UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO

Lucas Cordeiro Sgotti

Impacto da latência na verificação e do atraso acentuado na rotulação para detecção de falhas de software

Recife

2023

Lucas Cordeiro Sgotti

Impacto da latência na verificação e do atraso acentuado na rotulação para detecção de falhas de software

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação da Universidade Federal Rural de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Universidade Federal Rural de Pernambuco –

UFRPE Departamento de Computação

Curso de Bacharelado em Ciências da Computação

Orientador: George Gomes Cabral

Recife

2023

Dados Internacionais de Catalogação na Publicação
Universidade Federal Rural de Pernambuco
Sistema Integrado de Bibliotecas
Gerada automaticamente, mediante os dados fornecidos pelo(a) autor(a)

S523Li Sgotti, Lucas Cordeiro
Impacto da latência na verificação e do atraso acentuado na rotulação para detecção de falhas de software / Lucas Cordeiro Sgotti. - 2023.
40 f. : il.

Orientador: George Gomes Cabral.
Inclui referências, apêndice(s) e anexo(s).

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal Rural de Pernambuco, Bacharelado em Ciência da Computação, Recife, 2023.

1. detecção de falhas em software. 2. machine learning. 3. aprendizagem online. 4. desbalanceamento de classes. I. Cabral, George Gomes, orient. II. Título

CDD 004

MINISTÉRIO DA EDUCAÇÃO E DO DESPORTO
UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO (UFRPE)
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

http://www.bcc.ufrpe.br

Ficha de Aprovação do Trabalho de Conclusão de Curso

Trabalho defendido por Lucas Cordeiro Sgotti às 14 horas do dia 25 de abril de 2023, na sala 36 do departamento de computação, como requisito para conclusão do curso de Bacharelado em Ciência da Computação da Universidade Federal Rural de Pernambuco, intitulado "Impacto da latência na verificação e do atraso acentuado na rotulação para detecção de falhas de software", orientado por George Gomes Cabral e aprovado pela seguinte banca examinadora:

George Gomes Cabral
DC/UFRPE

Sidney de Carvalho Nogueira
DC/UFRPE

Agradecimentos

Agradeço primeiramente aos meus pais, Frank e Virgínia, por me ensinarem a persistir e atribuir o verdadeiro valor ao meu aprendizado e a todo apoio que me deram nos momentos de dificuldade e indecisão.

Ao professor George Gomes Cabral pela paciência, orientação e ensinamentos durante esta disciplina.

Aos meus amigos Rodrigo Araújo Borges e Filipe Travassos Wiedemann com os quais contei com o apoio em momentos difíceis e a todos os outros amigos que também fazem parte da minha vida.

Por fim, agradeço a todos os profissionais da Universidade Federal Rural de Pernambuco que auxiliaram, e ainda auxiliam, na criação de uma experiência de sabedoria e aprendizado para mim e para cada aluno que ingressa na instituição.

“Depois de um tempo, você passa a não ligar para como os outros te chamam, e apenas confiar em você mesmo.”

(Shrek the Third 2007)

Resumo

A detecção de falhas de software é uma atividade inerente ao desenvolvimento de software e requer um esforço elevado de recursos humanos. Essa atividade muitas vezes não é priorizada no intuito de reduzir o custo final de um projeto. Just-in-Time Software Defect Prediction (JIT-SDP) é uma das abordagens utilizadas para predição de defeitos de software com o objetivo identificar de maneira automática através de métodos de aprendizagem de máquina artefatos de software propensos a conterem defeitos a partir de dados históricos. No entanto, a maioria das abordagens assume que as características do problema permanecem as mesmas com o passar do tempo, porém o desbalanceamento entre as classes é um problema que evolui com o tempo à medida que novos exemplos de treinamento vão chegando, por exemplo. Analisar o problema em um ambiente online significa que, além de outras coisas, há uma natureza cronológica intrínseca à abordagem que por sua vez, traz consigo alguns problemas, dentre eles o de latência na verificação, que se refere ao fato que os rótulos dos exemplos de treinamento podem chegar muito mais tarde do que suas características. Este trabalho visa investigar o impacto da latência na verificação no problema da detecção de defeitos em software, assim como o desempenho desses métodos de acordo com o grau de latência na verificação em exemplos da classe indutora de defeitos.

Palavras-chave: detecção de falhas em software, machine learning, aprendizagem online, desbalanceamento de classes.

Abstract

Software Defect Prediction is an activity inherent to software development and it requires a high amount of human effort. This activity is often not prioritized in order to reduce the project's expenses. Just-in-Time Software Defect Prediction (JIT-SDP) is one of the approaches used for predicting software defects in order to automatically identify, through machine learning methods, software artifacts likely to contain defects based on historical data. However, most approaches assume that the characteristics of the problem remain the same over time, but the imbalance between classes is a problem that evolves over time as new training examples arrive, for example. Analyzing the problem in an online environment means that, among other things, there is an intrinsic chronological aspect to be considered which, in turn, brings with it some issues, among them verification latency, which refers to the fact that training example labels can arrive much later than their characteristics. This work aims to investigate the impact of verification latency on the problem of detecting defects in software, as well as the performance of these methods according to the degree of verification latency in examples of the defect inducing class.

Keywords: Software defect prediction, machine learning, online learning, class imbalance.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

JIT-SDP	Just In Time - Software Development Prediction

Sumário

1. Introdução

A detecção de falhas em software é um problema desafiador, especialmente considerando que as equipes de qualidade de software têm recursos limitados de teste [1]. Como maneira de minimizar o gasto final com o desenvolvimento do produto, os recursos de garantia de qualidade de software são limitados e menos priorizados no orçamento, portanto devem ser concentrados em módulos do sistema que sejam mais suscetíveis à falha. Para este fim, modelos de aprendizagem de máquina para a previsão de defeitos são treinados usando dados históricos para identificar módulos de software propensos a defeitos. Depois de serem treinados usando dados históricos, esses modelos podem ser usados para direcionar o esforço do time de garantia de qualidade, de acordo com a propensão prevista a defeitos dos módulos de uma versão futura do software [2].

Predição de defeito a nível de mudança [1] ou predição de defeito "em tempo real", ou seja, em tempo de *commit* ao repositório de códigos, (JIT-SDP - *Just In Time Software Defect Prediction*) [3], é uma abordagem alternativa que traz consigo diversas vantagens [4]. Primeiro, como as mudanças geralmente são menores que módulos inteiros, o JIT-SDP permite fazer previsões de granularidade muito mais fina, possibilitando uma localização mais fácil do local de inspeção. Segundo, enquanto os módulos têm um grupo de autores diferentes, mudanças têm apenas um autor, o que torna a triagem de previsões do JIT-SDP mais fáceis, sendo possível atribuir previsões ao autor da alteração e portanto rastreando mais facilmente o causador da falha. Por último, ao contrário da previsão a nível de módulo, os modelos JIT podem verificar as alterações logo após elas serem produzidas, o que significa que os problemas podem ser inspecionados durante o projeto ao passo que decisões ainda estão frescas na mente dos desenvolvedores.

A maioria dos trabalhos que consideram JIT-SDP com um problema de classificação online pressupõem que as alterações de software que causam defeitos no passado tendem a ser semelhantes às futuras [5]. Segundo McIntosh e Kamei [2], as características que induzem defeitos e as alterações de software mudam durante o ciclo de vida de um software. Essas flutuações podem afetar negativamente o desempenho preditivo dos classificadores treinados em dados antigos necessitando que o modelo se adapte ao longo do tempo. Algoritmos que aprendem novos exemplos de treinamento ao longo do tempo geralmente são chamados de algoritmos de aprendizado online [7] . Além de mudanças nas características de indução de defeitos [3], o JIT-SDP também pode sofrer com a evolução do desbalanceamento entre as classes. Isso significa que o nível de desbalanceamento entre as classes evolui com o tempo [5].

Trabalhos existentes assumem que o desequilíbrio de classe é estático, ou seja, não evolui com o tempo. E, supondo que ele realmente evolui com o tempo, tais abordagens se tornam impróprias para lidar com o problema [5].

Técnicas de reamostragem são comumente usadas em problemas de classificação de problemas com classes desbalanceadas. Muitas vezes, é desejável ter um desempenho elevado na classe minoritária, mantendo um bom desempenho na classe majoritária [6]. No entanto, pouco foi explorado sobre o uso de técnicas de reamostragem em ambientes de aprendizado online.

Dentre outros, este trabalho foca no problema da rotulação dos dados e *commits* no JIT-SDP em um ambiente online. A rotulação de cada instância leva um tempo para que possa ser corretamente efetuada, pois não há como saber como classificar imediatamente uma instância no momento em que ela foi criada, este problema é chamado de latência na verificação e este trabalho visa definir a importância e ramificações causadas por ele.

Problema de Pesquisa

Como já mencionado modelos JIT-SDP em ambientes online podem resultar em diversos benefícios para o campo de detecção de falhas de software podendo apontar falhas em pedaços de código no momento do *commit* de artefatos de software ao repositório, possibilitando ao time de desenvolvimento maiores chances para solucionar os possíveis problemas de maneira imediata. Nesse contexto, um classificador online pode também facilmente se adaptar às mudanças relacionadas à falhas durante o ciclo de desenvolvimento do software. Porém um ambiente online traz problemas caracterizados pela evolução de classes não balanceadas, uma vez que a tendência é que exemplos de uma classe sejam sempre mais abundantes que a outra.

O problema de "verification latency", ou latência na verificação, como apontado por Ditzler *et al.* [7], se refere ao fato de que os rótulos associados a exemplos de treinamento chegam com atrasos. Não é possível saber no momento se um *commit* de uma nova alteração de software induz um defeito ou não. No caso de JIT-SDP, o atraso na rotulação de um commit indutor de defeito é computado como o intervalo entre o commit indutor de defeito e o commit que resolve esse defeito. Esse processo é realizado pelo algoritmo SZZ [13]. Para commits que não induzem defeitos (ou seja, commits limpos), os rótulos são atribuídos após um determinado período em dias sem que nenhum defeito seja identificado. Este é um aspecto crucial para um tratamento realístico do problema abordado. Além disso, existe a potencial ameaça ao desempenho preditivo dos classificadores apresentada pela evolução do desbalanceamento entre classes, especialmente em um ambiente online. Considerando isto, é também interessante mitigar

os problemas trazidos por este desequilíbrio de classes e utilizar o rebalanceamento dos dados para alcançar resultados mais precisos.

Tan et al. [8] descobriu que ignorar a latência na verificação leva a estimativas excessivamente otimistas do desempenho preditivo. Novos exemplos para o treinamento ficam disponíveis para compor novos lotes somente após um tempo de espera previamente definido. Este tempo de espera deve refletir o tempo que leva para que se possa classificar as mudanças ocorridas no software como não indutoras de defeitos. A figura abaixo, disponível em Cabral et al. [5], demonstra o atraso na descoberta de defeitos. Os atrasos variaram de 1 a 4210 dias (Aproximadamente 11 anos e meio). As medianas dos atrasos variaram de 15 a 416,5 dias, e foram menores ou perto de 90 dias (linha tracejada vermelha inferior) em 8 de 10 dos projetos (Fabric8, Jgroups, Camel, Brackets, Neutron, Broadleaf, Nova e NPM). Portanto, utilizar um tempo de espera de 90 a 100 dias implica que mais do que, ou aproximadamente, metade das mudanças indutoras de defeitos seriam rotuladas corretamente no momento de seu uso para aprendizado pelo modelo de classificador.

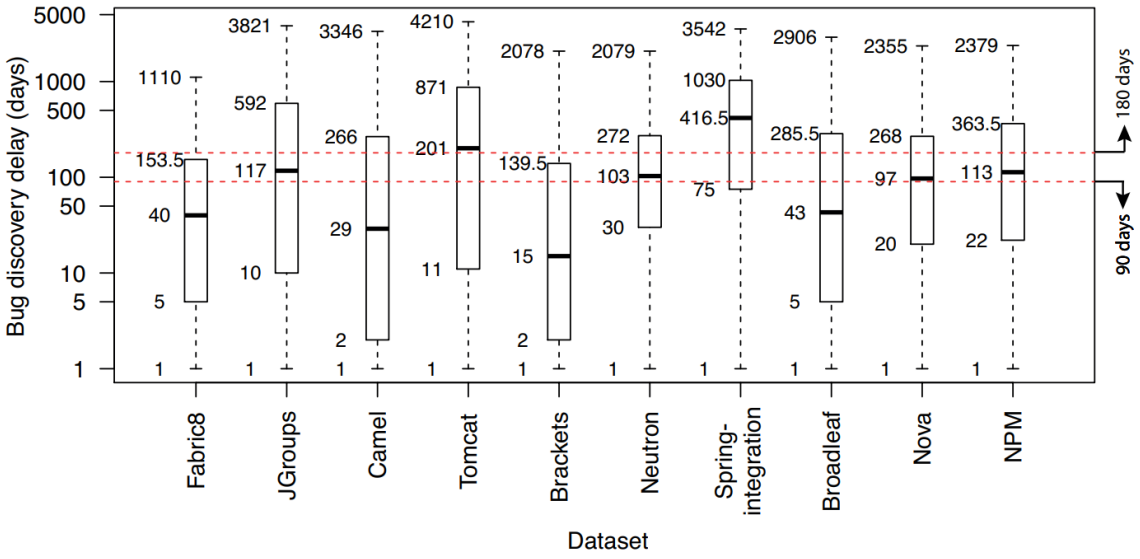

Figura 1. Atraso na descoberta de defeitos (em dias) para cada um dos projetos.

O problema de pesquisa deste trabalho aborda o impacto da não consideração do problema da latência na verificação, o quão importante é a ordem cronológica dos dados utilizados, assim como também avaliar o poder de generalização do classificador online através de exemplos (*commits*) contendo artefatos de software antigos de diferentes projetos.

2. Justificativa

O gerenciamento de riscos desempenha um papel crucial no sucesso do gerenciamento de projetos. Isto é especialmente verdade para projetos de software. F. Michael Dedolph [10] mostrou que, de maneira ampla à indústria de desenvolvimento de software, apenas 16,2% dos projetos de software estão dentro do prazo e do orçamento. Do resto, 52,7% são entregues com funcionalidade reduzida e 31,1% são cancelados. A principal razão para essa grande quantidade de projetos atrasados é a falta de gerenciamento adequado de riscos de software (ou seja, atividades usadas para gerenciar a possibilidade de dano ou perda) [9, 10].

Devido à importância do gerenciamento de riscos no sucesso de projetos de software, a indústria tem se interessado na área de gerenciamento de riscos de software e uma linha de trabalho e pesquisa que recebeu muita atenção recentemente é a previsão de falhas de software, em que métricas extraídas do código submetido são usadas para prever quais novos artefatos são mais suscetíveis a induzirem novos defeitos [4].

Mudanças arriscadas, identificadas pelos próprios desenvolvedores que as desenvolveram como aquelas que necessitam de mais cuidado, análise e testes envolvidos [4], podem introduzir defeitos de software, e assim também podem atrasar a entrega dos projetos e impactar negativamente a satisfação do cliente. Adicionalmente, quaisquer mudanças de software que resultem em um alto impacto no projeto são consideradas arriscadas, independentemente de apresentarem ou não erros. O risco é causado pela incerteza introduzida pelas mudanças [4].

Para mitigar estes efeitos negativos na indústria de software a utilização de uma metodologia para predição de erros de software automatizada pode ser de grande ajuda, porém deve se considerar as mudanças ao longo do tempo no contexto do software como apontado por S. Wang ***et al.*** em [11], que tipos de mudanças estão sendo realizadas e em que módulo do sistema estão sendo implementadas além das novas mudanças que são implementadas ao longo do tempo criando um ambiente online para otimizar o uso da metodologia ao longo do desenvolvimento de software.

3. Objetivos

Objetivos Gerais:

- Avaliar o impacto da consideração ou não consideração da verificação de latência no desempenho do problema de JIT-SDP, assim como verificar se falhas de software descobertas com muito atraso tendem a ser mais difíceis de serem detectadas que falhas descobertas com baixa latência na verificação.

Objetivos Específicos:

a) Identificar na literatura os problemas relacionados ao uso de um ambiente online para aprendizagem de máquina e suas especificidades. Problemas relacionados à ordem cronológica, problemas na latência na verificação e mudança no conceito.
b) Realizar diferentes experimentos com diferentes classificadores online sob diferentes condições, janelas de retreinamento e métodos de reamostragem com o objetivo de determinar o impacto da ordem cronológica e, consequentemente, da latência na verificação, uma vez que se desconsiderarmos a natureza cronológica do problema da latência na verificação, a classificação das instâncias como indutora ou não de defeitos poderia ser mais facilmente tratada em um ambiente offline.
c) Registrar e analisar, através de gráficos e tabelas de resultados, a diferença entre os diferentes experimentos para avaliar as a eficiência dos métodos testados, além dos efeitos causados pela ordem cronológica dos dados.

4. Definições Preliminares

As definições a seguir foram adaptadas do trabalho de Cabral et al. [19]

Definição 1 - **Instante do commit** (*commit time step*)

Um índice sequencial representando a ordem de chegada das alterações de software em termos de seu tempo de *commit*. Cada mudança de software é um exemplo de teste que requer uma previsão no tempo de *commit*.

Definição 2 - **Exemplo de teste**

Um exemplo de teste é uma alteração de software representada por um vetor de característica que precisa ser classificado como “limpo” ou “indutor de defeitos” Neste trabalho, adotamos como características de alteração de software propostos por Kamei et al. [3], incluindo: NS - número de subsistemas modificados; ND - número de diretórios modificados; NF - número de arquivos modificados; Entropia - distribuição do código modificado em cada arquivo; LA - linhas de código adicionadas; LD - linhas de código deletadas; LT - linhas de código em um arquivo antes da alteração; FIX - *flag* indicando se a alteração é uma correção de defeito; NDEV - número de desenvolvedores que mexeram nos arquivos; AGE- tempo médio do intervalo entre a última mudança e a

atual; NUC - número das últimas alterações exclusivas nos arquivos; EXP - desenvolvedor experiência; REXP - experiência recente do desenvolvedor; e SEXP - experiência do desenvolvedor em um subsistema. Essas características do trabalho de Kamei foram escolhidas devido ao reconhecimento do trabalho e ao fato do mesmo ser referência na área de JIT-SDP

Definição 3 - **Instante de treinamento** (training time step)

Um índice sequencial representando a ordem de chegada dos exemplos de treinamento. Cada exemplo de treinamento é usado para atualizar o classificador JIT-SDP assim que estiver disponível.

Definição 4 - **Exemplo de treinamento**

Um exemplo de treinamento é uma alteração de software representada por um vetor de característica e seu respectivo rótulo de classe. As características são as mesmas descritas na definição de exemplo de teste. O rótulo da classe pode ser limpo (representado por 0) ou indutor de defeitos (representado por 1).

Definição 5 - **Latência na verificação** (verification latency)

Latência na verificação se refere a demora na obtenção do rótulo (limpo ou indutor de defeito) de uma alteração de software. Considere uma alteração de software realizada em um certo timestamp Unix “ui”. O rótulo correspondente a esta alteração só pode ficar disponível em um timestamp Unix “uk” de maneira que (uk > ui). Isso ocorre porque, no momento do commit, não é possível determinar se a alteração de software realmente induzirá ou não

defeitos. Além disso, esta é a principal razão pela qual as alterações de software

precisam ser previstas como “limpa” ou “indutor de defeito” no JIT-SDP. Portanto, a latência na verificação é inerente ao problema JIT-SDP. Para este trabalho é utilizado o *framework* proposto por Cabral et al. [5], que produz rótulos com base nos casos abaixo onde “w” é o tempo de espera.

• Nenhum defeito foi encontrado que tenha sido induzido pela alteração de software durante w dias após seu commit - será rotulado como limpo uma vez que w dias se passaram, produzindo um exemplo de treinamento limpo.

• Um defeito é encontrado que tenha sido induzido pela alteração de software em $t < w$ dias após seu commit – será rotulado como indutor de defeito uma vez que t dias se passaram, produzindo um exemplo de treinamento indutor de defeito.

• Um defeito é encontrado que tenha sido induzido pela alteração de software em $t > w$ dias após seu commit – será primeiro rotulado como limpo uma vez que w dias se passaram e usado para produzir um exemplo de treinamento limpo e, em seguida, será rotulado como indutor de defeitos uma vez que t dias se passaram e usado para produzir um exemplo de treinamento indutor de defeito.

Definição 6 - **Desbalanceamento entre as classes**

Um problema em que o número de exemplos de treinamento de uma determinada classe é muito menor do que o número de exemplos de treinamento de outra classe é referido como um problema de desequilíbrio de classes. JIT-SDP é tipicamente um problema de desequilíbrio de classe, onde a classe indutora de defeitos é uma minoria. Se não for tratado, o desequilíbrio de classe pode fazer com que os classificadores supervalorizem a classe majoritária em detrimento da classe minoritária.

Definição 7 - **Aprendizagem online**

Considere um fluxo de dados composto de exemplos de treinamento ordenados pelo tempo em que foram produzidos. A aprendizagem online mantém um classificador que é atualizado sempre que um novo exemplo de treinamento torna-se disponível. Essa atualização pode ou não exigir acesso a exemplos de treinamento anteriores, dependendo do algoritmo de aprendizado de máquina que está sendo usado.

Definição 8 - **Conceito**

Um conceito é uma distribuição de probabilidade conjunta subjacente a um problema de aprendizado de máquina em uma determinada etapa de tempo. No JIT-SDP, isso pode ser visto como o status do processo gerador de defeitos na etapa de tempo de *commit* subjacente. Ele representa a função subjacente que captura a relação entre recursos e classes, as chances de observar exemplos de cada classe e as chances de observar cada valor de recurso

Definição 9 - **Desvio de conceito** (concept drift)

Um desvio de conceito ocorre quando o conceito muda ao longo do tempo. Existem diferentes tipos de desvio de conceito, com base no componente da distribuição de probabilidade conjunta que eles afetam:

1) Mudanças nas proporções nos exemplos de cada classe. Essas proporções são representadas pelas probabilidades anteriores das classes. Em um problema de desbalanceamento de classes, tais mudanças são referidas como evolução do desbalanceamento de classes.
2) Probabilidade evolutiva de observar diferentes valores de característica em determinada classe. Tal evolução pode ser de um ou ambos dos seguintes tipos:

   • Alterações na probabilidade de um exemplo pertencer a uma dada classe (indutora de defeitos ou limpa) considerando seus valores de característica, por exemplo, nas probabilidades posteriores. Esses desvios de conceito significam que as mudanças descritas por características que normalmente estariam associadas à classe limpa agora podem ser associadas à classe indutora de defeitos, ou vice-versa.

   • Mudanças na frequência de observação de diferentes valores de recursos, ou seja, mudanças na distribuição de probabilidade dos recursos. Esses desvios de conceito significam que os valores típicos dos recursos variam ao longo do tempo.

5. Trabalhos Relacionados

Nesta seção serão discutidas características da predição de defeito em software que estão ligadas a este trabalho, apresentando os possíveis problemas e a utilização de métodos tradicionais de aprendizagem de máquina e soluções online.

5.1 Machine Learning - Online e Offline

Neste trabalho o método de aprendizagem de máquina em foco será o Online. Aprendizado de máquina online é um método de aprendizado de máquina no qual os dados ficam disponíveis em uma ordem sequencial e é usado para atualizar o melhor preditor em intervalos regulares [7]. Este método é mais próximo a cenários reais em que

os *commits* serão analisados e ficarão disponíveis de maneira incremental, diferente de técnicas de aprendizado offline que geram o melhor preditor a partir de um conjunto de dados de uma só vez. Algoritmos de aprendizado online também são propensos a interferências. No caso deste trabalho os maiores problemas são “Concept Drift” e “Verification Latency”, muitos desses problemas necessitam de abordagens de aprendizado incremental [7].

5.2 Just-In-Time Software Defect Prediction

Os trabalhos em JIT-SDP consideram uma variedade de características e mudanças para classificar instâncias como defeituosas ou não. Essas características podem ser termos adicionados, diretório ou arquivo das mudanças, métricas de complexidade, etc. Alguns estudos como [12], [13], [14] também consideram fatores “externos” como características a se estudar. Fatores como dia, horário e desenvolvedor responsável pelo *commit* podem também influenciar na predição de defeitos. A questão do desenvolvedor foi especialmente estudada por [15], podendo se considerar taxa de bugs que ocorreram por número de commits, tempo de experiência como desenvolvedor, número de bugs reportados por instâncias de mudança no código são algumas das métricas utilizadas.

As características a serem usadas neste trabalho serão as mesmas utilizadas nos estudos conduzidos por Kamei et al. [3] em que foram usados uma variedade de fatores extraídos de commits e relatórios de bugs, que foram considerados bons indicadores de mudanças defeituosas.

5.3 Verification Latency

Verification Latency refere-se ao fato de que os rótulos dos exemplos de treinamento podem chegar muito mais tarde do que suas características de entrada. Ignorar esse atraso significa treinar modelos com dados ainda não disponíveis na prática, o que é uma séria ameaça à validade. Segundo os estudos de Tan et al. [8], ignorar a Verification Latency acarreta em resultados otimistas e irreais. Eles propuseram uma abordagem que leva Verification Latency em consideração. Ele armazena novos lotes de exemplos de treinamento ao longo do tempo e usa todos os lotes recebidos até agora para

construir um classificador. Exemplos de treinamento ficam disponíveis para criar novos batches após um tempo de espera. Este tempo de espera deve refletir o tempo que leva para alguém possa dizer com confiança se uma mudança causa ou não defeito no software. No entanto, o estudo não analisa quanto tempo normalmente leva para que os defeitos sejam encontrados, e sua abordagem proposta assume que não há classe evolução do desequilíbrio.

5.4 Concept Drift

Concept drift é uma mudança no processo em que novos commits são gerados, afetando a classificação de instâncias subsequentes [16]. Geralmente o concept drift corresponde a mudanças no processo de geração de instâncias, e pode ocorrer devido à evolução ou processo de amadurecimento de um projeto de software. Por exemplo,os primeiros commits de um projeto podem ser aqueles relacionados ao desenvolvimento da interface do usuário, após um tempo, os desenvolvedores podem mudar o foco para regras de negócios, levando a um desvio no conceito do que está sendo implementado e afetando a classificação de uma instância como causadora ou não de defeito já que cada commit em diferentes contextos possuem diferentes características. Além disso, a própria evolução no desbalanceamento entre as classes é um tipo de mudança de conceito. À medida que as alterações vão sendo efetivadas a tendência é que uma das classes se torne prevalente e se a situação não for tratada, o desbalanceamento acaba se tornando cada vez maior. Quando associado à latência na verificação, pode ocorrer que os dados de treinamento rotulados vindos do mesmo processo indutor de defeitos do software do que exemplos que estão atualmente sendo rotulados podem não estar disponíveis para uso, dificultando ainda mais o desempenho preditivo [19]. Os estudos existentes que levam em conta a cronologia revelam que o desempenho preditivo dos classificadores JIT-SDP fica pior e flutua ao longo do tempo potencialmente como resultado do concept drift exacerbado pela latência na verificação. [19]

5.5 Machine Learning e Verification Latency

Alguns trabalhos sobre o aprendizado utilizando fluxos de dados levaram a Verification Latency em consideração, no entanto, eles apenas consideram cenários

específicos de aprendizagem e não refletem totalmente um cenário realístico. Zhang et al. [17] assume que alguns exemplos de treinamento se tornam disponíveis de maneira organizada, enquanto outros nunca têm seus rótulos revelados. Dyer et al. [18] assume que exemplos de treinamento rotulados estão disponíveis apenas durante um estágio inicial de aprendizado e nenhum exemplo rotulado é fornecido posteriormente. Nenhum dos cenários de aprendizagem acima corresponde ao caso de JIT-SDP, onde qualquer exemplo poderia receber seu verdadeiro rótulo antecipadamente (se um defeito associado a ele for encontrado rapidamente), no final do tempo de espera (se o exemplo for considerado limpo) ou após o tempo de espera (se um exemplo anteriormente considerado limpo for encontrado como realmente indutor de defeitos após o tempo de espera).

Embora seja uma abordagem estado-da-arte para lidar com a evolução do desequilíbrio de classe, ela tem três problemas potenciais no contexto do JIT-SDP: Não considera a Verification Latency, assumem que ajustando a taxa de reamostragem para que as proporções de classe nos dados de treinamento se aproximem de (1:1) é suficiente para obter um desempenho preditivo equilibrado em diferentes classes e se uma das classes é mais difícil de aprender o resultado ainda será desbalanceado mesmo que a proporção seja ajeitada. Este trabalho utiliza o framework proposto por Cabral et al. [5] e ajusta os parâmetros do experimento para determinar diferentes maneiras de atacar o problema.

6. Metodologia de Pesquisa

A metodologia usada neste trabalho tem como objetivo responder às seguintes perguntas:

RQ1 - Até que ponto a ordem cronológica é importante para o treinamento em JIT-SDP? Para a qual serão utilizados os seguintes experimentos:

RQ1E1 - Treinar o modelo de maneira offline com todos os dados disponíveis, desrespeitando as restrições de tempo e de forma embaralhada usando 10-fold cross validation.

RQ1E2 - Treinar utilizando abordagem online e utilizando os dados de maneira incremental, respeitando as restrições de tempo e de maneira não embaralhada.

RQ2 - Qual o desempenho de um modelo treinado com dados recentes em commits que introduzem defeitos ocorridos há um longo período de tempo?

Para a qual serão utilizados os resultados do experimento RQ1E2.

6.1 Bases de dados e pré processamento

Para cada um dos experimentos citados acima, foram usadas 10 diferentes bases que foram separadas de um único arquivo em diferentes arquivos para o uso nos experimentos. Estas bases, são as mesmas utilizadas no trabalho de Cabral et al. [5] e foram escolhidas pelo fácil acesso e por já conterem as características relevantes apresentadas no trabalho de Kamei et al. [3]. Cada uma dessas bases de dados foram obtidas de diferentes projetos *open source* no *GitHub* e possuem um número inicial de instâncias que representam alterações realizadas ao longo do projeto de software.

TABELA I

ESTATÍSTICAS DOS PROJETOS UTILIZADOS

Base de Dados	Número de Instâncias	Número de Instâncias após processamento	Percentual instâncias defeituosas	Linguagem de programação	Período coleta de dados
brackets	17311	35890	23%	JavaScript	12/2011 - 12/2017
Broadleaf Commerce	14911	31003	17%	Java	11/2008 - 12/2017
camel	30517	63531	20%	Java	03/2007 - 12/2017
fabric8	13004	27043	20%	Java	12/2011 - 12/2017
JGroups	18317	38266	17%	Java	09/2003 - 12/2017
neutron	19451	40136	24%	Python	12/2010 - 12/2017
nova	48938	103003	25%	Python	08/2010 - 01/2018
npm	7893	16557	18%	JavaScript	09/2009 - 11/2017

spring integration	8692	19135	27%	Java	11/2007 - 01/2018
tomcat	18877	40841	28%	Java	03/2006 - 12/2017

Para simular o treinamento offline sem acesso a informações que só estariam disponíveis após um certo número, todas as instâncias marcadas como bug foram replicadas após o tempo necessário para que fosse detectada como bug. Todas as instâncias nas quais o tempo para detecção do bug fossem maior que o tempo limite de 100 dias foram replicadas e marcadas como não bug após o tempo limite como maneira de simular a falha na detecção do bug antes do limite. O diagrama abaixo exemplifica como uma instância de bug é replicada e marcada para treinamento:

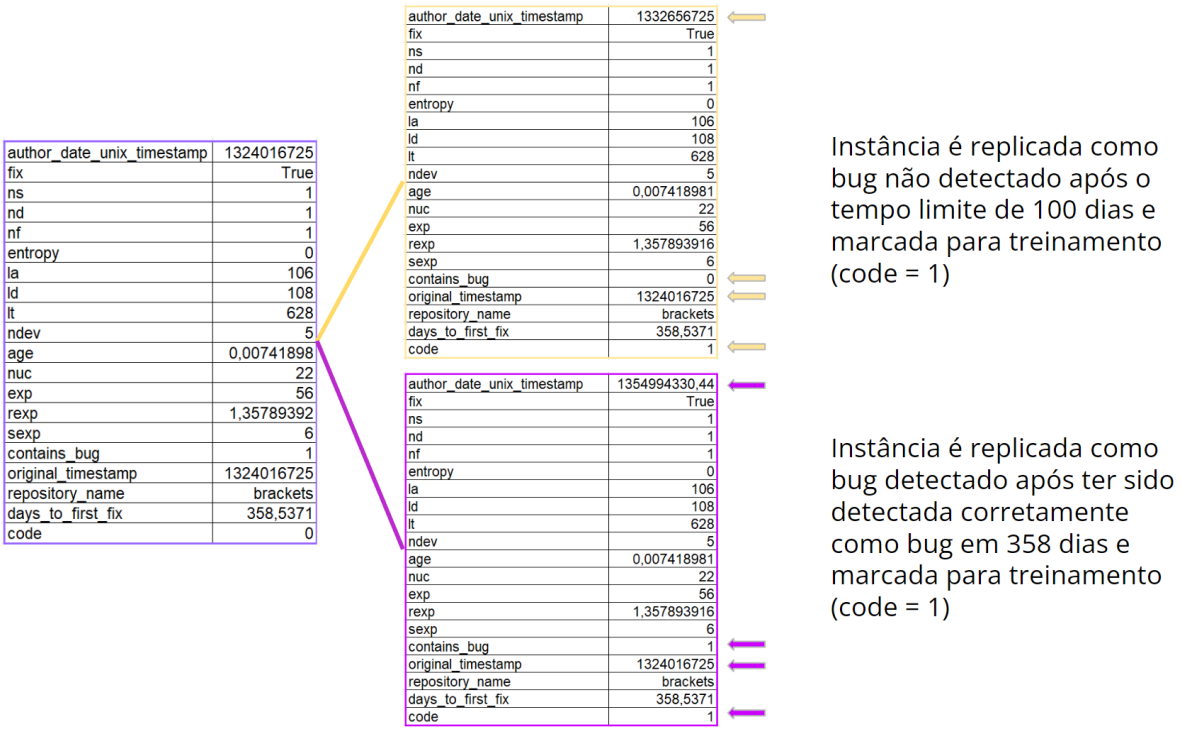

Figura 2. Processo de replicamento de instâncias.

Após realizar o processo de replicação as instâncias o conjunto marcado com code = 1 é separado para o uso em treinamento, no entanto, o dataset ainda encontra-se desbalanceado e fora de ordem. O conjunto de instâncias é então dividido para determinar a classe minoritária ou majoritária e o processo de resampling é utilizado no

conjunto determinado como minoritário. O método usado para o resample, é o método disponível no scikit learn [20] (sklearn.utils.resample) sem o uso de substituição. Após o resampling os dois conjuntos são unidos novamente e ordenados conforme o experimentos que serão usados (respeitando a ordem cronológica ou embaralhando).

6.2 Preparo de parâmetros do experimento

Após o pré processamento dos dados, as variáveis e parâmetros do experimento são inicializadas. A tabela abaixo apresenta os principais parâmetros e uma breve descrição.

TABELA II

PARÂMETROS DA BASE DE DADOS

Parâmetro	Descrição
periodicidadeTreinamento	Tempo em dias passados para um batch de dados ser usado para treinamento (100, 500 e 1000)
w (dias_a_esperar)	Número de dias a ser considerado para o verification latency (100 por padrão)
classifier	Classificador a ser usado no treinamento (HoeffdingTree e KNNClassifier)

As diferentes configurações de periodicidadeTreinamento foram escolhidas para avaliar possíveis diferenças causadas pela latência na verificação e pela mudança no conceito em diferentes etapas do experimento. Os dois classificadores utilizados foram escolhidos por serem comumente utilizados em projetos de aprendizagem de máquina, em específico, HoeffdingTree foi selecionado para a utilização em um ambiente online.

Para a análise do primeiro objetivo de pesquisa, foram considerados dois cenários: (i) levando em conta a ordem cronológica dos dados e a latência na verificação e (ii) desconsiderando ambas, ou seja, utilizando dados embaralhados.

Para o cenário (i), o experimento utilizou principalmente os parâmetros periodicidadeTreinamento e classifier, assim como oversampling e undersampling como

métodos de reamostragem dos dados para tratar o desbalanceamento entre as classes. Para o parâmetro periodicidadeTreinamento, são utilizados 3 diferentes configurações durante o experimento que determinam quantos dias o classificador terá que esperar para realizar o treinamento, sendo elas 100, 500 e 1000 dias. O intervalo na periodicidade do treino tem como objetivo analisar como o classificador se comporta em relação ao concept drift e latência na verificação considerando diferentes intervalos de tempo de treinamento. Ao final deste período, o classificador é atualizado com todas as instâncias acumuladas nas listas de instâncias. Realizando o experimento 5 vezes para cada uma das 10 bases de dados e para cada uma das permutações de configurações dos parâmetros, foram realizados um total de 300 experimentos.

Para o cenário (ii), os experimentos então são executados novamente com o dataset embaralhado, neste caso o conjunto é dividido em 10 diferentes subconjuntos para realizar um 10-fold cross-validation, as métricas são calculadas e o experimento é realizado a cada fold e os resultados finais são armazenados para cada um dos datasets.

Para a análise do segundo experimento, os mesmos parâmetros dos experimentos anteriores foram utilizados. Após a realização dos experimentos as instâncias do dataset são divididas em 10 conjuntos contendo instâncias da classe indutora de defeito inseridas nos mesmos intervalos de latência na verificação e um gráfico é gerado contabilizando o recall do classificador para cada um desses conjuntos. O intuito disto é analisar o desempenho do classificador em diferentes níveis na latência na verificação e possivelmente diferentes níveis de influência de concept drift.

6.3 Métricas de Avaliação

Para avaliar o desempenho preditivo do classificador foram empregadas as seguintes métricas de avaliação:

O presente trabalho convenciona como a classe 0 a classe de exemplos não indutores de defeito e classe 1 a classe com exemplos indutores de defeito.

Recall - Recall é a métrica que possibilita medir o desempenho preditivo do classificador para uma classe específica. O Recall pode ser calculado pela seguinte

fórmula, em que TP são os resultados classificados como verdadeiros positivos e FN os resultados classificados como falsos negativos:

$$Recall = \frac{TP}{TP + FN}$$

GMean - Gmean é a média geométrica calculada a partir do recall0 (recall da classe 0) e recall1 (recall da classe 1) calculada conforme a equação a seguir:

$$GMean = \sqrt{(recall0 \times recall1)}$$

As métricas descritas acima oferecem uma visão específica por classe (recall) e geral do desempenho do classificador (gmean) onde no geral, é esperado os valores dos recalls obtendo também valores similares para os recalls de classes diferentes. Por consequencia, um gmean alto reflete um bom desempenho geral do classficador.

7. Resultados

Para a primeira questão de pesquisa os resultados foram compilados nas seguintes tabelas que estarão disponíveis como o Anexo A na sessão de Anexos. As médias dos dados que foram guardados nas listas de recall e gmean à medida que o experimento foi sendo executado, para a comparação e análise dos resultados será utilizada a tabela com os melhores resultados que será apresentada abaixo.

A partir dos resultados apresentados nas tabelas do Anexo A, podemos inferir alguns pontos importantes. Primeiramente, de maneira geral, os resultados tendem a piorar em janelas de intervalo de periodicidade maiores, isto é ainda mais pronunciado em bases maiores como JGroups, neutron e tomcat. Devido ao número maior de instâncias é possível que uma janela maior de treinamento cause com que a "similaridade" entre duas instâncias possa ser mais acentuada por estarem mais distantes uma da outra em termos de dias e consequentemente mais suscetíveis ao concept drift. Também é facilmente observado que bases menores como npm tem uma performance muito mais baixa e menos consistente, mesmo considerando o preparo inicial da base e o uso do resampling. Adicionalmente também podemos observar que hoeffding tree se sobressai na maioria dos casos como classificador. A partir dessas observações podemos

gerar uma nova tabela considerando apenas os melhores resultados para comparar com os resultados embaralhados.

TABELA IV

MELHORES RESULTADOS DOS EXPERIMENTOS NÃO EMBARALHADOS

Base de Dados	Classificador	Periodicidade	Reamostragem	recall0	recall1	gmean
brackets	Hoefding Tree	100	undersampling	0,766	0,741	0,747
Broadleaf Commerce	Hoefding Tree	500	oversampling	0,762	0,605	0,663
camel	Hoefding Tree	100	oversampling	0,705	0,719	0,710
fabric8	Hoefding Tree	100	oversampling	0,638	0,719	0,670
Jgroups	Hoefding Tree	100	oversampling	0,703	0,595	0,638
neutron	Hoefding Tree	100	oversampling	0,776	0,923	0,844
nova	Hoefding Tree	100	oversampling	0,764	0,859	0,808
npm	knn	100	oversampling	0,583	0,613	0,595
Spring integration	Hoefding Tree	100	oversampling	0,655	0,738	0,687
tomcat	Hoefding Tree	100	oversampling	0,762	0,572	0,653

TABELA V

RESULTADOS DOS EXPERIMENTOS EMBARALHADOS UTILIZANDO 10-FOLD CROSS VALIDATION

Base de Dados	Reamostragem	recall0	recall1	gmean
brackets	oversampling	0.898	0.964	0.930
	undersampling	0.784	0.927	0.852
Broadleaf Commerce	oversampling	0.906	0.967	0.936
	undersampling	0.724	0.935	0.823
camel	oversampling	0.894	0.960	0.926
	undersampling	0.736	0.923	0.824
fabric8	oversampling	0.911	0.966	0.938

	undersampling	0.770	0.941	0.851
Jgroups	oversampling	0.944	0.973	0.959
	undersampling	0.825	0.954	0.887
neutron	oversampling	0.902	0.956	0.929
	undersampling	0.797	0.925	0.858
nova	oversampling	0.892	0.948	0.920
	undersampling	0.793	0.908	0.849
npm	oversampling	0.889	0.952	0.920
	undersampling	0.695	0.916	0.797
Spring integration	oversampling	0.859	0.918	0.888
	undersampling	0.722	0.865	0.790
tomcat	oversampling	0.849	0.911	0.880
	undersampling	0.723	0.868	0.792

A tabela VI a seguir demonstra a diferença percentual entre a utilização de dados embaralhados e dados em ordem cronológica.

TABELA VI

PERCENTUAL DE DIFERENÇA ENTRE OS RESULTADOS EMBARALHADOS E ORDEM CRONOLÓGICA

Base de Dados	Dados embaralhados	Dados em ordem cronológica	Percentual diferença
brackets	0.930	0.747	24.5%
Broadleaf Commerce	0.936	0.663	41.2%
camel	0.926	0.710	30.4%
fabric8	0.938	0.670	40%
Jgroups	0.959	0.638	50.3%
neutron	0.929	0.844	10%

nova	0.920	0.808	13.9%
npm	0.920	0.595	54.6%
Spring integration	0.888	0.687	29.3%
tomcat	0.880	0.653	34.8%
Média	0.922	0.701	32.9%

É possível observar que no geral os resultados do experimento embaralhado são melhores que o experimento não embaralhado, com uma porcentagem média de diferença de 32.9% a favor dos dados embaralhados, no entanto isto representa um otimismo exagerado que também pode ser encontrado em trabalhos na literatura, uma vez que os dados são embaralhados os efeitos causados pelo concept drift e principalmente, pela natureza cronológica da latência na verificação, passam a ser mitigados pois instâncias que poderiam fazer parte de diferentes períodos da implementação de um software agora estão distribuídas de maneira aleatória, o que afeta o treinamento do classificador. Isto não apresenta um cenário realístico onde podemos identificar bugs durante o commit pois essa estratégia utiliza dados futuros para treinamento. Neste aspecto, apesar de suas limitações e de ter apresentado um resultado inferior, o modelo utilizado nos experimentos não embaralhados representa uma maior aproximação a um cenário real.

Para a segunda questão de pesquisa foram gerados diferentes gráficos para cada projeto. Cada gráfico apresenta 10 percentis com base na quantidade de dias necessários para a rotulação dos commits que introduzem defeitos. Dessa forma, para o projeto Brackets, por exemplo, a primeira barra considera o desempenho em todos os commits com defeitos descobertos entre 0 e 0.1167 dias, a segunda barra considera todos os commits com defeitos descobertos entre 0.1167 e 0.8949 dias, etc. A altura da barra apresenta o recall da classe indutora de defeito para aquela parcela de commits. Como mencionado acima, a intuição por trás dessa análise é avaliar se commits que introduzem defeitos acometidos por um alto nível de latência na verificação (atraso na rotulação) são mais difíceis ou não de serem descobertos pelo classificador.

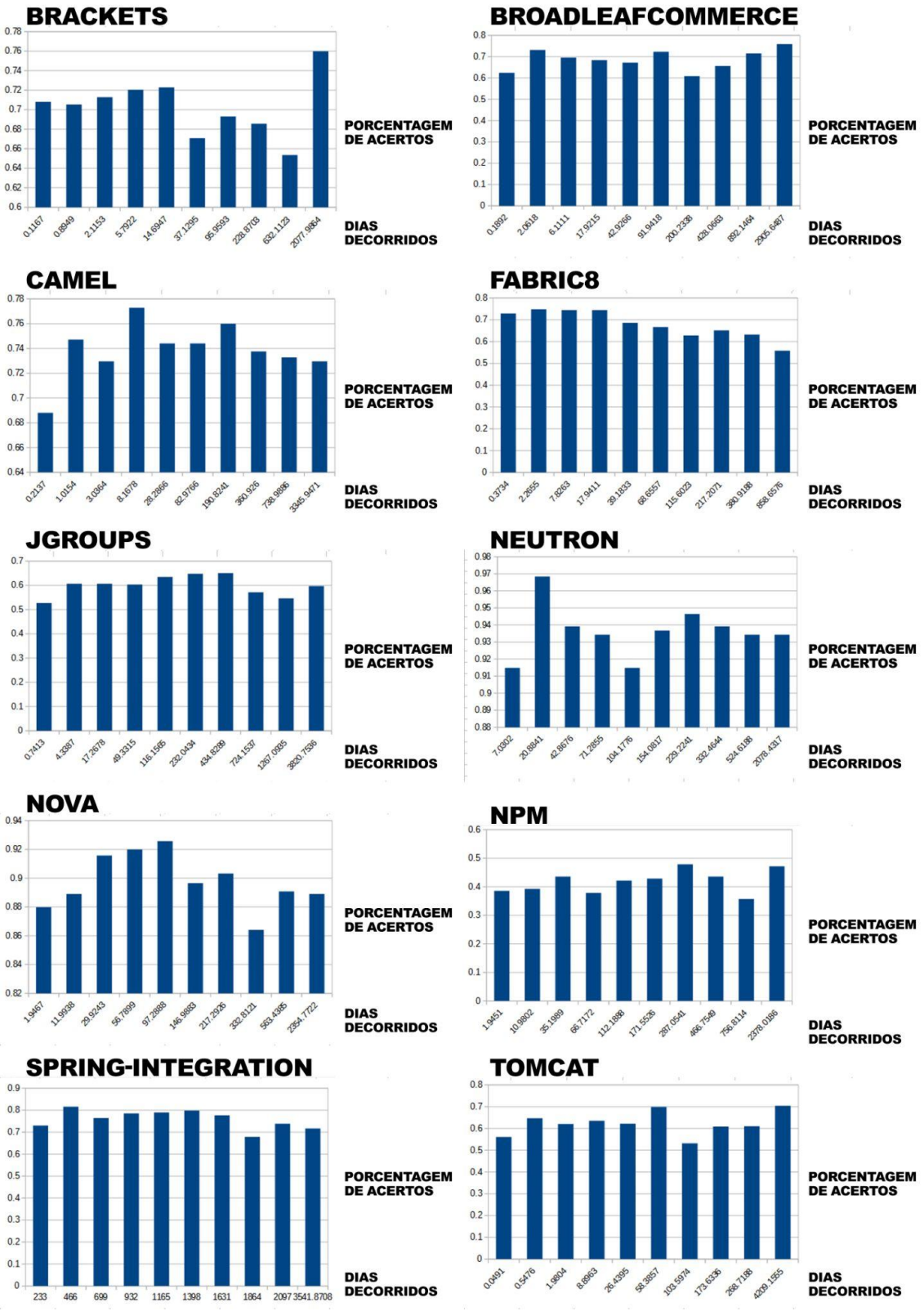

Figura 3. Recall da classe indutora de defeito por estrato de latência na verificação em dias.

Avaliando-se visualmente todos os projetos, não somos capazes de identificar um padrão de comportamento no desempenho que indique que commits indutores de defeitos

com um maior atraso de rotulação estejam relacionados a uma maior dificuldade de detecção desses por parte do classificador. Isso nos permite, indiretamente, concluir que classificadores com retreinamentos periódicos e respeitando a latência na verificação são capazes de persistir informações aprendidas em um passado relativamente distante (por exemplo, informações do momento onde o defeito foi introduzido) de maneira satisfatória.

É interessante ressaltar que o resultado apresentado na RQ2 refuta a hipótese de que o classificador teria um bom desempenho em instâncias mais recentes (que pertencem ao concento mais recentemente aprendido) em detrimento de instâncias mais antigas (ou seja, já afetadas pela mudança de conceito). Esse é um resultado particularmente interessante pois aponta para a manutenção do desempenho preditivo do classificador ao longo do tempo.

8. Conclusão e Trabalhos Futuros

O trabalho realizado teve como objetivo investigar o problema inerente a JIT-SDP da latência na verificação e a influência que o mesmo traz a diferentes abordagens considerando a ordem dos dados disponíveis para treino do classificador. Em particular, o trabalho investigou dois diferentes aspectos desse problema, sendo eles a investigação da importância na ordem cronológica e o desempenho do classificador se treinado com dados recentes em commits que introduzem defeitos ocorridos há um longo período de tempo.

No caso da [RQ1] pudemos observar que bases menores e mais desbalanceadas tendem a ser menos consistentes mesmo com as medidas de balanceamento impostas, além disso é possível observar que as bases embaralhadas, que não respeitam a latência na verificação, tendem a ter um melhor desempenho do que as bases não embaralhadas com uma diferença de, em média, 32,9% no classificador. O embaralhamento dos dados disponíveis, apesar de mitigar os problemas causados pela latência na verificação, alguns aspectos do concept drift e aumentar o rendimento do classificador, também compromete a fidelidade a um cenário verossímil em que os dados não serão constantemente embaralhados e em que a natureza cronológica do problema implica que não podemos simplesmente ignorar a latência na verificação. Ainda com estas observações, os resultados respeitando a ordem cronológica atingiram gmean acima de 60% na maioria das bases, com os melhores resultados ultrapassando 80% e apenas a menor das bases apresentando um desempenho inferior a 60%.

No estudo da [RQ2] pudemos perceber que mesmo analisando todos os projetos visualmente, não é possível determinar um padrão no desempenho ou uma correlação que mostre que commits indutores de defeitos com um maior atraso de rotulação causem uma maior dificuldade para que o classificador os classifique. A maior inferência que podemos fazer é que os classificadores possuem uma habilidade de persistir informações aprendidas não só recentemente, como também num passado distante, particularmente classificadores com retreinamentos periódicos e respeitando a latência na verificação.

Propostas de trabalhos futuros podem incluir (1) investigação de novas técnicas de reamostragem de dados para tratar o desbalanceamento entre as classes, possivelmente as adaptando para ambientes online e/ou a base de dados específicas; (2) investigação de um maior número de projetos e em um cenário mais verossímil para se chegar a uma conclusão mais robusta; (3) investigação de um maior número de classificadores com o intuito de encontrar quais classificadores são mais adequados a determinados cenários; e (4) investigação da utilização de métodos de seleção de características de forma a dar pesos diferentes a características mais importantes na solução do problema.

9. Anexos

ANEXO A.

TABELA III.1

EXPERIMENTOS NÃO EMBARALHADOS NA BASE BRACKETS

Base de Dados	Classificador	Periodicidade	Reamostragem	recall0	recall1	gmean
brackets	Hoefding Tree	100	oversampling	0,730	0,734	0,726
		500		0,718	0,731	0,714
		1000		0,731	0,682	0,688
		100	undersampling	0,698	0,626	0,647
		500		0,675	0,638	0,638
		1000		0,681	0,572	0,589
	knn	100	oversampling	0,703	0,739	0,719
		500		0,703	0,708	0,700
		1000		0,704	0,687	0,684
		100	undersampling	0,682	0,642	0,646
		500		0,683	0,544	0,580
		1000		0,690	0,582	0,601

TABELA III.2

EXPERIMENTOS NÃO EMBARALHADOS NA BASE BROADLEAFCOMMERCE

BroadleafCommerce	Hoefding Tree	100	oversampling	0,760	0,584	0,653
		500		0,762	0,605	0,663
		1000		0,758	0,578	0,641
		100	undersampling	0,774	0,478	0,590
		500		0,769	0,545	0,626
		1000		0,805	0,446	0,568
	knn	100	oversampli	0,657	0,665	0,660

		500	ng	0,658	0,639	0,643
		1000		0,664	0,622	0,63
		100	undersampling	0,654	0,652	0,650
		500		0,658	0,631	0,634
		1000		0,671	0,609	0,621

TABELA III.3

EXPERIMENTOS NÃO EMBARALHADOS NA BASE CAMEL

camel	Hoefding Tree	100	oversampling	0,705	0,719	0,710
		500		0,702	0,718	0,706
		1000		0,709	0,704	0,701
		100	undersampling	0,702	0,704	0,699
		500		0,705	0,689	0,691
		1000		0,722	0,646	0,674
	knn	100	oversampling	0,651	0,658	0,653
		500		0,661	0,647	0,648
		1000		0,664	0,626	0,633
		100	undersampling	0,596	0,611	0,602
		500		0,609	0,604	0,602
		1000		0,609	0,592	0,592

TABELA III.4

EXPERIMENTOS NÃO EMBARALHADOS NA BASE FABRIC8

fabric8	Hoefding Tree	100	oversampling	0,638	0,719	0,670
		500		0,656	0,695	0,667
		1000		0,670	0,645	0,643
		100	undersampling	0,667	0,630	0,639
		500		0,689	0,621	0,639
		1000		0,688	0,609	0,627
	knn	100	oversampling	0,656	0,655	0,654
		500		0,656	0,640	0,642
		1000		0,665	0,617	0,629
		100	undersampling	0,649	0,683	0,664
		500		0,658	0,647	0,642
		1000		0,664	0,620	0,622

TABELA III.5

EXPERIMENTOS NÃO EMBARALHADOS NA BASE JGROUPS

Jgroups	Hoefding Tree	100	oversampling	0,703	0,595	0,638
		500		0,719	0,564	0,627
		1000		0,721	0,557	0,619
		100	undersampling	0,752	0,461	0,581
		500		0,766	0,412	0,544
		1000		0,760	0,430	0,550
	knn	100	oversampling	0,625	0,583	0,603
		500		0,629	0,567	0,592
		1000		0,631	0,556	0,583
		100	undersampling	0,615	0,575	0,592
		500		0,625	0,558	0,583
		1000		0,637	0,543	0,573

TABELA III.6

EXPERIMENTOS NÃO EMBARALHADOS NA BASE NEUTRON

neutron	Hoefding Tree	100	oversampling	0,776	0,923	0,844
		500		0,781	0,899	0,831
		1000		0,780	0,895	0,826
		100	undersampling	0,733	0,902	0,809
		500		0,741	0,871	0,794
		1000		0,753	0,856	0,786
	knn	100	oversampling	0,797	0,790	0,793
		500		0,795	0,776	0,781
		1000		0,796	0,758	0,769
		100	undersampling	0,795	0,785	0,788
		500		0,795	0,773	0,778
		1000		0,794	0,748	0,759

TABELA III.7

EXPERIMENTOS NÃO EMBARALHADOS NA BASE NOVA

nova	Hoefding Tree	100	oversampling	0,764	0,859	0,808
		500		0,762	0,860	0,806
		1000		0,769	0,848	0,803
		100	undersampling	0,793	0,816	0,802
		500		0,793	0,801	0,792
		1000		0,793	0,797	0,790
	knn	100	oversampling	0,743	0,752	0,747
		500		0,744	0,746	0,743
		1000		0,743	0,741	0,739
		100	undersampling	0,743	0,755	0,748
		500		0,742	0,746	0,742
		1000		0,742	0,741	0,737

TABELA III.8

EXPERIMENTOS NÃO EMBARALHADOS NA BASE NPM

npm	Hoefding Tree	100	oversampling	0,735	0,367	0,502
		500		0,719	0,379	0,486
		1000		0,705	0,385	0,485
		100	undersampling	0,796	0,272	0,445
		500		0,829	0,249	0,427
		1000		0,819	0,264	0,414
	knn	100	oversampling	0,583	0,613	0,595
		500		0,597	0,577	0,576
		1000		0,609	0,547	0,555
		100	undersampling	0,568	0,622	0,590
		500		0,591	0,552	0,553
		1000		0,612	0,506	0,523

TABELA III.9

EXPERIMENTOS NÃO EMBARALHADOS NA BASE SPRING-INTEGRATION

Spring-integration	Hoefding Tree	100	oversampling	0,655	0,738	0,687
		500		0,681	0,691	0,672
		1000		0,704	0,623	0,644
		100	undersampling	0,701	0,613	0,645
		500		0,66	0,635	0,626
		1000		0,692	0,619	0,627
	knn	100	oversampling	0,655	0,589	0,619
		500		0,662	0,560	0,600
		1000		0,660	0,538	0,580
		100	undersampling	0,650	0,585	0,613
		500		0,651	0,556	0,589
		1000		0,674	0,526	0,573

TABELA III.10

EXPERIMENTOS NÃO EMBARALHADOS NA BASE TOMCAT

tomcat	Hoefding Tree	100	oversampling	0,762	0,572	0,653
		500		0,763	0,556	0,639
		1000		0,767	0,555	0,635
		100	undersampling	0,825	0,461	0,610
		500		0,817	0,454	0,595
		1000		0,825	0,439	0,581
	knn	100	oversampling	0,572	0,541	0,556
		500		0,583	0,529	0,551
		1000		0,579	0,527	0,544
		100	undersampling	0,570	0,536	0,551
		500		0,581	0,526	0,547
		1000		0,577	0,518	0,536

1. Referências Bibliográficas

[1] A. Mockus and D. M. Weiss, “Predicting risk of software changes,” Bell Labs Technical Journal, vol. 5, no. 2, pp. 169–180, 2000.

[2] S. McIntosh and Y. Kamei, “Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction,” IEEE Transactions on Software Engineering (TSE), vol. 44, no. 5, pp. 412–428, 2018.

[3] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, “A large-scale empirical study of just-in-time quality assurance,” IEEE Transactions on Software Engineering (TSE), vol. 39, no. 6, pp. 757–773, 2013.

[4] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, “An Industrial Study on the Risk of Software Changes,” in Proc. of the 20th Int’l Symposium on the Foundations of Software Engineering (FSE), 2012, pp. 62:1–62:11

[5] G. G. Cabral, L. L. Minku, E. Shihab and S. Mujahid, "Class Imbalance Evolution and Verification Latency in Just-in-Time Software Defect Prediction," 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 2019, pp. 666-676.

[6] More, Ajinkya, “Survey of resampling techniques for improving classification performance in unbalanced datasets.” , 2016.

[7] G. Ditzler, M. Roveri, C. Alippi, and R. Polikar, “Learning in nonstationary environments: A survey,” IEEE Computational Intelligence Magazine, vol. 10, no. 4, pp. 12–25, 2015.

[8] M. Tan, L. Tan, S. Dara, and C. Mayeux, “Online defect prediction for imbalanced data,” in Proceedings of the 37th International Conference on Software Engineering (ICSE), 2015, pp. 99–108.

[9] Barry W. Boehm. Software risk management: Principles and practices. IEEE Softw., 8(1):32–41, January 1991.

[10] F. M. Dedolph. The neglected management activity: Software risk management. Bell Labs Tech. Journal, 8(3):91–95, 2003.

[11] S. Wang, L. L. Minku, and X. Yao, “A systematic study of online class imbalance learning with concept drift,” IEEE Transactions on Neural Networks and Learning Systems (TNNLS), 2018.

[12] S. Kim, E. J. W. Jr., and Y. Zhang, “Classifying software changes: Clean or buggy?” IEEE Transactions on Software Engineering (TSE), vol. 34, no. 2, pp. 181–196, 2008.

[13] J. Sliwerski, T. Zimmermann, and A. Zeller, “When do changes induce ´ fixes?” in Proceedings of the 17th International Workshop on Mining Software Repositories, ser. MSR ’05, 2005, pp. 1–5.

[14] J. Eyolfson, L. Tan, and P. Lam, “Do time of day and developer experience affect commit bugginess?” in Proceedings of the 8th Working Conference on Mining Software Repositories (MSR), 2011, pp. 153–162.

[15] A. T. Misirli, E. Shihab, and Y. Kamei, “Studying high impact fix inducing changes,” Empirical Software Engineering Journal (EMSE), vol. 21, no. 2, pp. 605–641, 2016.

[16] Ren, Siqi & Zhu, Wen & Liao, Bo & Li, Zeng & Wang, Peng & Li, Keqin & Chen, Min & Li, Zejun. (2018). Selection-based resampling ensemble algorithm for nonstationary imbalanced stream data learning. Knowledge-Based Systems. 163. 10.1016/j.knosys.2018.09.032.

[17] P. Zhang, X. Zhu, J. Tan, and L. Guo, “Classifier and cluster ensembles for mining concept drifting data streams,” in Proceedings of the 2010 IEEE International

[18] K. B. Dyer, R. Capo, and R. Polikar, “Compose: A semi-supervised learning framework for initially labeled non-stationary streaming data,” IEEE Transactions on Neural Network

[19] G. G. Cabral and L. L. Minku, "Towards Reliable Online Just-in-Time Software Defect Prediction," in IEEE Transactions on Software Engineering, vol. 49, no. 3, pp. 1342-1358, 1 March 2023, doi: 10.1109/TSE.2022.3175789.

[20] https://scikit-learn.org/