UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

RAFAEL RAMPIM SORATTO

VOCABULÁRIO DE TESTES INSTÁVEIS EM JAVASCRIPT

CAMPO MOURÃO

2022

RAFAEL RAMPIM SORATTO

VOCABULÁRIO DE TESTES INSTÁVEIS EM JAVASCRIPT

Vocabulary of flaky tests in Javascript

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Ciência da Computação do Curso de Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Marco Aurélio Graciotto Silva

CAMPO MOURÃO

2022

RAFAEL RAMPIM SORATTO

VOCABULÁRIO DE TESTES INSTÁVEIS EM JAVASCRIPT

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Ciência da Computação do Curso de Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 15/junho/2022

Marco Aurélio Graciotto Silva
Doutorado em Ciências de Computação e Matemática Computacional
Universidade Tecnológica Federal do Paraná

Juliano Henrique Foleiss
Doutorado em Engenharia Elétrica
Universidade Tecnológica Federal do Paraná

Reginaldo Ré
Doutorado em Ciências de Computação e Matemática Computacional
Universidade Tecnológica Federal do Paraná

CAMPO MOURÃO
2022

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.

A Secretaria do Curso, pela cooperação.

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.

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 teste instáveis (*flaky test*). Este tipo de instabilidade pode atrasar o lançamento do software e reduz a confiança dos testes. Uma forma de identificar tais testes instáveis é pela reexecução dos testes, mas isso possui um custo computacional elevado. Uma alterativa 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 que abordam aplicações em Java, embora outras linguagens, como Javascript, também sejam amplamente utilizadas no desenvolvimento de software. **Objetivo:** O objetivo deste trabalho foi a identificação de testes instáveis em aplicações Javascript sem realizar a execução do mesmo, visando poupar recursos relacionados. **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. A seguir foi criado um modelo de classificação, considerando o vocabulário de instabilidade na linguagem Javascript, construído a partir do conjunto de dados estabelecido neste trabalho. **Resultados:** Observamos que os oito algoritmos de aprendizado de máquina utilizados tiveram bom desempenho na distinção de testes quanto à instabilidade. Em particular, a regressão logística teve a melhor precisão (0,984) e foi melhor em termos de revocação (0,98). O vocabulário instável obtido contém palavras relacionadas com espera assíncrona (por exemplo, ‘then’, ‘await’, ‘return’) e à visualização e interação com usuário (por exemplo, ‘layout’ e ‘click’). Também Encontramos duas palavras associadas a casos de teste instáveis que estão relacionadas com a utilização do framework Cypress, são elas : ‘cy’ e ‘getByTestID’. O termo que possui maior relevância em termos de ganho de informação é o ‘then’, que está fortemente associado a instabilidades ocasionadas pela espera assíncrona. **Conclusões:** Neste trabalho são apresentados resultados relevantes para identificação de *flaky tests* em projetos que utilizam Javascript. São necessários mais estudos empíricos para consolidar a abordagem de vocabulário instável como uma técnica confiável para identificação de instabilidades em testes.

Palavras-chave: teste de software; teste instável; javascript; vocabulário.

ABSTRACT

Context: Regression testing is a software verification and validation activity in modern software engineering. In this activity, tests can fail without any implementation change, characterizing a flaky test. Flaky tests may delay the release of the software and reduce testing confidence. One way to identify flaky tests is by re-running the tests, but this has a high computational cost. An alternative to re-execution is the static analysis of the code of the test cases, identifying patterns related to flaky tests. Considering this approach, however, to the best of your knowledge, only works that address Java applications are observed, although other languages, such as Javascript, are also widely used in software development. **Objective:** The objective of this work was to identify flaky tests in Javascript applications without executing them, saving computing resources. **Method:** A dataset was built with flaky test cases extracted from open projects hosted on Github that are implemented in Javascript. Next, a classification model was created, considering the source code of flaky tests in the Javascript language, built from the classification model established in this work. **Results:** We observed that the learning algorithms considered in this study achieved a good performance in the classification of tests regarding flakiness. In particular, logistic regression had the best accuracy (0.984) and was the best in terms of recall (0.98). The vocabulary for flaky tests contains words related to asynchronous wait (e.g., ‘then’, ‘await’, ‘return’) and visualization and user interaction (e.g., ‘layout’ and ‘click’). We also found two words associated with flaky test cases related to the usage of the Cypress framework: ‘cy’ and ‘getByTestID’. The term with the most significant relevance to the information gain is ‘then’, which is mainly associated with asynchronous waits. **Conclusions:** This work presents relevant results for identifying flaky tests in projects that use Javascript. Further studies are required to consolidate the reliable classification of tests regarding flakiness methodology.

Keywords: software testing; flaky test; javascript; vocabulary.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTAGEM DE CÓDIGOS FONTE

LISTA DE ABREVIATURAS E SIGLAS

Siglas

ATAF	All Tests Are Flaky
AUC	Area Under The Curve
CD	Continuous Deployment
CI	Continuous Integration
GUI	Graphical User Interface
JVM	Java Virtual Machine
K-NN	K Nearest Neighbors
LDA	Linear Discriminant Analysis
MCC	Matthews Correlation Coefficient
ML	Machine Learning
PR	Pull Request
SVM	Support Vector Machines

SUMÁRIO

1 INTRODUÇÃO

O teste de regressão é uma atividade importante para verificação e validação de sistemas presente na engenharia de software moderna (MIRANDA *et al.*, 2021). Ele é responsável por garantir que as funcionalidades do software não sejam danificadas com futuras atualizações, garantindo o comportamento esperado na especificação do produto.

Assim como outras funcionalidades do software, o teste precisa ser implementado no código-fonte para então ser executado. Espera-se que o resultado do teste seja mantido se o código-fonte não for alterado. Porém, ele pode variar (e.g. de falso para verdadeiro e vice-versa) em diferentes execuções sem alterações de implementação. Este evento caracteriza o teste como *flaky test* ou teste instável.

A instabilidade na maioria das vezes é resultado da dificuldade de se manter cenários de testes idênticos durante múltiplas execuções. Por exemplo, durante a execução de testes de sistemas é difícil garantir que a disponibilidade da rede se mantenha, que as plataformas de execução sejam idênticas, e que os fatores ambientais envolvidos não interfiram no processo de teste (MORáN *et al.*, 2020). Este cenário instável é fértil para revelar problemas relacionados à assincronicidade, diferença de plataformas e renderização de recursos (LUO *et al.*, 2014; ROMANO *et al.*, 2021).

Flaky test é uma realidade inconveniente em diversos projetos de software, e pode ou não ser relevante para o bom funcionamento do software, porém consome muito recurso de sistemas de implantação (LUO *et al.*, 2014; HERZIG; NAGAPPAN, 2015; MICCO, 2016; MICCO, 2017; HARMAN; O'HEARN, 2018a). Trabalhos anteriores mostraram que *flaky tests* são comuns em suítes de teste de regressão (BELL *et al.*, 2018; ECK *et al.*, 2019; HARMAN; O'HEARN, 2018b; LISTFIELD, 2017; LUO *et al.*, 2014).

Este termo aparece também em empresas sólidas no ramo de tecnologia, tais como: Microsoft (HERZIG; NAGAPPAN, 2015; LAM *et al.*, 2019a), Spotify (PALMER, 2019), Google (MICCO, 2016; MICCO, 2017), e Facebook (HARMAN; O'HEARN, 2018a).

Na empresa Google, em 2016, foi identificado que 1,5% dos testes apresentavam instabilidades (MICCO, 2016). Com avanço dos estudos em 2017, a propriedade *flakiness* foi identificada como um grande problema que pode atrasar as entregas do produto. No ano de 2017, foi relatado 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.

Na empresa Facebook foi proposto um sistema chamado Sapienz desde setembro de 2017 para projetar casos de teste, localizar e fazer a triagem de falhas para desenvolvedores e monitorar suas correções. De acordo com relatórios deste sistema, a detecção de testes instáveis é uma área de pesquisa em aberto que necessita de estudos empíricos. Ainda faltam informações sobre as principais causas e correções do termo *flakiness* em testes (ALSHAHWAN *et al.*, 2018). Considerando que instabilidades em testes são inevitáveis, no Facebook, propuseram a

seguinte abordagem: assumir que todos testes possuem seu grau de instabilidade é mais seguro para que o desenvolvedor tenha maior atenção durante sua implementação e também reconheça os padrões de instabilidade em cada cenário de teste. Essa abordagem chama-se All Tests Are Flaky (ATAF) (HARMAN; O'HEARN, 2018a). A perspectiva ATAF fortalece a ideia de que a instabilidade é inevitável em alguns cenários após diversas reexecuções.

A maneira mais comum para se identificar *flaky tests* é a reexecução do mesmo, geralmente com a variação das condições do teste (*e.g.*, diferentes plataformas). Muito recurso é gasto para identificar instabilidades pois cada execução tem um custo associado (LAM *et al.*, 2019b; MICCO, 2016). Esta abordagem para identificação de *flaky tests* pode ser considerada dinâmica. Abordagens estáticas, que não utilizam a reexecução dos testes, podem auxiliar o processo de identificação de instabilidades com intuito de reduzir os recursos gastos para isto. Existem abordagens mistas que identificam a instabilidade utilizando a combinação de duas técnicas: pela diferença de duas versões de código e a aplicação de uma reexecução (BELL *et al.*, 2018).

Existem abordagens de aprendizado de máquina que exploram regras de associação entre etapas de teste individuais em dezenas de milhões de alarmes de teste falsos (HERZIG; NAGAPPAN, 2015). Neste sentido, a predição de instabilidade é uma abordagem que visa reduzir o custo da reexecução de testes (BERTOLINO *et al.*, 2020). Outra abordagem é a predição de instabilidade a partir da criação de uma lista de 'tokens' para memorizar identificadores relacionados à instabilidade, surgindo o conceito de vocabulário instável, com estudos relacionados à aplicações Java (PINTO *et al.*, 2020; CAMARA *et al.*, 2021b).

O custo baixo para identificação estática de instabilidades em testes motiva o desenvolvimento de abordagens de predição, por exemplo, a abordagem de vocabulário instável. A abordagem de vocabulário instável une as áreas de engenharia de software e inteligência artificial para sugerir ao desenvolvedor que o código possui a chance de falhar sem a necessidade de múltiplas execuções de testes em ambientes de produção de software. São exemplos de ambientes de aplicações: pipelines, Continuous Integration (CI), Continuous Deployment (CD), sugestão de código e *linters*. Para avaliar a técnica de vocabulário em um contexto diferente dos trabalhos apresentados anteriormente, o presente trabalho visa utilizar a predição de *flakiness* em aplicações Javascript.

O principal objetivo deste trabalho é identificar instabilidades em aplicações Javascript, utilizando a abordagem de vocabulário instável. Desta forma, as metas deste trabalho abrangeram: a criação de um conjunto de dados sobre *flaky tests* para Javascript, utilizando um conjunto de dados inicial contendo casos de teste instáveis (ROMANO *et al.*, 2021); criação e avaliação de modelos de classificação, considerando o vocabulários dos casos de testes instáveis.

O primeiro requisito para análise do código *flaky* é sua identificação. Para criar um vocabulário de *flakiness* é necessário um conjunto de dados com o código relacionado aos casos de teste *flaky*. Foi utilizado um estudo no conjunto de dados sobre citações de instabilidades

por desenvolvedores no Github (ROMANO *et al.*, 2021). Com estes dados pudemos localizar informações essenciais para a predição, por exemplo, o conteúdo do caso de teste *flaky*.

A partir do conjunto de dados, foi utilizada a abordagem de mineração de dados em casos de testes para encontrar instabilidades. Para isto, foram criados modelos de classificação de instabilidade. Os resultados obtidos pelo trabalho são: conjuntos de dados contendo testes rotulados como instáveis e normais, um experimento utilizando a abordagem de vocabulário instável no Javascript, e por fim, um conjunto de palavras que estão fortemente relacionadas à instabilidades em casos de testes Javascript. Levantamos duas questões de pesquisa relacionadas com a validade de um vocabulário instável no cenário proposto.

Este trabalho é 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 no Javascript 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* de palavras Javascript. Os resultados quanto da avaliação da qualidade dos classificadores são apresentados e discutidos no Capítulo 4. Finalmente, conclusões deste estudo frente ao objetivo proposto e delineamento de trabalhos futuros estão descritos no Capítulo 5.

2 TRABALHOS RELACIONADOS

Nesse capítulo é definido o que é um caso de teste instável ou *flaky test* (Seção 2.1) e apresentadas técnicas para identificação de casos de teste instáveis (Seção 2.2). Também são apresentados alguns conjuntos de dados sobre *flaky tests* (Seção 2.3). Por fim, apresentamos as considerações finais do capítulo (Seção 2.4).

São apresentadas diferentes abordagens, estáticas e dinâmicas, para identificação de instabilidade em testes. A abordagem dinâmica utiliza a reexecução do caso de teste enquanto a abordagem estática utiliza informações presentes em testes rotulados como instáveis para identificar novas instabilidades. Além de adotar uma abordagem estática para identificação de instabilidades apresentamos dados empíricos sobre *flaky tests* em aplicações Javascript.

2.1 Caso de teste instável (flaky test)

O *flaky test* representa uma variação no resultado do teste sem variação de implementação, em diferentes execuções. Este comportamento não determinístico remove parte da confiança nos testes de regressão e geralmente é 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 até que ele não seja corrigido. Já a instabilidade quebra o valor somente algumas vezes durante uma sequência de testes sem nenhuma alteração no código fonte (ELOUSSI, 2015).

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 para aplicação após a sequência de eventos, ou certa interface esperada. Isto possibilita a simulação de ações que são 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. 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. Todos esses elementos podem variar de certa maneira inesperada, ocasionando um *flaky test* (MEMON; COHEN, 2013).

O contexto da instabilidade é visível no Pull Request (PR) (#17935) de uma *branch* para o projeto Angular [1]. Visualizando o único *commit* deste PR é encontrada a seguinte mensagem do desenvolvedor: “test: menu tests flaky due to missing animation flush”. Neste *commit* são identificados casos de testes instáveis da seguinte maneira: os casos de testes testes nos quais foram inseridos um atraso de tempo possuem instabilidade. Este *flaky test* é apresentado na Listagem 1 e refere-se ao tempo de animação, sendo resolvido com um *delay* ou *tick* de 500

[1] https://github.com/angular/components/pull/17935

ms na linha 7. Note que este atraso é inserido antes do *expect* para garantir que a transição de estado da aplicação tenha terminado.

Listagem 1 – Exemplo de flaky test coletado do Github

```
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 }));
```

No caso do caso de teste apresentado na Listagem 2, implementado em TypeScript, é possível visualizar o método de correção de *flaky test*: utilizando *timeouts* aninhados atrasam a execução garantindo que a mudança de foco tenha terminado. De acordo com o desenvolvedor que realizou este *commit* o foco precisa mudar antes do *assert* da linha 11. Este método é bem semelhante ao *tick* mencionado na Listagem 1.

Outro exemplo de *flaky test* ocasionado por tempo de execução excedido é apresentado na Listagem 3. Como método de alerta o desenvolvedor configurou o parâmetro *timeout* $= 10000ms$ em uma função de acesso a uma API. Em uma execução normal após o resultado da API na linha 2 o foco é alterado, e uma navegação de páginas ocorre conforme é apresentado na linha 6 da Listagem 3. Porém em uma execução instável, a função presente na linha 2 da Listagem 3 demora mais de 10000 mili segundos retornar o resultado, ocasionando um erro nos *asserts* da linha 5 e 7. Ainda sobre a Listagem 3, configura-se um *timeout* pelo desenvolvedor evitando que muito tempo seja perdido para executar a função da linha 2, levantando uma exceção e interrompendo a execução do teste.

Na Listagem 4 é apresentada uma asserção de ordem instável quando executada na plataforma Firefox. Para solucionar essa ordem inesperada, o desenvolvedor ordenou ambos elementos antes de realizar a asserção na linha 8. Outro ponto relevante no *commit* é o comentário do desenvolvedor: ‘fix flakey test in firefox’.

2.2 Técnicas para identificação

Existem técnicas para prevenção, identificação, previsão e correção de *flaky tests* (BELL *et al.*, 2018; PINTO *et al.*, 2020; CAMARA *et al.*, 2021a; MORáN *et al.*, 2020; HARMAN;

Listagem 2 – Exemplo de flaky test com timeout

```
it("returns focus to overlay if enforceFocus=true", done => {
            let buttonRef: HTMLElement;
            const focusBtnAndAssert = () => {
                buttonRef.focus();
                // nested setTimeouts delay execution until the next frame, not
                // just to the end of the current frame. necessary to wait for
                // focus to change.
                setTimeout(() => {
                    setTimeout(() => {
                        wrapper.update();
                        assert.notStrictEqual(buttonRef, document.activeElement);
                        done();
                    });
                });
            };

            wrapper = mount(
                <div>
                    <button ref={ref => (buttonRef = ref)} />
                    <Overlay enforceFocus={true} inline={false} isOpen={true}>
                        <input ref={ref => focusBtnAndAssert()} />
                    </Overlay>
                </div>,
                { attachTo: testsContainerElement },
            );
        });
```

O'HEARN, 2018a). Um grande problema a ser tratado na etapa de identificação da instabilidade é o número de execuções necessárias para isto. A reexecução é uma abordagem dinâmica, pois necessita executar o caso de teste. Por exemplo, ao executar 10 vezes uma bateria de testes é possível que nenhuma instabilidade ocorra; porém, se este número de execuções aumentar para 100 a chance de uma instabilidade aparecer aumenta. Da mesma forma, ao executar mil vezes um caso de teste, a chance de ocorrer uma instabilidade aumenta. Seguindo essa lógica, para obter uma identificação de instabilidade eficiente com a abordagem de reexecução é necessário um número alto de reexecuções (PINTO *et al.*, 2020). Para diminuir o custo deste processo de identificação, surgem técnicas estáticas, que identificam instabilidades utilizando informações estáticas presentes em casos de testes. Por exemplo, existe uma abordagem para identificação de *flaky tests* que utiliza as diferenças entre duas versões do código fonte de testes para localizar instabilidades (MORáN *et al.*, 2020).

Um tipo comum de *flaky tests* são testes dependentes de ordem, que passam ou falham dependendo da ordem em que os testes são executados. A correção de testes dependentes da

Listagem 3 – Exemplo de flaky test relacionado com timeout de API

```
it(`Focus router wrapper after navigation to regular page (from
    index)`, () => {
     cy.visit(`/`).waitForAPIorTimeout(`onRouteUpdate`, { timeout:
    10000 })

     cy.changeFocus()
     cy.assertRouterWrapperFocus(false)
     cy.navigateAndWaitForRouteChange(`/page-2/`)
     cy.assertRouterWrapperFocus(true)
  })
```

Listagem 4 – Exemplo de flaky test relacionado à ordem dos elementos HTML

```
function sortAttributes(html) {
  return html.replace(regex)/gi, (s, pre, attrs, after) => {
    let list = attrs.match(regex/gi).sort( (a, b) => a>b ? 1 : -1 )
    ;
    if (after.indexOf('/')) after = '></'+pre+'>';
    return '<' + pre + list.join('') + after;
  });
}
expect(sortAttributes(scratch.innerHTML)).to.equal(sortAttributes(
    '<div foo="bar" j="4" i="5">inner</div>'));
```

ordem geralmente é demorada. Existem trabalhos relacionados com a detecção de testes instáveis relacionados com ordem de execução. De modo geral, ele envolve a alteração da ordem da execução dos casos de teste. Caso algum caso de teste inadvertidamente dependa de outro caso de teste, a alteração da ordem de execução permite evidenciar essas instabilidades Shi *et al.* (2019). Esta abordagem é interessante pois reduz o número de execuções necessárias para encontrar uma instabilidade do tipo 'ordem de execução'.

Instabilidades são descobertas após diversas execuções de testes, tornando sua identificação lenta na maioria das vezes (ELOUSSI, 2015). O *flaky test* encontrado ou corrigido em projetos de código aberto torna-se útil para os pesquisadores identificarem futuras instabilidades criando abordagens variadas para isto. Por exemplo, testes rotulados como instáveis podem ser úteis para prever futuras instabilidades utilizando a abordagem de reexecução (LUO *et al.*, 2014; ELOUSSI, 2015).

Portanto, existem as seguintes abordagens para identificação e *flaky tests*: dinâmica, estática e mista. Onde a dinâmica utiliza a reexecução de testes, a estática utiliza dados presentes nos testes para predição de instabilidade, e a mista utiliza ambas abordagens. Neste capítulo são descritos trabalhos relacionados com *flaky tests* que utilizam tais abordagens. Mais além, é descrita a motivação para utilizar a abordagem estática de vocabulário instável neste trabalho.

2.2.1 Identificação por reexecução

A abordagem mais amplamente conhecida para identificar *flaky tests* são múltiplas execuções do teste até a falha ocorrer sem alterações no código-fonte. 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. Pode-se considerar o adiamento do primeiro teste executado em tais casos, utilizando uma nova máquina virtual para cada caso de teste (ELOUSSI, 2015).

Existem três abordagens dinâmicas para identificação de *flaky tests* que apresentam bons resultados: adiando a reexecução, utilizando reexecução em diferentes ambientes (e.g. Java Virtual Machine (JVM) para testes de Java), e cruzando a cobertura de teste com as alterações mais recentes (ELOUSSI, 2015).

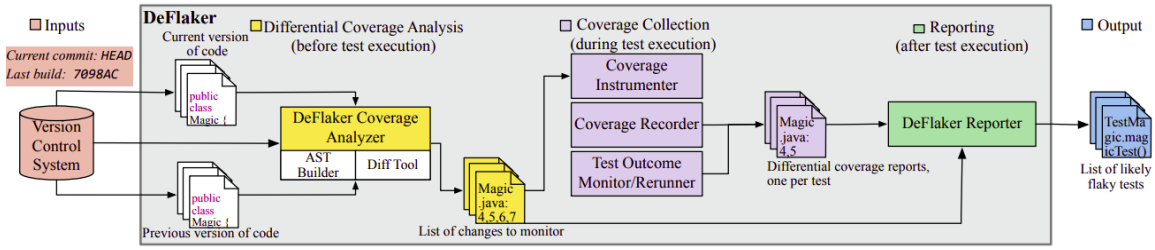
Os testes de sistemas são um grande desafio pois possuem comunicações complexas e assíncronas, simultaneidade entre os clientes-servidores e um padrão dos recursos empregados. É difícil manter a condição destes testes pois eles podem ser executados de formas variadas, como por exemplo: com diversos fatores ambientais, diferentes disponibilidades de rede e memória, e até mesmo com variação da resolução da tela. Essa condição variável pode resultar em instabilidades nos testes. O trabalho "*FlakyLoc*" utiliza a reexecução em diferentes configurações de teste utilizando uma abordagem combinatória para localizar a instabilidade. Neste estudo, a combinação de 64 configurações por fator ambiental realiza-se a análise utilizando a metodologia "*spectrum-based*" para se obter um *ranking* dos fatores de causa (MORáN *et al.*, 2020).

2.2.2 Identificação por análise diferencial de cobertura

Durante o processo de detecção de erros em um software, a suíte de testes é responsável por verificar regressões de funcionalidades durante atualizações de código fonte. A instabilidade de um teste revela-se como um evento não determinístico que pode retardar o ciclo de desenvolvimento. Visando reduzir o tempo gasto durante múltiplas execuções para identificar instabilidades é proposta a abordagem de análise diferencial de cobertura chamado *DeFlaker* (BELL *et al.*, 2018). O *DeFlaker* monitora a cobertura das últimas alterações de código e rotula como instável testes com falha recente sem alterações no código fonte, utilizando 3 fases de processamento conforme apresentado na Figura 1.

Na primeira fase esse identificador de instabilidades recebe como entrada dois dados de versionamento do projeto, são eles: um identificador de commit atual e o identificador do último *build* do projeto. Essas duas informações fornecem o estado atual do código e o estado anterior, utilizados no método de análise diferencial da cobertura de testes. Este método é utilizado antes da execução. O analisador de diferenças utiliza duas informações: um arquivo de diferenças

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).

.

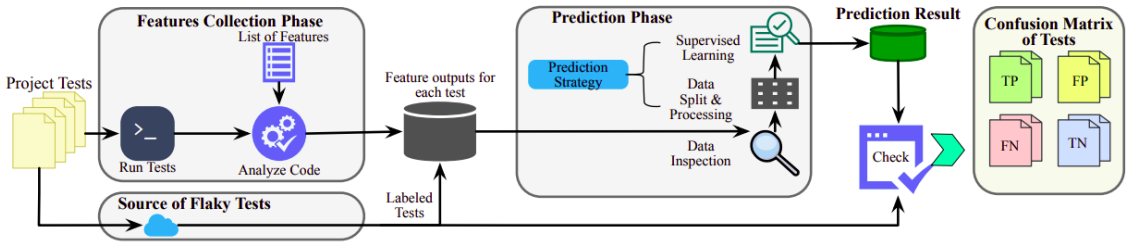
sintáticas e um *AST builder* para identificar uma lista de mudanças a serem rastreadas para os arquivos de origem do programa.

Na segunda fase o DeFlaker utiliza a coleção de cobertura obtida durante a execução do teste, utilizando um instrumentador e gravador de cobertura e um monitor de reexecução. A terceira fase é o relatório pós execução, que resulta na lista de instabilidades de um projeto, utilizando o método de identificação por análise diferencial de cobertura.

Esse sistema foi implementado durante o processo de *build* de 96 projetos Java no TravisCI, e encontrando 87 *flaky tests* até então desconhecidos em 10 desses projetos. O DeFlaker também executou experimentos em históricos de projetos, detectando 1.874 testes instáveis de 4.846 falhas, com uma taxa baixa de falsos positivos (1,5%). DeFlaker teve um recall maior (95,5% vs. 23%) de *flaky tests* confirmados do que o detector de teste instáveis padrão da Maven, que emprega reexecução simples (BELL *et al.*, 2018).

A ferramenta chamada *FlakyFlagger* propõe a predição de instabilidade com base em recursos comportamentais semelhantes (ALSHAMMARI *et al.*, 2021). Esta ferramenta coletou instabilidades executando 10 mil vezes suítes de testes em 24 projetos e rotulou corretamente grande parte dos dados de teste, motivando trabalhos a utilizar técnica de predição para economizar tempo dos desenvolvedores utilizando o a abordagem ilustrada na Figura 2.

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



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

De acordo com a 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 anali-

sadas. 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, sendo possível avaliar os falsos positivos para estimar a porcentagem de erro.

O sistema chamado *Flast* adota uma abordagem estática para predição de instabilidade. Nesta processo, ele realiza etapas de modelagem de espaço vetorial, pesquisa de similaridade, redução de dimensionalidade e classificação de *flaky tests*. Para avaliar a eficácia desta predição utilizaram-se 13 projetos do mundo real, com um total de 1.383 testes instáveis e 26.702 testes não instáveis. Os resultados coletados demonstram que o *Flast* fornece uma abordagem rápida que pode ser usada para orientar a nova execução do teste ou para permitir a inclusão de novos testes potencialmente instáveis em projetos Java (VERDECCHIA *et al.*, 2021).

2.2.3 Test Smells

A abordagem chamada *test smells* é um conjunto de sinais de más escolhas na codificação de teste (DEURSEN *et al.*, 2001). Estes sinais podem alertar ou até identificar possíveis falhas no teste, motivando diversos estudos relacionados (DEURSEN *et al.*, 2001; BAVOTA *et al.*, 2012; TUFANO *et al.*, 2016). Em geral, eles são introduzidos durante a criação do teste e duram por muito tempo sobrevivendo até milhares de *commits* em ambientes de produção.

Por estar relacionado com testes e com falhas (intermitentes ou não), este assunto foi objeto de estudo em conjunto com *flaky tests* (CAMARA *et al.*, 2021a). A ideia principal nesta abordagem é que a instabilidade em testes pode deixar 'rastros', por exemplo, *test smells*.

Foi realizado estudo empírico utilizando *tests smells* como preditores de *flaky tests* em projetos de contextos cruzados com a finalidade de otimizar o desempenho de classificadores para prever instabilidades (CAMARA *et al.*, 2021a). Para isto foram analisados: o ganho de informação utilizando cada *test smell* em relação ao dicionário instável. Para utilizar esta abordagem foi utilizado o conjunto de dados construído por (PINTO *et al.*, 2020) com base em 24 projetos apresentados no trabalho de (BELL *et al.*, 2018). Este conjunto de dados tem 49.919 casos de teste: 44.428 não instáveis, 5.069 instáveis. Para avaliar os modelos de perspectiva de projeto cruzado, usamos os dados de um trabalho (CAMARA *et al.*, 2021b), construído com base nos projetos idFlakies de Lam *et al.* (2019c). Este conjunto de dados contém apenas 422 *flaky tests* de 72 projetos diferentes. Utilizando estes dados e a abordagem de *test smells*, foram levantadas as seguintes questões de pesquisa por Camara *et al.* (2021a):

1. Com que precisão podemos prever *flaky tests* utilizando *test smells* ?

Resposta: Os resultados mostram que a técnica de *test smells* pode ser utilizada para predição de instabilidade. Porém, o desempenho cai consideravelmente no contexto interprojeto. 8 Modelos de classificação tiveram resultados razoáveis Exceto por Naive-Bayes, atingiram valores em uma faixa de 74% até 83% de precisão, recall, F1-Score e AUC. Todos MCC foram superiores a 0,5 onde

1 é uma classificação perfeita. O melhor resultado foi obtido com Random Forest Camara *et al.* (2021a).

2. Quais *test smells* estão mais fortemente associados à previsão de instabilidades?

   Resposta: Os *test smells* que mais estão relacionados à previsão de instabilidades são: TestSleepy Test e Constructor Initialization Camara *et al.* (2021a). Isso é demonstrado em certo conjunto de dados pela distribuição de números em testes instáveis e também pelo ganho de informação.

3. Como a abordagem baseada em *test smells* se compara à abordagem baseada em vocabulário existente?

   Resposta: O objetivo foi comparar os resultados obtidos com os abordagem baseada em vocabulário (CAMARA *et al.*, 2021a; PINTO *et al.*, 2020). O uso de *test smells* pode ser uma boa alternativa para superar algumas limitações da abordagem baseada em vocabulário, e os modelos baseados no *test smell* podem ser mais generalizáveis.

As abordagens que utilizam reexecução geralmente são mais demoradas, e consomem mais recursos. As abordagens existentes baseadas no uso de predição de instabilidade podem ser sensíveis ao contexto e propensas a sobre-ajuste, apresentando baixo desempenho quando executado em um cenário de projeto cruzado. Este problema do sobre-ajuste em um cenário de projeto cruzado ocorreu em trabalhos de *test smells* e vocabulário instável. Este é um problema pervasivo em Machine Learning (ML) presente também no contexto de projetos de software. Este trabalho não verifica o desempenho dos classificadores com projetos cruzados pois os trabalhos anteriores já relataram uma diferença na precisão da predição. Porém, avaliamos se existe a intersecção de palavras em vocabulários de projetos de diferentes contextos e linguagens para verificar a generalidade das palavras instáveis.

2.2.4 Identificação de instabilidades com Vocabulário Instável

Abordagens estáticas foram propostas para reduzir o custo de identificação de instabilidade geralmente utilizando técnicas de ML (HERZIG; NAGAPPAN, 2015; PINTO *et al.*, 2020; KING *et al.*, 2018). Por exemplo, a abordagem chamada vocabulário instável, visa encontrar identificadores textuais que estão fortemente relacionados com *flaky tests*. Nesta técnica é necessário extrair identificadores de casos de testes (e.g. nome de métodos) e verificar quais identificadores estão fortemente associados à instabilidade. Quando esses identificadores aparecem recorrentemente em testes instáveis pode-se utilizar modelos de classificação para se construir um vocabulário instável (PINTO *et al.*, 2020).

O primeiro trabalho sobre o vocabulário instável utilizou o conjunto de dados construído por Pinto *et al.* (2020) com base em 24 projetos analisados pelo DeFlaker (BELL *et al.*, 2018). Utilizando estes dados como base, o autor apresenta quatro importantes questões de pesquisa, sendo a última, uma justificativa para criação de vocabulários instáveis (PINTO *et al.*, 2020):

1. Quão prevalentes e elusivos são os *flaky tests*?

   Resposta: Por meio da reexecução de casos de testes obteu-se o resultado de que este problema afeta 6 dos 24 projetos analisados, em um total de 86 *flaky tests*. Os resultados indicam que a instabilidade é um problema comum em projetos relacionados a entrada e saída de dados. Além disso, detectar *flaky tests* com a reexecução do teste é um desafio.

2. Com que precisão pode-se prever a instabilidade do teste com base em identificadores de código-fonte nos casos de teste?

   Resposta: Todos os classificadores tiveram um bom desempenho no conjunto de dados proposto. No geral, Random Forest foi o classificador que melhor desempenhou.

3. Qual o valor de novas *features* para o classificador?

   Resposta: Embora o impacto de algumas etapas de pré-processamento seja insignificante, a divisão de identificadores tem um impacto positivo no desempenho do classificador.

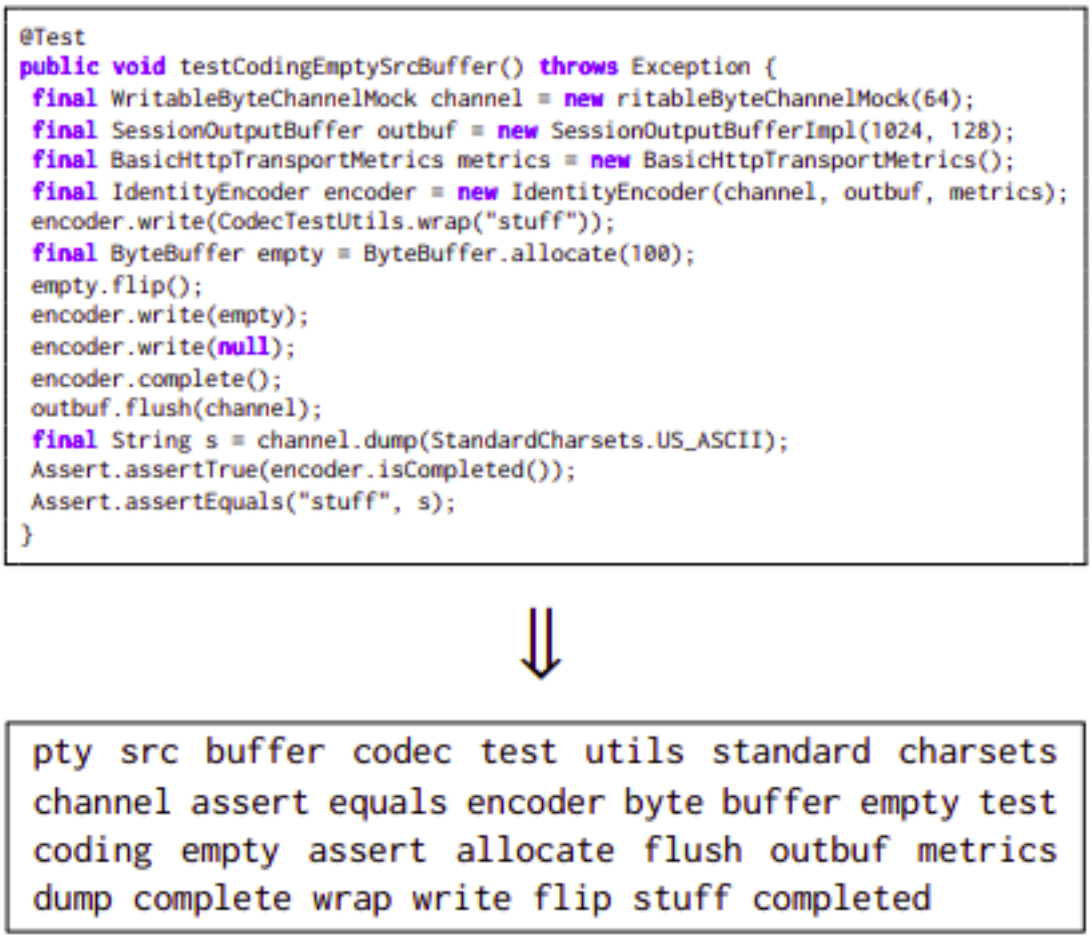
4. Quais identificadores de código de teste estão fortemente associados com *flaky tests*?

   Resposta: O vocabulário associado aos testes flaky contém palavras como ‘job’, ‘table’, ‘action’, muitas das quais estão associados à execução de tarefas remotamente e/ou usando uma fila de eventos.

No primeiro trabalho sobre vocabulário instável foram encontrados somente 86 casos de testes instáveis utilizando a reexecução pois são necessárias mais de 100 execuções para encontrar instabilidades. Este fato mostra que a detecção utilizando reexecução é um desafio que pode ser mitigado com a análise estática (PINTO *et al.*, 2020).

Vocabulários necessitam que os casos de testes sejam transformados em identificadores textuais ou características para utilização do método de predição. Para se utilizar dados textuais são 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 (PINTO *et al.*, 2020). O resultado deste pré-processamento textual é apresentado na parte inferior da Figura 3.

Figura 3 – Extração de identificadores textuais em casos de testes



```
@Test
public void testCodingEmptySrcBuffer() throws Exception {
 final WritableByteChannelMock channel = new ritableByteChannelMock(64);
 final SessionOutputBuffer outbuf = new SessionOutputBufferImpl(1024, 128);
 final BasicHttpTransportMetrics metrics = new BasicHttpTransportMetrics();
 final IdentityEncoder encoder = new IdentityEncoder(channel, outbuf, metrics);
 encoder.write(CodecTestUtils.wrap("stuff"));
 final ByteBuffer empty = ByteBuffer.allocate(100);
 empty.flip();
 encoder.write(empty);
 encoder.write(null);
 encoder.complete();
 outbuf.flush(channel);
 final String s = channel.dump(StandardCharsets.US_ASCII);
 Assert.assertTrue(encoder.isCompleted());
 Assert.assertEquals("stuff", s);
}
```


Fonte: Pinto *et al.* (2020, p.3).

No vocabulário, é analisado o impacto da divisão de identificadores compostos (por letras minúsculas) como entrada para o texto classificação (PINTO *et al.*, 2020). Por exemplo, o identificador "*getStatus*" seria representado usando três maneiras: "*get*", "*status*" e "*getstatus*". Para representar um caso de teste no formato de *features* foi utilizado o número de linhas de código do caso de teste a quantidade de identificadores (*tokens*) e o valor único de cada identificador. Por exemplo, na Figura 3 é apresentado o resultado do processo de extração de identificadores de um teste. Além desses identificadores, é possível obter mais informações como o número de linhas do teste e também se ele é instável ou não. Com essas informações é possível criar um modelo de classificação de instabilidade utilizando as palavras que estão nos testes (PINTO *et al.*, 2020).

O resultado deste pré-processamento dos casos de testes instáveis e não instáveis é utilizado por técnicas de ML para construção de um vocabulário instável. Neste cenário de aprendizado de máquina os casos de testes são representados por *features*, são elas: número de linhas do teste, número de palavras Java, e uma *feature* para cada *token* distinto encontrado em casos de teste; As *features* referentes aos *tokens* resultam em um grande número de *features*. No

pré-processamento de identificadores nos casos de teste usa-se a seleção de atributos para remover características com pouco ganho de informação utilizando um limiar de $0{,}02$ (TREUDE; ROBILLARD, 2016; PINTO *et al.*, 2020).

Após o pré-processamento textual e a criação de *features* legíveis para os classificadores, avalia-se o desempenho de 5 classificadores de aprendizado utilizando o conjunto de dados pré-processados (PINTO *et al.*, 2020). Foram replicados métodos de classificação já utilizados em contexto de engenharia de software, tais como: Random Forest, Decision Tree, Naive Bayes, Support Vector Machines (SVM), K Nearest Neighbors (K-NN) (SOUZA; CAMPOS; MAIA, 2014; TREUDE; ROBILLARD, 2016).

Considerando que testes rotulados corretamente como *flaky* são "verdadeiros positivos" e os testes rotulados incorretamente são "falsos positivos" foi obtida a seguinte métrica de precisão: Em termos de matriz de confusão, a precisão pode ser representada pelo número de verdadeiros positivos, dividido pelo número de verdadeiros positivos somado ao número de falsos positivos.

O primeiro trabalho sobre vocabulário instável proprõe uma medida de precisão diferente de uma acurácia convencional pois desconsidera a predição de "não instáveis" removendo-se os "verdadeiros negativos" e "falsos negativos". O foco é dado na pontuação F1 (a média harmônica de precisão e *recall*) pois há um maior interesse em prever corretamente *flaky tests* ao invés de não *flakies*. Neste primeiro trabalho, para obter os dados necessários sobre *flaky tests* foram executados 100 vezes os 64 mil casos de teste de 24 projetos Java. Sinalizaram um teste como instável se houve discordância nos resultados do teste. Foram extraídos todos identificadores dos casos de teste usando procedimentos tradicionais de tokenização. Finalmente, os casos de teste instáveis e não instáveis pré-processados foram usados como entrada para cinco algoritmos de aprendizado de máquina Pinto *et al.* (2020).

O trabalho apresentado por Pinto *et al.* (2020) foram identificados alguns pontos interessantes: Primeiro, foram encontrados seis projetos com testes, e um total de 86 testes instáveis neles. Um total de 55% desses *flaky tests* falharam apenas uma vez, o que significa que o O limite de 100 execuções pode ter limitado a observação de testes instáveis (ou seja, é provável que possamos encontrar mais *flaky tests* com mais execuções). Em particular, Random Forest teve a melhor precisão (0,99), enquanto Support Vector Machine teve um desempenho ligeiramente superior Random Forest em termos de recordação (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 e/ou usando uma fila de eventos (PINTO *et al.*, 2020).

2.2.5 Replicação da abordagem de Vocabulário Instável

Experimentos precisam ser replicados em diferentes contextos, múltiplas vezes, e sob diferentes condições antes que eles possam produzir conhecimentos genéricos (CAMPBELL;

STANLEY, 1963). Neste sentido, a abordagem sobre vocabulário instável precisa ser replicada em novos contextos para apresentar resultados sólidos (CAMARA *et al.*, 2021b). Para o teste de projetos cruzado (modelo treinado em um conjunto de dados e testado em outro), o resultado não foi positivo. Por este motivo é necessário estudar o *recall* dos métodos de classificação em diferentes contextos e também sobre a generalidade de lista de palavras instáveis em cenários de teste (CAMARA *et al.*, 2021b).

É relevante avaliar a generalização dos resultados de predição apresentados em um vocabulário em diferentes linguagens e modelos de classificação. Desta forma, é possível visualizar quais palavras instáveis em diferentes contextos, linguagens e ambientes. O resultado do estudo de replicação para abordagem de vocabulário instável é a remoção de possíveis vícios presentes do vocabulário ocasionado pelo domínio da aplicação ou pelas técnicas de aprendizado de máquina (CAMARA *et al.*, 2021b). É proposto um estudo de replicação para lidar com as ameaças e possíveis vícios do estudo original, seguindo as questões de pesquisa (CAMARA *et al.*, 2021b):

1. Outros algoritmos de ML ou os mesmos algoritmos com implementações distintas resultam na mesma precisão?

    1.1 Com que precisão pode-se prever a *flaky tests* com base em identificadores de código-fonte nos casos de teste?

Resposta: Os classificadores tiveram um desempenho muito bom, semelhante ao estudo original. A diferença dos resultados devido a frameworks de aprendizado de máquina é pequena. Os classificadores adicionais obtiveram resultados semelhantes aos investigados anteriormente. LR apresentou o melhor valor de recall.

    1.2 Qual valor de diferentes *features* adicionadas ao classificador?

Resposta: Assim como no estudo original, características diferentes não mostraram muito impacto no classificadores, exceto quando tokens que não são identificadores não são considerados.

    1.3 Quais identificadores de código de teste são mais fortemente associados à fragilidade do teste?

Resposta: O vocabulário instável está relacionado com execução, coordenação de tarefas e persistência. O conjunto de palavras é muito semelhante ao do estudo original.

2. Os resultados são válidos para outros conjuntos de dados, incluindo projetos diferentes?

    2.1 Qual o sucesso de um classificador treinado e testado nos mesmos projetos?

Resposta: O desempenho do modelo de classificação para identificar testes instáveis dentro do mesmo projeto foi baixa. As funcionalidades com maior ganho de informação são distintas das identificadas anteriormente e estão relacionadas a operações de entrada e saída de dados, execução, coordenação de tarefas e palavras-chave Java.

2.2 Qual o sucesso de um classificador treinado em um projeto e testado em outro?

Resposta: Classificadores treinados com um conjunto de dados e testados com outros conjuntos não apresentam bons resultados. As características com maior ganho de informação estão relacionadas a chamadas assíncronas e são distintas daqueles previamente identificados.

Como resposta da pergunta 2.2 são apresentados resultados ruins para classificadores treinados em projetos diferentes dos projetos de teste. Ou seja, variar o contexto do treino, pode afetar o resultado do teste de predição (CAMARA *et al.*, 2021b). A criação de vocabulários é interessante pois economiza recursos relacionados a reexecução, porém, necessita definir os limites da generalização desta técnica através de estudos empíricos.

A ideia que originou a criação de um vocabulário de *flaky tests* foi a presença de padrões sintáticos presentes em dados sobre testes instáveis, como, por exemplo, a espera assíncrona como causa da instabilidade (PINTO *et al.*, 2020). Os padrões sintáticos podem estar relacionados com diversos fatores (e.g. espera assíncrona), e podem ser observados nos dados de testes de diversos estudos (BELL *et al.*, 2018; ECK *et al.*, 2019; LUO *et al.*, 2014; PALOMBA; ZAIDMAN, 2017; PINTO *et al.*, 2020).

Técnicas para identificar *flaky tests* geralmente utilizam de dados empíricos sobre instabilidades já ocorridas que podem se repetir no futuro. Estes dados são utilizados em métodos diversos para se treinar uma identificação automática.Quando o método estiver maduro o suficiente para identificar instabilidades em conjunto de dados aleatórios pode-se comprovar a eficiência deste método naquele cenário de projetos (BELL *et al.*, 2018; ALSHAMMARI *et al.*, 2021; ELOUSSI, 2015; PINTO *et al.*, 2020; CAMARA *et al.*, 2021b).

Para validar a abordagem de vocabulário instável realizou-se um estudo de replicação, acrescentando três novos modelos de classificação (CAMARA *et al.*, 2021b). Foram apresentados resultados positivos, indicando que a utilização de vocabulários é replicável e extensível. Porém o nível de generalização do vocabulário pode variar em diferentes contextos e aplicações, tornando o vocabulário muito dependente do contexto (CAMARA *et al.*, 2021b).

2.2.6 Discussões sobre a utilização de vocabulários

O uso de *tokens* extraídos de casos de teste é eficaz na identificação de *flaky tests*. No entanto, considerando casos de teste para os mesmos projetos (que devem compartilhar o voca-

bulário) e para projetos diferentes (que podem não sustentar tal suposição), o *recall* foi bastante baixo. Isso sugere um *overfitting* do modelo de classificação (CAMARA *et al.*, 2021b).

As palavras irrelevantes têm pouco impacto sobre o desempenho do classificador. No entanto, isso pode ser devido a uma lista de palavras ineficazes. Por exemplo, palavras de pouca relevância (*stopwords*) – como verbos auxiliares comuns (ser, é), preposições (de) e símbolos (}) – poderiam ser desconsiderados. Este fato revela a importância no processo de *parser* do código referente ao teste instável (CAMARA *et al.*, 2021b).

Identificar de maneira assertiva o nome de métodos, variáveis, funções, classes e detalhes da linguagem aumenta o valor das *features* utilizadas pelos classificadores. Neste sentido, o presente trabalho visa construir um FlakyLexer especializado para linguagem Javascript conforme descrito no Capítulo 3.

2.3 Conjuntos de dados empíricos sobre flaky tests

Estudos sobre *flaky tests* necessitam de relatos e ocorrências empíricas sobre instabilidade para que sejam projetadas soluções viáveis para o problema de instabilidade em testes em um cenário real. Com estes dados empíricos são projetadas abordagens para mitigar ou solucionar a instabilidade. Desta forma podemos validar as abordagens de identificação de instabilidades em um cenário de produção de software real. Existem estudos que utilizam conjuntos de dados empíricos sobre casos de testes implementados na linguagem Java para servir de base para abordagens de identificação de instabilidade, por exemplo, o vocabulário instável (BELL *et al.*, 2018; PINTO *et al.*, 2020; CAMARA *et al.*, 2021b).

Trabalhos anteriores apresentaram bons resultados para a identificação de *flaky tests* com o vocabulário instável com projetos Java (PINTO *et al.*, 2020; CAMARA *et al.*, 2021b). Os autores ressaltam a importância e a necessidade de mais estudos empíricos utilizando esta abordagem. Neste sentido, utilizaremos um conjunto de dados com casos de testes Javascript para realizar experimentos relacionados ao vocabulário instável.

Neste trabalho analisamos um conjunto de dados sobre *flaky tests* misto, que contém projetos que utilizam Java e Javascript (ROMANO *et al.*, 2021). Filtramos todos os projetos Javascript que continham em seu histórico commits com palavras relacionadas à instabilidade. Algumas informações relevantes sobre instabilidade neste conjunto de dados foram levantadas, por exemplo: a causa típica de testes *flaky* neste escopo de projetos são: assincronicidade, Ambiente de execução, Falhas em integração com API e implementação. Essas informações podem impactar diretamente nos resultados das abordagens de identificação de *flaky tests*. Por exemplo, utilizando a abordagem de vocabulário instável, podem conter um número de palavras maior que estão associadas a estes tópicos de instabilidade.

2.4 Considerações Finais

Neste capítulo foram apresentadas as principais causas de instabilidades em casos de testes presentes em projetos reais tais como: problemas com renderização de recursos, assincronicidade de métodos, ordem de testes em uma suíte, variação de ambiente e plataforma, dentre outros. Foram apresentadas as principais técnicas para identificação de *flaky tests* no estado atual da arte, tais como: múltiplas execuções de testes, identificação por análise diferencial de cobertura, e identificação por análise de texto e *tests smells*. Para realização do presente trabalho foi escolhida a técnica de identificação estática, mais especificamente a construção de vocabulários (PINTO *et al.*, 2020; CAMARA *et al.*, 2021b), para otimizar o gasto de recursos relacionados à reexecução.

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). Desta forma, pode-se utilizar um vocabulário geral removendo palavras específicas de projeto. Neste sentido, o presente projeto visa utilizar um vocabulário Javascript. Existe a suposição de que as palavras que se repetem em ambos vocabulários são mais generalizáveis.

3 ABORDAGEM

Este trabalho utiliza a abordagem de vocabulário instável para identificar *flaky tests* em casos de teste Javascript. Os trabalhos anteriores sobre o vocabulário instável utilizam conjuntos de dados com casos de testes em Java. Para validar esta abordagem em uma linguagem amplamente utilizada na Web, coletamos casos de testes implementados com Javascript obtidos a partir do conjunto de testes de Romano *et al.* (2021). Com este conjunto de dados podemos avaliar a predição de instabilidade com vocabulário em um cenário diferente do proposto pelo trabalho original. As questões de pesquisa tratadas são:

QP1

Com que precisão podemos prever casos de testes instáveis com base nos identificadores presentes no código-fonte de casos de teste em Javascript?

QP2

Quais identificadores presentes em códigos de testes em Javascript estão fortemente associados à instabilidade?

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; PIATETSKY-SHAPIRO; SMYTH, 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 dos casos de testes.

O restante da organização deste capítulo faz-se conforme a definição apresentada de mineração de dados. O conhecimento do domínio sobre *flaky test* e os dados utilizados são descritos na Seção 3.1. São apresentadas técnicas para o pré-processamento de teste instáveis na Seção 3.2 e de extração de padrões na Seção 3.3. Finalizamos o capítulo apresentando as ameaças à validade (Seção 3.4) e considerações finais (Seção 3.5). A avaliação dos modelos gerados é apresentada no Capítulo 4.

3.1 Conhecimento do domínio

A identificação de casos instáveis pode ser feita analisando commits de desenvolvedores que relataram *flaky tests* (LUO *et al.*, 2014). Por exemplo, em trabalhos anteriores foram identificadas instabilidades em softwares desenvolvidos pela Apache Software Foundation (ASF): eles identificaram com sucesso um conjunto de causas comuns explicando por que um teste é

instável. Por exemplo, uma de suas descobertas foi que a “espera assíncrona” é a fonte mais comum do termo *flakiness*, responsável por 45% dos casos analisados, e ocorre quando um teste não espera corretamente pelo resultado de uma chamada assíncrona (LUO *et al.*, 2014). Foi analisado o histórico completo de 1.129 *commits* do repositório central (ASF) que possuem a palavra-chave *flaky* ou *intermitent* e, em seguida, inspecionou-se manualmente todos eles. Como resultado foram propostas 10 categorias de causas raiz da instabilidade e as estratégias mais comuns para repará-los (LUO *et al.*, 2014).

Esta abordagem de análise manual de *commits* é útil para construir conjuntos de dados com informações e relatos sobre instabilidade. Com estes dados, são aplicadas abordagens para detectar automaticamente novas instabilidades utilizando informações existentes nos *flaky tests* conhecidos. Neste trabalho utilizaremos a abordagem de revisão e *commits* em projetos que relataram instabilidades no GitHub para construir um conjunto de dados de testes instáveis e não instáveis. Com esses dados aplicamos a abordagem de predição de instabilidade chamada vocabulário instável (PINTO *et al.*, 2020).

A primeira etapa do processo é localizar o código fonte da instabilidade. Neste trabalho isto foi feito utilizando as colunas de ”Código afetado“e ”URL“ presentes no conjunto de dados inicial (ROMANO *et al.*, 2021). Utilizando a biblioteca *@typescript-eslint*[1] foram extraídas as palavras dos testes para os modelos classificação. Com esta biblioteca é possível ler um trecho de código em Javascript e Typescript e transformá-los em tokens referentes às palavras do código com seu valor e tipo. Este processo é essencial para transformar casos de testes em texto apropriado para as técnicas de mineração de texto e obter um vocabulário instável. Cada token possui então seu valor e tipo. Os tipos de cada token contêm especificações sobre a categoria aquele dado, são elas:

1. Identifier : são identificadores de código que geralmente estão relacionados com nome de variáveis, funções e nome de classes;
2. Keyword : são as palavras reservadas da linguagem. Por exemplo ‘if’, ‘else’ e ‘this’;
3. RegularExpression : expressões regulares;
4. Template : strings que utilizam acento ao invés de aspas simples e que geralmente contém o valor de uma variável string em seu conteúdo;
5. JSXIdentifier : Identificadores de elementos utilizados em projetos de interface;
6. JSXText : identificadores de texto presentes em projetos de interface.

Portanto, o conjunto de dados proposto neste trabalho são *tokens* de casos de teste. Para obter este conjunto de dados utilizaremos um conjunto de dados inicial descrito na próxima subseção.

[1] https://github.com/typescript-eslint/typescript-eslint

3.1.1 Conjunto de dados inicial

Estudos sobre mineração de dados referentes a software, denominados de mineração de repositórios de software (MSR), geralmente recuperam os dados a partir de repositórios de projetos específicos, de portais associados a uma organização ou de plataformas sociais 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 do gênero (KALLIAMVAKOU *et al.*, 2014), ela foi escolhida como fonte de dados para este trabalho.

A quantidade significativa de recursos que são gastos com a reexecução é um grande problema para a identificação de *flaky tests* (MICCO, 2017). Para amenizar este problema, quando a instabilidade é identificada por algum desenvolvedor ela pode ser útil para identificar futuras falhas intermitentes. Portanto, este trabalho utiliza instabilidades já existentes (identificadas pelos desenvolvedores) para a criação de um vocabulário instável, evitando múltiplas execuções.

A construção de um vocabulário de teste instável utiliza dados de projetos reais, geralmente presentes no Github. Essa coleta não é trivial visto que é necessário um critério para seleção de projetos reais com certa relevância. Para facilitar a etapa de coleta de casos de testes instáveis utilizamos a seguinte coleta contendo: 235 instâncias de interações de desenvolvedores que indicam instabilidades em testes automáticos projetos de código aberto no Github (ROMANO *et al.*, 2021).

No conjunto de dados do estudo de Romano *et al.* (2021), apresenta-se uma abordagem de coleta de instabilidades em testes que identifica a palavra *flaky* e *flakiness* em alterações de código (*commit*), solicitações de alteração (*pull request*) e relatos de problemas (*issues*) em projetos abertos Java e Javascript. O resultado desta coleta foi um total de 235 instâncias de instabilidades em testes automáticos de projetos que utilizam Javascript e Java. Cada instância possui um conjunto de atributos textuais que ajudam a identificar a natureza da instabilidade utilizando as informações fornecidas pelo desenvolvedor que relatou a instabilidade e o seu contexto. Existem instâncias do conjunto de dados que contêm o link para os *commits* nos quais o autor utiliza termos relacionados a instabilidade em testes. Os atributos de cada instância do conjunto de dados são as seguintes:

- *Title*: É a mensagem do desenvolvedor que identificou instabilidades em testes que contém citações sobre instabilidades de testes presente em *commits* em softwares livres. Exemplos: “[test] Fix flaky popper.js test”, “[core] Ignore a few flaky visual tests”.
- *URL*: É o link para acessar o *commit*, contendo uma visualização de diferença entre o arquivo antes e depois do *commit* em projetos do GitHub.
- *Root Cause of Flakiness*: De acordo com a mensagem utilizada pelo autor do *commit*, esta coluna indica o motivo que causou a instabilidade. Por exemplo: “Image load may fail", “Improper waiting for elements", “ Waiting time varies”.

- *Root Cause Category*: A coluna anterior (Root Cause of Flakiness) identifica a causa raiz da instabilidade. Essa informação se repete entre diferentes linhas do conjunto de dados. Neste dataset, as causas são: Renderização de recursos, problema de plataforma, problema de tempo de animação, interação incorreta do executor de teste, ordem incorreta de carregamento de recurso, carregamento de recurso de rede, problema de seletor de dom, dependência de ordem de teste, diferença de layout, coleções não ordenadas, aleatoriedade e tempo.

- *Tests/Code Affected (URL)*: De acordo com o *commit* presente na coluna URL, esta coluna indica qual arquivo do projeto foi alterado naquele *commit*.

- *Testing Environment/Runner*: Indica em qual ambiente o teste está sendo executado. São exemplos: software de execução de pipeline (e.g., CircleCI) e framework para execução de testes (e.g., Jest).

- *Manifestation Category*: Indica como foi verificada ou diagnosticada a instabilidade do teste. São exemplos: Especificando uma plataforma com problemas, forçando condições de teste, alterando o intervalo de tempo para execução, executando o teste individualmente (fora do contexto da suíte de testes), executando o teste múltiplas vezes.

- *How Fixed in Code*: Esta coluna indica qual foi o tratamento utilizado pelo desenvolvedor para contornar ou resolver a instabilidade. A informação presente nesta coluna está associada com o commit do desenvolvedor.

- *Fix Category*: De acordo com o dado presente na coluna anterior, foram identificados alguns padrões que se repetem quando ao tratamento para correção de instabilidades. São eles: correção de assincronicidade, remoção do teste, refatoração da lógica de implementação, alteração na conexão com API, alteração da versão de bibliotecas.

- *Link to Pull Request*: Esta coluna é opcional e contém o link do *pull request* da correção para o teste instável (se existir).

No contexto deste projeto, duas limitações importantes são observáveis. Uma se refere quanto à linguagem utilizada em cada projeto. O conjunto de dados não diferencia os projetos escritos respectivamente em Javascript ou Java. Para esse problema, felizmente, é possível detectar facilmente a linguagem empregada nos casos de teste instáveis, observando-se a extensão do arquivo envolvido.

Porém, uma segunda limitação é a não identificação do trecho de código de teste instável. Apenas um relato do autor do *commit* indica que naquela contribuição alguma instabilidade de teste foi corrigida. Como um *commit* pode alterar vários trechos de um arquivo ou vários arquivos, não é possível identificar automaticamente, de modo geral, qual foi o caso de teste afetado pela contribuição. Portanto, o caso de teste instável não é identificado no conjunto de

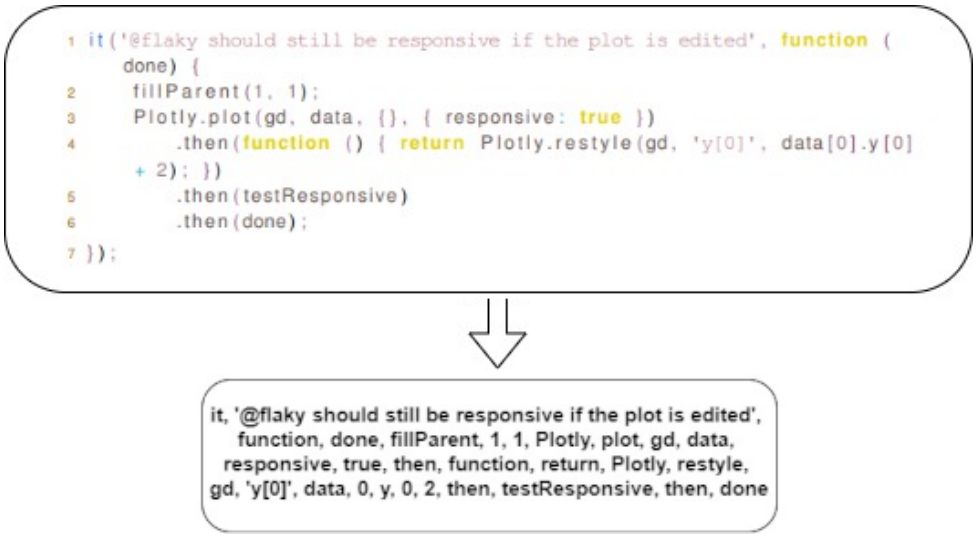
dados inicial, o que dificulta a extração dos termos necessários para construção do vocabulário, objetivo deste trabalho.

Assim, a identificação de *flaky tests* neste conjunto de dados não é trivial, necessitando de revisão manual dos códigos afetados e o estudo sobre a suíte de testes 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. Identificando o cenário da instabilidade, processos de análise manual do código e reexecução podem auxiliar a rotular testes como instáveis. Neste trabalho foram desenvolvidas ferramentas para automatizar a coleta de informações sobre casos de testes, apresentados na próxima seção, tais como: filtragem de projetos, coleta de trechos de código a partir de commits e processamento léxico do código-fonte.

3.1.2 Conjunto de dados proposto

O conjunto de dados inicial é útil pois revela alterações no código fonte do projeto (geralmente na implementação de testes) que indicam instabilidades (ROMANO *et al.*, 2021). A construção de um vocabulário neste trabalho necessita do conteúdo do teste instável no formato de texto, conforme resumido na Figura 4: a partir de um caso de teste se deseja obter uma lista de palavras.

Figura 4 – Exemplo de tokens presentes em um caso de teste.

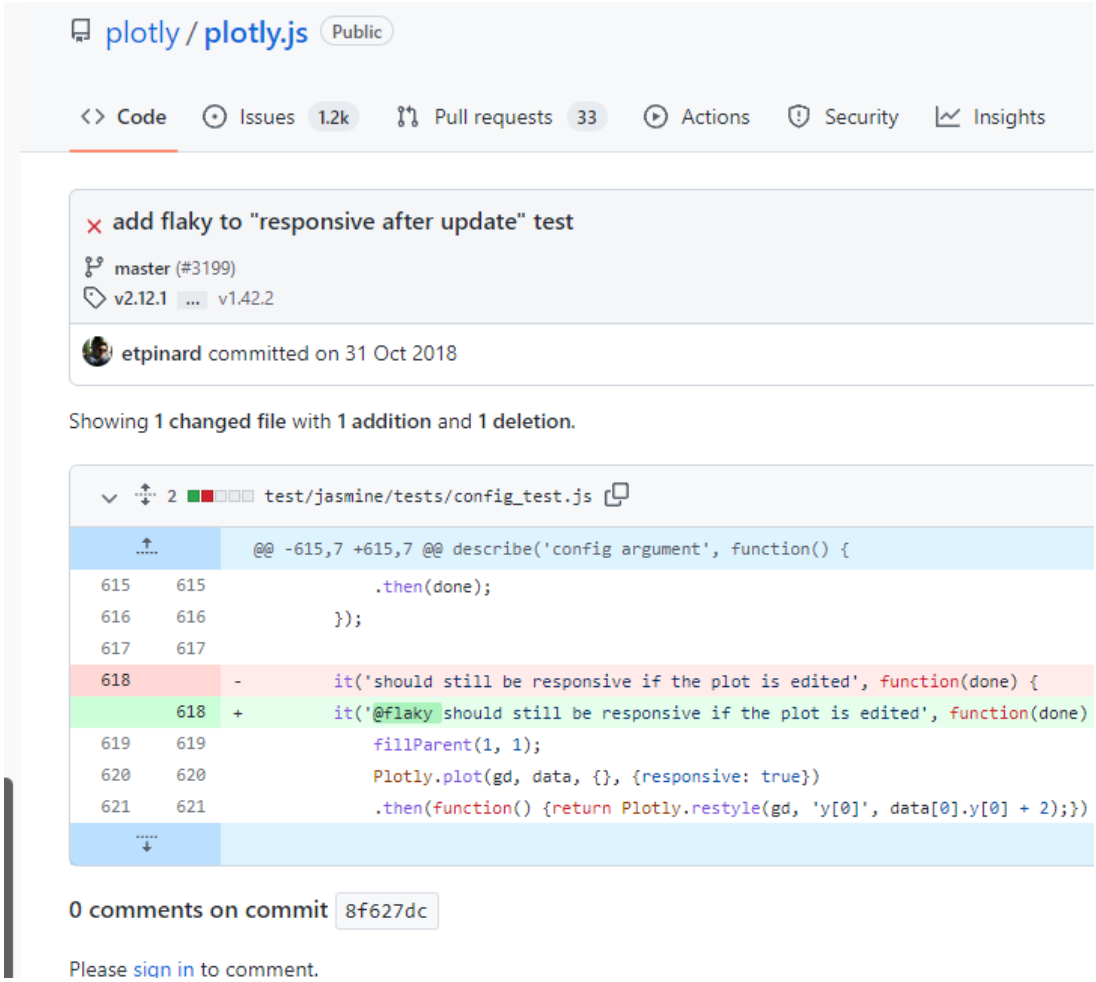


Fonte: Autoria Própria (2022).

Porém as alterações de código descritas no conjunto de dados inicial podem não evidenciar um caso de teste instável diretamente. Assim, esta seção propõe uma análise de commits presentes no conjunto de dados inicial, com a finalidade de identificar o início e o fim de cada caso de teste instável.

Um exemplo de como um *flaky test* pode ser identificado a partir de um arquivo afetado[2] em um commit[3] é ilustrado na Figura 5. Nela estão presentes as informações sobre instabilidades coletadas do Github: o comentário do autor do *commit* que identifica a instabilidade, uma mensagem associada ao *commit*, o arquivo do caso de teste e o que foi alterado (linha removida e linha adicionada). Neste caso, foi inserido um marcador para identificar a instabilidade e provavelmente ignorar o teste ou realizar algum tratamento especial durante a execução de uma bateria de testes.

Figura 5 – Exemplo de identificação de instabilidade à partir de um *commit*.



Fonte: Autoria Própria (2022).

Porém, diferentemente do caso apresentado na Figura 5, existem casos onde a identificação de um caso de teste instável não é trivial, necessitando de uma revisão do código afetado ou da suíte de testes. Para garantir o bom funcionamento do processo de obtenção de um vocabulário instável, a identificação de testes instáveis é parte essencial do processo, pois garante a assertividade dos modelos de classificação sobre palavras instáveis. Quando alterações de desenvolvedores são aplicadas diretamente ou indiretamente em casos de testes, é possível iden-

[2] https://github.com/plotly/plotly.js/blob/8f627dcacc40a031a7da1516c4e438dc5f795618/test/jasmine/tests/config_test.js

[3] https://github.com/plotly/plotly.js/commit/8f627dcacc40a031a7da1516c4e438dc5f795618

tificar o código fonte do teste instável utilizando a revisão de *commits* e dos arquivos afetados. Nesses casos, procuramos identificadores que comumente estão relacionados a instabilidades (e.g., ‘retries’, ‘timeout’, ‘sleepFor’, ‘tick’, ’skip’).

Para obter um conjunto de testes instáveis este trabalho analisou todos os 235 dados apresentados anteriormente no conjunto de dados inicial e selecionou as instâncias contendo um arquivo do código fonte que possui pelo menos um *flaky test* em Javascript ou Typescript. Para identificar o início e fim de cada caso de teste instável dentro de um arquivo, os dados foram construídos no formato ilustrado na Tabela 1:

Tabela 1 – Instância de arquivo que contém flaky test

Commit	Tests/Code Affected	is_test_code	test1_start_line	test1_end_line,
commit_id	project_file	true	175	20

Fonte: Autoria Própria (2022).

A Tabela 1 foi preenchida manualmente, após a análise de cada um dos dados. Em seu conteúdo, foram apresentados até 10 flakies em cada arquivo descrito em ‘*project_file*’, incrementando as colunas de início de fim do caso de teste de 1 a 10, contabilizando um total de 144 flaky tests identificados.

Para a realização desta atividade, foi desenvolvida uma ferramenta para coletar dados sobre *flakiness* em projetos de código aberto no GitHub que utilizam Javascript e TypeScript. Foram desenvolvidas funções para baixar repositórios de código livre na versão na qual a instabilidade foi identificada, localizar os testes do projeto, e realizar o processamento de texto referente ao código-fonte de testes automáticos implementados com Javascript.

Conhecendo o arquivo, a linha de início e fim de um caso de teste podemos obter seu código-fonte. Os casos de teste foram transformados em uma lista de identificadores (tokens) e armazenados em um grande JSON no formato presente na Listagem 5. Cada objeto deste JSON representa uma instabilidade de teste. Os tokens presentes nestes objetos representam o caso de teste no formato de texto. Esses tokens são utilizados para se obter características dos testes. Por exemplo, os casos de testes contém informações textuais que podem ser agrupadas em categorias, ou tipos de texto, tais como: Identifier, Keyword, RegularExpression, Template, JSXIdentifier, JSXText, Boolean, String, Numeric, Undefined, Null, Object. É importante ressaltar que a grande quantidade de dados do tipo ‘identificador’ está relacionada com informações que o desenvolvedor utilizou para implementar o teste, como por exemplo: nome de variáveis, funções e métodos. Coletando essa informação textual é possível visualizar quais palavras se repetem em *flaky tests*.

Com essas três informações (id do *commit*, arquivo contendo o *flaky test*, e identificadores de início e fim do caso de teste), é realizado o *checkout* e a coleta dos testes instáveis. Para cada teste instável, coleta-se seu conteúdo no formato de tokens. Por exemplo, o *flaky test* apresentado na Listagem 6 pode ser representado por tokens conforme apresenta a Listagem 5 em formato de JSON. Em cada um destes objetos estão presentes informações que identificam

Listagem 5 – Resultado do processamento léxico de um caso de teste

```
{
    "URL": "https://github.com/plotly/plotly.js/blob/8
    f627dcacc40a031a7da1516c4e438dc5f795618/test/jasmine/tests/
    config_test.js",
    "is_flaky": True,
    "commit": "8f627dcacc40a031a7da1516c4e438dc5f795618",
    "project_name": "plotly.js",
    "project_author": "plotly",
    "file": "/test/jasmine/tests/config_test.js",
    "start_line": 618,
    "end_line": 624,
    "test_code": "it('@flaky should still be responsive if the
    plot is edited', function (done) {\n    fillParent(1, 1);\n
    Plotly.plot(gd, data, {}, { responsive: true })\n        .then(
    function () { return Plotly.restyle(gd, 'y[0]', data[0].y[0] +
    2); })\n        .then(testResponsive)\n        .then(done);\n})
    ;\n",
    "tokens": [{
        "value": "it",
        "type": "Identifier"
      },{
        "value": "'@flaky should still be responsive if the plot
    is edited'",
        "type": "String"
      },{
        "value": "function",
        "type": "Keyword"
      }, ...
    ]
  },
```

casos de instabilidades, tais como: a URL referente ao arquivo de teste; a linha de início e fim do caso de teste neste arquivo; o commit no qual a instabilidade foi encontrada; o nome do projeto e do autor do projeto; o conteúdo do código-fonte referente ao teste; e a lista de tokens referente ao conteúdo do teste no formato de texto; uma variável booleana indicando se o teste é instável ou não (‘is_flaky’).

Listagem 6 – Exemplo de flaky test relacionado à visualização dos elementos

```
it('@flaky should still be responsive if the plot is edited',
    function (done) {
    fillParent(1, 1);
    Plotly.plot(gd, data, {}, { responsive: true })
        .then(function () { return Plotly.restyle(gd, 'y[0]', data
    [0].y[0] + 2); })
        .then(testResponsive)
        .then(done);
});
```

A linha de início e fim de cada *flaky test* foi identificada neste trabalho utilizando a diferença de código adicionado ou removido no commit. A partir do trecho de código identificado é utilizada a biblioteca @*typescript-eslint* para obter os tokens do caso de teste. Quando o atributo 'is_flaky' tem o valor 'True', garante que aquele teste foi indicado como flaky por desenvolvedores. Quando tem o valor 'False', significa que ele ainda não foi relatado como instável por alguém até o momento. Portanto, esta variável booleana não garante que aquele teste é instável de fato, mas que algum desenvolvedor indicou que aquele teste teve uma execução instável em alguma execução.

Tabela 2 – Projetos e número de casos de testes analisados

Github ID	Name	Flaky Tests
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 flakies**

Fonte: Autoria Própria (2022).

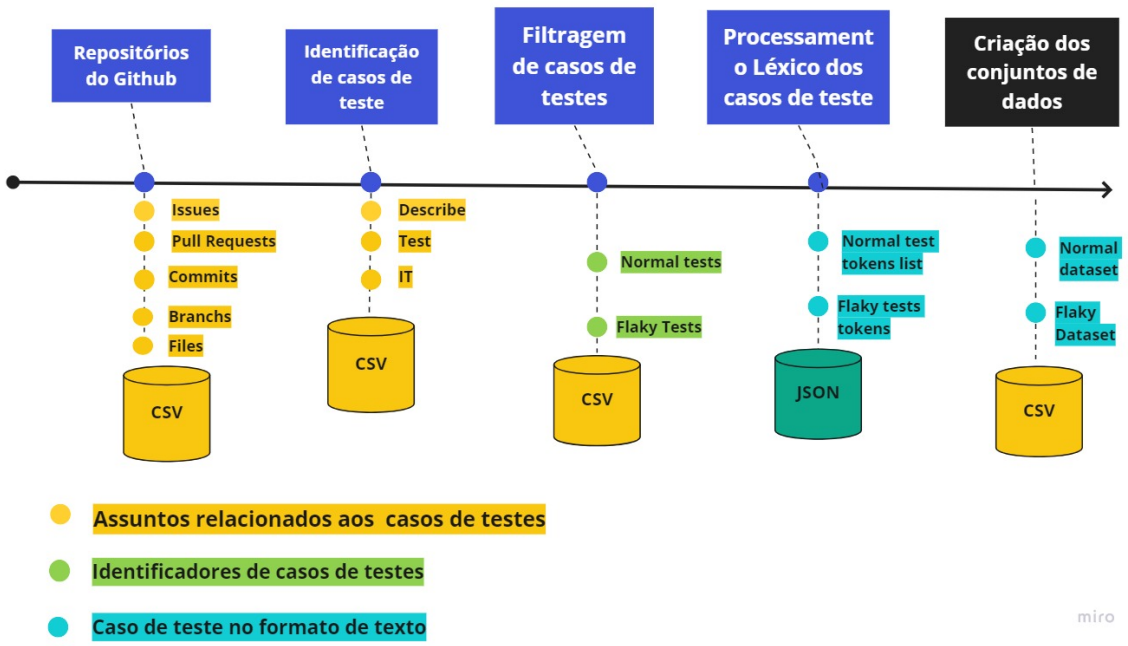
Revisamos os *commits* do conjunto de dados inicial (ROMANO *et al.*, 2021) e identificamos 144 casos de testes instáveis, conforme relacionado na Listagem 2. Esses elementos foram

salvos no formato de objetos apresentados na Listagem 5. Cada objeto representa um *flaky test* e possui uma lista de tokens para representar seu código-fonte associado. Desta maneira, fica evidente quais são as palavras referentes aos casos de testes instáveis ou não. Utilizamos esses dados para construção de classificadores de instabilidades.

3.2 Coleta de dados e Pré-processamento Léxico

Obter casos de testes do Github e extrair suas palavras é útil para diversas questões de pesquisas relacionadas à testes. Neste trabalho, estes dados servem de entrada para um processamento que resulta em um vocabulário instável. Para realizar a coleta e processamento léxico, foi desenvolvido um processo para obtenção de casos de testes do Github em formato de tokens, conforme ilustrado na Figura 6.

Figura 6 – Fluxo para coleta de tokens de casos de testes do Github.



Fonte: Autoria Própria (2022).

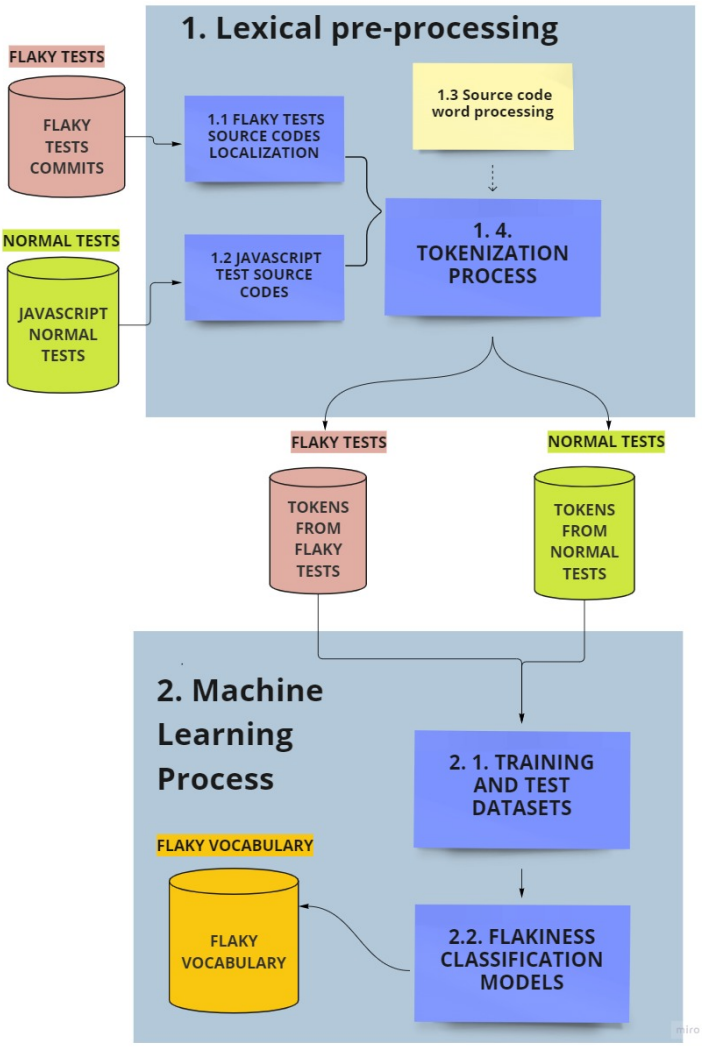
Foram desenvolvidas ferramentas para coleta de dados sobre testes automáticos no Github com intuito de criar uma base para os modelos de classificação. Para coletar casos de testes de um repositório basta informar o autor e o nome do projeto que o algoritmo proposto coleta essas informações no formato de texto. Definindo uma lista de repositórios no arquivo repositories.txt dentro da pasta config separados por quebra de linha, coleta-se todos os casos de testes daquele repositório. O objetivo é coletar dois arquivos no formato JSON contendo tokens de casos de testes rotulados como instáveis e normais. Foi desenvolvida a classe GithubTestCollector para obter esses tokens 'normais' e 'instáveis' a partir de dados que identificam repositórios e trechos de códigos instáveis.

Para entender um pouco sobre do processo de coleta de dados do Github apresentamos a estrutura de arquivos desenvolvida:

- dataset.csv
- identified-flaky.csv
- non-identified.csv
- configs
  - availables-test-folders.txt
  - availables-test-names.txt
  - repositories.txt
- datasets
  - dataframes/flakies/1.csv
  - dataframes/normal/1.csv
- parsed-tests
  - flaky-parsed.json
  - normal-tests.json

Utilizamos um arquivo inicial chamado ‘dataset.csv’ contendo uma lista de instâncias contendo relatos de instabilidades no Github, servindo neste trabalho como um conjunto de dados inicial para identificar instabilidades. Neste arquivo estão presentes *commits*, *pull requests* e *issues* que contêm os identificadores de instabilidade em seu conteúdo. A partir deste arquivo selecionamos os *commits* que identificavam *flaky tests* em projetos Javascript e salvamos no arquivo ‘identified-flaky.csv’, e os não identificados no arquivo ‘non-identified.csv’. Desenvolvemos uma função para obter os repositórios destes dados e salvar no arquivo ‘repositories.txt’. A partir do arquivo ‘identified-flaky.csv’, fizemos o clone de cada repositório, o *checkout* nos *commits* relacionados à instabilidade, e a coleta do conteúdo dos testes. Realizamos o processo de tokenização do conteúdo dos testes e salvamos no arquivo ‘flaky-parsed.json’ e ‘normal-tests.json’. Convertendo os dados dos arquivos do formato JSON para o formato CSV, geramos os arquivos apresentados no diretório ‘datasets’. Como existem muito mais casos de teste que não são instáveis do que instáveis, para obter o arquivo ‘norma-tests.json’ é possível amostrar uma porcentagem fixa de casos de teste presentes no conjunto de dados. A partir desses arquivos CSV, podemos utilizar a linguagem Python para aplicação de modelos de classificação destas duas classes de testes: instável e não instável.

Figura 7 – Processo para construção de um vocabulário instável



Fonte: Autoria própria (2021).

3.3 Extração de padrões com Vocabulário instável em Javascript

A criação de um vocabulário instável em Javascript é o resultado de um processo de mineração de dados presentes nos testes automáticos de software. Desta maneira, é possível descobrir quais palavras estão presentes em instabilidades de testes. Para realizar este processo são propostas duas etapas: a representação dos casos de teste, considerando a frequência dos tokens do conjunto de dados; e a aplicação dos modelos de classificação para validar a criação de um vocabulário instável. Estas etapadas estão organizados conforme esquematizado na Figura 7, indicando a seguinte ordem de tarefas:

1. Representação dos casos de teste, considerando os tokens, em um modelo *bag-of-words*;
2. Criação de modelos de classificação;
3. Construção de um vocabulário *flakiness* a partir dos resultados do classificador;

4. Avaliação de modelos de classificação e vocabulário *flakiness* em projetos implementados em Javascript;

5. Comparação de vocabulário e modelos de classificação com trabalhos relacionados.

3.3.1 Extração de características

São extraídas as seguintes 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 testes instáveis, utilizando modelos de aprendizado de máquina. Existem palavras que são comuns em ambos testes estáveis e instáveis. Para resolver isto, propomos a construção do vocabulário instável utilizando palavras com alto ganho de informação para identificar instabilidades. Podem existir palavras específicas de contexto que são muito boas para identificar instabilidade em um cenário específico. Da mesma forma, existem palavras que podem existir em diferentes contextos. 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 testes instáveis mais extensos tenham mais palavras consideradas instáveis.

O valor e o tipo de um token são utilizados para identificar cada palavra nas colunas do conjunto de dados: as linhas são as instâncias de testes, e o conteúdo do dataset é a quantidade de vezes que aquela palavra se repetiu no teste. Desta forma, criamos uma tabela no seguinte formato: as linhas são os *flaky tests* e as colunas são as palavras instáveis no formato de tokens. O conteúdo da tabela refere-se à quantidade de vezes que os tokens de cada coluna se repetiram na linha referente ao *flaky test*. Este conjunto de dados é apresentado na Tabela 3.

Tabela 3 – Tokens presentes em flaky tests

Flaky	1º token do 1º teste	1º token do 2º teste	...	token N - teste N
Flaky_1_tokens	3	6	...	1
Flaky_2_tokens	0	0	...	0
Flaky_3_tokens	3	9	...	0

Fonte: Autoria Própria (2022).

Seguindo a lógica apresentada na Tabela 3, é possível visualizar que a construção de tokens tem dimensão proporcional à variedade do conteúdo apresentado nos flaky tests e também ao número de flakies. Em muitos casos os tokens são diferentes em conteúdo no sentido literal, porém são idênticos no tipo e semelhantes no conteúdo. Por exemplo, um token do tipo *string* que possui em seu conteúdo o texto ‘red’ pode ser classificado como uma cor, da mesma forma que outro token com o texto ‘dark red’ também, porém no sentido literal eles são diferentes.

Outro exemplo é a utilização de textos referentes a variáveis booleanas, seu conteúdo (falso ou verdadeiro) pode ser diferente entre tokens, porém indicamos que um agrupamento pode ser realizado. Outro exemplo são strings referentes a requisições de API: elas podem ser diferentes entre testes, porém possuem significado semelhante. Assim, verificamos se este agrupamento dos dados de acordo com seu tipo e valor podem ser relevantes para melhorar os resultados obtidos nos modelos de avaliação. Portanto, foram adotadas 3 propostas para criação das colunas do dataset:

1. Somente o valor do token: o texto puro do caso de teste.
2. O valor do token concatenado com seu tipo: texto puro concatenado com seu tipo (e.g. string, numeric, etc).
3. Somente o valor do token agrupado de acordo com seu significado (e.g. cores, urls de apis, variáveis booleanas).

Estes três conjuntos de dados possuem colunas diferentes porém as linhas permanecem sendo os casos de teste. O conteúdo do conjunto de dados refere-se a quantidade de vezes que um token se repete em um teste. Desta maneira é possível aplicar modelos de avaliação nos três conjuntos de dados e comparar os resultados. Assumindo que os conjuntos de dados de casos de testes normais e instáveis estejam em um destes três formatos, é possível construir conjuntos de dados de treino e teste que serão utilizados pelos classificadores.

Devido ao tamanho do conjunto de dados obtido (de *tokens* referentes aos casos de teste), esse trabalho utilizou somente o primeiro tipo de *dataset*, analisando somente o valor do *token*. Em trabalhos futuros será possível analisar o resultado de se considerar agrupamentos de *tokens*. Este agrupamento por significado pode ser realizado de diferentes formas. Por exemplo uma cor pode ser escrita de diferentes maneiras em casos de testes: RGB, hexadecimal, por extenso em português e inglês. Portanto, ao detectar tokens referentes à cores, os valores desses tokens são substituídos pela palavra ‘cor’.

3.3.2 Modelos de classificação

Serão replicados os classificadores comuns no contexto de engenharia de software, tais como: Random Forest, Decision Tree, Naive Bayes, SVM, K-NN, regressão logística, perceptron, Linear Discriminant Analysis (LDA) (SOUZA; CAMPOS; MAIA, 2014; TREUDE; ROBILLARD, 2016).

A implementação 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 parecido com os trabalhos anteriores, separamos o conjunto de dados contendo testes normais e instáveis em 20% para teste e 80% para treinamento. Para ter uma estimativa melhor do desempenho do sistema, seria interessante usar um esquema de validação cruzada em k partições. Isto faz com que todos os dados do dataset possam ser usados para avaliar o sistema. Portanto, a validação cruzada é uma sugestão para trabalhos futuros.

Após a classificação foram obtidas as seguintes métricas:

- Precisão: número total de verdadeiros positivos dividido pelo número total de instâncias classificadas como positivas.;
- Recall: o número de testes flaky classificados corretamente dividido pelo número total de testes flaky reais no conjunto de teste;
- F1-score: a média harmônica de precisão e recall.
- MCC (coeficiente de correlação de Matthews): mede a correlação entre as classes previstas (ou seja, flaky vs. não flaky);
- AUC: mede a área sob a curva que visualiza o trade-off entre a taxa de verdadeiros positivos e taxa de falsos positivos;

Concentramos nossas discussões no F1-score, pois estamos mais interessados em prever corretamente a instabilidade em vez da 'não instabilidade' (PINTO *et al.*, 2020).

3.4 Ameaças à validade

Para os projetos analisados, foi considerado que os casos de teste foram implementados na mesma linguagem utilizada para a parte da aplicação sob teste. De forma geral, isso se mantém quando consideram-se testes de unidade e de integração, que são aqueles geralmente encontrados em projetos de software. Entretanto, seria possível ter casos de testes escritos em outras linguagens, distintas daquela utilizada para a parte sob teste da aplicação. Por exemplo, em testes de sistema seria possível utilizar scripts com comandos do interpretador do sistema (Bash) ou de configuração de containers de software (Docker). Embora esses tipos de teste não sejam contemplados por esse estudo, cabe destacar que, para os projetos analisados, não foram encontrados testes com tais características.

Uma ameaça à validade interna é a representatividade dos dados dentro do domínio. Por exemplo, para se obter dados de aplicações típicas Javascript é necessário selecionar projetos relevantes. Para solucionar essa ameaça foram selecionados projetos considerados relevantes pelo número de estrelas. Utilizamos projetos com alta relevância no Github, por exemplo: o 'influxdata/influxdb.git' que possui 23 mil estrelas, o 'plotly/plotly.js.git' que possui 14 mil

estrelas, ‘vercel/next.js.git’ que possui 87 mil estrelas, entre outros. Num total de 32 projetos selecionados obtivemos um conjunto de dados com somente 144 instâncias sobre casos de testes instáveis que foram revisados manualmente à partir de *commits*. O número baixo de dados sobre casos de testes instáveis pode ser uma ameaça à validade interna. Além disso, é possível que alguns dos casos de teste que consideramos como não instáveis sejam, na verdade, instáveis. Estes problemas estão relacionados com o custo alto para identificação de *flaky tests* em abordagens convencionais. Sugerimos a aplicação de abordagens de reexecução e um conjunto de projetos maior para construir uma base de dados mais robusta para mitigar esta ameaça.

Uma ameaça a validade externa é a limitação do escopo do conjunto de dados utilizado: o modelo pode se limitar para o domínio Javascript para interfaces web. Dessa forma, devem ser considerados projetos de diversos domínios e características para constituir o conjunto de dados para criação e avaliação dos modelos de classificação.

O desbalanceamento da base de dados construída neste trabalho em relação as classes *flaky* e não *flaky* é uma ameaça a validade externa que afeta os resultados dos modelos de classificação. Este problema está relacionado com a quantidade de tempo e recurso necessário para identificação de *flaky tests* em Javascript. Portanto, em trabalhos futuros podemos balancear a base de dados, inserindo instâncias da classe *flaky* para validar os resultados. Utilizar a validação cruzada com k-partições poderia melhorar os resultados porém optamos por ser fiel aos trabalhos anteriores, este ponto também pode ser alterado em trabalhos futuros.

3.5 Considerações finais

Neste capítulo está presente a abordagem de vocabulário instável escolhida para facilitar a identificação de *flaky tests* utilizando palavras.

Avaliamos este processo de mineração de casos de teste propondo duas questões de pesquisa: Com que precisão podemos prever casos de testes instáveis com base nos identificadores presentes no código-fonte de casos de teste em Javascript? E quais identificadores presentes em códigos de testes estão fortemente associados à instabilidade? Para responder essas questões, construímos um conjunto de dados sobre casos de testes instáveis e aplicamos modelos de classificação.

No desenvolvimento do capítulo é apresentado um conjunto de dados inicial sobre instabilidades que serve de base para a criação do nosso conjunto de dados proposto. Construímos um conjunto de dados contendo as palavras de cada caso de teste e um identificador de instabilidade. No processo de construção do conjunto de dados estão presentes as seguintes etapas: identificação e clone de repositórios, identificação do código-fonte dos casos de teste, pre-processamento léxico do código-fonte e construção de ‘datasets’ no formato necessário para os modelos de classificação. Os resultados obtidos dos experimentos de classificação estão no próximo capítulo.

Por fim apresentamos as ameaças a validade interna, externa e de construção do trabalho.

Todo o processo de coleta de casos de testes do Github e sua tokenização foi automatizado e está disponível no repositório jsflaky-dictionary[4].

[4] https://github.com/sorattorafa/jsflaky-dictionary

4 RESULTADOS DA ABORDAGEM

Este capítulo apresenta os experimentos realizados com os modelos de classificação de dados descritos no capítulo anterior para responder às questões de pesquisa QP1 e QP2:

- QP1: Com que precisão podemos prever instabilidades em casos de teste escritos em Javascript?
- QP2: Quais identificadores presentes em códigos de testes estão fortemente associados à instabilidade?

4.1 Resposta para questão de pesquisa QP1: com que precisão podemos prever instabilidades em casos de teste escritos em Javascript

O primeiro resultado deste trabalho busca responder a questão de pesquisa: Com que precisão podemos prever instabilidades de teste com base nos identificadores presentes no código-fonte de casos de teste escritos em Javascript?

A maioria dos classificadores tiveram um desempenho muito bom em nosso conjunto de dados. No geral, o algoritmo de regressão logística teve o melhor resultado, o K-NN foi o segundo melhor. O classificador com o pior desempenho foi o Naive Bayes, que errou 243 vezes ao rotular 238 'não instáveis' como instáveis e 5 instáveis como 'não instáveis' em um conjunto de dados com 762 instâncias de testes. Este classificador também obteve o pior desempenho nos trabalhos anteriores (PINTO *et al.*, 2020; CAMARA *et al.*, 2021b). O classificador que teve melhor desempenho nos trabalhos anteriores foi o *Random Forest*, diferentemente do nosso (regressão logística).

As métricas dos modelos de classificação foram obtidas a partir da matriz de confusão de cada modelo. Nesta matriz estão presentes o número de predições corretas e incorretas de cada classificador. Com base nestes dados é possível analisar os casos de fracasso para entender melhor os motivos dos erros e também verificar os melhores resultados nos casos de sucesso. As duas classes presentes neste problema são os casos instáveis e não instáveis. No conjunto de dados obtido por este trabalho foram identificadas 144 instâncias *flaky tests* e 3.661 casos de testes normais coletados dos repositórios listados na Tabela 2.

Sobre os resultados observados, usamos o mesmo conjunto de algoritmos de aprendizado de máquina para a classificação de casos de teste apresentado em estudos anteriores, como exemplo do estudo apresentado por Souza, Campos e Maia (2014). A Tabela 4 apresenta o desempenho de oito algoritmos de aprendizado de máquina em nosso conjunto de dados em termos de métricas que são comumente usadas na literatura: precisão, recall, F1-score, Matthews Correlation Coefficient (MCC) e Area Under The Curve (AUC). Números em negrito destacam o algoritmo que teve o melhor desempenho para uma determinada métrica.

Nas primeiras colunas da Tabela 4 são apresentados os resultados médios dos algoritmos de classificação. Na última coluna temos o F1-score da classe positiva (true) que é um bom indicativo para avaliar o desempenho durante a classificação de *flaky tests*. As matrizes de confusão apresentadas nesta subseção são referentes aos resultados médios dos classificadores.

Tabela 4 – Resultados dos modelos de classificação

Algoritmo	**Precisão**	**Recall**	**F1 Score (weighted avg)**	**MCC**	**AUC**	F1 Score (True)
Random Forest	0.97	0.97	0.96	0.63	**0.98**	0.57
Decision Tree	0.97	0.97	0.76	**0.82**	0.85	0.76
Naive Bayes	0.94	0.68	0.77	0.24	0.77	0.20
SVM	0.96	0.95	0.93	0.23	0.79	0.10
K-NN	0.98	0.98	0.98	0.82	0.84	0.80
Logistic Regression	**0.98**	**0.98**	**0.98**	0.82	0.95	**0.81**
Perceptron	0.96	0.96	0.95	0.56	0.9	0.48
Lda	0.95	0.95	0.95	0.58	0.76	0.60

Fonte: Autoria Própria (2022).

Nosso foco é dado na pontuação F1 (média harmônica de precisão e *recall*) pois há um maior interesse em prever corretamente *flaky tests* ao invés de não *flakies* (PINTO *et al.*, 2020). Na Tabela 4 é possível visualizar que os resultados de *recall* dos algoritmos de classificação variam entre 0.68 até 0.98. Já o F1-score, considerando a média ponderada, apresenta resultados em um intervalo de 0.76 até 0.98, e para classe positiva de 0.10 à 0.81. A coluna MCC representa uma predição perfeita quanto mais próxima do número 1. Apenas os algoritmos Naive Bayes e SVM apresentaram um valor abaixo do mínimo esperado (0.5).

Cada modelo de classificação apresentou uma matriz de confusão representada em um gráfico em que o eixo y são os valores reais sobre instabilidade e o eixo x são os valores das predições. Em todos gráficos a seguir são apresentadas 761 instâncias de testes.

Na Figura 8 está presente o resultado da classificação utilizando o algoritmo Decision Tree: 26 casos de testes rotulados corretamente como instáveis e 719 rotulados corretamente como não instáveis. Neste gráfico vemos um total de 16 erros: 11 *flaky tests* rotulados como não instáveis e 5 casos de testes normais rotulados como instáveis.

Na Figura 9 está presente o resultado da classificação utilizando o algoritmo K-NN: 25 casos de testes rotulados corretamente como instáveis e 724 rotulados corretamente como não instáveis. Neste gráfico vemos um total de 12 erros: 12 *flaky tests* rotulados como não instáveis e nenhum teste normal rotulado incorretamente como instável.

Na Figura 10 está presente o resultado da classificação utilizando o algoritmo LDA: 23 casos de testes rotulados corretamente como instáveis e 708 rotulados corretamente como não

instáveis. Neste gráfico vemos um total de 30 erros: 14 *flaky tests* rotulados como não instáveis e 16 casos de testes normais rotulados como instáveis.

Na Figura 11 está presente o resultado da classificação utilizando o algoritmo de regressão logística: 27 casos de testes rotulados corretamente como instáveis e 722 rotulados corretamente como não instáveis. Neste gráfico vemos um total de 12 erros: 10 *flaky tests* rotulados como não instáveis e 2 casos de testes normais rotulados como instáveis. Este gráfico apresenta o melhor resultado na questão de precisão de identificação de *flaky tests*.

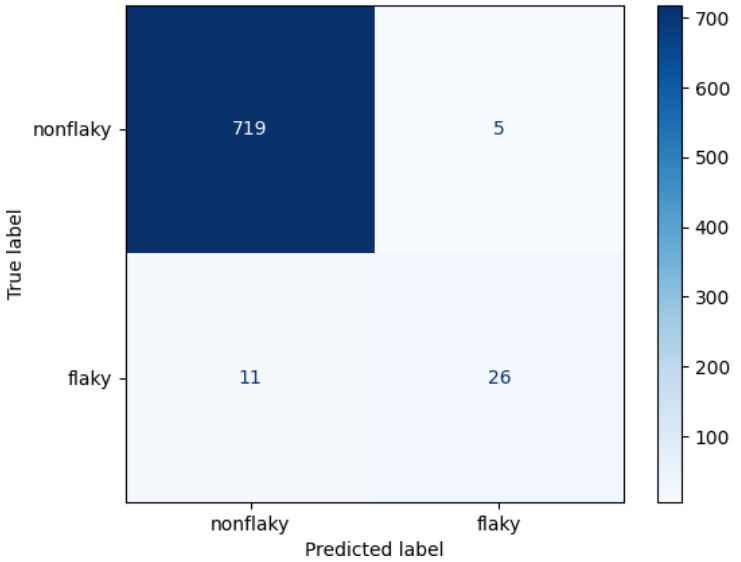
Na Figura 12 está presente o resultado da classificação utilizando o algoritmo Naive Bayes: 32 casos de testes rotulados corretamente como instáveis e 486 rotulados corretamente como não instáveis. Neste gráfico vemos um total de 243 erros: 5 *flaky tests* rotulados como não instáveis e 238 casos de testes normais rotulados como instáveis. Os resultados deste algoritmo não são bons, apresentando MCC de 0.24 e o *recall* de 0.68.

Na Figura 13 está presente o resultado da classificação utilizando o algoritmo Perceptron: 12 casos de testes rotulados corretamente como instáveis e 724 rotulados corretamente como não instáveis. Neste gráfico vemos um total de 25 erros: 25 *flaky tests* rotulados como não instáveis e nenhum teste normal rotulado incorretamente como instável.

Na Figura 14 está presente o resultado da classificação utilizando o algoritmo Random Forest: 15 casos de testes rotulados corretamente como instáveis e 724 rotulados corretamente como não instáveis. Neste gráfico vemos um total de 2 erros: 22 *flaky tests* rotulados como não instáveis e nenhum teste normal rotulado incorretamente como instável.

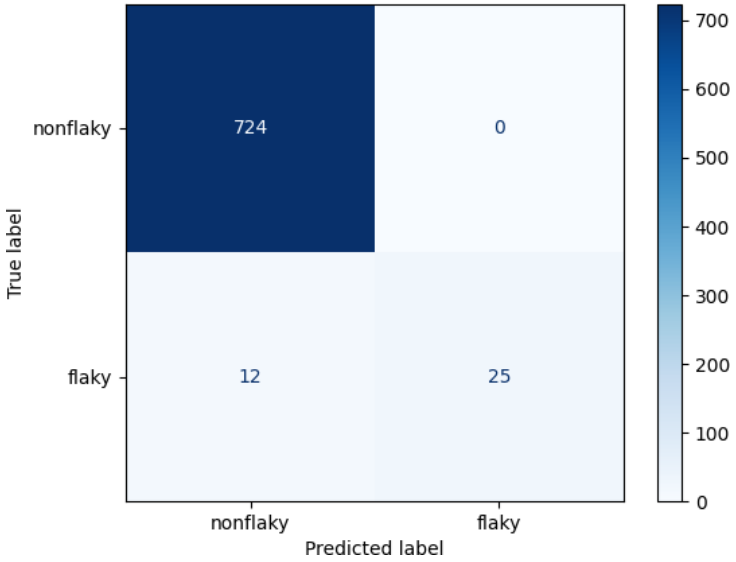
Na Figura 15 está presente o resultado da classificação utilizando o algoritmo SVM: 2 casos de testes rotulados corretamente como instáveis e 724 rotulados corretamente como não instáveis. Neste gráfico vemos um total de 35 erros: 35 *flaky tests* rotulados como não instáveis

Figura 8 – Matriz de confusão para o algoritmo Decision Tree



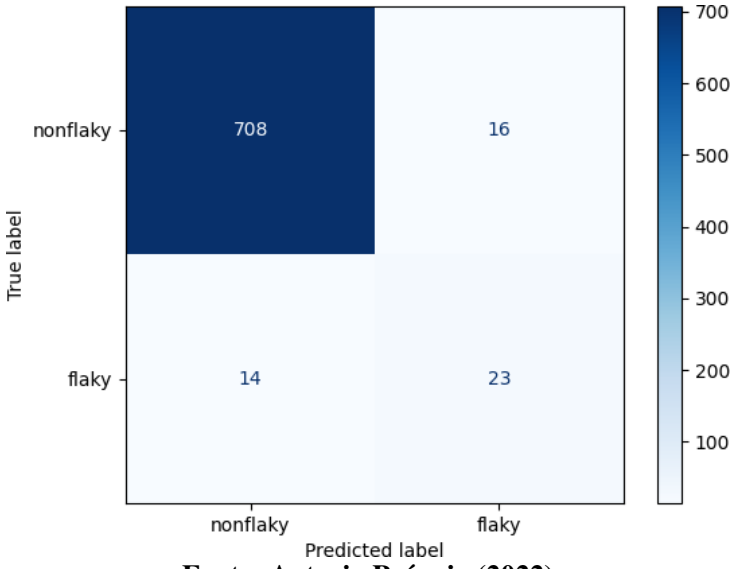
Fonte: Autoria Própria (2022).

Figura 9 – Matriz de confusão para o algoritmo K-nn



Fonte: Autoria Própria (2022).

Figura 10 – Matriz de confusão para o algoritmo LDA



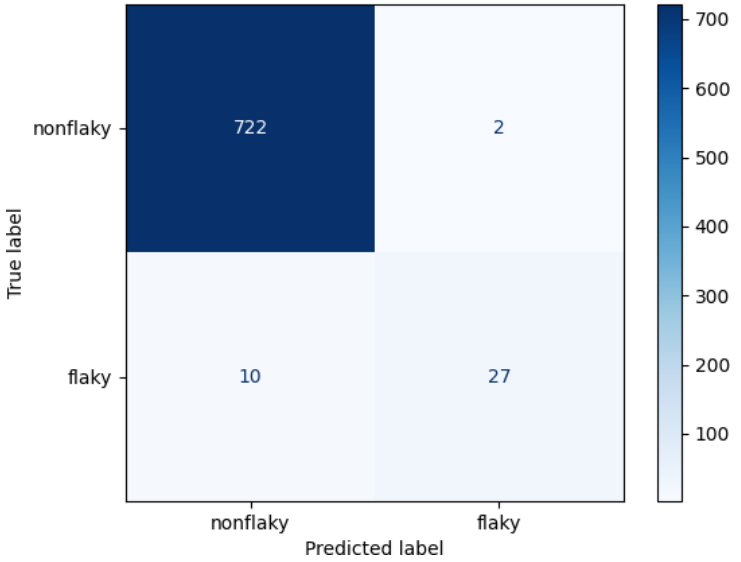
Fonte: Autoria Própria (2022).

e nenhum teste normal rotulado incorretamente como instável. Este algoritmo não apresentou bons resultados pois foi o que menos encontrou *flaky tests* no conjunto de dados proposto.

Resposta para QP1:

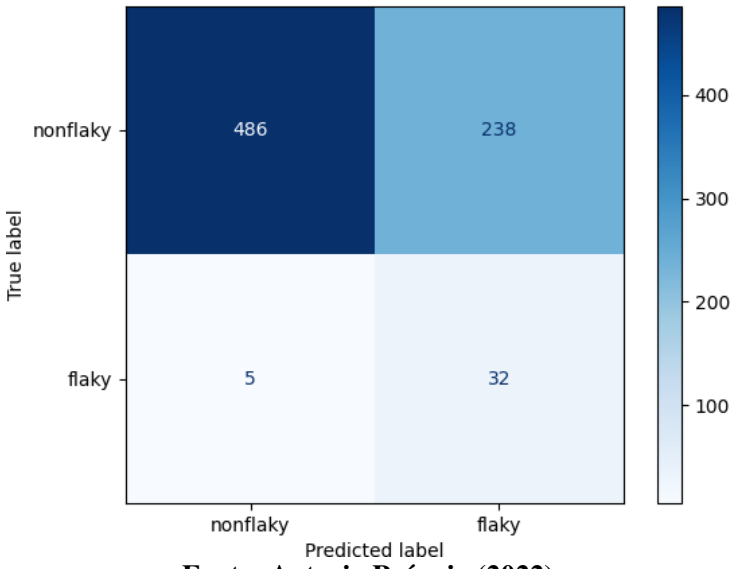
Apresentamos bons resultados durante a execução da maioria dos classificadores, tendo como o melhor resultado o algoritmo de regressão logística apresentando uma precisão, *recall* e f1-score de 98%; e uma área sob a curva de 95%. O algoritmo K-NN apresentou precisão, *recall* e f1-score similares à regressão logística, embora com AUC inferior.

Figura 11 – Matriz de confusão para o algoritmo Logistic Regression



Fonte: Autoria Própria (2022).

Figura 12 – Matriz de confusão para o algoritmo Naive Bayes



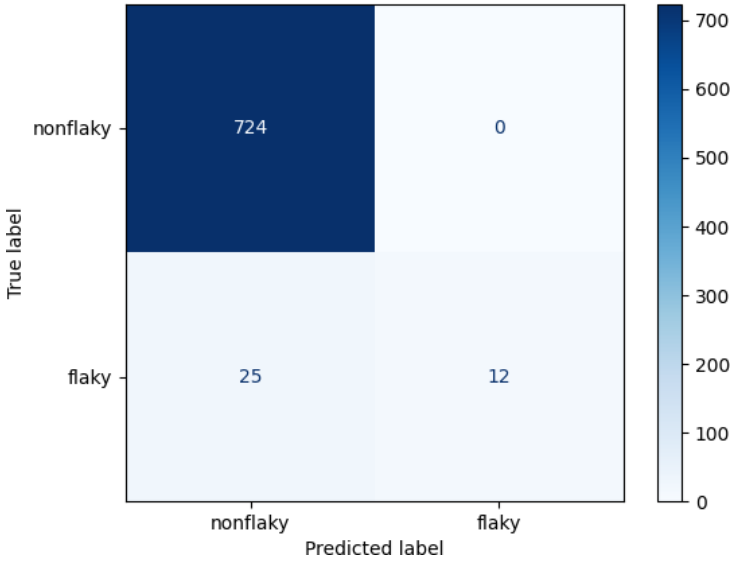
Fonte: Autoria Própria (2022).

4.2 Resposta para a questão de pesquisa QP2: Quais identificadores presentes em códigos de testes estão fortemente associados à instabilidade?

A Tabela 5 mostra as 20 features com maior ganho de informação juntamente com sua frequência em casos de teste instáveis e não instáveis. O ganho de informação foi calculado de maneira idêntica ao trabalho apresentado por Camara *et al.* (2021b): utilizando a função 'mutual_info_classif' presente na biblioteca sklearn em python.

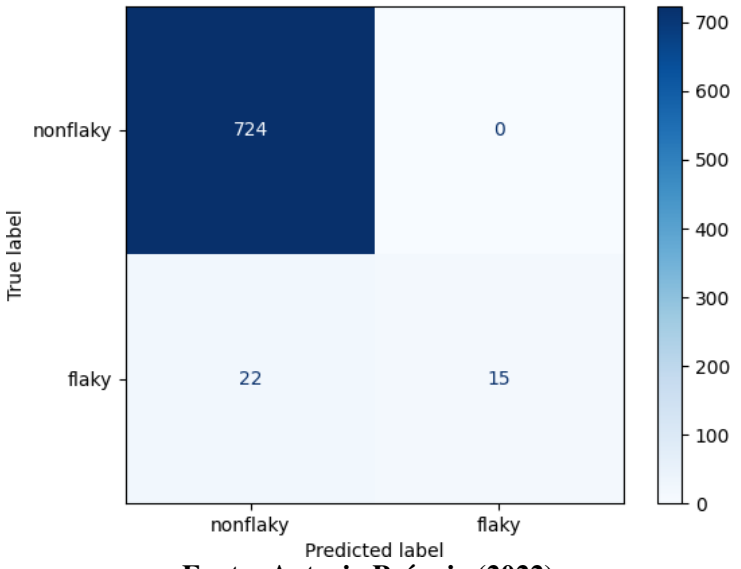
A presença do termo 'then' em primeiro lugar e do termo 'await' em 15° podem representar problemas com a espera assíncrona (LUO *et al.*, 2014). Porém, é possível notar uma grande diferença entre os dois termos: o termo 'then' está presente em *flakies* mais do que o

Figura 13 – Matriz de confusão para o algoritmo Perceptron



Fonte: Autoria Própria (2022).

Figura 14 – Matriz de confusão para o algoritmo Random Forest

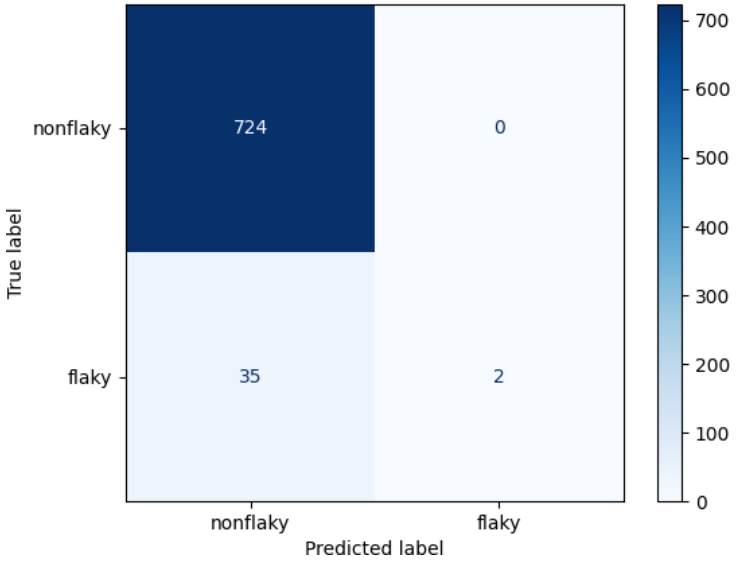


Fonte: Autoria Própria (2022).

termo ‘await’, e este por sua vez está presente em diversos casos de testes rotulados como *non flaky*. Esses termos possuem a mesma finalidade: esperar uma promessa ser executada, porém, geralmente são utilizadas em contextos diferentes. A palavra ‘then’ é utilizada quando existem diversas *promises* à serem aninhadas e executadas sequencialmente, por exemplo, em testes de sistema e de interface. Já a palavra ‘await’ pode ser utilizada pontualmente para esperar um recurso do servidor.

Quando funções assíncronas são executadas e demoram para retornar seu resultado, ocorre o *timeout* no teste. Esse evento não ocorrerá em todas execuções, somente naquelas que demorarem para executar. Isso ocorre na prática e muitos desenvolvedores, por não conseguir prever a instabilidade, adicionam *delays* no código para mitigá-las (ROMANO *et al.*, 2021).

Figura 15 – Matriz de confusão para o algoritmo SVM



Fonte: Autoria Própria (2022).

Tabela 5 – As 20 features com melhor ganho de informação

Rank	token	Ganho de informação	Ocorrências	Ocorrências em Flakies	Ocorrências em Não Flakies
1	then	0.034	46	41	5
2	gd	0.023	27	27	0
3	function	0.021	64	32	32
4	done	0.021	67	36	31
5	cy	0.020	26	24	2
6	getByTestID	0.018	21	21	0
7	click	0.018	67	31	36
8	it	0.016	19	19	0
9	var	0.015	58	27	31
10	0	0.015	276	44	232
11	Plotly	0.014	16	16	0
12	layout	0.013	21	17	4
13	1	0.013	247	37	210
14	should	0.011	36	17	19
15	await	0.011	400	33	367
16	return	0.010	79	23	56
17	type	0.010	60	20	40
18	data	0.009	88	19	69
19	const	0.008	1039	46	993
20	page	0.008	23	13	10

Fonte: Autoria Própria (2022).

Os termos ‘gd’ e ‘Plotly’ representam classes de variáveis utilizadas unicamente no projeto Plotly [1] e estão fortemente associados à instabilidade. Esses dois termos são específicos do domínio do framework plolty.js. Podem existir vocabulários específicos de domínio, ou seja: palavras que são utilizadas em contextos específicos. Neste sentido, se o conjunto de dados

[1] https://github.com/plotly/plotly.js

abrangesse mais projetos, algumas palavras (e.g. ‘Plotly’ e ‘gd’) poderiam perder seu ganho de informação por serem específicas a um domínio. Ao mesmo tempo, é possível que outras bibliotecas utilizem o framework com testes instáveis, o que reforçaria a importância desses termos ou deste domínio de aplicação quanto à instabilidade dos casos de teste.

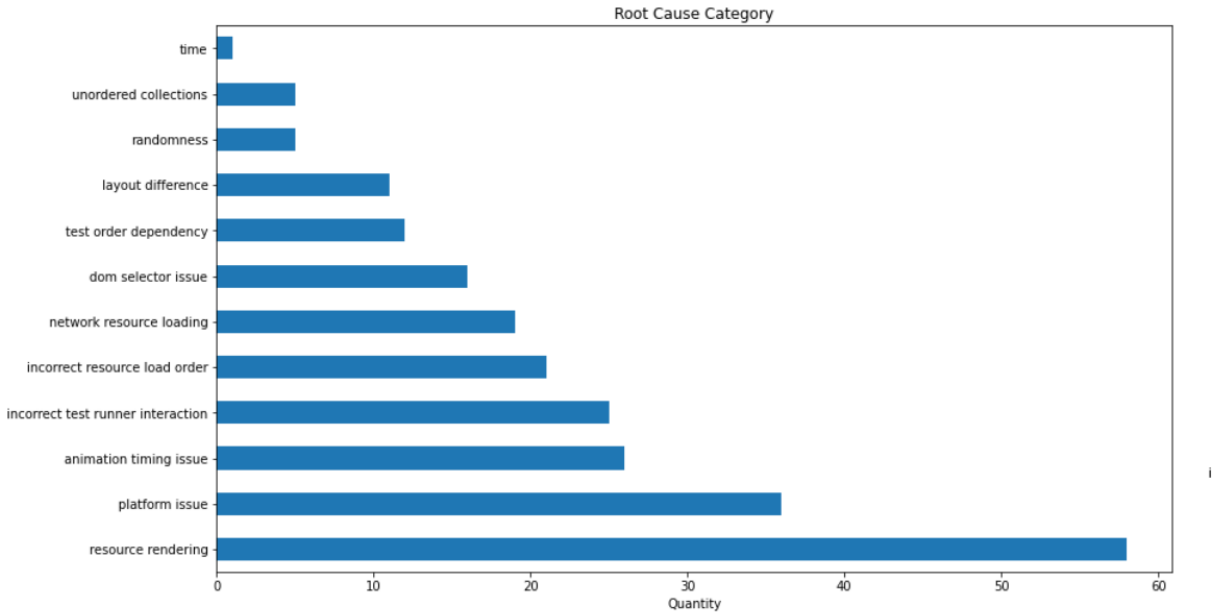
Encontramos duas palavras associadas à instabilidade que estão relacionadas com a utilização da ferramenta de teste de front-end web Cypress[2], são elas : ‘cy’ e ‘getByTestID’. No trabalho apresentado por Pinto *et al.* (2020) aparecem palavras semelhantes, por exemplo: ‘id’ e ‘getId’. Surge um questionamento que pode ser analisado em trabalhos futuros: ‘Ferramentas para execução de testes possuem elementos que causam instabilidades ou eles facilitam a detecção de instabilidades em testes?’. Por um lado, no conjunto de dados utilizado, existem várias instabilidades associadas à interações com a ferramenta de teste (*incorrect test runner interaction*), o que pode ser um indicador de que algumas ferramentas de teste são mais propensas às instabilidades. Por outro lado, acreditamos que tais ferramentas para execução de testes facilitam a identificação de *flaky tests* pois permitem a execução de testes de sistemas, que são mais propensos à instabilidade. Testes de sistemas estão fortemente relacionados com instabilidade pois são extensos, necessitam de alta integração de recursos e podem ser executados em diferentes ordens. No trabalho apresentado por Pinto *et al.* (2020), é utilizada uma característica relacionada ao tamanho do teste (linhas de código). Existe a hipótese de que testes de sistema são mais longos em questão de números de linhas, tornando essa medida importante no vocabulário estabelecido naquele trabalho.

Para entender melhor se as palavras instáveis estão relacionadas com causas de instabilidades realizamos uma visualização no conjunto de dados original (ROMANO *et al.*, 2021) na Figura 16. Associamos as palavras instáveis com as causas de instabilidade. Na Figura 16 é possível visualizar que as principais categorias de causa raiz de instabilidades estão associadas com palavras do vocabulário de testes instáveis. Por exemplo, as palavras ‘layout’, ‘gd’, ‘click’ estão relacionadas com a renderização de recursos. As palavras ‘await’, ‘then’, ‘return’ podem estar relacionadas com espera assíncrona. Assim, esta abordagem alerta desenvolvedores quais palavras estão relacionadas com instabilidades. Porém, quando um teste possui aquela palavra não implica que certamente ele será instável, e sim que ele possui uma chance de falhar em diversas execuções.

Foram analisados as top-20 *features* com ganho de informação pois este dado revela a importância da palavra durante a identificação de uma instabilidade. Não foram analisados os top-20 palavras que mais aparecem em instabilidades pois este dado pode estar ‘viciado’ com palavras que são comuns à maioria de testes Javascript. Por exemplo, a palavra que possui mais ocorrências em testes instáveis é ‘expect’ conforme apresentado na Tabela 6. No entanto, a palavra ‘expect’ é comum e não necessariamente está associada com instabilidades. Portanto, podemos dar maior relevância para os termos com ganho de informação em um vocabulário instável.

[2] https://github.com/cypress-io/cypress

Figura 16 – Categorias de causas raiz de instabilidade no conjunto de dados inicial.



Fonte: Autoria Própria (2022).

Resposta para QP2

O vocabulário relacionado com a instabilidade possui palavras relacionadas com espera assíncrona, por exemplo: ‘then’,‘await’,‘return’. Existem palavras relacionadas à visualização, interação com usuário, por exemplo: ‘layout’ e ‘click’. Encontramos duas palavras instáveis que estão relacionadas com a utilização do framework Cypress: ‘cy’ e ‘getByTestID’.

4.3 Discussões

Este trabalho seguiu a implementação dos mesmos 5 modelos de classificação utilizados por Pinto *et al.* (2020) e incrementados por Camara *et al.* (2021b). A Tabela 7 apresenta um resumo do melhor e pior resultado dos algoritmos utilizados por Pinto *et al.* (2020). O algortimo que obteve o melhor desempenho foi o Random Forest com os seguintes valores: 0.99 de precisão, 0.91 de *recall*, 0.95 de F1-score, 0.90 de MCC e 0.98 de AUC. O algoritmo que teve o pior desempenho foi o Naive Bayes com os seguintes valores: 0.93 de precisão, 0.80 de *recall*, 0.86 de F1-score, 0.74 de MCC e 0.93 de AUC.

A Tabela 8 apresenta um resumo do melhor e pior resultado dos algoritmos utilizados por Camara *et al.* (2021b). 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-score,

Tabela 6 – As 20 features com mais ocorrências em flaky tests

Rank	token	Ganho de informação	Ocorrências	Ocorrências em Flakies	Ocorrências em Não Flakies
1	expect	0.01	1496	65	1431
2	const	0.01	1039	46	993
3	0	0.02	276	44	232
4	then	0.03	46	41	5
5	1	0.01	247	37	210
6	done	0.02	67	36	31
7	await	0.01	400	33	367
8	function	0.02	64	32	32
9	click	0.02	67	31	36
10	gd	0.02	27	27	0
11	var	0.02	58	27	31
12	true	0	302	27	275
13	cy	0.02	26	24	2
14	return	0.01	79	23	56
15	toBe	0	529	23	506
16	getByTestID	0.02	21	21	0
17	type	0.01	60	20	40
18	to	0.01	156	20	136
19	it	0.02	19	19	0
20	data	0.01	88	19	69

Fonte: Autoria Própria (2022).

Tabela 7 – Resumo dos resultados de Pinto *et al.* (2020)

Algoritmo	**Precisão**	**Recall**	**F1 Score**	**MCC**	**AUC**
Melhor : Random Forest	0.99	0.91	0.95	0.90	0.98
Pior : Naive Bayes	0.93	0.80	0.86	0.74	0.93

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

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-score, 0,63 de MCC e 0,87 de AUC.

Tabela 8 – Resumo dos resultados de Camara *et al.* (2021b)

Algoritmo	**Precisão**	**Recall**	**F1 Score**	**MCC**	**AUC**
Melhor : Regressão Logística	0.91	0.91	0.91	0.84	0.96
Pior : Decision Tree	0.87	0.86	0.86	0.74	0.87

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

A Tabela 9 apresenta um resumo do melhor e pior resultado obtidos durante a execução dos modelos de classificação neste trabalho. O algoritmo que apresentou melhor desempenho foi o de Regressão Logística, com os seguintes resultados: 0.98 de precisão, 0.98 de *recall*, 0.98 de F1-score, 0.82 de MCC e 0.95 de AUC. O algoritmo que teve o pior desempenho foi o Naive Bayes com os seguintes valores: 0.94 de precisão, 0.68 de *recall*, 0.77 de F1-score, 0.24 de MCC e 0.77 de AUC.

Tabela 9 – Faixa de desempenho dos algoritmos de classificação.

Algoritmo	Precisão	Recall	F1 Score	MCC	AUC
Melhor : Regressão Logística	0.98	0.98	0.98	0.82	0.95
Pior : Naive Bayes	0.94	0.68	0.77	0.24	0.77

Fonte: Autoria Própria (2022).

Enquanto no estudo de Pinto *et al.* (2020) o Random Forest obteve o melhor resultado, nos estudos de Camara *et al.* (2021b) e neste trabalho a Regressão Logística foram melhores. Entretanto, todos os estudos os melhores classificadores apresentaram bom desempenho. Em relação ao algoritmo com pior resultado, este estudo e o de Pinto *et al.* (2020) tiveram resultados concordantes, embora a revocação, F1 Score, MMC e AUC observadas neste trabalho tenham obtido resultado inferiores àquele.

As palavras mais comuns no vocabulário original, proposto por Pinto *et al.* (2020), são: 'job', 'table' e 'action'. As palavras mais comuns no vocabulário proposto por Camara *et al.* (2021b) são: 'job', 'table' e 'service'. Tal semelhança entre os trabalhos anteriores se dá pelo fato de utilizarem conjuntos de dados semelhantes, sobre casos de testes de aplicações Java.

No vocabulário obtido neste trabalho as 4 palavras com maior ganho de informação são: 'then', 'gd', 'cy' e 'layout'. O termo que possui maior relevância em termos de ganho de informação é o 'then' que está fortemente associado a instabilidades ocasionadas pela espera assíncrona. Duas palavras presentes no vocabulário (entre projetos) de Camara *et al.* (2021b) que também estão presentes em nosso vocabulário são: 'await' e 'return'. Este fato comprova que a espera assíncrona é um problema existente em diversos contextos de casos de testes. Neste caso, observa-se uma coincidência da terminologia adotada nos dados analisados, apesar da diferença de linguagem adotada (Java e Javascript) e domínio das aplicações. Embora o estudo de Camara *et al.* (2021b) apresente indícios de que os modelos treinados para um conjunto de projetos não possua desempenho adequada para outros projetos de software na mesma linguagem, podemos observar que existe uma interseção nos vocabulários observados.

4.4 Considerações Finais

Podemos prever instabilidades em casos de testes Javascript utilizando vocabulário instável com uma precisão considerável. Esta abordagem é vantajosa e promissora pois economiza tempo necessário para identificação de instabilidades. Porém esta técnica apresenta limitações referentes ao conjunto de dados utilizado e à generalização dos termos instáveis para diferentes contextos. A predição de instabilidade ainda não é capaz de substituir a reexecução como abordagem principal para identificação de *flaky tests*. Porém, apresenta resultados bons como técnica complementar à reexecução, com intuito de reduzir o custo da identificação de instabilidades em testes.

O vocabulário instável possui palavras relacionadas com espera assíncrona e *callbacks*, por exemplo: ‘then’, ‘await’, ‘return’, ‘function’, ‘done’. O termo que possui maior relevância em termos de ganho de informação é o ‘then’ que está fortemente associado a instabilidades ocasionadas pela espera assíncrona. Existem palavras relacionadas à visualização, interação com usuário, por exemplo: ‘layout’ e ‘click’. Encontramos duas palavras que estão relacionadas com a utilização do framework Cypress, são elas : ‘cy’ e ‘getByTestID’. Também encontramos palavras específicas de frameworks, tais como: ‘gd’ e ‘Plotly’.

5 CONCLUSÕES

Testes instáveis são aqueles que as vezes passam e as vezes falham, sem qualquer mudança no código de teste. Eles diminuem significativamente a confiança de um conjunto de testes de regressão automatizado. 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 não ignorá-los.

A maioria dos trabalhos se concentram em identificar e caracterizar as causas de instabilidades em testes. Porém, pouco esforço tem sido gasto para reconhecer de maneira eficiente um teste instável (CAMARA *et al.*, 2021b). Os estudos se concentram em um contexto de projetos Java e pouco recurso é gasto para projetos Javascript. Portanto, este trabalho se concentra na predição de testes instáveis utilizando identificadores de código-fonte Javascript (e.g. nomes de métodos e variáveis). São respondidas duas questões de pesquisa:

- QP1: Com qual precisão podemos prever casos de teste instáveis em Javascript ?
  - **Resposta:** Observamos que os oito algoritmos de aprendizado de máquina utilizados tiveram bom desempenho na distinção testes não instáveis. Em particular, a regressão logística teve a melhor precisão (0,984) e foi melhor em termos de recordação (0,98). O segundo melhor resultado foi do algoritmo K-NN.
- QP2: Quais identificadores presentes em códigos de testes em Javascript estão fortemente associados à instabilidade?
  - **Resposta:** O vocabulário instável obtido contém palavras relacionadas com espera assíncrona, por exemplo: ‘then’, ‘await’, ‘return’. O termo que possui maior relavância em termos de ganho de informação é o ‘then’ que está fortemente associado a instabilidades ocasionadas pela espera assíncrona. Existem palavras relacionadas à visualização, interação com usuário, por exemplo: ‘layout’ e ‘click’. Encontramos duas palavras instáveis que estão relacionadas com a utilização do framework Cypress [1], são elas : ‘cy’ e ‘getByTestID’.

Para as questões de pesquisa começamos extraindo casos de teste de um conjunto de dados proposto por Romano *et al.* (2021). No conjunto de dados inicial eram presentes 235 instâncias de interações de desenvolvedores no Github relatando *flaky tests*. Buscamos neste conjunto *commits* para localizar o código-fonte de casos de testes rotulados como instáveis. O conjunto de dados proposto possui casos de 144 testes no formato de tokens, rotulados como

[1] https://github.com/cypress-io/cypress

instáveis ou não. Extraímos todos os identificadores de casos de teste usando procedimentos tradicionais de tokenização. Finalmente, casos de testes pré-processados foram usados como entrada para oito algoritmos de aprendizado de máquina. Com estes dados foram aplicados modelos de classificação de instabilidades. Os resultados são: conjuntos de dados sobre testes 'tokenizados' e rotulados como instáveis e normais; a precisão de diferentes modelos de classificação de instabilidade e um vocabulário instável.

A visualização de *flaky tests* no formato de texto permite comparar quais identificadores são comuns em diferentes linguagens e contextos, e também quais são específicas de seu escopo. Neste sentido, o presente trabalho visa contribuir para o conhecimento de vocabulários instáveis em um escopo de projetos implementados com JavaScript.

A predição de *flaky tests* a partir da análise estática do texto dos casos de teste é uma abordagem nova que visa reduzir o custo gasto com reexecuções de testes para identificar instabilidades. Pinto *et al.* (2020) e Camara *et al.* (2021b) obtiveram bons resultados para esta abordagem em um cenário de projetos Java. Neste trabalho são apresentados resultados relevantes para identificação de *flaky tests* em projetos que utilizam Javascript. Não obstante, são necessários mais estudos empíricos para consolidar esta abordagem como uma técnica confiável para identificação de instabilidades em testes.

Existem outros trabalhos nesta linha de pesquisa. Uma possível linha seria a construção de ferramentas que possam ajudar os desenvolvedores a identificar testes instáveis utilizando a sugestão de código-fonte para testes implementados no Javascript. Inicialmente, essas ferramentas poderiam receber como entrada as features que encontramos com o maior ganho de informação, e futuramente com novas features. De acordo com Pinto *et al.* (2020) essas ferramentas também permitem que os desenvolvedores confirmem se um teste é instável ou não, e com base nessa decisão, essas ferramentas podem melhorar interativamente seu próprio dicionário de palavras relacionadas a flaky. Neste sentido, o vocabulário instável na linguagem Javascript pode ser incrementado com diversos casos de testes de diferentes contextos.

Um outro ponto a ser trabalhado é quanto aos conjuntos de dados sobre testes instáveis para Javascript. O conjunto atual possui poucos testes instáveis. A criação de conjuntos de melhor qualidade, que empreguem técnicas de reexecução, propiciará a execução de estudos de predição de instabilidade com mais rigor. Neste sentido, trabalhos sobre mineração de testes instáveis no Github podem servir de base para diversos outros trabalhos neste segmento.

REFERÊNCIAS

ALSHAHWAN, N.; GAO, X.; HARMAN, M.; JIA, Y.; MAO, K.; MOLS, A.; TEI, T.; ZORIN, I. Deploying search based software engineering with sapienz at facebook. *In*: COLANZI, T. E.; MCMINN, P. (Ed.). **10th International Symposium on Search-Based Software Engineering**. [*S.l.*]: Springer, 2018. (Lecture Notes in Computer Science, v. 11036), p. 3–45. ISBN 978-3-319-99240-2. ISSN 0302-9743.

ALSHAMMARI, A.; MORRIS, C.; HILTON, M.; BELL, J. FlakeFlagger: Predicting flakiness without rerunning tests. *In*: **43rd International Conference on Software Engineering**. [*S.l.*]: IEEE, 2021. p. 1572–1584.

BAVOTA, G.; QUSEF, A.; OLIVETO, R.; LUCIA, A. D.; BINKLEY, D. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. *In*: **28th IEEE International Conference on Software Maintenance**. [*S.l.*: *s.n.*], 2012. p. 56–65. ISBN 978-1-4673-2313-0. ISSN 1063-6773.

BELL, J.; LEGUNSEN, O.; HILTON, M.; ELOUSSI, L.; YUNG, T.; MARINOV, D. 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, A.; CRUCIANI, E.; MIRANDA, B. A. F. de; VERDECCHIA, R. **Know your neighbor: fast static prediction of test flakiness**. Pisa, Itália, 2020.

CAMARA, B.; SILVA, M.; ENDO, A.; VERGILIO, S. On the use of test smells for prediction of flaky tests. *In*: ELER, M.; ASSUNçãO, W. 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, B. H. P.; SILVA, M. A. G.; ENDO, A. T.; VERGILIO, S. R. 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.

CAMPBELL, D. T.; STANLEY, J. C. **Experimental and quasi-experimental designs for research**. 1. ed. Chicago, IL, EUA: Houghton Mifflin, 1963. 84 p. ISBN 0-395-30787-2.

DEURSEN, A. V.; MOONEN, L.; BERGH, A.; KOK, G. Refactoring test code. *In*: MARCHESI, M. (Ed.). **Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP 2001)**. [*S.l.*: *s.n.*], 2001. p. 92–95.

ECK, M.; PALOMBA, F.; CASTELLUCCIO, M.; BACCHELLI, A. Understanding flaky tests: The developer's perspective. *In*: APEL, S.; RUSSO, A. (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, L. **Determining Flaky Tests from Test Failures**. abr. 2015. Dissertação (mathesis) — University of Illinois at Urbana-Champaign, Urbana, Illinois,, abr. 2015. Disponível em: http://hdl.handle.net/2142/78543.

FAYYAD, U.; PIATETSKY-SHAPIRO, G.; SMYTH, P. 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.

HARMAN, M.; O'HEARN, P. 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.

HARMAN, M.; O'HEARN, P. 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.

HERZIG, K.; NAGAPPAN, N. 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, E.; GOUSIOS, G.; BLINCOE, K.; SINGER, L.; GERMAN, D. M.; DAMIAN, D. 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.

KING, T. M.; SANTIAGO, D.; PHILLIPS, J.; CLARKE, P. J. Towards a bayesian network model for predicting flaky automated tests. *In*: **2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)**. [*S.l.*]: IEEE, 2018. p. 100–107. ISBN 978-1-5386-7840-4.

LAM, W.; GODEFROID, P.; NATH, S.; SANTHIAR, A.; THUMMALAPENTA, S. Root causing flaky tests in a large-scale industrial setting. *In*: **Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis**. [*S.l.*: *s.n.*], 2019. p. 101–111.

LAM, W.; GODEFROID, P.; NATH, S.; SANTHIAR, A.; THUMMALAPENTA, S. Root causing flaky tests in a large-scale industrial setting. *In*: **Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis**. New York, NY, EUA: ACM, 2019. p. 101–111. ISBN 9781450362245.

LAM, W.; OEI, R.; SHI, A.; MARINOV, D.; XIE, T. 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.

LISTFIELD, J. **Where do our flaky tests come from?** 2017. Página Web. Disponível em: https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html.

LUO, Q.; HARIRI, F.; ELOUSSI, L.; MARINOV, D. 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.

MEMON, A. M.; COHEN, M. 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, J. Flaky tests at google and how we mitigate them. **Online] https://testing. googleblog. com/2016/05/flaky-tests-at-google-and-how-w e. html**, 2016.

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

MIRANDA, C.; AVELINO, G.; NETO, P. S.; SILVA, V. 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, J.; AUGUSTO, C.; BERTOLINO, A.; RIVA, C. D. L.; TUYA, J. 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.

PALMER, J. **Test Flakiness – Methods for identifying and dealing with flaky tests**. 2019. Disponível em: https://engineering.atspotify.com/2019/11/18/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/.

PALOMBA, F.; ZAIDMAN, A. 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.

PINTO, G.; MIRANDA, B.; DISSANAYAKE, S.; D'AMORIM, M.; TREUDE, C.; BERTOLINO, A. What is the vocabulary of flaky tests? *In*: GOUSIOS, G.; NADI, S. (Ed.). **17th International Conference on Mining Software Repositories (MSR)**. New York, NY, EUA: ACM, 2020. ISBN 9781450375177.

PRESTON-WERNER, T.; WANSTRATH, C.; HYETT, P. J.; CHACON, S. **GitHub**. 2008. Programa de computador. Disponível em: https://github.com.

REZENDE, S. O. **Sistemas inteligentes: fundamentos e aplicações**. 1. ed. Barueri, SP, Brasil: Manole, 2005. 550 p. ISBN 8520416837.

ROMANO, A.; SONG, Z.; GRANDHI, S.; YANG, W.; WANG, W. An empirical analysis of UI-based flaky tests. *In*: **43rd International Conference on Software Engineering**. [*S.l.*]: IEEE, 2021. p. 1585–1597.

SHI, A.; LAM, W.; OEI, R.; XIE, T.; MARINOV, D. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. *In*: APEL, S.; RUSSO, A. (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.

SOUZA, L. B. D.; CAMPOS, E. C.; MAIA, M. d. 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.

TREUDE, C.; ROBILLARD, M. P. Augmenting api documentation with insights from stack overflow. *In*: IEEE. **2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)**. [*S.l.*], 2016. p. 392–403.

TUFANO, M.; PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; LUCIA, A. D.; POSHYVANYK, D. An empirical investigation into the nature of test smells. *In*: **31st IEEE/ACM International Conference on Automated Software Engineering**. New York, NY, EUA: ACM, 2016. p. 4–15. ISBN 978-1-4503-3845-5.

VERDECCHIA, R.; CRUCIANI, E.; MIRANDA, B.; BERTOLINO, A. Know you neighbor: Fast static prediction of test flakiness. **IEEE Access**, IEEE, 2021.