Jest tests
14        uses: sorattorafa/shaker-js@main
15        with:
16          tool: jest
17          tests_command: yarn test:source
18          runs: 1
19          output_folder: project_name/output

```

Fonte: Autoria própria (2025).

positório ¹⁹. Neste repositório separamos os dados JavaScript em duas pastas: *rerun* e *review*. No caso da pasta *rerun* todos casos de testes estão separados por diretórios de cada projeto e possuem um JavaScript Object Notation (JSON) com todas as informações da execução e da falha intermitente.

Tabela 5 – Casos de testes em JavaScript rotulados por meio da reexecução.

Git	Versão	#FT	#NFT
vercel/next.js	99490de	11	1664
twbs/bootstrap	40c6d8a	2	739
facebook/react	45d61cf	32	1848
facebook/react-native	a1b318c	8	777
facebook/metro	4127714	12	165
remix-run/react-router	e6814d5	9	59
react-hook-form/react-hook-form	c91e816	1	703
vueComponent/ant-design-vue	12c9e0c	3	196
ant-design/ant-design	4f1ca4c	5	101
nativefier/nativefier	051622d	3	20
strapi/strapi	8917530	6	153
gatsbyjs/gatsby	dc82d92	7	5
moleculerjs/moleculer	275eda5	3	54

Fonte: Autoria própria (2025).

3.2 pré-processamento dos casos de teste

Após obter o código-fonte referente aos casos de teste podemos extrair informações relevantes para as abordagens de identificação de *flakiness*. Implementamos uma ferramenta que recebe como parâmetro o código-fonte a ser processado e retorna uma lista de palavras respeitando a gramática da linguagem utilizada no teste. Em seguida, são extraídas as seguintes

¹⁹ <https://github.com/sorattorafa/flaky-across-languages/tree/main/datasets/javascript/data/>

características de testes: seu valor de instabilidade (*flaky* ou não *flaky*), e a quantidade de vezes que as palavras se repetem em cada caso de teste. As palavras de cada caso de teste são representadas como tokens que possuem os seguintes atributos: valor, tipo, e quantidade de vezes que aquela palavra se repetiu no teste instável. O motivo da escolha destas características é conhecer as palavras que são mais comuns em *flaky tests*, utilizando modelos de aprendizado de máquina. O número de tokens para cada caso de teste varia: existem testes pequenos, com poucas palavras; e existem testes extensos (por exemplo, testes de sistema), com mais palavras. Neste sentido, é possível que os casos de *flaky tests* mais extensos tenham mais palavras consideradas instáveis. Para amenizar este problema selecionaremos casos de teste com o número de linhas semelhantes para buscar balancear o tamanho dos casos de teste.

Na Tabela 6 as linhas são os *flaky tests*, onde cada teste é representado por sua respectiva lista de palavras do código-fonte. As colunas são os termos presentes em *flaky tests*. O conteúdo da tabela refere-se à quantidade de vezes que os termos de cada coluna se repetiram em cada caso de teste. Neste formato, podemos quantificar a repetição dos termos entre *flaky tests*. A estrutura deste conjunto de dados é apresentada na Tabela 6.

Tabela 6 – Tokens presentes em FT

Teste Instável (lista de tokens)	1º token do 1º FT	2º token do 1º FT	...	token (N) do FT (N)
1º FT	3	6	...	1
2º FT	0	0	...	0
⋮	3	9	...	0
Nº FT	3	9	...	0

Fonte: Autoria própria (2025).

Seguindo a lógica apresentada na Tabela 6, a construção de tokens tem dimensão proporcional à variedade e tamanho do conteúdo apresentado no código-fonte de *flaky tests*. Se o conjunto tiver testes extensos, ou muitos testes, conseqüentemente o número de termos será maior (colunas da Tabela 6). Portanto, as colunas dependem das linhas nesta tabela.

3.3 Extração de padrões

Os métodos utilizados são replicados dos trabalhos anteriores sobre o vocabulários instáveis (Pinto *et al.*, 2020; Camara *et al.*, 2021b; Haben *et al.*, 2021). Neles, estão presentes os modelos de aprendizado de máquina e métricas de ganho de informação entre variáveis.

3.3.1 Modelos de classificação

Neste trabalho são replicados os classificadores comuns no contexto de engenharia de software, tais como: Random Forest (RF), Decision Tree (DT), Naive Bayes (NB), Support Vector Machine (SVM), K-Nearest Neighbors (K-NN), Logistic Regression (LR), Linear Discriminant Analysis (LDA) (Souza *et al.*, 2014; Pinto *et al.*, 2020; Lampel *et al.*, 2021). A im-

plementação dos modelos de classificação é realizada utilizando a linguagem Python seguindo duas etapas: o carregamento dos conjuntos de dados com tokens de testes normais e instáveis utilizando a biblioteca pandas, e a classificação destes dados utilizando a biblioteca sklearn. Para definir um ambiente de classificação avaliamos diferentes abordagens de seleção para conjuntos de dados de testes normais e instáveis. Por exemplo: separando em 20% para teste e 80% para treinamento, validação cruzada em k partições, entre outras abordagens. Isto faz com que todos os dados do *dataset* possam ser usados para avaliar o sistema.

Em seguida, serão obtidas as seguintes métricas:

- **Precisão (Precision):** Esta métrica é definida como o número total de verdadeiros positivos (TP) dividido pelo número total de instâncias que foram classificadas como positivas, ou seja, TP dividido pela soma de TP e falsos positivos (FP). A precisão é um indicador importante quando o custo de uma classificação falsa positiva é alto.

$$\text{Precisão} = \frac{TP}{TP + FP} \quad (2)$$

- **Recall:** O recall, também conhecido como sensibilidade ou verdadeira taxa positiva, é calculado dividindo o número de testes flaky corretamente classificados pelo número total de testes flaky reais no conjunto de teste. O recall é crucial em contextos onde é vital identificar a maior quantidade possível de casos positivos, mesmo que isso signifique aceitar um aumento em falsos positivos.

$$\text{Recall} = \frac{TP}{TP + FN} \quad (3)$$

- **F1-score:** Esta métrica é a média harmônica entre precisão e recall, sendo uma forma útil de capturar tanto falsos positivos quanto falsos negativos em um único valor. O F1-score é particularmente benéfico quando buscamos um equilíbrio entre precisão e recall.

$$F1 = 2 \frac{\text{Precisão} \cdot \text{Recall}}{\text{Precisão} + \text{Recall}} \quad (4)$$

- **MCC (Matthews Correlation Coefficient):** O MCC é uma medida que considera todas as quatro classes de uma matriz de confusão — verdadeiros positivos, falsos positivos, verdadeiros negativos e falsos negativos. Essa métrica fornece uma avaliação global da performance, sendo especialmente útil para classes desbalanceadas.

$$MCC = \frac{(TP \cdot TN) - (FP \cdot FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \quad (5)$$

- **AUC (Area Under the Curve):** Esta métrica representa a área sob a curva ROC (Receiver Operating Characteristic), que plota a taxa de verdadeiros positivos (TPR) contra a taxa de falsos positivos (FPR) em diferentes limiares de classificação. A AUC fornece uma única métrica que resume a performance do classificador em todos os limiares possíveis.

$$AUC = \int_0^1 TPR dFPR \quad (6)$$

Estudos anteriores avaliaram o *F1-score* pois o foco é o reconhecimento de *flaky* e não da classe *non-flaky*.

No conjunto de dados JavaScript, utilizamos o processo de construção de tokens específico para a linguagem. Este processo obtém o valor e o tipo de cada palavra presente no código-fonte. Ele também é responsável por remover palavras desnecessárias, por exemplo, pontuadores. A ferramenta para converter um código-fonte JavaScript em uma lista de tokens (com valor e tipo da variável) está disponível no link ²⁰. Utilizamos ela para obter os tokens das 246 instâncias de FT e 6481 instâncias de NFT apresentados na Tabela 9.

Também construímos uma ferramenta para obtenção dos tokens em outras linguagens disponível no link ²¹. Utilizamos essa ferramenta para obter os tokens dos conjuntos de código-fonte Python e Java. Obtivemos os tokens de 14661 instâncias de NFT, e 1392 instâncias de FT em Java; e também 908 instâncias de FT e mais de 20 mil instâncias de NFT em Python.

A principal diferença dos dados utilizados neste experimento, além da linguagem utilizada, é que nos trabalhos de (Pinto *et al.*, 2020) e (Haben *et al.*, 2021) os dados rotulados como NFT não foram identificados utilizando a reexecução. Somente as instâncias de JavaScript obtidas neste trabalho são confiavelmente rotuladas como NFT, pois utilizamos o resultado de diversas execuções para isto.

A construção dos modelos utilizando diferentes dados de treinamento e teste é realizada no arquivo ²². Este arquivo Python realiza a carga e a análise de dois conjuntos de dados distintos, contendo casos de teste rotulados como *flaky* ou *não flaky*. Cada conjunto pode conter a label designada como *is_flaky* (utilizado em nosso conjunto JavaScript) ou *Label* (utilizado por trabalhos anteriores em Java e Python), e o código implementa uma verificação para identificar qual rótulo está presente. O código utiliza várias bibliotecas para processamento de dados e modelagem, incluindo: *pandas* para manipulação de dados, *numpy* para operações numéricas, *sklearn* para aplicação de algoritmos de machine learning, como RF, DT, NB, entre outros, e *matplotlib* para visualização de dados. O código é estruturado em funções claras que realizam as seguintes tarefas: *loadFilesFromPath(folder_path)* percorre um diretório e carrega todos os arquivos JSON, agregando-os em um *DataFrame* do *pandas*; *displayScores(scores)*

²⁰ <https://github.com/sorattorafa/flaky-across-languages/blob/main/scripts/tokenizer/javascript/main.js>

²¹ https://github.com/sorattorafa/flaky-across-languages/blob/main/scripts/tokenizer/code_tokenizer.py

²² https://github.com/sorattorafa/flaky-across-languages/blob/main/scripts/vocabulary/languages_model.py

exibe as métricas de desempenho, incluindo *precision*, *recall*, MCC, e AUC de forma resumida; *checkUsage()* verifica os argumentos passados ao programa para garantir que os diretórios especificados existem; e *getTrainAndTestData(train_data, test_data)* processa os conjuntos de dados de treinamento e teste, realizando a extração de características e a criação de vetores para o treinamento dos modelos. A função principal *main()* orquestra o fluxo do programa, incluindo: carregamento dos dados de treinamento e teste, aplicação de amostragem opcional nos dados, preparação dos dados para treinamento e teste, treinamento dos classificadores e avaliação através das métricas definidas. O código permite executar um modelo preditivo para identificar casos *flaky* em um conjunto de dados, com a flexibilidade de usar diferentes classificadores, e fornece relatórios estruturados das métricas de desempenho dos modelos treinados.

3.4 Pós-processamento

A análise do ganho de informação é utilizada para estabelecer o vocabulário para casos de teste instáveis. Essa etapa é executada após a validação dos modelos de classificação. Ela é útil para obter as palavras que mais impactam nos modelos de classificação. As palavras com o maior ganho de informação para predição de instabilidade nos conjuntos fazem parte do vocabulário instável. Podemos construir o vocabulário à partir dos dados de um único projeto, de um conjunto de projetos ou até mesmo de uma linguagem de programação. Para construir o vocabulário instável é necessário aplicar a etapa de ganho de informação dos termos em relação à instabilidade (Pinto *et al.*, 2020). O ganho de informação (Information Gain (IG)) é uma medida da redução na incerteza de uma variável aleatória Y após a observação de outra variável aleatória X . Ele é calculado pela diferença na entropia antes e depois da observação (Quinlan, 1986). A fórmula para o ganho de informação é dada por:

$$\text{Ganho de Informação}(X; Y) = H(Y) - H(Y|X) \quad (7)$$

onde $H(Y)$ é a entropia inicial da variável Y , que é calculada como:

$$H(Y) = - \sum_{y \in Y} P(Y = y) \log P(Y = y) \quad (8)$$

$H(Y|X)$ é a entropia da variável Y dada a variável X , que é calculada como:

$$H(Y|X) = - \sum_{x \in X} P(X = x) \sum_{y \in Y} P(Y = y|X = x) \log P(Y = y|X = x) \quad (9)$$

Aqui, $P(Y = y)$ é a probabilidade de Y assumir o valor y , e $P(Y = y|X = x)$ é a probabilidade condicional de Y assumir o valor y dado que X assume o valor x .

Assim, o ganho de informação quantifica a informação que a variável X fornece sobre a variável Y , ajudando a entender como a observação de uma variável pode reduzir a incerteza sobre a outra. Em relação a análise dos resultados, as palavras que geram um maior ganho de informação indicam uma maior capacidade de previsão sobre as instabilidades em testes. Com isso, pode-se determinar quais palavras são mais relevantes para prever as instabilidades durante a construção e manutenção dos testes. Essa abordagem permite identificar as palavras mais informativas, ajudando a direcionar esforços para compreender as causas das instabilidades em testes.

3.5 Utilização do conhecimento

Para conhecer a relação entre as palavras e a instabilidade em testes não basta verificar a repetição das palavras. Trabalhos anteriores utilizaram o ganho de informação para isto (Pinto *et al.*, 2020; Camara *et al.*, 2021b).

Utilizamos a função ‘mutual_info_classif’ do módulo ‘sklearn.feature_selection’ para calcular a informação mútua entre cada característica (feature) e a variável alvo (target). Informação mútua é uma medida da dependência entre duas variáveis. Em termos simples, ela quantifica a quantidade de informação que uma variável fornece sobre a outra. Neste caso, queremos ver a quantidade de informação que as palavras fornecem para prever a instabilidades em testes. É importante utilizar essa métrica ao invés do número de repetições de determinada palavra em FT para garantir que a palavra está de fato relacionada com a causa raiz do FT.

Avaliar o vocabulário de projetos e linguagens permite conhecer as causas raízes de *flaky tests* em diferentes cenários. Além da comparação das métricas e modelos de classificação, serão executadas as comparações com os resultados dos vocabulários. Verificar semelhanças e diferenças do vocabulário instável entre diferentes cenários permite avaliar os limites da abordagem em conjuntos mais amplos de dados. Neste sentido, a primeira parte da análise se concentra no vocabulário de projetos JavaScript com base nos dados obtidos neste trabalho. A segunda parte da análise se concentra na comparação do vocabulário entre linguagens e projetos.

Os resultados de cada classificador são armazenados para análise posterior, permitindo a identificação das melhores e piores configurações. Por fim, gráficos de calibração são gerados para demonstrar a confiabilidade das previsões dos modelos, oferecendo *insights* adicionais sobre o desempenho do aprendizado de máquina implementado.

3.6 Cenários propostos para responder às questões de pesquisa

O principal objetivo deste trabalho é investigar a técnica de vocabulário para predição de instabilidades entre projetos de diferentes domínios e linguagens de programação. Trabalhos anteriores mostram bons resultados nas linguagens Java, Python e JavaScript (Pinto *et al.*, 2020;

Camara *et al.*, 2021b; Haben *et al.*, 2021; Soratto; Silva, 2023). Porém, ainda não existem trabalhos verificando a possibilidade de generalização de um vocabulário instável entre diferentes projetos e linguagens. A utilização dos modelos de predição em um cenário real de teste de software pode ocorrer durante a integração contínua (CI), onde os testes rotulados como instáveis são ignorados temporariamente para um estudo profundo ser realizado após a entrega do produto (Haben *et al.*, 2021). É necessário que os modelos construídos possuam um desempenho considerável, para evitar que possíveis falhas intermitentes sejam ignoradas, ou que falsos positivos gerem alerta desnecessário. Neste sentido, é interessante conhecer a precisão dos modelos de classificação em um conjunto mais amplo de dados. Para conhecer melhor a técnica de vocabulário e responder às questões de pesquisa, propomos os experimentos nos seguintes cenários:

Cenário proposto para QP1

QP1: Com que qualidade podemos prever instabilidades em casos de teste implementados na linguagem JavaScript utilizando vocabulário?

Cenário: Coleta de dados (código-fonte de *flaky tests* e não instáveis em JavaScript);

Cenário: Predição de instabilidade entre projetos JavaScript;

Cenário proposto para QP2

QP2: Quais são os principais termos presentes em códigos de testes associados à instabilidade na linguagem JavaScript?

Cenário: Análise das palavras com maior ganho de informação para predição de instabilidade na linguagem JavaScript.

Cenário proposto para QP3

QP3: Com que qualidade podemos prever instabilidades em casos de teste entre diferentes projetos e linguagens utilizando vocabulário?

Cenário: Predição de instabilidade entre projetos JavaScript, Java e Python;

Cenário proposto para QP4

QP4: Quais são os principais termos presentes em códigos de testes associados à instabilidade entre diferentes linguagens?

Cenário: Análise das palavras com maior ganho de informação para predição de instabilidade entre linguagens;

4 RESULTADOS

4.1 QP1: Com que qualidade podemos prever instabilidades em casos de teste implementados na linguagem JavaScript utilizando vocabulário?

Resposta para QP1

Podemos prever instabilidades em casos de teste implementados na linguagem JavaScript utilizando o vocabulário instável com uma qualidade razoável. Em um cenário intra-projeto os modelos de predição possuem um F1 superior a 80%. Em um cenário entre projetos, com uma natureza de instabilidades diversificada, o resultado do F1 foi reduzido para 62%. Um método mais robusto para obtenção dos dados utilizados (*rerun*) também pode reduzir os resultados dos modelos e trazer um cenário mais real para o treinamento dos mesmos.

Modelos baseados em vocabulário apresentaram bons resultados para a linguagem JavaScript em trabalhos anteriores (Soratto, 2022; Soratto; Silva, 2023). Porém, existe limitação quanto ao conjunto de dados daqueles trabalhos (que usaram exclusivamente testes classificados como instáveis a partir de observação empírica). Os testes rotulados como não instáveis dependem da reexecução para garantir a assertividade dos modelos de precisão. Então, verificamos se os resultados obtidos nos trabalhos anteriores se mantêm para os dados obtidos por reexecução. Camara *et al.* (2021b) apresentaram a limitação do desempenho dos modelos entre projetos implementados em Java. Também verificamos essa limitação dos modelos avaliados entre diferentes projetos JavaScript. Para responder essa questão de pesquisa, as seguintes atividades foram executadas:

1. Coleta de dados:
 - a) Coleta do código-fonte de casos de testes rotulados como FT utilizando técnicas estáticas, dinâmicas e mistas. Isto foi feito extraindo *features* de casos de testes conhecidos por conjuntos de trabalhos anteriores, e executando os casos de testes em JavaScript para criação de um novo conjunto.
 - b) Coleta do Código-fonte de casos de testes rotulados como NFT: utilizando técnicas dinâmicas (*rerun*).
2. Teste do modelo com dados de um projeto JavaScript obtidos por meio da execução;
3. Teste de 6 modelos de classificação de *flaky tests* (LDA,K-NN,DT,LR,RF,NB) utilizando dados de vários projetos JavaScript obtidos por meio da execução e revisão de casos de testes.

Em trabalhos anteriores sobre predição de *flaky tests* em JavaScript utilizando o vocabulário foram utilizados testes obtidos por meio da revisão de *commits* (Soratto, 2022). Neste trabalho realizamos a execução dos casos de teste para verificar um conjunto diferente de dados. Aplicamos os modelos nos dois tipos de conjuntos: obtidos pela execução e revisão. Verificamos que alguns termos obtidos pela revisão podem não se fazer presente no conjunto obtido pela execução, o que pode impactar nos resultados dos modelos. Por exemplo, o termo *it.skip* não poderia ter sido encontrado pelo conjunto de reexecução, visto que ele ignora o teste em questão. Mas como também consideramos um conjunto obtido por revisão, o termo existe no conjunto. Na Tabela 18, onde os dados são obtidos somente por técnicas dinâmicas, a maioria dos *flaky tests* possuem em sua declaração as palavras *it* e *describe*. Por este motivo aplicamos os modelos em diferentes conjuntos.

4.1.1 Teste dos modelos com dados de um projeto JavaScript obtidos por meio da execução

O fluxo completo de aprendizado de máquina foi desenvolvido no arquivo *javascript_model_default.py*¹ com o objetivo de prever a ocorrência de *flaky tests* com base em dados de testes armazenados em arquivos JSON. Este processo pode ser dividido nas seguintes etapas principais:

Primeiramente, o programa começa com a verificação dos parâmetros de entrada, garantindo que o usuário forneça um diretório válido onde os arquivos de dados estão localizados. Após essa validação, os dados são carregados utilizando a biblioteca *pandas*. Cada arquivo JSON é lido e concatenado em um único *DataFrame*, permitindo a manipulação e análise dos dados como um todo.

Em seguida, o programa identifica qual coluna representa o tipo dos testes (*flaky* e *non-flaky*) e divide os dados em duas subclasses: *FT* para *flaky tests* (denotados como 1) e *NFT* para *non-flaky tests* (denotados como 0). Para lidar com um possível desbalanceamento entre essas classes, o programa permite a aplicação da técnica SMOTE (Synthetic Minority Over-sampling Technique), que cria cópias sintéticas da classe minoritária, embora isso esteja opcional no código.

Após a segmentação dos dados, o próximo passo envolve a conversão do texto dos testes para uma representação numérica. Isso é realizado através do uso do *CountVectorizer*, que transforma sentenças em vetores de contagem de termos. A matriz resultante é então utilizada como entrada para o treinamento dos classificadores, enquanto as respectivas classes (*FT* ou *NFT*) são armazenadas na variável *y*.

O programa então divide os dados em conjuntos de treinamento e teste por meio da função *train_test_split*. O uso do argumento *stratify* assegura que a distribuição original das

¹ https://github.com/sorattorafa/flaky-across-languages/blob/main/scripts/vocabulary/javascript_model_default.py

classes seja preservada em ambos os conjuntos, o que é crucial para uma avaliação justa do modelo.

Uma lista de classificadores é definida, incluindo *RandomForestClassifier*, *DecisionTreeClassifier*, *GaussianNB*, *KNeighborsClassifier*, *LogisticRegression* e *LinearDiscriminantAnalysis*. O programa itera sobre cada classificador, onde cada um é ajustado ao conjunto de treinamento utilizando o método *fit*. As previsões são feitas em ambos os conjuntos (treinamento e teste), e as probabilidades são computadas para avaliação futura.

As métricas de desempenho, como *precision*, *recall*, F1, AUC e MCC, são calculadas após a execução dos classificadores. Essas métricas permitem uma comparação detalhada do desempenho de cada classificador. As matrizes de confusão são geradas e plotadas, apresentando uma visualização clara dos resultados.

Utilizamos os dados apresentados na Tabela 5 para verificar a precisão dos classificadores de FT em projetos JavaScript. Selecionamos os projetos que apresentaram 3 ou mais instâncias de FT durante as execuções e testamos os modelos citados anteriormente para cada projeto. Também realizamos a execução dos modelos com todos os dados entre projetos. Conforme apresenta a Tabela 7, podemos observar que os modelos também apresentam bons resultados para os dados obtidos pela reexecução em JavaScript. Nesta tabela, apresentamos o resultado do classificador que obteve o melhor desempenho para cada projeto.

Tabela 7 – Métricas do Classificadores em Projetos JavaScript

Projeto (Git)	#FT	#NFT	Precisão	Recall	F1	MCC	AUC
ant-design/ant-design	5	101	1.00	0.83	0.90	0.91	0.92
facebook/react	32	1848	1.0	0.50	0.66	0.70	0.75
facebook/react-native	8	777	1.0	0.75	0.86	0.86	0.88
facebook/metro	12	165	0.83	0.76	0.78	0.78	0.84
gatsbyjs/gatsby	5	10	1.00	0.83	0.90	0.85	0.92
moleculerjs/moleculer	3	54	1.00	1.00	1.00	1.00	1.00
nativefier/nativefier	3	20	1.00	0.75	0.83	0.84	0.88
remix-run/react-router	9	59	1.00	0.56	0.56	1.00	0.78
strapi/strapi	6	153	1.00	0.83	0.90	0.91	0.92
vercel/next.js	11	1664	1.0	0.50	0.66	0.70	0.75
Entre Projetos	102	6481	0.73	0.55	0.62	0.63	0.77

Fonte: Autoria própria (2025).

O projeto que obteve melhor desempenho foi o Moleculer, com a pontuação máxima do classificador. O pior resultado foi do projeto React, apresentando F1 de 0,44. O projeto com mais instâncias de NFT que obteve o melhor resultado foi o React-Native, com a precisão de 100% e F1 de 86%. O projeto com mais instâncias de FT que obteve o melhor resultado foi o Strapi, com precisão de 100% e F1 de 90%.

A maioria dos projetos apresentaram uma precisão de 100%, e uma boa pontuação de F1. Os dois projetos que apresentaram o melhor resultado de F1 foram: Moleculer (100%) e Ant-Desing (90%). Os projetos que apresentaram o pior resultado foram projetos relacionados com React.js: React, React-Router e Next.js. Mesmo realizando o balanceamento de das classes de *flaky tests* e *non flaky tests* verificamos que o ecossistema React possui testes variados, de

diversos componentes e funcionalidades. Isto pode afetar na *recall* do modelo e consequentemente no *f1-score*. o React-Native foi o único projeto do ecossistema React que obteve um *f1* razoável de 86%.

Na Tabela 7 verificamos que ao testar o modelo de classificação entre projetos a métrica de *recall* cai, mesmo com uma precisão razoável de 96%. Este problema já foi relatado anteriormente por (Camara *et al.*, 2021b) na linguagem Java e acontece pela variedade de projetos e categorias de testes utilizados no conjunto. Enquanto Camara *et al.* (2021b) obtiveram um *recall* de 48% em um cenário entre projetos Java, nós obtivemos o *recall* de 77% no cenário entre projetos JavaScript.

A variedade de palavras e contextos pode reduzir a qualidade dos classificadores mesmo dentro de um único projeto. Nestes casos, precisamos de mais relatos sobre FT para incrementar o conjunto de treinamento. Para isso, no próximo experimento utilizamos os FT obtidos também por meio de revisão de alterações dos desenvolvedores.

4.1.2 Teste de modelos com dados obtidos por meio da execução e revisão de casos de testes

Nesta configuração utilizaremos todos os dados coletados anteriormente: as instâncias de FT obtidas pela revisão e reexecução dos casos de teste, e as instâncias de NFT obtidas somente pela reexecução. Coletamos um total de **246** FT e **6481** NFT listados na Tabela 8. Todos os *non flaky tests* foram coletados após mais de 1500 execuções com o mesmo resultado estável. Já os *flaky tests* foram coletados de forma mista: pela reexecução e pela revisão de *commits* feitas por Soratto (2022).

Conforme apresentado na Tabela 8, temos um total de **6727** casos de testes rotulados em JavaScript, onde 246 estão rotulados como *flaky tests* e 6481 como *non-flaky tests*. Deste conjunto, separamos 20% para teste, resultando em **5381** instâncias para treinamento e **1346** para teste. O conjunto de treinamento fica com **197** FT e **5184** NFT; e o conjunto de teste com **49** FT, e **1297** NFT. Aplicamos esta configuração para diversos classificadores, cujos resultados são apresentados na Tabela 9.

Em relação aos resultados entre projetos da Tabela 7, o modelo com o conjunto ampliado de casos de teste, com resultados apresentados na Tabela 9, apresentou melhor desempenho. Isto ocorreu pela natureza dos FT incluídos. A melhor precisão foi apresentada pelo modelo K-NN com o valor de **0,96**. Na predição de FT é importante analisar a métrica de *recall*. Ele mede a capacidade do modelo de identificar corretamente todas as instâncias positivas (FT). Em um contexto de predição de falhas intermitentes, isso significa que o modelo deve ser capaz de identificar o máximo de FT reais possíveis. O modelo com melhor *recall* é o DT com *recall* de **0,77** e F1 de **0,83**. Em relação aos valores de *recall* dos modelos, percebemos uma baixa variação: o melhor resultado de **0,77** e o pior de **0,59**. E com exceção dos modelos LDA e NB todos os modelos apresentaram precisão superior a 90% e F1 superior a 76%. Os resultados

Tabela 8 – Flaky tests e non-flaky tests obtidos pela revisão e reexecução de projetos JavaScript.

Projeto	#FT (Revisão)	#FT (Reexecução)	#NFT (Reexecução)
vercel/next.js	1	11	1664
twbs/bootstrap	1	2	739
facebook/react	-	32	1848
facebook/react-native	-	8	777
facebook/metro	-	12	165
remix-run/react-router	-	9	59
react-hook-form/react-hook-form	-	1	703
vueComponent/ant-design-vue	-	3	196
ant-design/ant-design	-	5	101
nativesfier/nativesfier	-	3	20
strapi/strapi	-	6	153
gatsbyjs/gatsby	1	7	5
moleculerjs/moleculer	-	3	54
styleguidist/react-styleguidist	1	-	-
outline/outline	2	-	-
apollographql/react-apollo	1	-	-
react-cosmos/react-cosmos	1	-	-
HospitalRun/hospitalrun-frontend	1	-	-
gotify/server	6	-	-
Hacker0x01/react-datepicker	3	-	-
skbkontur/retail-ui	1	-	-
thaliproject/postcardapp	5	-	-
OfficeDev/office-ui-fabric-react	1	-	-
rjsf-team/react-jsonschema-form	1	-	-
palantir/blueprint	3	-	-
plotly/plotly.js	32	-	-
stream-labs/streamlabs-obs	2	-	-
JetBrains/ring-ui	2	-	-
NativeScript/nativescript-angular	1	-	-
angular/components	10	-	-
angular/angular	6	-	-
coralproject/talk	1	-	-
pinterest/gestalt	4	-	-
uber/baseweb	7	-	-
jhipster/generator-jhipster	3	-	-
nfl/react-helmet	1	-	-
influxdata/influxdb	22	-	-
enzymejs/enzyme	1	-	-
mobxjs/mobx	2	-	-
preactjs/preact	17	-	-
Total:	144	102	6481

Fonte: Autoria própria (2025).

deste experimento confirmam ser possível ter um modelo de classificação de FT para projetos JavaScript.

Na Tabela 9 é possível verificar que ao misturar as instâncias de FT que foram obtidas por técnicas estáticas com as obtidas por técnicas dinâmicas, os resultados dos modelos são afetados. O trabalho de Soratto (2022) apresenta a melhor métrica de precisão (98%) para o modelo LR, em nosso trabalho é o K-NN (96%). Porém, nosso maior *recall* foi de apenas 77% para o modelo DT, enquanto no trabalho anterior o maior valor foi de 98% para o modelo LR. O mesmo aconteceu para métrica de F1: enquanto no trabalho original o valor foi 98% para

Tabela 9 – Resultados dos classificadores em JavaScript

Classificador	Conjunto		Precisão	Recall	F1	MCC	AUC
	#FT	#NFT					
Linear Discriminant Analysis	246	6481	0.31	0.59	0.41	0.40	0.77
K Nearest Neighbours	246	6481	0.96	0.65	0.78	0.78	0.82
Decision Tree	246	6481	0.90	0.77	0.83	0.83	0.88
Logistic Regression	246	6481	0.91	0.69	0.79	0.79	0.84
Random Forest	246	6481	0.96	0.63	0.76	0.77	0.81
Naive Bayes	246	6481	0.53	0.75	0.62	0.62	0.86

Fonte: Autoria própria (2025).

o modelo LR, neste trabalho o modelo DT obteve o valor máximo de 83%. Isto acontece pela variedade maior de instabilidades em nosso conjunto, e a validação robusta das classes pelo *rerun*.

Na Figura 4, apresentamos uma comparação dos modelos criados na Tabela 9. Essa comparação é feita utilizando a Reliability Curve (RC) (curva de confiabilidade), uma ferramenta usada em aprendizado de máquina e estatística para avaliar a calibração de modelos de classificação probabilística. A curva de confiabilidade compara as probabilidades previstas pelo modelo com as frequências reais observadas. Se a curva de confiabilidade está próxima da linha $y = x$, isso indica que o modelo está bem calibrado. Se a curva está longe da linha $y = x$, isso indica que o modelo não está bem calibrado. Se a curva está consistentemente acima da linha $y = x$, o modelo está superestimando as probabilidades. Se a curva está consistentemente abaixo da linha $y = x$, o modelo está subestimando as probabilidades.

A análise dos classificadores foi realizada por meio de gráficos de calibração, que avaliam a precisão das previsões probabilísticas na Figura 4. Também são apresentados os números deste gráfico na Tabela 10.

Tabela 10 – Resultados de Calibração dos Classificadores

Classificador	Valor Previsto Médio	Fração de Positivos
Random Forest	0.1	0.08
Decision Tree	0.2	0.15
GaussianNB	0.3	0.28
KNN	0.4	0.35
Logistic Regression	0.5	0.52
Linear Discriminant Analysis	0.6	0.60

Fonte: Autoria própria (2025).

A seguir, são apresentadas as conclusões com base no gráfico analisado.

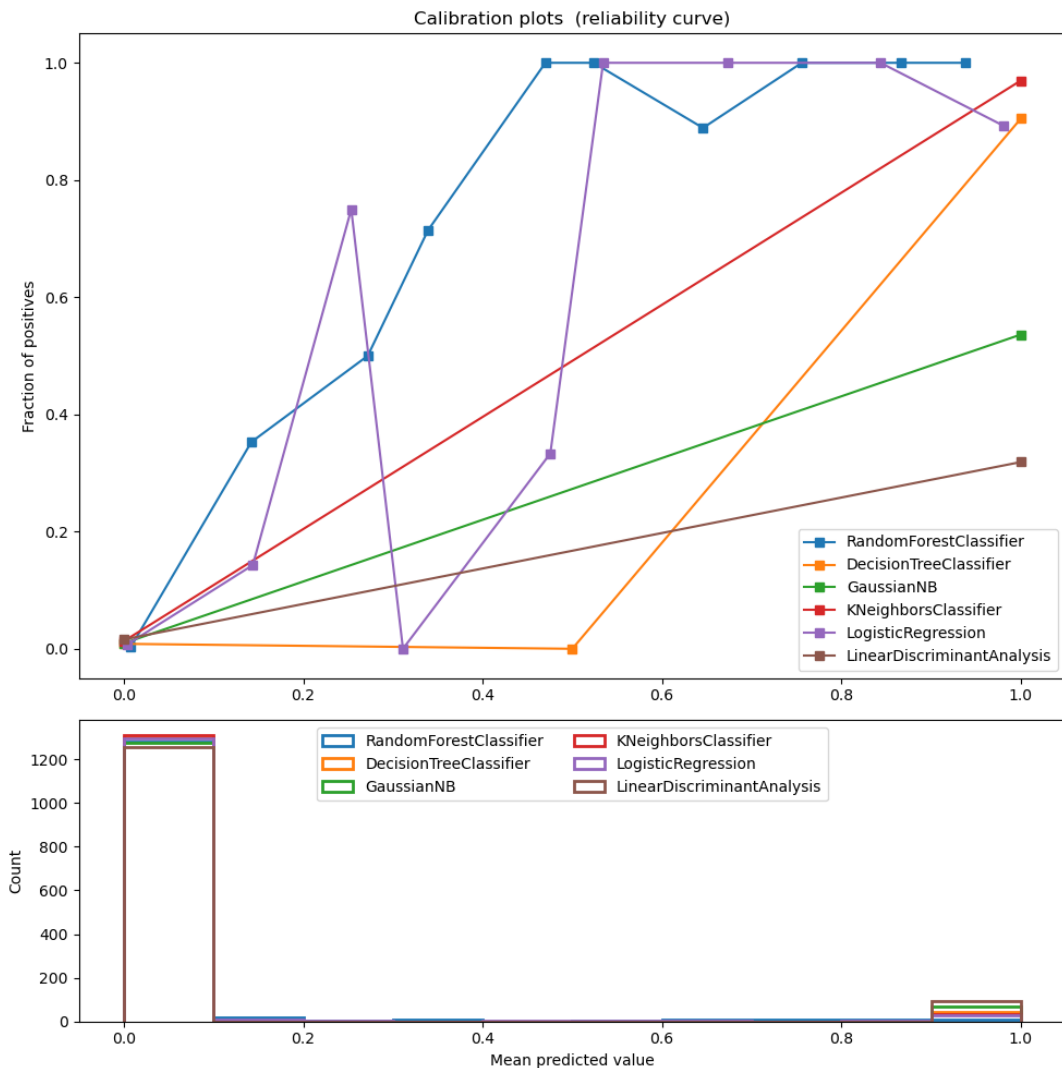
Entre os classificadores avaliados, o LDA apresentou o melhor desempenho, conforme indicado pela proximidade de sua curva de calibração com a linha de referência ideal (linha diagonal). Isso demonstra uma boa calibração, onde as probabilidades previstas refletem com precisão as proporções reais de eventos positivos.

O LR também apresentou bom desempenho, com a curva de calibração próxima à linha ideal em quase todos os intervalos de previsão.

Em contrapartida, o DT e o **rf** mostraram uma calibração mais pobre, com desvios significativos da linha ideal em vários pontos. Isso sugere a necessidade de ajustes adicionais ou de calibração dos modelos para melhorar a precisão das previsões probabilísticas.

Conclui-se que, para este conjunto de dados e métricas analisadas, o **lda** é o classificador mais confiável em termos de calibração de probabilidades.

Figura 4 – Comparação dos Classificadores em JavaScript



Fonte: Autoria própria (2025).

Na Figura 5 e na Figura 6 são apresentados os melhores resultados de matriz de confusão. Entre as **1346** instâncias de teste, o DT errou somente **15** vezes e o K-NN errou somente **18** vezes. O modelo DT foi o que mais encontrou FT, um total de **38** de 49. Na Figura 7 e na Figura 8 são apresentados os resultados de RF e LR respectivamente. Ambos apresentaram resultados bons, porém uma quantidade significativa de falsos negativos (Falsos Negativos, do inglês *False Negative* (FN)). O modelo NB não apresentou um bom resultado na Figura 9, pois mesmo acertando **37** previsões de FT, ele apresentou **32** falsos alarmes de FT. O mesmo aconteceu para o modelo LDA na Figura 10, onde ele apresentou **62** falsos alarmes.

Figura 5 – Matriz de confusão do Modelo DT

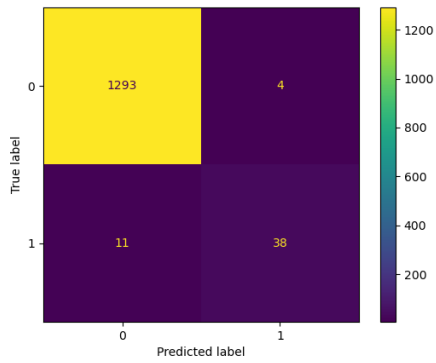


Figura 6 – Matriz de confusão do Modelo K-NN

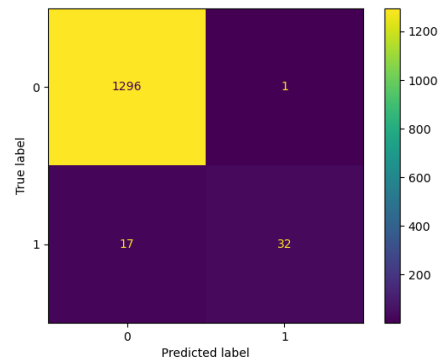


Figura 7 – Matriz de confusão do Modelo RF

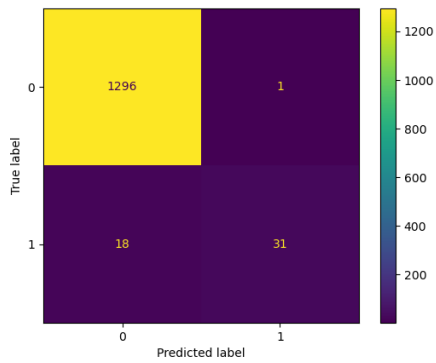


Figura 8 – Matriz de confusão do Modelo LR

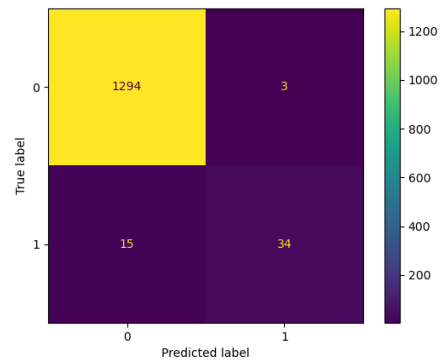


Figura 9 – Matriz de confusão do Modelo NB

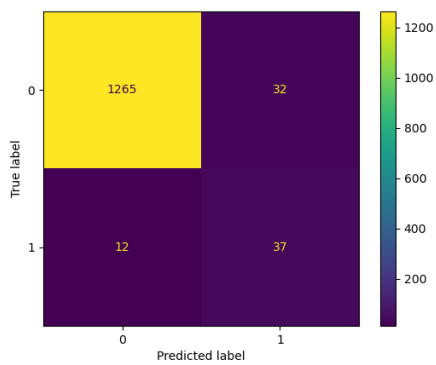
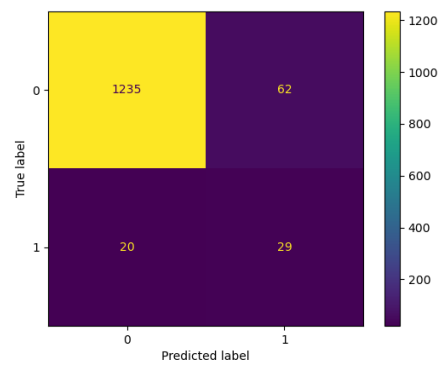


Figura 10 – Matriz de confusão do Modelo LDA



4.2 QP2: quais são os principais termos presentes em códigos de testes associados à instabilidade na linguagem JavaScript?

Resposta para QP2

Os principais termos associados à instabilidade em testes na linguagem JavaScript estão distribuídos em várias categorias. No tratamento de erros, termos como **throw**, **catch** e **fail** são comuns, indicando instabilidades relacionadas a falhas na manipulação de exceções. No contexto do Hypertext Markup Language (HTML), termos como **container**, **querySelector** e **click** aparecem frequentemente, evidenciando desafios na manipulação do DOM. A espera assíncrona, com termos como **await** e **promise**, também é uma fonte significativa de instabilidade devido a problemas de sincronização. A presença de **pathname**, **writeFile** e **readFile** na categoria de arquivos sugere dificuldades com manipulação de sistemas de arquivos. Testes usando expressões regulares e **frameworks** de teste que exibem termos como **toBe** e **toMatch** indicam complexidades inerentes à verificação de saídas esperadas contra resultados efetivos. Elementos relacionados a dados, expressos através de termos como **data** e **type**, também contribuem para a instabilidade dos testes, refletindo questões na formatação e manipulação de informações.

Com os projetos JavaScript apresentados na Tabela 5, obtivemos as top-20 palavras com maior ganho de informação para cada projeto e para todos os projetos juntos (Tabela 18). O vocabulário interno de cada projeto possui palavras que tiveram maior ganho de informação para predição de *flaky tests*. Visto que cada projeto pode ter um vocabulário específico, e temos mais de 10 projetos, apresentamos apenas os vocabulários específicos de projetos que são baseados em React, nas Tabelas 11 à 15.

O valor de ganho de informação é totalmente dependente da quantidade de dados sobre FT e NFT disponíveis. Normalmente, um projeto possui mais testes estáveis do que instáveis. Por exemplo, na Tabela 11 o termo com maior ganho de informação para predição de instabilidades no React é o *toBe*, com o valor de apenas 0,013. Este valor é baixo, pois ele aparece em 752 NFT e em apenas em 11 FT. Todos os outros tokens também possuem o valor menor que este, pois neste projeto existem muitas instâncias de NFT. Observe que esse termo (*toBe*) é bem comum devido ao fato de ser utilizado nas assertivas (faz parte da linguagem do *framework* de teste). Porém, a linguagem também possui outros termos para fazer assertivas (por exemplo, *toEqual*). Com isso, podemos concluir que essa assertiva é a que mais apresenta instabilidades no React. Portanto, mesmo com ganho de informação baixo, os *tokens* com melhor ganho de informação de um projeto são aqueles que auxiliam os modelos de classificação a performa-

rem bem. Com esses termos podemos analisar os elementos do código-fonte que mais estão relacionados com instabilidades no conjunto.

Existem palavras que se repetem muito, mas não estão necessariamente relacionadas com a falha intermitente. Por exemplo, no Código-fonte 2 temos um FT onde a palavra que causa a instabilidade do caso de teste (*now*) aparece somente uma vez no mesmo. Utilizando o conjunto de dados obtidos pela reexecução, verificamos as métricas de repetição e ganho de informação por palavra e descobrimos que o termo *now* é o sexto que mais se repete nas instâncias de FT do projeto em questão (Moleculer). Porém, conforme a Tabela 16 apresenta, ela é a primeira palavra do ranking de ganho de informação. Essa métrica apresenta quais palavras possuem influência sobre os modelos de classificação.

Na Tabela 11 verificamos que os termos que aparecem nos *flaky tests* do projeto React vão desde elementos relacionados ao conteúdo da página HTML até componentes específicos do projeto, como, por exemplo, as palavras: *suspend*, *fallback*, *loading*, *suspense*, *innerHTML*, *finalhtml*, *text*. Termos referentes a erros tiveram um alto ganho de informação para predição de *flaky tests* em React (*throw*, *error*). No React, para construir um componente, na maioria das vezes os termos *function* e *return* são utilizados. Esses termos apresentam alto ganho de informação neste projeto e também no trabalho de Soratto e Silva (2023), onde são utilizados dados do projeto Next.js (um framework que utiliza o React como base). De acordo com a documentação do React² todo componente é uma função que retorna um elemento HTML. Portanto, podemos notar uma categoria de instabilidade representada pelos termos *function* e *return*: a criação e utilização de componentes visuais no React.

Tabela 11 – Ganho de informação das palavras para predição de FT no projeto React

Posição	Token	IG	Ocorrências	#FT	#NFT
1	tobe	0.013	763	11	752
2	scheduler	0.012	149	14	135
3	throw	0.012	138	12	126
4	container	0.011	822	11	811
5	return	0.011	1015	29	986
6	expect	0.010	1624	24	1600
7	suspend	0.010	43	8	35
8	log	0.010	181	13	168
9	innerHTML	0.009	144	12	132
10	text	0.009	269	12	257
11	hello	0.009	266	8	258
12	finalhtml	0.008	4	4	0
13	oops	0.008	4	4	0
14	function	0.008	758	24	734
15	loading	0.008	135	8	127
16	fallback	0.007	167	8	159
17	suspense	0.007	173	8	165
18	refreshreg	0.007	24	6	18
19	patch	0.007	20	4	16
20	error	0.007	155	8	147

Fonte: Autoria própria (2025).

² <https://react.dev/reference/react>

É comum que diferentes projetos possuam semelhanças e diferenças entre seus vocabulários instáveis. Por exemplo, na Tabela 12 verificamos os termos com melhor resultado para essa métrica em um projeto baseado no React, chamado Ant-Design. Neste projeto o termo com maior ganho de informação é o *const* com valor de 0,19. O termo *toBe* manteve seu bom desempenho neste projeto, sendo o segundo lugar também com valor de 0,19. Outros termos referentes ao HTML foram encontrados neste projeto, são eles: *htmldivelement*, *click*, *wrapper*, *queryselector*, *component*, *container*, *key*. Também foram apresentados termos relacionados com a espera assíncrona: *async*, *await*, *waitfaketimer*.

Tabela 12 – Ganho de informação das palavras para predição de FT no projeto Ant-Design

Posição	Token	IG	Ocorrências	#FT	#NFT
1	const	0.190	102	5	97
2	tobe	0.190	25	5	20
3	string	0.190	11	5	6
4	waitfaketimer	0.177	9	5	4
5	if	0.177	8	5	3
6	queryselectorall	0.177	42	5	37
7	test	0.172	10	5	5
8	component	0.172	7	5	2
9	wrapper	0.166	13	5	8
10	expect	0.166	106	5	101
11	queryselector	0.166	47	5	42
12	unmount	0.164	7	5	2
13	container	0.160	79	5	74
14	async	0.158	16	5	11
15	ant	0.157	50	5	45
16	htmldivelement	0.154	7	5	2
17	tobefalsy	0.150	13	5	8
18	key	0.145	15	5	10
19	await	0.141	13	5	8
20	click	0.140	33	4	29

Fonte: Autoria própria (2025).

Na Tabela 13 apresentamos o ganho de informação das palavras de outro projeto baseado no React, o React-Native. O termo com a melhor métrica neste projeto é o *tomatchsnapshot* com valor de 0.03. Em geral, os termos instáveis deste projeto estão mais relacionados com a interação de arquivos, são eles: *file*, *source*, *exclude*, *readfile*, *sourcefiles*, *files_with_known_errors*. Verificamos a implementação de alguns casos de testes que possuem esses termos no projeto e podemos confirmar a existência de elementos que comumente estão associados com causas raízes de instabilidades. Por exemplo, no primeiro teste do arquivo ³ de *flaky tests* do React-Native já encontramos a presença destes termos. Neste arquivo, existem todas as informações dos *flaky tests*. Esse em questão, fica no arquivo de teste de uma API ⁴, começa na linha 68 e termina na linha 111. Nele podemos visualizar diversas possíveis causas raízes de

³ <https://github.com/sorattorafa/flaky-across-languages/blob/main/datasets/javascript/data/rerun/mix/react-related/react-native/react-native-e9d9de1a1e6a9d0b4636d5da3c123be02f637311-2308-reruns-tokens.json>

⁴ https://github.com/facebook/react-native/blob/e9d9de1a1e6a9d0b4636d5da3c123be02f637311/packages/react-native/Libraries/__tests__/public-api-test.js

instabilidades: interação com o sistema operacional que executa o teste, execução de serviços externos, operações de entrada e saída, tarefas com espera assíncrona. Outro ponto, que esse teste executa um laço de repetição que cria a verificação de diversos módulos do projeto.

A palavra que obteve bom resultado no projeto Ant-Design(Tabela 12) e também no React-Native (Tabela 13) é a condicional *if*. Ela teve o quinto melhor resultado para o Ant Design com valor de 0,17 e o segundo melhor resultado para o React Native com valor de 0.03.

Tabela 13 – Ganho de informação das palavras para predição de FT no projeto React-Native

Posição	Token	IG	Ocorrências	#FT	#NFT
1	tomatchsnapshot	0.039	162	6	156
2	if	0.039	96	6	90
3	file	0.039	38	6	32
4	source	0.039	38	6	32
5	has	0.039	17	6	11
6	unsupported	0.039	8	6	2
7	skipping	0.039	7	6	1
8	success	0.039	7	6	1
9	api	0.039	6	6	0
10	catch	0.039	6	6	0
11	each	0.039	6	6	0
12	exclude	0.039	6	6	0
13	files_with_known_errors	0.039	6	6	0
14	package_root	0.039	6	6	0
15	readfile	0.039	6	6	0
16	sourcefiles	0.039	6	6	0
17	suppression	0.039	6	6	0
18	tests	0.039	6	6	0
19	translateflowtoexportedapi	0.039	6	6	0
20	unable	0.039	6	6	0

Fonte: Autoria própria (2025).

Na Tabela 14 apresentamos os termos com maior ganho de informação para predição de FT no projeto React Router. O termo *index* apresentou o valor mais alto, de 0.186. O termo *router* refere-se a classe principal do projeto React-Router e também apresentou um valor alto. Assim como os outros projetos baseados no React, ele possui termos relacionados ao HTML, são eles: *gethtml*, *container*, *div*, *getbytext*. Outro termo apresentado em outros projetos deste contexto é o **fallback**. De acordo com a documentação deste componente no React ⁵, o *suspense* é uma ferramenta no React para lidar com carregamentos assíncronos de forma eficiente e fluida. Ele permite que desenvolvedores criem experiências de usuário melhores ao fornecer feedback (**fallback**) visual durante o carregamento, e suas integrações com otimizações avançadas possibilitam construir aplicações mais rápidas e responsivas. Neste caso, os termos *suspense*, *fallback* e *loading* podem estar associados tanto a execução de tarefas assíncronas quanto a adaptação de interface responsiva. Alguns termos relacionados com a espera assíncrona também estão presentes neste projeto: *await*, *waitFor*, *async* e *future*. O termo *tomatchinlinesnapshot* aparece entre o top-20 do React Router, semelhantemente ao termo *toMatchSnapshot* do projeto React Native.

⁵ <https://react.dev/reference/react/Suspense>

Tabela 14 – Ganho de informação das palavras para predição de FT no projeto React-Router

Posição	Token	IG	Ocorrências	#FT	#NFT
1	index	0.186	23	6	17
2	expect	0.178	68	9	59
3	dfd	0.133	7	5	2
4	await	0.133	44	9	35
5	router	0.133	52	8	44
6	container	0.133	18	6	12
7	gethtml	0.133	18	6	12
8	let	0.125	67	9	58
9	root	0.123	17	5	12
10	getbytext	0.119	11	6	5
11	screen	0.119	11	6	5
12	state	0.118	38	5	33
13	waitfor	0.116	11	6	5
14	tomatchinlinesnapshot	0.108	28	5	23
15	div	0.106	20	5	15
16	async	0.106	45	9	36
17	data	0.099	21	6	15
18	fallback	0.096	3	3	0
19	future	0.096	3	3	0
20	hydratefallback	0.096	3	3	0

Fonte: Autoria própria (2025).

Na Tabela 15 apresentamos os termos com maior ganho de informação para predição de FT no projeto Next.js. Em primeiro lugar, o termo *await* revela problemas de espera assíncrona neste projeto. Analisando as palavras dos FT deste projeto, verificamos que os termos *reject* e *dev* também são utilizados para forçar o atraso da execução do código. Assim como os outros projetos baseados no React, os termos *fallback* e *tobe* estão presentes. Também são apresentados termos relacionados com o dispositivo de execução, são eles: *browser*, *window* e *itdidnotrefresh*. Podemos notar a variedade de características dos termos presentes neste projeto, onde as 10 primeiras palavras são de contextos diferentes. Temos termos relacionados a matemática, texto, espera assíncrona, navegadores, entre outros.

Na Tabela 16 apresentamos os termos com maior ganho de informação para predição de FT no projeto Molecular. Todos os flaky tests encontrados neste projeto são da mesma categoria. Trata-se de um problema de serialização de dados. Neste caso, sabemos exatamente a causa raiz da instabilidade deles, ao serem corrigidos no PR ⁶. Em resumo, a utilização da data e hora atual causa a instabilidade em um teste que verifica o tamanho de um objeto serializado. Como esse objeto tem um atributo variável, o tamanho do objeto serializado pode variar também, causando uma falha intermitente quando o teste é executado em um horário específico. Na Tabela 16 é confirmado que o termo que mais interfere na instabilidade dos testes é o *now*, com um IG de 0,206. Os outros termos presentes na Tabela 16 também indicam pistas de onde a causa raiz do FT está. Por exemplo, existem outros termos que indicam que a falha ocorre durante a verificação de tamanho de um objeto de data serializado, são eles: *obj*, *packet*, *serialize*,

⁶ <https://github.com/moleculerjs/moleculer/pull/1151>

Tabela 15 – Ganho de informação das palavras para predição de FT no projeto Next.js

Posição	Token	IG	Ocorrências	#FT	#NFT
1	await	0.010	1371	10	1361
2	browser	0.008	530	6	524
3	manual	0.007	56	4	52
4	expect	0.007	1454	8	1446
5	window	0.007	181	3	178
6	dev	0.006	22	2	20
7	math	0.006	8	2	6
8	reject	0.006	2	2	0
9	strings	0.006	2	2	0
10	eval	0.005	242	3	239
11	const	0.005	1488	9	1479
12	fallback	0.005	53	3	50
13	tobe	0.005	852	6	846
14	pathname	0.005	145	5	140
15	itdidnotrefresh	0.004	5	2	3
16	text	0.004	574	6	568
17	tomatch	0.004	113	3	110
18	blocking	0.004	24	2	22
19	stale	0.004	6	2	4
20	beforevalidate	0.004	4	2	2

Fonte: Autoria própria (2025).

serializer, deserialize, length, date. Portanto, essa métrica pode auxiliar diretamente na análise de instabilidades em testes, facilitando a identificação e correção da instabilidade em testes.

Tabela 16 – Ganho de informação das palavras para predição de FT no projeto Molecular

Posição	Token	IG	Ocorrências	#FT	#NFT
1	now	0.206	4	3	1
2	obj	0.206	4	3	1
3	packet	0.206	4	3	1
4	res	0.206	14	3	11
5	serialize	0.206	5	3	2
6	test	0.206	5	3	2
7	uuid	0.206	3	3	0
8	broadcast	0.206	3	3	0
9	clonedep	0.206	3	3	0
10	it	0.206	54	0	54
11	packet_event	0.206	3	3	0
12	sender	0.206	3	3	0
13	serializer	0.206	3	3	0
14	ver	0.206	3	3	0
15	date	0.166	4	3	1
16	created	0.166	6	3	3
17	length	0.166	6	3	3
18	const	0.166	19	3	16
19	deserialize	0.147	5	3	2
20	event	0.147	14	3	11

Fonte: Autoria própria (2025).

Na Tabela 17 são apresentadas as top-20 palavras que mais se repetem no código-fonte de FT e também as top-20 palavras com maior ganho de informação em projetos baseados no React. Verificamos que existem tokens que não aparecem no top-20 com mais repetições FT

porém aparecem no top-20 com maior ganho de informação. Destacamos essas palavras em negrito na Tabela 17.

Alguns tokens possuem bom ganho de informação entre diferentes projetos baseados no React, são eles: fallback, expect, await, waitfor, async, refresh, html, string, async, inner, suspense, text, loading, function, return.

Tabela 17 – Palavras com maior IG para FT obtidos no conjunto de execução de projetos relacionados ao React.

Projeto	Top 20 palavras (repetição)	Top 20 palavras (IG)
React	toBe, return, container, const, function, await, text, span, Hello, Scheduler log, firstChild, Suspense, render, innerHTML, ReactNoop, React, async, true	should , tobe, scheduler, throw , container, return, expect, suspend , log, innerhtml , text, hello, finalhtml , oops , function, loading , fallback , suspense, refreshreg , patch
React-Native	file, if, InteractionManager, source, jest, const, toBeCalled, console toMatchSnapshot, success, not, describe, runAllTimers, await, has, error, e, FILES_WITH_KNOWN_ERRORS, it, interactionStart, interactionComplete	tomatchsnapshot, if, file, source, has, unsupported , skipping , success, api , catch , each , exclude, files_with_known_errors, package_root, readfile, sourcefiles , suppression , tests , translateflowtoexportedapi
React-Hook-Form	const, control, expect, setCustomValidity, reportValidity, focus, jest, fn, message, FormValues, field, fireEvent, screen, getByRole, toBeCalledTimes, type, test, function, Input, return, div, input, ref, App, handleSubmit, form, click	tobecalleddtimes, focus, reportvalidity, setcustomvalidity, fn, native , jest, shouldusenativevalidation , this , tobecalledwith , message, ref, button, tobecalled , function, waitfor , expect, required , usecontroller , const
React-Router	let, router, state, await, container, t, data, p, getHtml, resolve, "/", async, path, return, toMatchInlineSnapshot, toBe, screen, getByText, true	index , expect , await, router, container, gethtml, let, root , getbytext, screen , state, waitfor , tomatchinlinesnapshot, div , async, data , fallback , future , hydratefallback
Next.js	await, browser, const, expect, eval, toBe, text, next, url, async, Math, toMatch, window, scrollTopMemories, floor, fetchViaHTTP, check, res, success, pathname	await, browser, manual , expect, window, dev, math, reject , strings , eval, const, fallback , tobe, pathname, itdidnotrefresh , text, tomatch, blocking , stale , beforeinvalidate
Ant-Design	0, const, container, expect, name, copyTest, Base, '.ant-typography-copy', querySelectorAll, iconClassNames, tooltipTexts, jest, icon, fireEvent, tooltips, 1, toBe, querySelector, spy	const, tobe, string , waitfaketimer , if , queryselectorall, test, component , wrapper , expect, queryselector, unmount , container, async , ant, htmldivement , tobefalsy , key , await , click
React-Based	expect, const, await, container, toBe, 0, return, name, it, text, async, jest, render, true, copyTest, if, Base, function, let, 1,	expect, if, const, container, oops , text, await, tobe, should, log , waitfor , let, loading , source , success , true, tests , mockreset , skipping , files_with_known_errors

Fonte: Autoria própria (2025).

Também verificamos o ganho de informação entre as palavras de todos os projetos. Na Tabela 19 utilizamos os dados dos FT obtidos pela revisão e execução, e na Tabela 18 utilizamos somente os dados obtidos pela execução. Em ambas tabelas, os valores de ganho de informação são naturalmente baixos por existir uma quantidade alta de instâncias de todos os projetos. Como essa variedade de palavras e projetos é maior, selecionamos os top-30 termos com maior IG por ordem decrescente.

Na Tabela 18 podemos verificar que o termo com maior ganho de informação para identificar instabilidades entre projetos JavaScript é o *await* com o valor de 0,011. O termo *async* também está incluso nesse contexto, e aparece em terceiro lugar com o valor de 0,009. Outro termo relacionado com o problema de espera assíncrona apareceu no top-30, o *waitFor*. Ele teve o valor de 0,005 pois também está presente em diversas instâncias de NFT. O termo *expect* apareceu em segundo lugar com valor de 0,010. A possível justificativa para este termo ter um bom ganho de informação é de que os termos que fazem diversas verificações em um único caso de teste, estão mais propensos a apresentar falhas. Isto ocorre, pois para o teste falhar por completo basta que apenas um *expect* falhe. A chance da falha fica ainda maior quando existem funções de espera assíncrona entre esses *expects*. Pode ser que a função demore mais do que o esperado e ocorram *timeouts*.

Na Tabela 18, o termo *if* apareceu em quarto lugar, com o valor de 0,009. Isto indica que o uso de condicionais podem causar instabilidades no conjunto apresentado. A maioria dos testes JavaScript começam a declaração do teste com esses termos, mas não necessariamente todos. Por exemplo, encontramos testes declarados com termos diferentes (*maybeTest*, *test*, entre outros). Na Tabela 19, onde os dados são obtidos com técnicas mistas, os termos *flakeIt* e *test* aparecem também representando declarações de teste. Em especial, o termo *it.skip* não poderia ter sido encontrado pelo conjunto de reexecução, visto que ele ignora o teste em questão. Mas como também consideramos um conjunto obtido por revisão, o termo existe no conjunto. Na Tabela 18, onde os dados são obtidos somente por técnicas dinâmicas, a maioria dos flaky tests possuem em sua declaração as palavras *it* e *describe*. Na posição 8 aparece o termo *container* que é muito utilizado no ambiente React para interagir com os elementos HTML. Neste código-fonte também são apresentados termos que apareceram neste top-30, são eles: *const*, *text*, *log* e *waitFor*. Esses termos são geralmente utilizados em conjunto do termo *container* para definir, visualizar e alterar elementos HTML. Neste top-30 também aparecem termos relacionados com arquivos: *file*, *join*, *source*, *files_with_known_errors* e *sourcefiles*; tratamento de erros: *oops* e *skipping*; e processos: *worker*.

Tabela 18 – Ganho de Informação por Token na linguagem JavaScript utilizando dados obtidos pela reexecução.

Posição	Token	Ganho de Informação	Ocorrências	#FT	#NFT
1	await	0.0115	2083	76	2007
2	expect	0.0108	4520	79	4441
3	async	0.0091	2156	76	2080
4	if	0.0091	975	33	942
5	path	0.0090	163	26	137
6	it	0.0089	4661	68	4593
7	describe	0.0088	13	11	2
8	container	0.0084	931	22	909
9	should	0.0080	3804	31	3773
10	const	0.0072	3859	59	3800
11	text	0.0070	372	17	355
12	oops	0.0067	8	8	0
13	console	0.0065	83	15	68
14	file	0.0064	94	10	84
15	join	0.0061	323	18	305
16	index	0.0059	241	11	230
17	worker	0.0054	10	7	3
18	source	0.0052	69	6	63
19	waitfor	0.0052	146	15	131
20	readfile	0.0051	12	8	4
21	test	0.0051	932	31	901
22	log	0.0051	214	19	195
23	skipping	0.0050	7	6	1
24	files_with_known_errors	0.0050	6	6	0
25	package_root	0.0050	6	6	0
26	sourcefiles	0.0050	6	6	0
27	suppression	0.0050	6	6	0
28	translateflowtoexportedapi	0.0050	6	6	0
29	unable	0.0050	6	6	0
30	untyped	0.0050	6	6	0

Fonte: Autoria própria (2025).

Na Tabela 19 utilizamos os dados dos FT obtidos pela revisão e execução. Por este motivo, a variedade de palavras e contextos de FT são maiores. Novamente, a palavra com maior ganho de informação está relacionado com a espera assíncrona. Porém, diferente da Tabela 18 onde o termo era *await*, neste conjunto o termo é *then*. Mas, na prática os dois significam o retorno de uma função assíncrona ⁷. Os termos *const* e *should* aparecem no top-3 palavras com maior IG. O termo *const* é utilizado para instanciar classes utilizadas no teste, já o termo *should* aparece no nome dos casos de testes agrupados por um *describe*. É uma prática comum nos testes JavaScript definir um conjunto de testes (identificados pelo *it should*) para uma única classe (definida no *describe*). Portanto, o termo *should* tem um bom ganho de informação pelo mesmo motivo do termo *expect*: são diversas verificações executadas em sequência para a mesma classe, basta uma verificação falhar para que o teste da classe falhe. Os termos *gd*, *plot* e *plotly* são utilizados no framework Plotly.js, e indicam que alguns gráficos podem apresentar o resultado intermitente. Os termos *cy* e *getbytestid* são utilizados pelo framework Cypress para gerenciamento dos testes. O termo *function* é utilizado em uma variedade de contextos para construir e interagir com os elementos do teste. Existem termos relacionados com coordenadas de gráficos: *xaxis*, *yaxis* e *range*. Semelhantemente aos resultados anteriores de IG, neste cenário também existem um termos utilizados para verificações que se destacaram, o *equal* e *flakeIt*. Existem termos para tratamento de erros: *catch* e *failtest*; e para interação com interfaces: *scratch* e *click*.

Analisando o resultado dessas tabelas de ganho de informação no JavaScript, verificamos características em comum entre os tokens. Agrupamos os tokens que possuem maior ganho de informação para predição de instabilidade entre os projetos JavaScript em diferentes características. Na Tabela 20 apresentamos a relação entre alguns tokens e suas características.

4.3 QP3: Com que qualidade podemos prever instabilidades em casos de teste entre diferentes projetos e linguagens utilizando vocabulário?

Resposta para QP3

Os resultados não são aplicáveis a um cenário de predição real entre diferentes contextos devido à alta relação entre os termos de cada contexto, o que limita a generalização da previsão de instabilidades em casos de teste variados.

Modelos baseados em vocabulário apresentaram bons resultados para as linguagens Python, Java e JavaScript (Pinto *et al.*, 2020; Camara *et al.*, 2021b; Haben *et al.*, 2021; Soratto, 2022; Soratto; Silva, 2023). Para verificar os limites desta técnica em diferentes cenários verificamos os resultados de predição para diferentes conjuntos de treinamento, incluindo conjuntos obtidos por meio da execução dos casos de teste. As configurações propostas para res-

⁷ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/async_function

Tabela 19 – Ganho de Informação por Token na linguagem JavaScript utilizando dados mistos (revisão + reexecução).

Posição	Token	Ganho de Informação	Ocorrências	#FT	#NFT
1	then	0.019680	135	53	82
2	const	0.016033	5427	134	5293
3	should	0.015998	5345	138	5207
4	expect	0.015503	6120	181	5939
5	gd	0.015029	30	30	0
6	plotly	0.014009	28	28	0
7	cy	0.011983	25	24	1
8	flaky	0.011978	24	24	0
9	var	0.011068	78	34	44
10	done	0.010225	131	37	94
11	plot	0.009956	20	20	0
12	getbytestid	0.009667	27	21	6
13	function	0.009132	1159	83	1076
14	equal	0.009017	54	27	27
15	to	0.008747	1029	84	945
16	it	0.008276	4850	185	4665
17	await	0.008164	3484	119	3365
18	catch	0.008086	138	37	101
19	if	0.007767	1252	74	1178
20	beforeeach	0.007444	15	15	0
21	click	0.006949	932	70	862
22	failtest	0.006944	14	14	0
23	bucket	0.006444	13	13	0
24	overlay	0.006072	21	14	7
25	scratch	0.006001	14	13	1
26	range	0.005962	35	16	19
27	xaxis	0.005945	12	12	0
28	yaxis	0.005945	12	12	0
29	flakeyit	0.005446	11	11	0
30	drag	0.005446	11	11	0

Fonte: Autoria própria (2025).

ponder esta questão de pesquisa são: teste de projetos previamente desconhecidos pelo modelo e modelos construídos em linguagens distintas.

4.3.1 Teste de projetos previamente desconhecidos pelo modelo

Verificamos a qualidade dos modelos em relação às falhas intermitentes totalmente desconhecidas. Para garantir que o contexto do teste não seja conhecido pelo conjunto de treinamento, utilizamos projetos diferentes nos conjuntos teste. Ou seja, o conjunto de treinamento possui todos 246 FT e 6481 NFT utilizados na Tabela 9, com exceção das instâncias do projeto a ser testado. Os resultados deste experimento são apresentados na Tabela 21. Neste cenário, os projetos Ant-Design e React Native apresentaram os melhores resultados com F1 de **0.88** e **0.87** respectivamente. Todos os outros projetos apresentaram um F1 menor que **0.44**.

Tabela 20 – Características presentes nos Flaky Tests em JavaScript

Característica	Palavras
Tratamento de Erros	throw, catch, failtest, skipping, fail, errors
HTML	container, querySelectorAll, querySelector, getByText, click, alert, innerhtml, finalhtml, text, htmldivement, div, wrapper, window, template, domcontentloaded, mainWindow
Espera assíncrona	await, waitFor, waitFakeTimer, future, promise, timeout, waitForLoadState, loading, fallback, suspense,
Arquivos	pathname, writeFile, readFile, sourceFiles, files_with_know_errors, source, file
Paginação de dados	data, page
Data e Hora	fn, now
Expressões Regulares	refreshreg, infoRegex, successRegex,
Fixtures	fixture, function, return, log, const, console, beforeEach
Dados	obj
Framework de teste	toBe, toMatch, toBeFalsy, toBeTruthy, toStrictEqual, mockimplementation
Valores binários	true, false, 0, 1
Gráficos	plot, xaxis, yaxis, plotly
Tipos de dados	type, untype, string(s)

Fonte: Autoria própria (2025).

Tabela 21 – Métricas dos modelos em projetos previamente desconhecidos pelo treinamento.

Treinamento		Projeto	Teste		Precisão	Recall	F1	MCC	AUC
NFT	FT		FT	NFT					
6581	241	Ant Design	5	101	1.00	0.80	0.88	0.89	0.90
6581	238	React Native	8	777	0.87	0.87	0.87	0.87	0.93
6581	235	Next.js	11	1664	0.33	0.09	0.14	0.17	0.54
6581	240	Strapi	6	153	0.33	1.00	0.5	0.55	0.96
6581	243	Nativefier	3	20	0.33	0.66	0.44	0.35	0.73
6581	237	React Router	9	59	0.21	0.44	0.28	0.14	0.59
6581	234	Metro	12	165	0.10	0.66	0.17	0.12	0.62
6581	243	Ant Design Vue	3	196	0.01	1.0	0.05	0.57	0.80
6581	243	Moleculer	3	54	0	0	0	-0.09	0.42

Fonte: Autoria própria (2025).

4.3.2 Teste de modelos construídos em linguagens distintas

Verificamos a qualidade dos modelos em relação às falhas intermitentes de diferentes linguagens de programação: JavaScript, Java e Python. Utilizamos o mesmo *workflow* utilizado anteriormente em JavaScript para as linguagens Java e Python, variando apenas os conjuntos. Utilizamos o conjunto contendo o código-fonte de casos de testes rotulados como FT/NFT na linguagem JavaScript apresentados neste trabalho. Em Python utilizamos os dados apresentados por Haben *et al.* e, para Java, o conjunto apresentado por Pinto *et al.*. Adicionamos o pré-processamento para extração de *tokens* e rotulagem para as novas linguagens (Java e Python) com base nos conjuntos anteriores.

O resultado deste cenário entre linguagens é apresentado na Tabela 22. O melhor resultado, foi para o modelo treinado na linguagem JavaScript e testado na linguagem Java, que apresentou o F1 de **0.63**. O segundo melhor resultado do modelo treinado em linguagem Python

e testado na linguagem Java, que apresentou o F1 de **0.62**. O pior resultado foi para o modelo treinado na linguagem Java e testado na linguagem JavaScript, que apresentou o F1 de **0.34**.

Tabela 22 – Métricas dos modelos entre linguagens distintas.

Treinamento			Teste			Precisão	Recall	F1	MCC	AUC
Ling.	FT	NFT	Ling.	FT	NFT					
Javascript	6481	246	Python	900	900	0.44	0.37	0.40	-0.90	0.45
Javascript	6481	246	Java	1392	1392	0.51	0.83	0.63	0.07	0.53
Python	6000	908	Javascript	246	246	0.46	0.26	0.34	-0.04	0.48
Python	6000	908	Java	1392	1392	0.54	0.73	0.62	0.11	0.55
Java	6000	1392	Javascript	246	246	0.40	0.30	0.34	-0.01	0.42
Java	6000	1392	Python	908	908	0.43	0.36	0.39	-0.01	0.44
Misto	3200	800	Misto	800	200	0.92	0.81	0.86	0.89	0.83

Fonte: Autoria própria (2025).

4.4 QP4: Quais são os principais termos presentes em códigos de testes associados à instabilidade entre diferentes linguagens?

Resposta para QP4

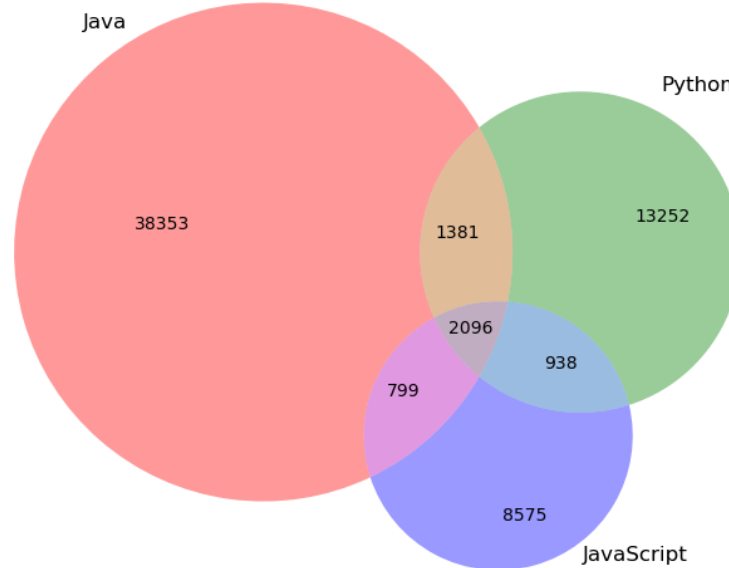
Entre as linguagens de programação Python e JavaScript podemos verificar termos relacionados com as seguintes causas raízes de *flaky tests*: comunicação assíncrona (**await**, **400**, **data**), concorrência, eventos de interface gráfica do Usuário (**plot**, **page**, **button**, **click** e **width**, dependência de Tempo (**time**), vazamentos de Recursos (**source**, **data**, **args**). Entre as linguagens Java e JavaScript, podemos verificar a intersecção de termos relacionados a concorrência (**manager**), espera assíncrona (**await**) e vazamentos de Recursos (**new**, **class**). Entre as linguagens Python e Java, podemos verificar termos relacionados com comunicação assíncrona (**waitFor**), concorrência (**get**, **select**), dependência de Tempo (**date**), vazamentos de recursos (**evaluate**, **catch**).

Para realizar a predição de *flaky tests* entre linguagens é necessário garantir que essas linguagens possuem termos em comum. Na Figura 11 podemos visualizar que existe a intersecção dos termos entre linguagens JavaScript, Java e Python.

Encontramos um total de 2096 palavras comuns nas três linguagens. Também foram identificadas 3417 (1381 + 2036) palavras em comum nas linguagens Java e Python; 2954 (918 + 2036) palavras em comum nas linguagens Python e JavaScript; e 2835 (7999 + 2036) palavras em comum nas linguagens JavaScript e Java. Essa intersecção indica que existem termos semelhantes entre linguagens. É necessário verificar quais destes termos podem auxiliar a predição de instabilidades entre linguagens. Portanto, é possível que as palavras possuam um bom ganho de informação para predição de FT entre as linguagens JavaScript, Python, e Java. Para verificar essa hipótese, selecionamos as 30 palavras com maior ganho de informação para cada

linguagem. Na Tabela 24 temos as palavras em Python, na Tabela 23 temos as palavras em Java, e na Tabela 19 as palavras JavaScript.

Figura 11 – Interseção do Vocabulário de Testes entre Linguagens (Java, Python e JavaScript)



Fonte: Autoria própria (2025).

O arquivo ⁸ é responsável por verificar quais termos podem auxiliar a predição de instabilidades em um conjunto fornecido. Ele foi executado para as 3 linguagens (JavaScript, Java e Python) presentes em nosso repositório ⁹.

Na Tabela 23 estão presentes as top-30 palavras com maior métrica de IG para a linguagem Java. Na primeira posição está a palavra *framework*. Em segundo temos o nome de um framework java, o *oozie*. O Oozie é um sistema agendador de fluxo de trabalho para gerenciar *jobs* do Apache Hadoop¹⁰. Na sequência, o terceiro termo com maior ganho de informação no conjunto Java é o *job*. Os termos *status*, *service*, *getstatus*, *getId*, *addrecordtocoordjobtable*, *execute*, *coordinatorjobbean*, *running*, *executor* e *services* aparecem com um bom ganho de informação e estão provavelmente relacionados com a execução de processos. Na sétima posição encontramos o termo *xml*, que é comumente utilizado em Java para construção de interface ou comunicação de dados. O termo *date* na posição 27 do ranking apresenta um contexto de data. O termo *exception* na posição 25 representa o teste de erros.

Na Tabela 24 estão presentes as top-30 palavras com maior métrica de ganho de informação na linguagem Python. Em primeiro lugar, encontramos a palavra *jira*, que se refere a uma ferramenta popular de gerenciamento de projetos e rastreamento de problemas. A seguir, na segunda posição, temos o termo *bot*, frequentemente relacionado a scripts automatizados ou bots que interagem em ambientes de chat ou plataformas de comunicação. O termo *test* aparece em terceiro lugar, indicando a relevância dos testes na prática de desenvolvimento em Python,

⁸ https://github.com/sorattorafa/flaky-across-languages/blob/main/scripts/vocabulary/information_gain_languages.py

⁹ <https://github.com/sorattorafa/flaky-across-languages/tree/main/datasets>

¹⁰ <https://oozie.apache.org/>

Tabela 23 – Ganho de Informação por palavras na linguagem Java

Posição	Token	IG	Ocorrências	#FT	#NFT
1	framework	0.1543	1131	944	187
2	oozie	0.1306	1826	1016	810
3	job	0.0731	543	475	68
4	status	0.0592	574	438	136
5	services	0.0578	467	394	73
6	service	0.0527	712	445	267
7	getid	0.0410	403	310	93
8	coord	0.0400	332	279	53
9	xml	0.0383	367	289	78
10	getstatus	0.0361	281	244	37
11	hbase	0.0343	5252	92	5160
12	jpaservice	0.0338	212	202	10
13	call	0.0324	579	291	288
14	workflow	0.0315	352	252	100
15	jpa	0.0314	227	203	24
16	xtestcase	0.0300	223	201	22
17	coordinatorjobbean	0.0291	202	188	14
18	addrecordtocoordjobtable	0.0265	165	161	4
19	running	0.0262	241	196	45
20	coordinatoraction	0.0260	194	175	19
21	command	0.0252	259	200	59
22	execute	0.0241	847	259	588
23	executor	0.0237	356	218	138
24	succeeded	0.0231	200	164	36
25	exception	0.0230	1093	385	708
26	action	0.0226	925	350	575
27	date	0.0222	323	205	118
28	class	0.0209	2228	541	1687
29	rest	0.0202	300	172	128
30	oozieclient	0.0193	148	132	16

Fonte: Autoria própria (2025).

especialmente em contextos de teste de software. Na quarta posição, o termo *return* é crucial, referindo-se à palavra-chave utilizada em funções para retornar valores. Outro termo de destaque é *chat_id* na quinta posição, que geralmente é utilizado em scripts que interagem com APIs de mensagens. Outros termos, como *bokeh_server_page*, *modify_doc*, *project_b* e *page*, indicam termos técnicos usados em manipulações de visualização e gestão de dados em aplicações *Python*, especialmente com bibliotecas populares como *Bokeh*. Essas palavras refletem a natureza colaborativa e dinâmica do desenvolvimento em *Python*, onde ferramentas de automação e *frameworks* são fundamentais para a eficiência e eficácia no gerenciamento de projetos e na criação de aplicações.

A Tabela 25 contém apenas os top 30 termos de cada linguagem. Mesmo que os termos dos top-30 de cada linguagem sejam diferentes, eles podem representar conceitos semelhantes. Por exemplo, no conjunto de dados JavaScript encontramos um termo de contexto semelhante, o *json* que teve o maior ganho de informação no projeto Strapi. O termo *date* na posição 27 do ranking apresenta um contexto de data que também foi visto anteriormente nos projetos JavaScript, com os termos *fn* e *now*. Outro exemplo da intersecção de contextos entre as linguagens é

Tabela 24 – Ganho de Informação por palavras na linguagem Python

Posição	Token	Ganho de Informação	Ocorrências	#FT	#NFT
1	jira	0.012975	113	104	9
2	bot	0.006568	357	102	255
3	test	0.006548	2577	285	2292
4	return	0.006286	5247	40	5207
5	chat_id	0.005427	173	72	101
6	bokeh_server_page	0.005270	39	39	0
7	project_b	0.004998	37	37	0
8	modify_doc	0.004728	45	39	6
9	page	0.004425	55	39	16
10	product	0.004183	31	31	0
11	group	0.003785	146	49	97
12	code	0.003481	210	57	153
13	issue	0.003359	71	35	36
14	plot	0.003325	167	29	138
15	y_range	0.002971	40	29	11
16	source	0.002903	242	42	200
17	plot_height	0.002873	42	29	13
18	manager	0.002845	80	37	43
19	plot_width	0.002828	43	29	14
20	time	0.002752	231	37	194
21	results	0.002711	219	43	176
22	min_border	0.002696	32	25	7
23	rangeId	0.002682	58	29	29
24	400	0.002655	42	27	15
25	test_manager	0.002640	25	22	3
26	mark	0.002611	235	55	180
27	add_tools	0.002567	24	19	5
28	click_custom_action	0.002559	19	19	0
29	timeout	0.002521	167	46	121
30	x_range	0.002493	52	29	23

Fonte: Autoria própria (2025).

o termo *exception* na posição 25. Em JavaScript foram encontrados alguns termos semelhantes, são eles: *throw*, *catch* e *errors*.

Outro ponto é sobre a quantidade de palavras de cada contexto. Verificamos o termo *plot* presente tanto no conjunto de Python quanto no conjunto JavaScript. Porém, em JavaScript este termo aparece mais abaixo no ranking. Portanto, essa análise de semelhança dos termos pode ser feita de uma forma mais ampla do que utilizar somente o top-30 por ganho de informação. Uma forma mais efetiva para observar a semelhança entre os vocabulários instáveis de diferentes linguagens é pela utilização de gráficos. Anteriormente, na Figura 11 verificamos que existem as seguintes interseções de tokens entre linguagens:

- Intersecção de 3477 (1381 + 2096) tokens nas linguagens Java e Python;
- Intersecção de 2895 (799 + 2096) tokens nas linguagens Java e JavaScript;
- Intersecção de 3034 (938 + 2096) tokens nas linguagens Python e JavaScript;
- Intersecção de 2096 tokens nas linguagens Java, JavaScript e Python;

Tabela 25 – Ganho de Informação por Token nas linguagens Java, Python e JavaScript (Rerun + Review)

Posição	Java	Python	Js (Rerun + Review)
1	framework	jira	then
2	oozie	bot	const
3	job	test	should
4	status	return	expect
5	services	chat_id	gd
6	service	bokeh_server_page	plotly
7	getid	project_b	cy
8	coord	modify_doc	flaky
9	xml	page	var
10	getstatus	product	done
11	hbase	group	plot
12	jpaservice	code	getbytestid
13	call	issue	function
14	workflow	plot	equal
15	jpa	y_range	to
16	xtestcase	source	it
17	coordinatorjobbean	manager	await
18	addrecordtocoordjobtable	plot_height	catch
19	running	plot_width	if
20	coordinatoraction	time	beforeeach
21	command	results	click
22	execute	min_border	failtest
23	executor	rangeId	bucket
24	succeeded	400	overlay
25	exception	test_manager	scratch
26	action	mark	range
27	date	add_tools	xaxis
28	class	click_custom_action	yaxis
29	rest	timeout	flakeyit
30	oozieclient	x_range	drag

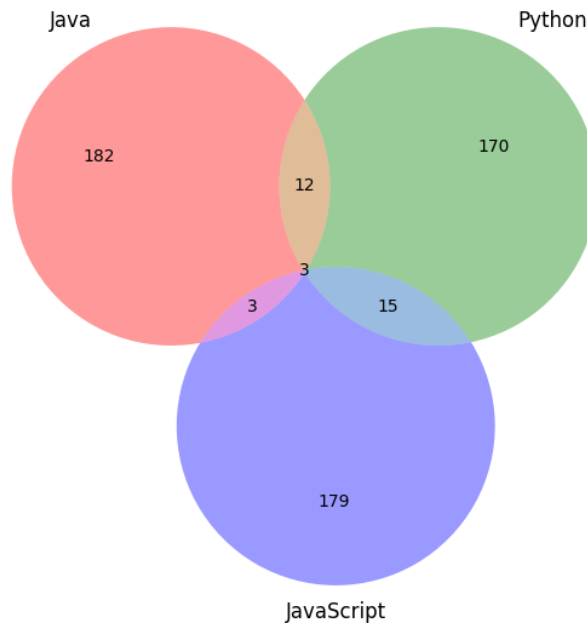
Fonte: Autoria própria (2025).

Verificamos que existem pelo menos 2096 termos (*tokens*) em comum entre diferentes linguagens.

É importante verificar quais termos dessas interseções possuem bom desempenho para predição de *flaky tests*. Com essa informação podemos descobrir possíveis causas raiz de instabilidade que são comuns entre diferentes projetos e linguagens de programação. Para isto, selecionamos os top-200 *tokens* com maior ganho de informação para predição de instabilidade de cada linguagem: Java, JavaScript e Python. Em seguida, verificamos a intersecção dessas palavras, conforme apresentado no diagrama de Venn da Figura 12.

Os termos presentes nas interseções dos conjuntos apresentados na Figura 12 são detalhados na Figura 13. Utilizamos os dados extraídos de ganho de informação das linguagens Java (Tabela 23), Python (Tabela 24) e JavaScript (Tabela 19) para criar uma visualização com a técnica Tree Mapping (TM) em Figura 13. O TM é uma visualização que funciona como um diagrama de árvore que utiliza retângulos dimensionados diferentemente para carregar valores numéricos para cada ramificação. Na visualização cada termo é representado por um retângulo de área proporcional ao seu ganho de informação para predição de instabilidades entre

Figura 12 – Intersecção dos top-200 palavras com maior ganho de informação para predição de instabilidade entre linguagens.



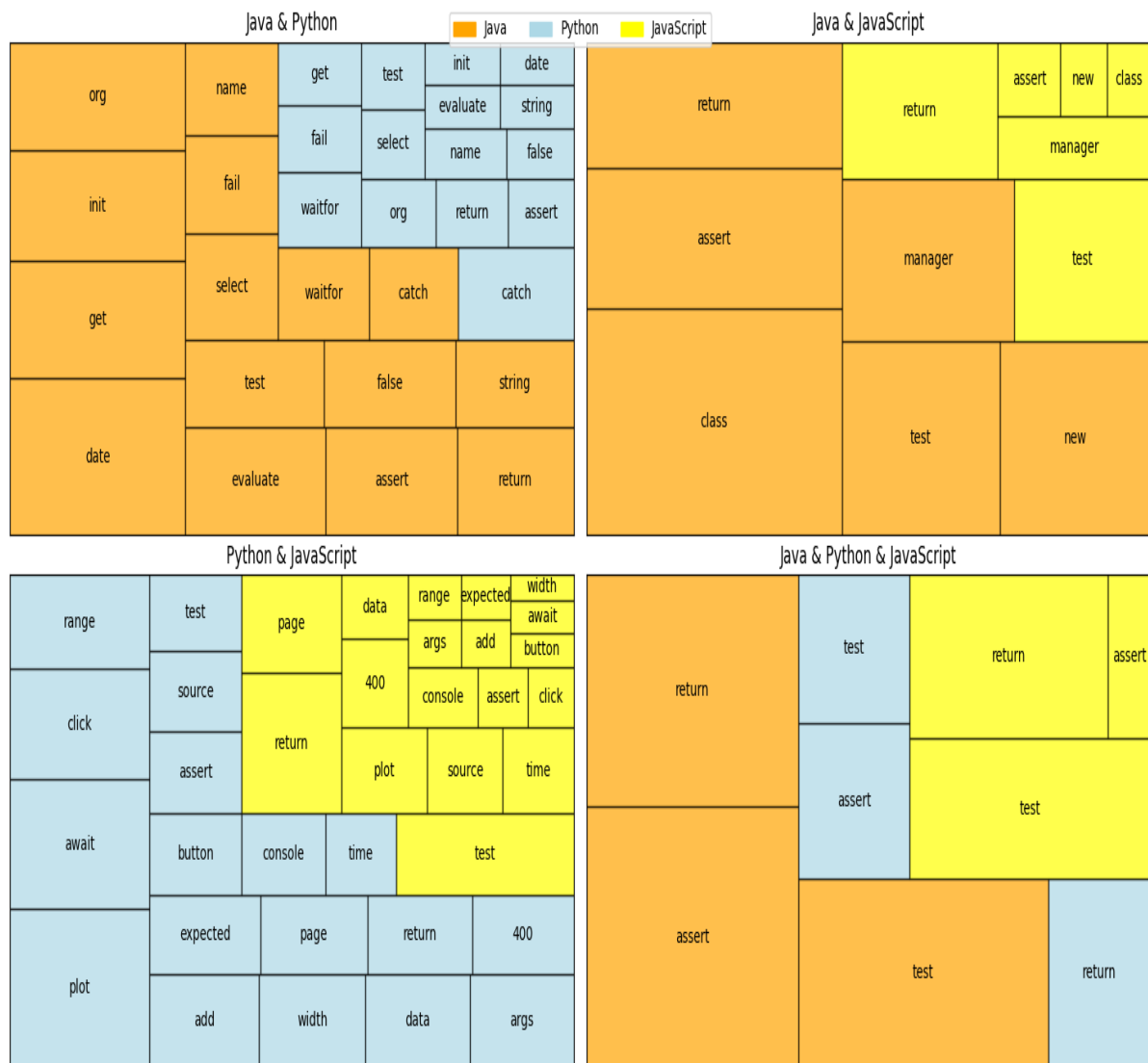
Fonte: Autoria própria (2025).

linguagens. Assim, buscamos mostrar a relevância dos tokens que possuem intersecção entre linguagens com a medida de ganho de informação.

Na Figura 13 temos 3 termos na intersecção dos top-200 termos com maior ganho de informação das 3 linguagens (Java, Python e JavaScript). Dois termos dessa intersecção, do ponto de vista de desenvolvimento, são termos comuns ao *framework* de teste e esperados para qualquer caso de teste automatizado. São eles: *assert* e *test*. Porém, o *return*, termo presente nesta intersecção está relacionado com instabilidades em testes de diversos projetos JavaScript. Conforme apresentado anteriormente, no React, o *return* é utilizado para construção de componentes e pode ser utilizado em diversos contextos. Mesmo sendo uma palavra que de fato identifica conjuntos de casos de testes, ela não pode ser utilizada com assertividade para prever instabilidades em testes.

Na Figura 13, os resultados são positivos ao avaliar a intersecção de somente duas linguagens. Neste cenário é possível verificar termos relacionados a causa raiz de instabilidades em testes. Por exemplo, nas linguagens Python e JavaScript podemos verificar termos relacionados com as causas apresentadas por Eloussi (2015): Comunicação Assíncrona (**await**, **400**, **data**), Concorrência, Eventos de Interface Gráfica do Usuário (**plot**, **page**, **button**, **click** e **width**, Dependência de Tempo (**time**), Vazamentos de Recursos (**source**, **data**, **args**).

Figura 13 – Visualização Treemapping dos termos que se repetem no ranking dos top-200 termos com maior ganho de informação para predição de instabilidades entre linguagens (Java, JavaScript, Python).



Fonte: Autoria própria (2025).

De forma semelhante, na Figura 13, observando a intersecção das linguagens Python e Java, podemos verificar termos relacionados com as causas apresentadas por Eloussi (2015): Comunicação Assíncrona (**waitFor**), Concorrência (**get**, **select**), Dependência de Tempo (**date**), Vazamentos de Recursos (**evaluate**, **catch**).

Por fim, na Figura 13, observando a intersecção das linguagens Java e JavaScript, podemos verificar termos relacionados com as causas apresentadas por Eloussi (2015): Concorrência (**manager**) e Vazamentos de Recursos (**new**, **class**).

4.5 Discussões

4.5.1 Modelos de Classificação

Os trabalhos relacionados de vocabulário relatam as métricas padrão de precisão, *recall* e F1. A precisão mostra a proporção de testes com falhas classificadas corretamente. A *recall* mostra a proporção de testes com falhas encontrados entre todos os existentes. Eles concentram sua análise na pontuação F1, que combina precisão e *recall* para avaliar o desempenho do modelo (Pinto *et al.*, 2020; Camara *et al.*, 2021b; Haben *et al.*, 2021). Este trabalho seguiu a implementação dos mesmos 5 modelos de classificação utilizados por Pinto *et al.* (2020) e incrementados por Camara *et al.* (2021b). Comparamos os resultados obtidos na linguagem JavaScript, Java e Python. Desta forma podemos conhecer as semelhanças e diferenças dos modelos de classificação e o vocabulário de cada linguagem.

A Tabela 26 apresenta os resultados de Pinto *et al.* (2020) para linguagem Java. O algoritmo que obteve o melhor desempenho foi o Random Forest com os seguintes valores: 0.99 de precisão, 0.91 de *recall* e 0.95 de F1. O algoritmo que teve o pior desempenho foi o Naive Bayes com os seguintes valores: 0.93 de precisão, 0.80 de *recall* e 0.86 de F1.

Tabela 26 – Resumo dos resultados de Pinto *et al.* (2020)

Algoritmo	Precisão	Recall	F1	MCC	AUC
Random Forest	0.99	0.91	0.95	0.90	0.98
Decision Tree	0.89	0.88	0.89	0.77	0.91
Naive Bayes	0.93	0.80	0.86	0.74	0.93
SVM	0.93	0.92	0.93	0.85	0.93
Nearest Neighbour	0.97	0.88	0.92	0.85	0.93

Fonte: Pinto *et al.* (2020, p.6).

A Tabela 27 apresenta os resultados de Camara *et al.* (2021b) na linguagem Java utilizando validação cruzada. O algoritmo que melhor desempenhou em termos de *recall* foi a regressão logística com os seguintes valores: 0.91 de precisão, 0.91 de *recall*, 0.91 de F1, 0.84 de MCC e 0.96 de AUC. O algoritmo que teve o pior desempenho foi o LDA com os seguintes valores: 0,83 de precisão, 0,78 de *recall*, 0,80 de F1, 0,63 de MCC e 0,87 de AUC.

Tabela 27 – Resumo dos resultados de Camara *et al.* (2021b)

Algoritmo	Precisão	Recall	F1	MCC	AUC
Random Forest	0.98	0.89	0.94	0.89	0.98
Decision Tree	0.87	0.86	0.86	0.74	0.87
Naive Bayes	0.95	0.84	0.89	0.81	0.90
SVM	0.93	0.86	0.90	0.81	0.96
Nearest Neighbour	0.98	0.81	0.89	0.81	0.90

Fonte: Camara *et al.* (2021b, p.7).

No estudo de Haben *et al.* (2021), para a linguagem Python, o modelo RF foi escolhido para avaliar os projetos pois obteve um F1 superior a 0.95. Os outros modelos também apre-

sentaram um bom desempenho porém Haben *et al.* (2021) não apresentaram uma comparação entre os modelos em Python.

Enquanto nos estudos de Pinto *et al.* (2020) e Haben *et al.* (2021) o Random Forest obteve o melhor resultado, no estudo de Camara *et al.* (2021b) a Regressão Logística foi o melhor modelo. Neste trabalho, a árvore de decisão foi o modelo que obteve a melhor métrica de F1 de 0,83.

Uma semelhança entre este e todos os trabalhos anteriores relacionados ao vocabulário instável é que o modelo NB é aquele que apresenta os piores resultados de F1 para predição de FT entre linguagens.

Tanto em JavaScript quanto em Python, os resultados são melhores quando os modelos são treinados e testados individualmente por cada projeto (mas pode gerar *overfitting*). Isto acontece pois diferentes projetos podem possuir diferentes contextos, implementações e aplicações. Todas essas variáveis afetam no código-fonte e conseqüentemente no vocabulário. Em Python, o projeto *python-telegram-bot* obteve o melhor desempenho para F1 (Haben *et al.*, 2021). Já em JavaScript, o projeto que obteve o melhor desempenho na Tabela 7 foi o *Moleculer*, superando os resultados do projeto em Python *python-telegram-bot*. Conseguimos o resultado máximo de classificação (F1 de 1.00) no projeto *Moleculer*, pois todas as instâncias de FT eram ocasionadas pela mesma causa raiz: a utilização de data e hora de maneira incorreta no código-fonte do teste. Diferentemente do *React*, que possui 32 FT de diferentes causas e mais de 1500 NFT. Neste tipo de projeto, que possuem uma variedade de causas de instabilidades e contextos de testes, a *recall* dos modelos podem ser afetada. Na Tabela 7 mostramos que mesmo com a precisão de 1,00 no projeto *React*, a *recall* foi afetada pela variedade das instâncias, apresentando o valor de 0,50. Conseqüentemente, o F1 do modelo neste projeto foi reduzido para 0,66.

Verificamos este mesmo comportamento também para alguns projetos Python, por exemplo, o *pipeenv*. Este projeto também possui 32 FT e mais de 1500 NFT, semelhantemente ao *React*. Ele também teve uma precisão considerável, mas teve baixo desempenho para *recall* (0.19). Isto afeta diretamente o F1 deste projeto, apresentando o valor de 0,30.

Portanto, a variedade de contexto entre as instâncias de FT e NFT de um projeto podem afetar diretamente na métrica de *recall* dos modelos.

4.5.2 Vocabulários

As palavras mais comuns no vocabulário original, proposto por Pinto *et al.* (2020), são: ‘job’, ‘table’, ‘id’ e ‘action’. As palavras mais comuns no vocabulário proposto por Camara *et al.* (2021b) são: ‘job’, ‘table’, ‘id’ e ‘service’. Tal semelhança entre os trabalhos anteriores se dá pelo fato de utilizarem conjuntos de dados semelhantes, sobre casos de testes de aplicações implementadas na linguagem Java. Porém, no estudo de Camara *et al.* (2021b) são realizados dois experimentos adicionais avaliando o ganho de informação nos cenários entre projetos e sepa-

radamente por projetos. O resultado do experimento entre projetos é apresentado na Tabela 28, onde estão presentes as top-20 palavras com maior ganho de informação para predição de instabilidade no conjunto Java. Neste experimento, foram identificadas diversas palavras relacionadas com a espera assíncrona (*await*, *export*, *handler*, *protocol*, *server*, *task*, *taskpayloadbuilder*, *url*).

Tabela 28 – Vocabulário Java em um cenário entre projetos

Token	Valor	Ocorrências
getname	0.8882	22
namingcontext	0.8882	22
same	0.8882	22
await	0.8674	14
export	0.8674	14
return keyword	0.8465	17
be	0.8465	17
handler	0.8465	30
is	0.8465	17
this keyword	0.8396	14
protocol	0.8396	14
}	0.8396	14
collections	0.8188	17
context	0.8188	19
server	0.8119	14
task	0.8049	19
of	0.7633	19
taskpayloadbuilder	0.7633	19
url	0.6939	40
try keyword	0.6661	51

Fonte: Camara *et al.* (2021b, p.11).

Tabela 29 – Vocabulário Java em um cenário em projetos separados

Feature	Inf. Gain	Ocorrências
public keyword	0.8188	80
acl	0.8188	6
created	0.8188	6
reader	0.8188	6
createdirwithhttp	0.0791	6
createsnapshot	0.0791	6
directory	0.0791	6
folder	0.0791	6
gethadoopusers	0.0791	6
had . . . helper	0.0791	6
init	0.0791	6
no	0.0791	6
touri	0.0791	6
write	0.0791	6
assertnonnull	1.1102	28

Fonte: Camara *et al.* (2021b, p.10).

O resultado do experimento em projetos separados na linguagem Java é apresentado na Tabela 29. Nela são apresentados termos relacionados com a execução e coordenação de tarefas (*init*, *createdirwithhttp*), operações de leitura e escrita de dados (*reader*, *write*, *directory*, *folder*), identificadores Java (*public*) (Camara *et al.*, 2021b). Neste cenário, alguns termos apareceram com ganho de informação zerado. Por fins de comparação, não utilizaremos esses termos pois são irrelevantes.

Nos estudos de Pinto *et al.* (2020) e Camara *et al.* (2021b) o método *CountVectorizer* foi utilizado para obter os tokens Java. Neste trabalho, utilizamos uma abordagem mais robusta para obter as palavras a partir do código-fonte. Ao invés do *CountVectorizer*, utilizamos o parse das próprias linguagens (Java, JavaScript e Python). Desta forma, além de obter as repetições das palavras, podemos obter informações adicionais, por exemplo, o tipo de cada palavra ou sua localização no código-fonte. O vocabulário da linguagem JavaScript foi obtido realizando este processo no conjunto de dados apresentado na Seção 3.1.4. No vocabulário Python, utilizamos o conjunto apresentado por Haben *et al.* (2021). No vocabulário Java, utilizamos o conjunto apresentado por Pinto *et al.* (2020).

Em Java, as palavras com maior ganho de informação foram: *framework*, *oozie*, *job*, *status* e *services*. Em Python, as palavras com maior ganho de informação foram: *jira*, *bot*, *test*, *return* e *chat_id*. Em JavaScript, utilizando dados mistos (mineração e execução de casos de testes) as palavras com maior ganho de informação foram: *then*, *const*, *gd*, *plotly* e *cy*. Em JavaScript, utilizando dados dinâmicos (obtidos pela reexecução), as palavras com maior ganho de informação foram: *await*, *expect*, *async*, *if* e *path*.

Com base nas top-5 palavras de cada vocabulário já podemos perceber algumas semelhanças entre os vocabulários de diferentes linguagens:

1. É comum encontrar *flaky tests* em ferramentas amplamente utilizadas (por exemplo, *oozie*, *jira* e *cypress*);
2. É possível que diferentes palavras do mesmo vocabulário estejam relacionadas ao mesmo contexto. Por exemplo, *oozie*, *job* na linguagem Java estão relacionados com um *framework* agendador de fluxo de trabalho para o Apache. Da mesma forma, os termos *bot* e *chat_id* em Python estão relacionados com *frameworks* de chatbot. O mesmo acontece com o *framework Bokeh* em Python: termos relacionados com gráficos tiveram bom ganho de informação (*y_range*, *plot*, *plot_height*, *plot_width*);
3. A espera assíncrona aparece to top-1 do vocabulário em JavaScript (representado pelo termo *then*) e também no cenário entre projetos da linguagem Java apresentado por Camara *et al.* (2021b) (representado pelo termo *await*);
4. Mesmo que os termos sejam diferentes entre linguagens, o contexto deles podem ser semelhantes, por exemplo, os termos *await* e *then*.

Na Tabela 30 categorizamos os top-30 termos com maior ganho de informação entre as linguagens para verificar a semelhança entre o contexto dos termos. Alguns termos, mesmo que distintos, podem abordar sobre o mesmo tema.

4.6 Ameaças à validade

Nesta seção são apresentadas as validades internas, externas e de construção.

Ameaças à validade do conclusão estão relacionadas à adequação das métricas utilizadas para avaliação dos resultados. As métricas de precisão, *recall*, F1, MCC e AUC foram utilizados em vários trabalhos de engenharia de software que utilizam a classificação (Kim *et al.*, 2008; Pinto *et al.*, 2020; Camara *et al.*, 2021b; Haben *et al.*, 2021).

Nossas conclusões são baseadas principalmente na precisão e na pontuação F1, uma vez que essas duas as métricas verificam a utilidade de um sistema de recomendação que pode alertar os desenvolvedores quando eles estão prestes a introduzir um teste instável (Pinto *et al.*, 2020).

Tabela 30 – Contexto do vocabulário entre linguagens

Características	Java	JavaScript	Python
Frameworks	framework, oozie e oozieclient	plotly e cy	jira e bokeh
Gráficos	-	plotly, plot, range, drag, xaxis e yaxis	y_range, plot_width, plot_height, y_range, x_range, min_border e plot
Chamadas Assíncronas	await, export, handler, protocol, server, task, taskpayloadbuilder, url	then, await, async, waitFor	timeout, time
Data e Hora	date	now	date, time
Operações de I/O	reader, write, directory, folder	readfile, files, sourcefiles, source	source
Tratamento de Erros	exception	throw, catch, failtest, skipping, fail, errors	400
Elementos de Interface	xml	container, html, div, wrapper, window, template	page
Execução de Tarefas	task, execute, action, running, workflow, call, job, services, status,	worker, function, done	click_custom_action, results
Requisições HTTP	url, toUri,	500	400, timeout

Fonte: Autoria própria (2025).

4.6.1 Ameaças à validade interna

Ameaças à validade interna comprometem a nossa confiança em estabelecer uma relação entre as variáveis independentes e dependentes (Pinto *et al.*, 2020). Nos casos de *flaky tests* temos evidências de execuções que apresentaram a falha intermitente. Já nos casos de testes não instáveis, é possível que algumas instâncias sejam instáveis. Isto acontece, pois mesmo executando as suítes de testes de cada projeto mais de mil vezes, algumas instabilidades não tem um número certo de repetições para serem reveladas. Porém, realizamos um número alto de execuções para garantir uma estabilidade mínima das instâncias classificadas como NFT. Quanto às instâncias de FT percebemos uma quantidade variada de falhas: alguns testes falham várias vezes durante as mil execuções, já alguns falham apenas uma vez semelhantemente ao trabalho de Pinto *et al.* (2020). Em geral, as falhas intermitentes que acontecem várias vezes estão relacionadas com *timeout*, verificação de interfaces ou operações de I/O. A quantidade de execuções utilizada para obter as instâncias de FT e NFT também é uma ameaça à validade interna. Ela pode ser incrementada e revelar resultados não conhecidos anteriormente.

Assim como nos trabalhos anteriores, parte do nosso conjunto de *flaky tests* foi obtido pelo rótulo dos desenvolvedores. Caso os casos de teste tenham sido rotulados erroneamente, isso pode ser uma ameaça a validade interna também. Porém, dado a escassez de *flaky tests* em linguagens diferentes de Java, utilizar os casos de testes previamente rotulados torna-se viável

(Haben *et al.*, 2021). Nosso processo de extração de *features* consegue considerar esses rótulos. Conseqüentemente, alguns rótulos (*flakeit* e *failtest*) tiveram um bom ganho de informação para prever instabilidades na Tabela 19.

O ambiente no qual o experimento está sendo executado pode ser uma ameaça à validade interna. O estudo de Pinto *et al.* (2020) utilizou o software Weka (Witten *et al.*, 1999). O estudo de Camara *et al.* (2021b) utilizou o software Scikit-Learn (Pedregosa *et al.*, 2011). Em nosso estudo utilizamos o Scikit-Learn porém implementamos um processo mais robusto para obtenção das *features* de acordo com cada linguagem. Mesmo com essa variação do ambiente e do processo de totemização, verificamos que os resultados seguem positivos.

Outra validade interna são os parâmetros escolhidos para os algoritmos de classificação. Eles podem ser ajustados para otimizar os resultados em trabalhos futuros. A reprodução do experimento pode ser uma ameaça à validade. Por isso descrevemos todo o processo de obtenção dos vocabulários em diferentes linguagens no repositório de replicação ¹¹.

4.6.2 Ameaças à validade externa

A ameaça à validade externa está principalmente relacionada com a habilidade de generalização dos resultados. Nossos experimentos se limitam aos projetos escolhidos e as linguagens Java, JavaScript e Python. Nos experimentos verificamos que existe a intersecção entre instabilidades de diferentes projetos e linguagens (Tabela 30). Mesmo existindo semelhanças entre os termos, é possível que eles sejam diferentes, afetando o desempenho dos classificadores. Para construir um vocabulário instável mais genérico, trabalhos futuros podem considerar o contexto em que cada palavra do código-fonte está inserida e aplicar a técnica em diferentes linguagens e projetos.

Assim como apresentado no estudo de Camara *et al.* (2021b), no cenário entre projetos em JavaScript, os resultados possuem uma redução de desempenho. Em trabalhos futuros podemos validar nossos resultados em conjuntos de dados maiores de FT. Em nosso experimento da QP1, verificamos que todos os projetos em JavaScript que relatam instabilidades possuem tarefas assíncronas, de criação de interfaces ou de entrada e saída de dados (I/O). Diferentemente de Java, onde os projetos são fortemente relacionados com entrada e saída de dados (Pinto *et al.*, 2020).

Trabalhos anteriores mostram que a técnica de vocabulário funciona muito bem quando utilizada em projetos com instâncias de FT e NFT definidas (Pinto *et al.*, 2020; Camara *et al.*, 2021b; Haben *et al.*, 2021). Verificamos este que este comportamento apresentado para as linguagens Java e Python também se repete para o JavaScript (Tabela 7). O motivo pelo qual os termos do código-fonte podem prever instabilidades tão bem está relacionado com a frequência desses termos em *flaky tests* e estáveis. A Tabela 18 mostra para diversas palavras-chave a frequência com que elas aparecem em *flaky tests* e não estáveis em JavaScript. Mesmo

¹¹ <https://github.com/sorattorafa/flaky-across-languages>

que essa frequência seja baixa, ao realizar o *ranking* por ganho de informação, conhecemos as possíveis palavras que podem estar relacionadas com a causa raiz da falha intermitente. Para algumas dessas palavras-chave, as diferenças são extremas, por exemplo, ‘oops’ ocorre em 8 *flaky tests* e em 0 testes não instáveis. Um classificador adivinhando que todos os testes contendo o termo ‘oops’ são instáveis, por definição, já atingiria uma precisão de 100% (8/8). Por outro lado, a diferença pode ser maior para os testes não instáveis. Por exemplo, a palavra-chave *await* ocorre em 76 *flaky tests* e em 2007 testes não instáveis. Por este motivo utilizamos o ganho de informação (IG) dos termos e não sua frequência, conforme apresenta a Tabela 18. Com proporções semelhantes para outras palavras-chave e o poder dos modelos de classificação, respectivamente, essas diferenças se traduzem em um excelente desempenho dos classificadores em termos de precisão.

A principal ameaça à validade externa é o tamanho e a natureza dos *datasets* utilizados. Para linguagem Java, utilizamos o conjunto de dados DeFlaker, por ser o maior conjunto de *flaky tests* de código aberto e já foi usado em muitos estudos de instabilidade (Pinto *et al.*, 2020; Bertolino *et al.*, 2020; Camara *et al.*, 2021b). Para linguagem Python, utilizamos o conjunto de dados apresentado por (Haben *et al.*, 2021) contendo 837 *flaky tests* rotulados por desenvolvedores e obtidos pela mineração de 9 repositórios no GitHub. Para a linguagem JavaScript, construímos um conjunto de dados de 246 *flaky tests* utilizando a reexecução e também a mineração de mais de 30 repositórios no GitHub. Para todas as linguagens, aumentar o número de instâncias de FT é necessário em trabalhos futuros. Desta forma será possível alcançar resultados mais genéricos.

Também é interessante prever instabilidades em projetos que nunca apresentaram um FT anteriormente. Neste sentido, investigamos o desempenho de classificadores treinados em diferentes projetos e verificamos ser possível generalizar os resultados mesmo com uma queda do desempenho. Trabalhos futuros podem incrementar os dados e as linguagens utilizadas para enriquecer ainda mais a variedade dos resultados. Projetos industriais também podem enriquecer o vocabulário ainda mais.

5 CONCLUSÃO

flaky tests são aqueles que às vezes passam e às vezes falham, sem qualquer mudança no código de teste. Eles diminuem significativamente a confiança de um conjunto de testes de regressão automatizado e atrasam o processo de entrega do software. Muitas vezes, testes rotulados como instáveis são até ignorados em bateria de testes para evitar o atraso no processo de implantação de sistemas. Por este motivo, o estudo sobre instabilidades no cenário de testes automatizados tem se intensificado na comunidade de engenharia de software com intuito de garantir o valor dos testes e reduzir o tempo gasto com falhas intermitentes.

Nos últimos anos, a pesquisa sobre a instabilidades em do teste ganhou um impulso significativo. Trabalhos anteriores focados em caracterizar o que é instável ou identificar a causa raiz dos *flaky tests*. No entanto, pouco esforço foi colocado em como reconhecer eficientemente um teste instável de forma automática. Este trabalho se concentra na questão de saber se existem identificadores de programação (por exemplo, nomes de métodos e variáveis) que poderiam ser usados para reconhecer automaticamente *flaky tests*. Mais precisamente, buscamos responder a pergunta: Com que qualidade podemos prever instabilidades em casos de teste entre diferentes projetos e linguagens utilizando vocabulário?

Para responder a essa pergunta, começamos extraindo casos de teste de um conjunto de dados bem conhecido de *flaky tests* em Java (Bell *et al.*, 2018), de um conjunto Python (Haben *et al.*, 2021) e um conjunto JavaScript (Soratto; Silva, 2023). Os trabalhos de Pinto *et al.* (2020) e Haben *et al.* (2021) fornecem o código-fonte de *flaky tests* e não instáveis em Java e Python respectivamente. Porém, não existe nenhum trabalho que forneça o código-fonte de testes não instáveis implementados em JavaScript no atual estado da arte. Este trabalho fornece um conjunto de dados de *flaky tests* e *non flaky tests* em projetos relevantes na linguagem JavaScript, obtidos pela reexecução. Para isso, reexecutamos mais de mil vezes mais de 40 projetos JavaScript considerados relevantes no Github. Sinalizamos um teste como instável se houver desacordo nos resultados do teste. Após a identificação de testes com falhas intermitentes, extraímos todos os identificadores dos casos de teste usando o processamento natural da linguagem de programação (a mesma utilizada pelo compilador da linguagem). Finalmente, os casos de teste instáveis e não instáveis pré-processados foram usados como entrada para cinco algoritmos de aprendizado de máquina.

Com base nesses dados e abordagem, pudemos observar várias descobertas interessantes. Primeiro, conseguimos encontrar 32 projetos JavaScript com 144 *flaky tests*. Diferentemente do estudo de Pinto *et al.* (2020), onde 55% dos falharam apenas uma vez, a maioria dos testes falharam mais de uma vez. Isto porque utilizamos um número de execuções 10 vezes maior do que o trabalho original de vocabulário (1000 execuções por projeto). Em segundo lugar, observamos que os seis algoritmos de aprendizado de máquina usados tiveram um bom desempenho na distinção entre FT e NFT.

Em JavaScript, semelhantemente ao Java, o modelo RF teve a melhor precisão (0,96), enquanto o modelo DT superou ligeiramente o RF em termos de recall (0,77 vs 0,63). Os resultados são melhores quando o modelo é treinado e testado no mesmo escopo de projeto e linguagem. Alguns projetos alcançaram resultados de 100% de precisão e recordação. Porém, verificamos que a qualidade dos modelos em relação às falhas intermitentes totalmente desconhecidas é baixa. Para garantir que o contexto do teste não seja conhecido pelo conjunto de treinamento, utilizamos projetos diferentes nos conjuntos teste. Neste cenário, em JavaScript, os projeto Ant-Design e React Native apresentaram os melhores resultados com F1 de **0.88** e **0.87** respectivamente. Todos os outros projetos apresentaram um desempenho ruim. Verificamos a qualidade dos modelos em relação às falhas intermitentes de diferentes linguagens de programação: JavaScript, Java e Python. O melhor resultado, foi para o modelo treinado na linguagem JavaScript e testado na linguagem Java, que apresentou o F1 de **0.63**. O segundo melhor resultado do modelo treinado em linguagem Python e testado na linguagem Java, que apresentou o F1 de **0.62**. O pior resultado foi para o modelo treinado na linguagem Java e testado na linguagem JavaScript, que apresentou o F1 de **0.34**.

Finalmente, em relação ao vocabulário de *flaky tests*, notamos que palavras são fortemente dependentes do contexto do projeto e com a linguagem de programação na qual ele é implementado. Por exemplo, no React, os termos mais relevantes para predição de instabilidades em testes são: `fallback`, `expect`, `await`, `waitfor`, `async`, `refresh`, `html`, `string`. Neste tipo de projetos, as falhas intermitentes estão relacionadas com a espera assíncrona, elementos de interface e execução de tarefas. Em projetos de diferentes contextos, foram encontradas diversas categorias de instabilidades: operações de entrada e saída, verificação binária, data e hora, gráficos, entre outros.

Existe uma intersecção entre contextos e causas raízes de instabilidades de diferentes projetos e linguagens. Porém, utilizar somente o conteúdo puro do código-fonte pode reduzir o desempenho dos modelos de predição. Uma sugestão para trabalhos futuros é associar o código-fonte com seu significado. Por exemplo, o tratamento de datas pode conter termos diferentes em linguagens distintas. Portanto, agrupar os termos para a categoria de data, pode melhorar o entendimento dos classificadores sobre o contexto de cada palavra. Por exemplo, na Tabela 30, mostramos que existem termos de diferentes linguagens que estão relacionados com a categorias semelhantes de falhas: chamadas assíncronas, data e hora, operações de entrada e saída, tratamento de erros, execução de tarefas, utilização de frameworks, elementos de interface e requisições Hypertext Transfer Protocol (HTTP). Trabalhos futuros podem incrementar os dados e as linguagens utilizadas para enriquecer ainda mais a variedade dos resultados. Projetos industriais também podem enriquecer o vocabulário ainda mais.

REFERÊNCIAS

- AGRAWAL, Rakesh; IMIELIŃSKI, Tomasz; SWAMI, Arun. Mining association rules between sets of items in large databases. *In: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. [S.l.: s.n.], 1993. p. 207–216.
- AKLI, Amal; HABEN, Guillaume; HABCHI, Sarra; PAPADAKIS, Mike; TRAON, Yves Le. Flakycat: Predicting flaky tests categories using few-shot learning. *In: 2023 IEEE/ACM International Conference on Automation of Software Test (AST)*. [S.l.: s.n.], 2023. p. 140–151.
- ALSHAMMARI, Abdulrahman; MORRIS, Christopher; HILTON, Michael; BELL, Jonathan. FlakeFlagger: Predicting flakiness without rerunning tests. *In: 43rd International Conference on Software Engineering*. [S.l.]: IEEE, 2021. p. 1572–1584.
- BARBOSA, Keila; FERREIRA, Ronivaldo; PINTO, Gustavo; D’AMORIM, Marcelo; MIRANDA, Breno. Test flakiness across programming languages. *IEEE Transactions on Software Engineering*, IEEE, v. 49, n. 4, p. 2039–2052, 2022.
- BELL, Jonathan; LEGUNSEN, Owolabi; HILTON, Michael; ELOUSSI, Lamyaa; YUNG, Tiffany; MARINOV, Darko. DeFlaker: Automatically detecting flaky tests. *In: 40th International Conference on Software Engineering*. New York, NY, EUA: ACM, 2018. p. 433–444. ISBN 978-1-4503-5638-1.
- BERTOLINO, Antonia; CRUCIANI, Emilio; MIRANDA, Breno Alexandre Ferreira de; VERDECCHIA, Roberto. **Know your neighbor: fast static prediction of test flakiness**. Pisa,, 2020.
- Box, Inc. **flaky**. 2014. #software#. Disponível em: <https://github.com/box/flaky>.
- CAMARA, Bruno; SILVA, Marco; ENDO, Andre; VERGILIO, Silvia. On the use of test smells for prediction of flaky tests. *In: ELER, Marcelo; ASSUNÇÃO, Wesley K. G. (Ed.). VI Brazilian Symposium on Systematic and Automated Software Testing (SAST’21)*. New York, NY, EUA: ACM, 2021. p. 46–54. ISBN 978-1-4503-8503-9.
- CAMARA, Bruno Henrique Pachulski; SILVA, Marco Aurélio Graciotto; ENDO, Andre Takeshi; VERGILIO, Silvia Regina. What is the vocabulary of flaky tests? an extended replication. *In: 29th IEEE/ACM International Conference on Program Comprehension (ICPC 2021)*. [S.l.]: IEEE/ACM, 2021. p. 444–454.
- CORDEIRO, Marcello; SILVA, Denini; TEIXEIRA, Leopoldo; MIRANDA, Breno; D’AMORIM, Marcelo. Shaker: a tool for detecting more flaky tests faster. *In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2021. p. 1281–1285.
- CORDEIRO, Marcello; SILVA, Denini; TEIXEIRA, Leopoldo; MIRANDA, Breno; D’AMORIM, Marcelo. Shaker: a tool for detecting more flaky tests faster. *In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.]: IEEE, 2021. p. 1281–1285. ISBN 978-1-6654-4784-3. ISSN 1938-4300.
- DUTTA, Saikat; SHI, August; CHOUDHARY, Rutvik; ZHANG, Zhekun; JAIN, Aryaman; MISAILOVIC, Sasa. Detecting flaky tests in probabilistic and machine learning applications.

In: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis. [S.l.: s.n.], 2020. p. 211–224.

ECK, Moritz; PALOMBA, Fabio; CASTELLUCCIO, Marco; BACCHELLI, Alberto. Understanding flaky tests: The developer’s perspective. *In: APEL, Sven; RUSSO, Alessandra (Ed.). Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* New York, NY, EUA: ACM, 2019. p. 830–840. ISBN 9781450355728.

ELOUSSI, Lamyaa. **Determining Flaky Tests from Test Failures.** abr. 2015. Dissertação (mathesis) — University of Illinois at Urbana-Champaign, Urbana, Illinois, EUA, abr. 2015. Disponível em: <http://hdl.handle.net/2142/78543>.

FATIMA, Sakina; GHALEB, Taher A; BRIAND, Lionel. Flakify: A black-box, language model-based predictor for flaky tests. **IEEE Transactions on Software Engineering**, IEEE, v. 49, n. 4, p. 1912–1927, 2022.

FAYYAD, Usama; PIATETSKY-SHAPIRO, Gregory; SMYTH, Padhraic. From data mining to knowledge discovery in databases. **AI Magazine**, Association for the Advancement of Artificial Intelligence (AAAI), Palo Alto, CA, EUA, v. 17, n. 3, p. 37–54, set.–nov. 1996. ISSN 0738-4602.

FENG, Zhangyin; GUO, Daya; TANG, Duyu; DUAN, Nan; FENG, Xiaocheng; GONG, Ming; SHOU, Linjun; QIN, Bing; LIU, Ting; JIANG, Daxin *et al.* Codebert: A pre-trained model for programming and natural languages. **arXiv preprint arXiv:2002.08155**, 2020.

GILL, Geoffrey K; KEMERER, Chris F. Cyclomatic complexity density and software maintenance productivity. **IEEE transactions on software engineering**, v. 17, n. 12, p. 1284–1288, 1991.

HABCHI, Sarra; HABEN, Guillaume; SOHN, Jeongju; FRANCI, Adriano; PAPADAKIS, Mike; CORDY, Maxime; TRAON, Yves Le. What made this test flake? pinpointing classes responsible for test flakiness. *In: IEEE. 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME).* [S.l.], 2022. p. 352–363.

HABEN, Guillaume; HABCHI, Sarra; PAPADAKIS, Mike; CORDY, Maxime; TRAON, Yves Le. A replication study on the usability of code vocabulary in predicting flaky tests. *In: .* [S.l.: s.n.], 2021. p. 219–229.

HARMAN, Mark; O’HEARN, Peter. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. *In: 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM).* [S.l.]: IEEE, 2018. p. 1–23. ISBN 978-1-5386-8291-3. ISSN 1942-5430.

HASHEMI, Negar; TAHIR, Amjed; RASHEED, Shawn. An empirical study of flaky tests in Javascript. *In: .* [S.l.: s.n.], 2022. p. 24–34.

HERZIG, Kim; NAGAPPAN, Nachiappan. Empirically detecting false test alarms using association rules. *In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering.* [S.l.]: IEEE, 2015. p. 39–48. ISSN 0270-5257.

KALLIAMVAKOU, Eirini; GOUSIOS, Georgios; BLINCOE, Kelly; SINGER, Leif; GERMAN, Daniel M.; DAMIAN, Daniela. The promises and perils of mining GitHub. *In: 11th Working Conference on Mining Software Repositories.* New York, NY, EUA: ACM, 2014. p. 92–101. ISBN 978-1-4503-2863-0.

KIM, Sunghun; WHITEHEAD, E James; ZHANG, Yi. Classifying software changes: Clean or buggy? **IEEE Transactions on Software Engineering**, IEEE, v. 34, n. 2, p. 181–196, 2008.

KING, Colin Ian *et al.* **stress-ng**. 2013. Programa de computador. Disponível em: <https://github.com/ColinIanKing/stress-ng>.

KING, Tariq M; SANTIAGO, Dionny; PHILLIPS, Justin; CLARKE, Peter J. Towards a bayesian network model for predicting flaky automated tests. *In: IEEE. 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. [S.l.], 2018. p. 100–107.

LAM, Wing; OEI, Reed; SHI, August; MARINOV, Darko; XIE, Tao. iDFlakies: A framework for detecting and partially classifying flaky tests. *In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. [S.l.]: IEEE, 2019. p. 312–322. ISBN 978-1-7281-1737-9. ISSN 2159-4848.

LAMPEL, Johannes; JUST, Sascha; APEL, Sven; ZELLER, Andreas. When life gives you oranges: Detecting and diagnosing intermittent job failures at mozilla. *In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021. (ESEC/FSE 2021), p. 1381–1392. ISBN 9781450385626. Disponível em: <https://doi.org/10.1145/3468264.3473931>.

LUNDBERG, Scott M; ERION, Gabriel; CHEN, Hugh; DEGRAVE, Alex; PRUTKIN, Jordan M; NAIR, Bala; KATZ, Ronit; HIMMELFARB, Jonathan; BANSAL, Nisha; LEE, Su-In. From local explanations to global understanding with explainable ai for trees. **Nature machine intelligence**, Nature Publishing Group, v. 2, n. 1, p. 56–67, 2020.

LUO, Qingzhou; HARIRI, Farah; ELOUSSI, Lamyaa; MARINOV, Darko. An empirical analysis of flaky tests. *In: 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, EUA: ACM, 2014. p. 643–653. ISBN 978-1-4503-3056-5.

LUO, Qingzhou; HARIRI, Farah; ELOUSSI, Lamyaa; MARINOV, Darko. An empirical analysis of flaky tests. *In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.: s.n.], 2014. p. 643–653.

MANNING, Christopher D.; SCHÜTZE, Hinrich. **Foundations of statistical natural language processing**. [S.l.]: MIT press, 1999.

MEMON, Atif M.; COHEN, Myra B. Automated testing of GUI applications: Models, tools, and controlling flakiness. *In: 35th International Conference on Software Engineering*. Piscataway, NJ, EUA: IEEE, 2013. p. 1479–1480. ISBN 978-1-4673-3076-3. Disponível em: <http://comet.unl.edu/tutorial.php>.

MICCO, John. Flaky tests at google and how we mitigate them. **Online] <https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html>**, 2016.

MICCO, John. **The State of Continuous Integration Testing@ Google.(2017)**. 2017.

MIRANDA, Charles; AVELINO, Guilherme; NETO, Pedro Santos; SILVA, Victor da. Uma análise da co-evolução de teste em projetos de software no GitHub. *In: IX Workshop de Visualização, Evolução e Manutenção de Software (VEM 2021)*. Porto Alegre, RS, Brasil: SBC, 2021. p. 36–40.

- MORÁN, Jesús; AUGUSTO, Cristian; BERTOLINO, Antonia; RIVA, Claudio De La; TUYA, Javier. FlakyLoc: Flakiness localization for reliable test suites in web applications. **Journal of Web Engineering**, River Publishers, v. 19, n. 2, p. 267–296, jun. 2020. ISSN 1544-5976.
- PALOMBA, Fabio; ZAIDMAN, Andy. Does refactoring of test smells induce fixing flaky tests? *In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.]: IEEE, 2017. p. 1–12. ISBN 978-1-5386-0993-4. Paper retracted.
- PARRY, Owain; KAPFHAMMER, Gregory M; HILTON, Michael; MCMINN, Phil. Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models. **Empirical Software Engineering**, Springer, v. 28, n. 3, p. 72, 2023.
- PEDREGOSA, Fabian; VAROQUAUX, Gaël; GRAMFORT, Alexandre; MICHEL, Vincent; THIRION, Bertrand; GRISEL, Olivier; BLONDEL, Mathieu; PRETTENHOFER, Peter; WEISS, Ron; DUBOURG, Vincent *et al.* Scikit-learn: Machine learning in python. **the Journal of machine Learning research**, JMLR. org, v. 12, p. 2825–2830, 2011.
- PEITEK, Norman; APEL, Sven; PARNIN, Chris; BRECHMANN, André; SIEGMUND, Janet. Program comprehension and code complexity metrics: An fmri study. *In: IEEE. 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. [S.l.], 2021. p. 524–536.
- PERSON, Suzette; ELBAUM, Sebastian. Test analysis: Searching for faults in tests (n). *In: IEEE. 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2015. p. 149–154.
- PINTO, Gustavo; MIRANDA, Breno; DISSANAYAKE, Supun; D'AMORIM, Marcelo; TREUDE, Christoph; BERTOLINO, Antonia. What is the vocabulary of flaky tests? *In: GOUSIOS, Georgios; NADI, Sarah (Ed.). 17th International Conference on Mining Software Repositories (MSR)*. New York, NY, EUA: ACM, 2020. ISBN 9781450375177.
- PODGURSKI, Andy; LEON, David; FRANCIS, Patrick; MASRI, Wes; MINCH, Melinda; SUN, Jiayang; WANG, Bin. Automated support for classifying software failure reports. *In: IEEE. 25th International Conference on Software Engineering, 2003. Proceedings*. [S.l.], 2003. p. 465–475.
- PONTILLO, Valeria; PALOMBA, Fabio; FERRUCCI, Filomena. Static test flakiness prediction: How far can we go? **Empirical Softw. Engg.**, Kluwer Academic Publishers, USA, v. 27, n. 7, dec 2022. ISSN 1382-3256. Disponível em: <https://doi.org/10.1007/s10664-022-10227-1>.
- PRESTON-WERNER, Tom; WANSTRATH, Chris; HYETT, P. J.; CHACON, Scott. **GitHub**. 2008. Programa de computador. Disponível em: <https://github.com>.
- PYTEST. **Pytest Documentation**. 2025. Online. Acesso em 08/12/2024. Disponível em: <https://docs.pytest.org/en/stable/>.
- QUINLAN, J. R. Induction of decision trees. *In: . [S.l.]: Springer, 1986. v. 1, n. 1, p. 81–106.*
- REZENDE, Solange Oliveira. **Sistemas inteligentes: fundamentos e aplicações**. 1. ed. Barueri, SP, Brasil: Manole, 2005. 550 p. ISBN 8520416837.
- ROMANO, Alan; SONG, Zihe; GRANDHI, Sampath; YANG, Wei; WANG, Weihang. An empirical analysis of UI-based flaky tests. *In: 43rd International Conference on Software Engineering*. [S.l.]: IEEE, 2021. p. 1585–1597.

- SHI, August; GYORI, Alex; LEGUNSEN, Owolabi; MARINOV, Darko. Detecting assumptions on deterministic implementations of non-deterministic specifications. *In: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. [S.l.: s.n.], 2016. p. 80–90.
- SHI, August; LAM, Wing; OEI, Reed; XIE, Tao; MARINOV, Darko. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. *In: APEL, Sven; RUSSO, Alessandra (Ed.). Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, EUA: ACM, 2019. p. 545–555. ISBN 9781450355728.
- SILVA, Denini; TEIXEIRA, Leopoldo; D’AMORIM, Marcelo. Shake it! detecting flaky tests caused by concurrency with Shaker. *In: 2020 IEEE International Conference on Software Maintenance and Evolution*. [S.l.]: IEEE, 2020. p. 301–311.
- SILVA, Denini Gabriel. **Using noise to detect test flakiness**. fev. 2022. 62 p. Dissertação (mathesis) — Universidade Federal de Pernambuco – Centro de Informática, Recife, PE., fev. 2022. Disponível em: <https://repositorio.ufpe.br/handle/123456789/44567>.
- SORATTO, Rafael Rampim. **Vocabulário de testes instáveis em Javascript**. Campo Mourão, PR,: [s.n.], 2022. 62 p.
- SORATTO, Rafael Rampim; SILVA, Marco Aurélio Graciotto. Vocabulary of flaky tests in javascript. *In: XXII Simpósio Brasileiro de Qualidade de Software*. New York, NY, USA: ACM, 2023. p. 1–10.
- SOUZA, Lucas BL De; CAMPOS, Eduardo C; MAIA, Marcelo de A. Ranking crowd knowledge to assist software development. *In: Proceedings of the 22nd International Conference on Program Comprehension*. [S.l.: s.n.], 2014. p. 72–82.
- SUN, Xian; WANG, Bing; WANG, Zhirui; LI, Hao; LI, Hengchao; FU, Kun. Research progress on few-shot learning for remote sensing image interpretation. **IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing**, IEEE, v. 14, p. 2387–2402, 2021.
- VERDECCHIA, Roberto; CRUCIANI, Emilio; MIRANDA, Breno; BERTOLINO, Antonia. Know you neighbor: Fast static prediction of test flakiness. **IEEE Access**, IEEE, 2021.
- WELKER, Kurt D. The software maintainability index revisited. **CrossTalk**, Citeseer, v. 14, p. 18–21, 2001.
- WITTEN, Ian H.; FRANK, Eibe. Data mining: practical machine learning tools and techniques with java implementations. **SIGMOD Rec.**, Association for Computing Machinery, New York, NY, USA, v. 31, n. 1, p. 76–77, mar. 2002. ISSN 0163-5808. Disponível em: <https://doi.org/10.1145/507338.507355>.
- WITTEN, Ian H; FRANK, Eibe; TRIGG, Leonard E; HALL, Mark A; HOLMES, Geoffrey; CUNNINGHAM, Sally Jo. Weka: Practical machine learning tools and techniques with java implementations. 1999.
- ZHANG, Sai; JALALI, Darioush; WUTTKE, Jochen; MUS, Kundefinedvan; LAM, Wing; ERNST, Michael D.; NOTKIN, David. Empirically revisiting the test independence assumption. *In: 2014 International Symposium on Software Testing and Analysis*. New York, NY, EUA: ACM, 2014. p. 385–396. ISBN 978-1-4503-2645-2.