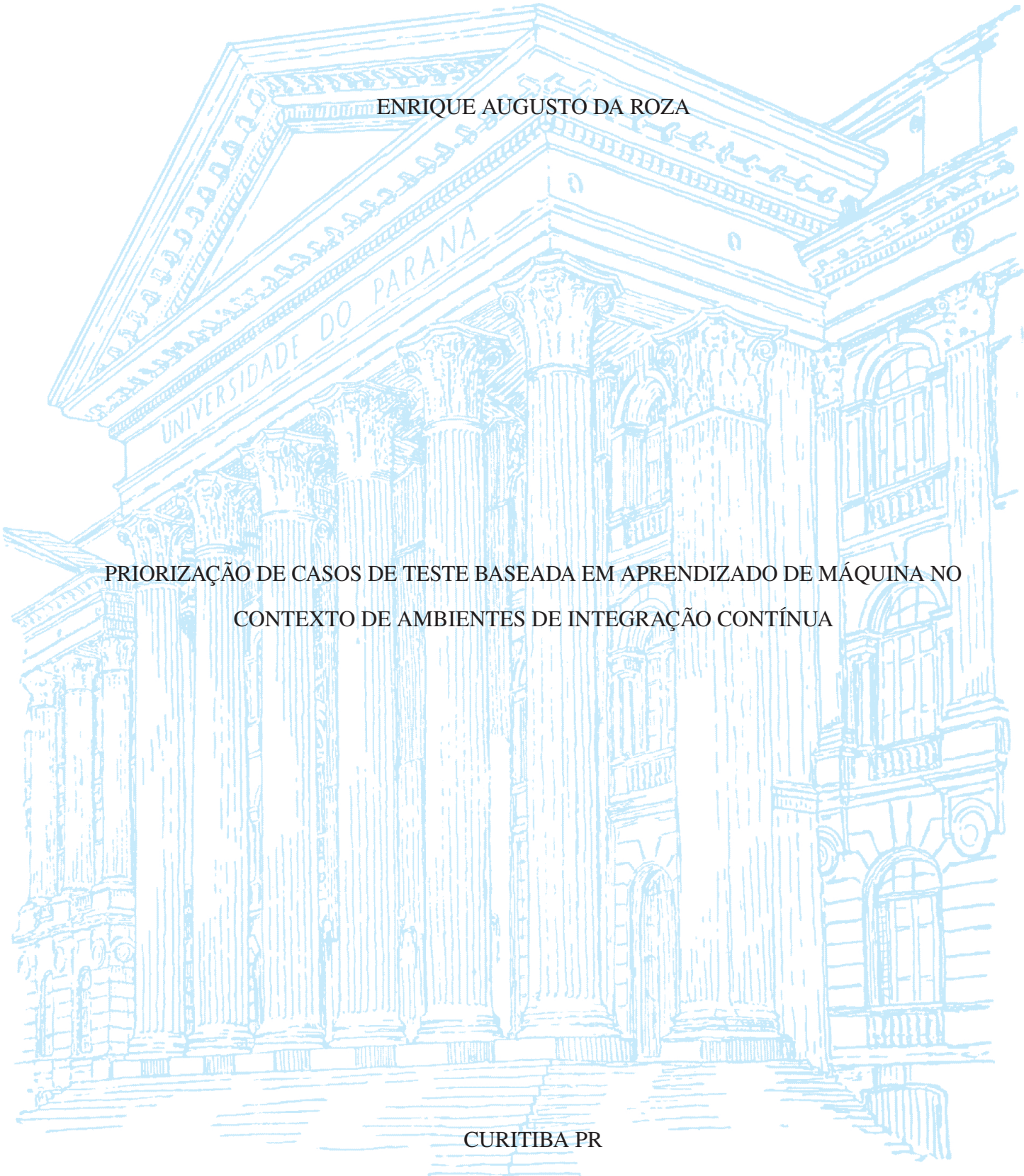
UNIVERSIDADE FEDERAL DO PARANÁ

ENRIQUE AUGUSTO DA ROZA



PRIORIZAÇÃO DE CASOS DE TESTE BASEADA EM APRENDIZADO DE MÁQUINA NO CONTEXTO DE AMBIENTES DE INTEGRAÇÃO CONTÍNUA

CURITIBA PR

2022

ENRIQUE AUGUSTO DA ROZA

PRIORIZAÇÃO DE CASOS DE TESTE BASEADA EM APRENDIZADO DE MÁQUINA NO CONTEXTO DE AMBIENTES DE INTEGRAÇÃO CONTÍNUA

Proposta de dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Silvia Regina Vergilio.

Coorientador: Jackson Antônio do Prado Lima.

CURITIBA PR

2022

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA CIÊNCIA E TECNOLOGIA

Roza, Enrique Augusto da
Priorização de casos de teste baseada em aprendizado de máquina no contexto de ambientes de integração contínua. / Enrique Augusto da Roza. – Curitiba, 2022.
1 recurso on-line : PDF.

Dissertação (Mestrado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática.

Orientadora: Silvia Regina Vergilio
Coorientador: Jackson Antônio do Prado Lima

1. Aprendizado por computador. 2. Análise de Regressão. I. Vergilio, Silvia Regina. II. Lima, Jackson Antônio do Prado. III. Universidade Federal do Paraná. Programa de pós-Graduação em Informática. IV. Título.

Bibliotecária: Roseny Rivelini Morciani CRB-9/1585

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **ENRIQUE AUGUSTO DA ROZA** intitulada: **Priorização de Casos de Teste Baseada em Aprendizado de Máquina no Contexto de Ambientes de Integração Contínua**, sob orientação da Profa. Dra. SILVIA REGINA VERGILIO, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 23 de Agosto de 2022.

Assinatura Eletrônica
25/08/2022 10:57:28.0
SILVIA REGINA VERGILIO
Presidente da Banca Examinadora

Assinatura Eletrônica
24/08/2022 13:32:13.0
AURORA TRINIDAD RAMIREZ POZO
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
24/08/2022 10:48:19.0
PLÍNIO DE SÁ LEITÃO JÚNIOR
Avaliador Externo (UNIVERSIDADE FEDERAL DE GOIÁS)

A todos que me apoiaram e acreditaram em mim, mesmo quando eu não acreditava.

AGRADECIMENTOS

Primeiramente, gostaria de agradecer minha família por sempre estarem comigo e me apoiarem sempre que me sentia incapaz de fazer algo. Quero agradecer especialmente minha namorada Flávia Eduarda Camargo por todo o apoio, carinho e paciência durante a jornada do mestrado. Por ter cuidado de mim quando eu me descuidei e me erguer quando caia.

Gostaria de agradecer minha orientadora Silvia Regina Vergilio e meu Co orientador Jackson do Prado Lima por terem me guiado durante o mestrado e pelo apoio durante todo o trabalho.

Por fim gostaria de agradecer a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo apoio financeiro com o numero de processo: 88887.501291/2020-00

RESUMO

Com a adoção de práticas de integração contínua pela maioria das empresas de software, o uso de técnicas de priorização de casos de teste (do inglês: *Test Case Prioritizaion*, *TCP*) tornou-se fundamental para reduzir o custo do teste de regressão. Abordagens promissoras são baseadas em aprendizado por reforço, que aprendem com priorizações passadas, guiada por uma função de recompensa. Tais abordagens são mais adequadas para priorização de casos de teste em ambientes de integração contínua (do inglês: *Test Case Prioritization in Continuous Integration Environments*, *TCPCI*) por se adaptarem melhor ao orçamento de teste e à volatilidade de casos de teste, isto é, o fato de que casos de teste podem ser adicionados ou removidos com o passar dos ciclos de integração contínua. Além disso, elas adotam o método de janela deslizante (do inglês: *Sliding Window*, *SW*) para avaliação dos casos de teste sem prejudicar seu desempenho em um estágio inicial. Considerando este fato, este trabalho apresenta uma abordagem para *TCPCI* que permite o uso do método SW com diferentes algoritmos de aprendizado de máquina (do inglês: *Machine Learngning*, *ML*) baseados em regressão. A hipótese é que o uso de outros algoritmos de *ML*, não baseados em aprendizado por reforço, possa levar a um melhor desempenho. Ao contrário de outras abordagens de *ML*, não é necessário retreinar o modelo para realizar a priorização e nem qualquer análise de código. Foram realizadas duas avaliações da abordagem introduzida. Primeiro, como alternativa para as abordagens baseadas em aprendizado por reforço, aplicou-se o algoritmo *Random Forest* (*RF*) e uma rede de aprendizado profundo *Long Short-Term Memory* (*LSTM*). Três orçamentos de tempo e onze sistemas foram utilizados. Os resultados mostram a aplicabilidade da abordagem considerando o tempo de priorização e o tempo entre os ciclos de integração contínua. Ambos os algoritmos levam menos que dez segundos para serem executados. O algoritmo *RF* obteve o melhor desempenho para orçamentos mais restritivos em comparação com as abordagens baseadas em aprendizado por reforço descritas na literatura, *COLEMAN* e *RETECS*. Considerando todos os sistemas e orçamentos, o *RF* atinge valores de Percentual Médio Normalizado de Falhas Detectadas (do inglês: *Normalized Average Percentage of Faults Detected*, *NAPFD*) que são os melhores ou estatisticamente equivalentes aos melhores em cerca de 72% dos casos, enquanto a rede *LSTM* em 55% deles. Na segunda avaliação, usando um subconjunto dos mesmos sistemas e os três orçamentos de tempo, a abordagem foi avaliada com outra versão do algoritmo *RF* que utiliza informações do contexto de teste, e comparada com um algoritmo contextual baseado em *Multi-Armed Bandit*. Resultados semelhantes à primeira avaliação foram obtidos. Os resultados mostram que o uso de informações de contexto não produz melhorias significativas no desempenho dos algoritmos.

Palavras-chave: Integração Continua. Aprendizado Profundo. Priorização. Teste de Regressão. Aprendizado de Máquina

ABSTRACT

With the adoption of Continous Integration practices by most software organizations, the use of Test Case Prioritization (TCP) techniques became fundamental to reduce regression testing costs. Promising approaches are based on Reinforcement Learning (RL), which learns with past prioritizations, guided by a reward function. Such approaches are more suitable for Test Case Prioritizaion in Continuous Integration Environments (TCPCI) because they adapt better to the test budget and test case volatility, that is, the fact that test cases may be added or removed over the CI cycles. Moreover, they adopt the Sliding Window (SW) method to evaluate a test case without it being hampered by its performance at a early stage. Considering this fact, this work introduces a TCPCI approach to allow the usage of the SW method with different ML regression algorithms. The hypothesis is that the use of other ML techniques, which are not based on RL, can lead to better performance. Unlike other ML approaches, it does not require retraining the model to perform the prioritization and any code analysis. We conducted two evaluations of the introduction approach. First, as an alternative for the RL approaches, we applied the Random Forest (RF) algorithm and a Long Short Term Memory (LSTM) deep learning network. We use three time budgets and eleven systems. The results show the applicability of the approach considering the prioritization time and the time between the CI cycles. Both algorithms take less than ten seconds to execute. The RF algorithm obtained the best performance for more restrictive budgets, compared to the RL approaches described in the literature, COLEMAN and RETECS. Considering all systems and budgets, RF reaches Normalized Average Percentage of Faults Detected (NAPFD) values that are the best or statistically equivalent to the best ones in around 72% of the cases, and the LSTM network in 55% of them. In the second evaluation, using a sub-set of the same systems and the three time budgets, we evaluated our approach with another version of the RF algorithm that uses test context information against a contextual Multi-Armed Bandit algorithm. Results similar to the first evaluation were obtained. The results show that the use of context information does not produce significant improvements in the performance of the algorithms.

Keywords: Continuous Integration. Deep Learning. Prioritization. Regression Testing. Machine Learning.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

APFD	*Average Percentage of Faults Detected*
APFDc	*Average Percentage of Faults Detected per Cost*
COLEMAN	*Combinatorial VOlatiLE Multi-Armed BANdit*
RETECS	*Reinforced Test Case Selection*
CRF	*Contextual Random Forest*
CMAB	*Contextul Multi-Armed Bandit*
DL	*Deep Learning*
EvE	*Exploration versus Exploitation*
FDE	*Fault-Detection Effectiveness*
FRRMAB	*Fitness-Rate-Rank based on Multi-Armed Bandit*
LSTM	*Long Short-Term Memory*
LinUCB	*Linear Upper Confidence Bound*
MAB	*Multi-Armed Bandit*
ML	*Machine Learning*
NAPFD	*Normalized Average Percentage of Faults Detected*
NTR	*Normalized Time Reduction*
PT	*Prioritization Time*
RF	*Random Forest*
RFTC	*Rank of the Failing Test Cases*
SW	*Sliding Window*
SW-LinUCB	*Sliding Window - Linear Upper Confidence Bound*
TCPCI	*Test Case Prioritization in Continuos Integration Environments*

SUMÁRIO

1 INTRODUÇÃO

Os ambientes de desenvolvimento de software têm evoluído com o passar do tempo e com o avanço tecnológico. Atualmente, há a necessidade de desenvolvimento e entregas rápidas, além de garantir o funcionamento regular e estável do produto entregue. As técnicas de desenvolvimento ágil foram propostas com o intuito de satisfazer algumas dessas necessidades [89]. O desenvolvimento ágil tem como princípios a satisfação do cliente, a entrega incremental como prioridade, simplicidade de desenvolvimento geral e comunicação ativa e constante entre os desenvolvedores e os clientes [80].

Integração contínua é uma prática do desenvolvimento ágil na qual o software em desenvolvimento é, usualmente, integrado e testado muitas vezes ao dia [80]. A cada ciclo, novas funcionalidades são acrescentadas, novos casos de teste são criados ou se tornam obsoletos. Nesse cenário tanto a execução de testes manuais do sistema, quanto a re-execução de testes automatizados previamente desenvolvidos, tornam-se potencialmente inviáveis. Por isso, reduzir o custo e tempo de execução dos testes é fundamental, pois nem sempre é possível executar todos os casos de teste disponíveis devido a restrições de recursos, de tempo e orçamento para o teste. Com este objetivo, algumas técnicas de teste de regressão são empregadas, tais como minimização, seleção e priorização dos casos de teste [84]. A minimização busca reduzir o conjunto de casos de teste eliminando testes redundantes; a seleção busca selecionar os casos de teste que satisfazem certas condições; e a priorização busca executar os casos de teste de acordo com uma ordem de prioridade. A priorização de casos de teste é a mais comumente utilizada em integração contínua [29]. Ela tem a vantagem de não descartar os casos de teste, e é o foco deste trabalho.

Para alcançar uma melhor qualidade durante a priorização, ou seja, revelar o máximo possível de defeitos em pouco tempo, são utilizadas algumas informações do sistema. Prado Lima e Vergilio [29] realizaram um mapeamento de estudos sobre priorização de casos de teste em ambientes de integração contínua. O estudo relata que a informação mais utilizada na literatura é o histórico de falhas, ou seja, as técnicas de priorização partem do princípio de que casos de teste que falharam em ciclos anteriores têm maior chance de falhar em ciclos futuros. Além disso, abordagens que levam muito tempo para executar, como abordagens baseadas em algoritmo de busca [60] são inadequadas para o contexto de ambientes de integração contínua. Abordagens baseadas em aprendizado de máquina (do inglês: *Machine Learning*, *ML*) se adaptam melhor às restrições de tempo destes ambientes.

Os algoritmos de aprendizado de máquina aprendem a reconhecer padrões e tomar decisões inteligentes com base em dados [53]. O processo para aprendizado com os dados geralmente implica em que o algoritmo generalize e construa um modelo que será utilizado no futuro para produzir saídas com novos dados. Métodos de *ML* são classificados de acordo com as características dos dados e tarefas para as quais eles são utilizados: i) aprendizado supervisionado utiliza dados rotulados; ii) aprendizado não supervisionado utiliza dados não rotulados; iii) aprendizado por reforço aprende com o ambiente de acordo com o *feedback* recebido. Estes métodos são utilizados na engenharia de software para problemas como: classificação, agrupamento e regressão. Contudo, observa-se mais recentemente um grande número de trabalhos utilizando redes de aprendizado profundo (do inglês: *Deep Learning*, *DL*) para realizar diferentes tarefas da Engenharia de Software [99]. *DL* tem origem em redes neuras convencionais, simulando o comportamento do cérebro humano, e tem se tornado uma

das principais sub-áreas de estudo de *ML*, com sucesso em várias áreas de processamento de linguagem natural, imagem, fala e processamento de áudio, dentre outras aplicações [78].

O trabalho de Bertolino et al. [10] compara duas estratégias de *ML* no ambiente de integração contínua e avalia o desempenho de diferentes algoritmos utilizando informações de contexto. A primeira estratégia avaliada, denominada *Learning-to-Rank*, utiliza aprendizado supervisionado para treinar um modelo baseado em alguns recursos de teste. O modelo é então usado para classificar conjuntos de teste em predições futuras. O problema com esta estratégia é que ela requer retreinamento do modelo, porque muitas vezes o modelo pode não ser mais representativo quando o contexto de predição mudar. A segunda estratégia avaliada, denominada *Ranking-to-Learn*, é mais adequada ao contexto dinâmico do ambiente de integração contínua. Esse tipo de estratégia em sua maioria é baseada em aprendizado por reforço. Os autores concluem que as estratégias *Ranking-to-Learn* são mais robustas em relação à volatilidade dos casos de teste, alterações de código e número de testes com falha.

Algumas abordagens do tipo *Ranking-to-Learn* para o contexto de priorização de casos de teste em ambientes de integração contínua (do inglês: *Test Case Prioritization in Continuos Integration Environments*, *TCPCI*) foram propostas utilizando aprendizado por reforço e o histórico de falhas [90, 28, 19]. Estas abordagens lidam apropriadamente com a volatilidade dos casos de teste, ou seja, o fato de eles serem adicionados e removidos ao longo dos ciclos de integração contínua, e são consideradas o estado-da-arte em *TCPCI*.

Dentre estas abordagens destacam-se *COLEMAN* [28] e *RETECS* [90] que utilizam o método de janela deslizante (do inglês: *Sliding Window*, *SW*) [27]. A abordagem *RETECS* [90] (do inglês: *Reinforced Test Case Selection*) usa uma representação de memória para delimitar quanto da informação passada é usado para aprender. A abordagem *COLEMAN* utiliza a técnica *Multi-Armed Bandit* (*MAB*) [82] que simula um jogador ganhando recompensas ao puxar o braço em uma máquina de caça-níqueis.

A *SW* ajuda a lidar adequadamente com o dilema de Exploração vs Intensificação [82] (do inglês: *Exploration versus Exploitation*, *EvE*) que diz respeito ao fato de que é necessário priorizar os casos de teste com alta probabilidade de falhar, mas também é importante incluir novos casos de teste no conjunto priorizado, caso contrário, alguns casos de teste nunca poderão ser executados devido ao orçamento de teste. Isso significa que, para tais problemas, são desejadas soluções com o melhor desempenho (Intensificação), mas também é importante garantir a diversidade, ou seja, soluções diferentes (Exploração).

Spieker et al. [90] e Chen et al. [15] relatam que um caminho promissor para futuras pesquisas na área de *TCPCI* é a utilização de redes de aprendizado profundo (do inglês: *Deep Learning*, *DL*). Utilizando o histórico de falhas, Wang et al. [103] utilizam uma rede profunda *LSTM* (do inglês: *Long Short-Term Memory*) para predição de defeitos em sistemas de aviões, demonstrando que a rede tem uma alta adaptabilidade e uma alta acurácia para o problema utilizando histórico de falhas. O trabalho de Xiao et al. [102], baseado nos resultados positivos do trabalho de Wang et al., utiliza uma rede *LSTM* para a resolução do problema de *TCPCI*, utilizando o histórico de falhas. Esse problema é tratado como um problema de séries temporais. O trabalho demonstra que a rede *LSTM* alcança bons resultados utilizando a métrica de qualidade *APFD*. Contudo, o trabalho não leva em consideração o orçamento de tempo dos ambientes de integração contínua.

1.1 MOTIVAÇÃO

No trabalho de Bertolino et al. [10] os algoritmos de *ML* apresentaram bons resultados utilizando informações do contexto de teste, tais como tempo de execução e tamanho do caso

de teste. Dentre estes algoritmos e considerando a estratégia *Ranking-to-Learn*, o algoritmo *Random Forest* foi o que apresentou melhor resultado [10]. Contudo, neste trabalho não foi utilizado o método de *SW* e não foram realizadas comparações com a abordagem *COLEMAN*.

A abordagem *COLEMAN* apresentou melhor desempenho que a abordagem *RETECS* e pode ser considerada o estado da arte em *TCPCI*. Ela utiliza aprendizado por reforço e a informação sobre o caso de teste, se este falha ou não. Mas esta abordagem não trabalha com outras informações do contexto de teste como tamanho do caso de teste, tempo de duração, etc.

A hipótese que serve de motivação para este trabalho é que a utilização de outros algoritmos de *ML*, não baseados em aprendizado por reforço, mas que aplicam o método de *SW* pode levar a um melhor desempenho no contexto de *TCPCI*. Além disso, o desempenho pode ser ainda melhor se a informação de contexto de teste for utilizada.

1.2 OBJETIVOS

Dado o contexto e as motivações descritas acima, este trabalho tem como objetivo geral introduzir uma abordagem para *TCPCI* baseada no método de *SW* a fim de permitir a utilização de diferentes algoritmos de *ML* baseados em regressão. Essa abordagem deverá ser avaliada com e sem informação de contexto, e seus resultados deverão ser avaliados considerando abordagens baseadas em aprendizado por reforço na literatura. Para satisfazer esse objetivo, esse trabalho tem os seguintes objetivos específicos:

- Avaliar a abordagem com o algoritmo *Random Forest*, que obteve o melhor resultado nos experimentos de Bertolino et al [10]. Foram implementadas duas versões deste algoritmo: uma versão chamada *Random Forest* (*RF*) que utiliza apenas a informação do resultado do caso de teste, se este falhou ou não; e a segunda versão, chamada *Contextual Random Forest* (*CRF*) que utiliza informação do contexto de teste;
- Avaliar a abordagem com um algoritmo de *DL*, cujo o uso tem sido apontado como uma oportunidade de pesquisa por muitos autores [90, 102]. Para isso foi implementada uma rede do tipo *Long Short-Term Memory* (*LSTM*) [41], que tem obtido bons resultados para modelar problemas de séries temporais [102];
- Comparar o uso de ambos algoritmos com a abordagem baseada em reforço *Combinatorial VOlatiLE Multi-Armed BANdit* (*COLEMAN*);
- Implementar uma versão da abordagem *COLEMAN* utilizando informação do contexto de software, com uma versão contextual da técnica *MAB* (*Contextul Multi-Armed Bandit*), a ser comparada com a versão contextual do algoritmo *Random Forest* (*CRF*).

Espera-se obter uma abordagem que resolva de maneira eficiente o problema de *TCPCI*, levando em consideração particularidades do ambiente. Além disso a abordagem e os resultados dos experimentos estão disponíveis em um repositório como material suplementar [86].

1.3 ORGANIZAÇÃO DO TRABALHO

O presente trabalho está organizado da seguinte maneira:

- **Capítulo 2 - Fundamentação Teórica**: apresenta os principais fundamentos sobre: Teste de Software, Integração Contínua, Aprendizado de Máquina, e Aprendizado por reforço.

- **Capítulo 3 - Trabalhos Relacionados**: apresenta trabalhos similares encontrados na literatura, que tratam de aplicações de *ML* para priorização de casos de teste.
- **Capítulo 4 - Abordagem Proposta**: apresenta a abordagem desenvolvida baseada na utilização de *ML* em conjunto com o método de *SW*. São apresentados aspectos de implementação e resultados de uma avaliação em comparação com abordagens baseadas em aprendizado por reforço presentes na literatura.
- **Capítulo 5 - Investigando o uso de informações de contexto**: apresenta resultados de uma avaliação da abordagem proposta com algoritmos de ML que utilizam informações de contexto. São apresentados alguns aspectos de implementação e uma resultados da comparação com abordagens não contextuais.
- **Capítulo 6 - Conclusão do trabalho**: Esse capítulo apresenta um resumo da dissertação, as limitações e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Nesse capítulo são apresentados conceitos relacionados a esta dissertação: Teste de software (Seção 2.1), Integração contínua (Seção 2.2), Aprendizado de Máquina (Seção 2.3), Aprendizado Profundo (Seção 2.4), Aprendizado por Reforço (Seção 2.5) e *Contextul Multi-Armed Bandit* (Seção 2.5.2).

2.1 TESTE DE SOFTWARE

O teste de software é um processo, ou uma série de processos, que visa a avaliar se o código do sistema faz o que foi projetado para fazer, e assim evitar situações não planejadas. O software deve ser previsível e consistente, sem oferecer surpresas aos usuários [73]. Dessa maneira o processo do teste tem como objetivo: descobrir situações em que o software se comporta de maneira incorreta e demonstrar ao desenvolvedor e ao cliente que o software atende aparentemente, aos seus requisitos [89]. Contudo, a realização de testes não indica que um software está livre de defeitos ou que seu comportamento estará sempre de acordo com o planejado. O teste de software pode mostrar apenas a presença de erros, e não a sua ausência [73].

Na Figura 2.1 é representado o fluxo seguido pela atividade de teste. O sistema a ser testado é considerado um sistema de caixa-preta (no qual não há referência à estrutura interna do sistema) [80]. O sistema espera um conjunto de entradas e em seguida, gera um conjunto de saídas. Os testes associados a entradas anômalas, ou seja, não previstas ou fora dos padrões, geram saídas anômalas que detectam defeitos. Essas saídas são o objeto a ser encontrado pelos testes, uma vez que elas revelam os problemas do sistema [89].

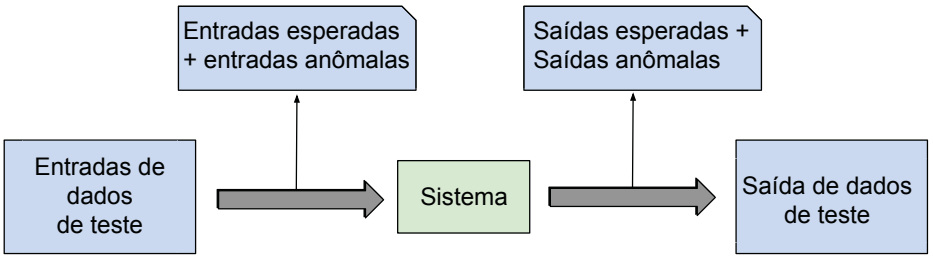


Figura 2.1: Fluxo do teste de software (adaptada de Sommerville [89]).

Com relação à qualidade dos testes, existe uma série de atividade chamadas: Validação Verificação e Teste (VV&T) [80]. Sendo verificação o processo de determinar se as saídas de uma fase do processo de desenvolvimento de software atendem aos requisitos estabelecidos na fase anterior. Enquanto, validação é o processo de avaliação de software no final do seu desenvolvimento para garantir a conformidade com o uso pretendido [2].

As atividades de VV&T podem auxiliar na economia de recursos quando aplicadas nas fases de desenvolvimento de software, permitindo a detecção de erros nas fases iniciais do processo [26]. Contudo, essas atividades têm uma longa duração e requerem recursos para serem executadas.

São definidos na literatura, alguns termos para auxiliar a compreensão do teste de software. São eles: defeito, engano, erro e falha [2]. Defeito é qualquer problema estático no código-fonte do software. O engano é uma ação humana a qual introduz um defeito. O erro é um

estado incorreto causado por um defeito. Enquanto a falha é a propagação do erro, o que gera um comportamento incorreto no software, ou seja uma saída incorreta com relação às especificações.

Ao longo do desenvolvimento do sistema, vários casos de testes são gerados buscando garantir com que o sistema siga as definições dos requisitos. Afim de avaliar se as mudanças introduzidas geram comportamentos anômalos no código, todos os testes podem ser executados buscando uma cobertura total do código fonte. [104].

No decorrer do desenvolvimento de software, novas funcionalidades são introduzidas, para garantir que essas funcionalidades estejam de acordo com o funcionamento das anteriores, o que ocasiona a execução de testes de regressão, os quais têm como objetivo garantir que não sejam introduzidos novos defeitos. Entretanto, durante a evolução do sistema, o número de casos de teste tende a crescer, tornando cara a execução do conjunto de testes e possivelmente inviabilizando seu uso. Devido a essa limitação, técnicas com o intuito de diminuir o custo necessário para a execução de testes de regressão foram desenvolvidas, destacando-se: minimização dos casos de teste, seleção dos casos de teste e priorização dos casos de teste.

A técnica de minimização tem como objetivo reduzir o tamanho do conjunto de testes eliminando casos de testes considerados redundantes. A minimização segundo Rothermel et al. [85] pode ser definida como: Dado um conjunto de casos de teste T e um conjunto de requisitos R que necessitam serem satisfeitos para provar o funcionamento correto do programa em teste, seleciona-se um sub conjunto de T' composto de casos de teste que satisfaçam os requisitos em R. A seleção de casos de teste também tem como objetivo reduzir o tamanho do conjunto de teste, porém a redução é temporária e focada em áreas do software com alterações recentes [104]. Rothermel e Harrold [84] definem a técnica de seleção como: Dado um programa P e considerando versões modificadas de P como P', encontrar um subconjunto de T, T' com o qual testar P' seguindo algum critério.

A priorização de casos de teste busca ordenar os casos de teste com o objetivo de maximizar alguma propriedade ou algum objetivo, como por exemplo produzir uma falha o mais rápido possível. A priorização não faz uma seleção dos casos de teste e assume que todos os testes devem ser executados na ordem em que a priorização produzir [104]. A priorização pode ser definida como: Dados um conjunto de teste T, um conjunto de permutações de T PT, e uma função a qual transforma cada elemento de PT em um número real, encontrar $T' \in PT$ no qual $(\forall T'')\ (T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.

Para a avaliação da performance da priorização, na literatura são utilizadas as métricas [28, 29]: *Average Percentage of Faults Detected* (*APFD*), *Average Percentage of Faults Detected per Cost* (*APFDc*) e *Normalized Average Percentage of Faults Detected* (*NAPFD*), enquanto *Rank of the Failing Test Cases* (*RFTC*), *Normalized Time Reduction* (*NTR*) e Tempo de Priorização (do inglês: *Prioritization Time*, *PT*) para avaliação em relação ao tempo utilizado para a priorização dos casos de teste.

A métrica *APFD* tem como objetivo indicar a rapidez de um conjunto de casos de teste para encontrar falhas presentes em uma aplicação em teste, seu valor é calculado de acordo com a média ponderada de percentuais de falhas detectadas.

A métrica *NAPFD* considera a relação entre falhas detectáveis pelos casos de teste disponíveis e falhas detectadas no conjunto. Esta métrica é utilizada considerando que em um ambiente de integração contínua, nem todos os casos de teste são executados. Portanto, algumas falhas podem não ser detectadas. No caso em que todas as falhas são detectadas, a métrica *NAPFD* é equivalente à métrica *APFD*.

$$NAPFD(T') = p - \frac{\sum_1^n rank(T'_i)}{m \text{ x } n} + \frac{p}{2n} \tag{2.1}$$

sendo m o número de falhas detectadas por todos os casos de teste, $rank(T'_i)$ é a posição de T'_i no conjunto T', caso T'_i não revele falhas, ele valerá zero. A variável n é o número de casos de teste presentes em T'. Por fim, p é o número de falhas detectadas por T', dividido por m.

A métrica *APFDc* assume que os casos de teste têm custos diferentes, ou seja, alguns podem ser mais caros de executar do que outros devido a fatores como a gravidade da falha ou tempo de execução. Os casos de teste geralmente são priorizados até que um custo máximo seja atingido, que seja viável de ser executado. Além disso, se a gravidade da falha e os custos do caso de teste forem iguais, o *APFDc* pode ser usado para calcular o valor do *APFD*.

$$APFDc(T'_t) = \frac{\sum_{i=1}^{m}(\sum_{j=TF_i}^{n} c_j - 0.5c_{TF_i})}{\sum_{j=1}^{n} c_j \times m} \tag{2.2}$$

em que c_i é o custo do caso de teste T_i, e TF_i é o primeiro caso de teste do conjunto priorizado T' que revela a falha i.

Considerando o contexto de integração contínua, o algoritmo deve ter uma execução que não supere o orçamento de recursos disponíveis. Para avaliar este aspecto, as métricas *RFTC* e *NTR* foram propostas [28]. A métrica *RFTC* tem como objetivo a avaliação da eficiência do conjunto de testes, atribuindo valores menores para casos de teste que apresentam uma detecção de falhas em um curto período de tempo. Também para avaliação de tempo, será utilizado a métrica *NTR*, a qual avalia o tempo necessário para que o primeiro caso de teste falhe r_t e o tempo total necessário para a execução de todos os casos de testes $\hat{r}_t$. Para a execução dessa métrica, serão avaliados somente *commits* que contêm casos de testes que falharam, representados por CI^{fail}.

$$NTR(\mathcal{A}) = \frac{\sum_{t=1}^{CI^{fail}}(\hat{r}_t - r_t)}{\sum_{t=1}^{CI^{fail}}(\hat{r}_t)} \tag{2.3}$$

A métrica Tempo de Priorização (*PT*) também se refere à avaliação de tempo, consistindo no tempo que o algoritmo leva para realizar a priorização. Enquanto as métricas *NAPFD* e *APFDc* consideram a ordem dos casos de testes reprovados. Para uma análise simples sobre a taxa de detecção de falhas sob uma restrição de tempo é utilizada Eficácia da detecção de falhas [29] (do inglês: *Fault-detection effectiveness*, *FDE*). Nesta métrica, considera-se que cada teste revela uma falha sem repetição, assim FDE mede a razão entre o número de falhas detectadas pelo conjunto de teste executado para um orçamento de tempo em relação ao seu conjunto de testes original (não reduzido).

2.2 AMBIENTE DE INTEGRAÇÃO CONTÍNUA

No passado, desenvolvedores trabalhavam separadamente por um longo período e submetiam suas alterações nos códigos-fonte para um único *branch*. Essa prática causava acúmulo de erros não resolvidos e diminuía a velocidade de atualizações.

Atualmente desenvolvedores têm a necessidade de fazer entregas rápidas para maior aproveitamento de oportunidades e respostas à competitividade [89]. A IBM introduziu o desenvolvimento incremental na década de 1980 [71]. No final da década de 1990 a ideia de desenvolvimento ágil tornou-se popular com o desenvolvimento de novas abordagens como: Método de Desenvolvimento Dinâmico do Sistema (do inglês: *Dynamic System Development Method*, DSDM) [91], Scrum [87] e Programação Extrema (do inglês: *Extreme Programming*) [8].

Com a popularização dessas técnicas, as práticas de engenharia de software contínua, tais como integração contínua, implantação contínua e entrega contínua, começaram a ser adotadas

por diversas organizações possibilitando uma integração frequente e testes rápidos de alterações de código [29]. A Figura 2.2 mostra a relação dessas práticas.

Nos ambientes de integração contínua, um desenvolvedor ou grupo de desenvolvedores realizam modificações no código fonte, as quais são enviadas para um servidor ou repositório. Em seguida, o servidor ou ambiente de integração contínua busca as mudanças e atualiza seu código local. O ambiente então, segue para fazer a construção (do inglês: *build*) do código, testes e em seguida prepara os resultados. Os resultados podem ser erros, avisos (do inglês: *warnings*) ou sucesso. Por fim, o ambiente notifica os desenvolvedores.

Em ambientes com integração contínua, o *build* do software bem como os testes são automatizados [56], auxiliando a escalabilidade e velocidade de desenvolvimento, permitindo com que desenvolvedores trabalhem paralelamente em diferentes recursos de maneira independente. Em ambientes com entrega contínua, uma versão com as alterações a serem enviadas para o servidor de produção (disponível ao usuário) é enviada para execução de um teste de aceitação. Nesse contexto, essa versão deve estar pronta para ser entregue para a produção em qualquer momento [29]. No entanto, em ambientes de implantação contínua, a distribuição e implementação da versão no servidor de produção são realizadas de maneira automática.

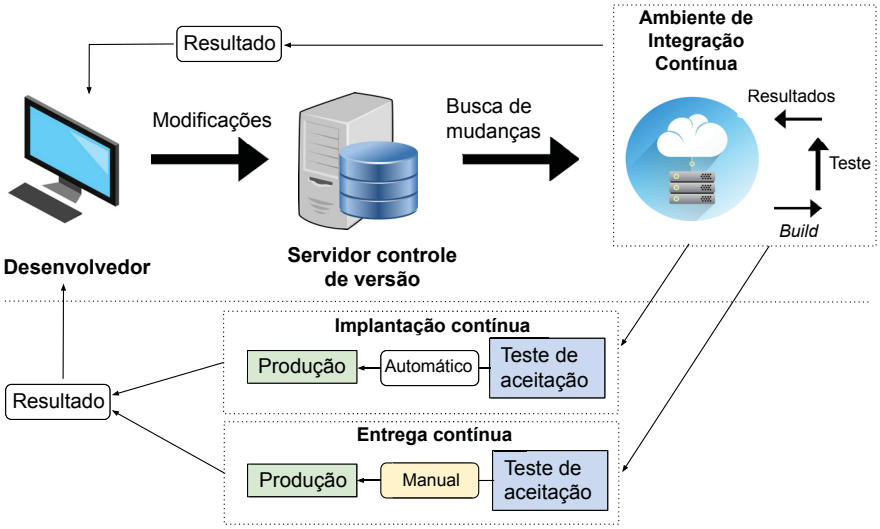


Figura 2.2: Fluxo de integração continua em ambientes de desenvolvimento de software (adaptada de Prado Lima e Vergilio [29]).

Nesse contexto, os testes de regressão são fundamentais, porém apresentam a desvantagem de serem muito custosos. Além disso, deve-se considerar que os ambientes de integração contínua precisam trabalhar com um limite de recursos para realização de tais testes (do inglês: *test budget*). Para minimizar os recursos utilizados, uma técnica bastante utilizada e mais popular é a priorização de casos de teste [29]. Contudo em ambientes de integração contínua existe uma grande volatilidade nos casos de teste, (criação, alteração e remoção de casos de testes com o passar do tempo). Para solucionar esse problema algoritmos de aprendizado de máquina têm sido propostos [28].

2.3 APRENDIZADO DE MÁQUINA

Com o avanço tecnológico, as técnicas de Aprendizado de Máquina (do inglês: *Machine Learning*, *ML*) têm ganhado destaque por serem aplicáveis em várias áreas como a classificação ou reconhecimento de imagens [61]. Utilizando essas técnicas, computadores conseguem realizar ações sem a necessidade de serem escritos para determinada ação.

Em consideração às áreas de estudo dentro da Ciência da Computação, a área que estuda maneiras de computadores replicarem a inteligência humana é a Inteligencia Artificial. A área de Aprendizado de Máquina é responsável por buscar técnicas ou tecnologias que permitem o computador a aprender sem a programação específica para tal função. Por fim, a área de estudo de aprendizado profundo (do inglês: *Deep Learning*, *DL*) concentra seus estudos em aprendizados de máquina que utilizam redes neurais de várias camadas [53]. A Figura 2.3 ilustra a organização dessas áreas de estudo.

Quanto aos problemas tratados por *ML*, eles podem ser divididos em três tipos são eles: problemas de regressão, classificação e agrupamento [53]. Os problemas de regressão têm como saída geralmente valores com expectativas não discretas, ou seja, são valores dentro de alguma faixa de valores. Em problemas de classificação, o algoritmo seleciona entre valores discretos, ou seja, são valores contáveis, com características mensuráveis. Por fim, os problemas de agrupamento ou organização de grupos (do inglês: *clusters*), tem como base a aproximação de valores com características semelhantes entre si.

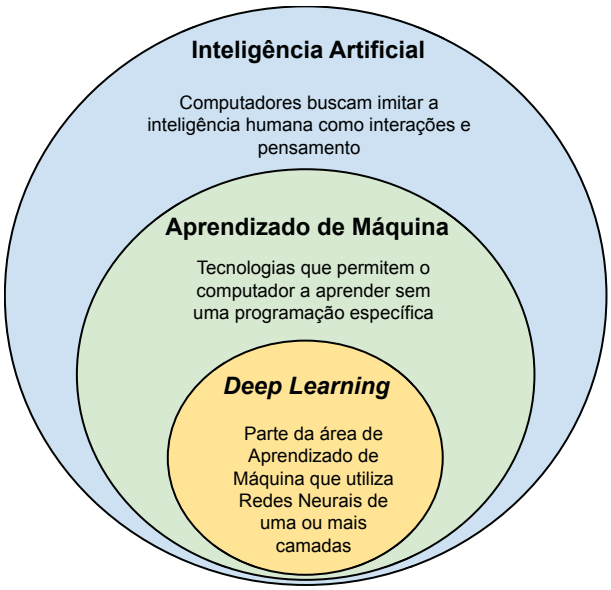


Figura 2.3: Diferenças entre Inteligência Artificial, Aprendizado de Máquina e Aprendizado Profundo.

O aprendizado de máquina pode ser dividido em três grandes áreas de acordo com o tipo de entrada, sendo eles: aprendizagem supervisionada, aprendizagem não supervisionada e aprendizagem por reforço. O aprendizado supervisionado é baseado nas entradas que já contem um "rótulo" dentre a própria entrada, um exemplo seria a classificação de imagens, onde a definição da imagem (rótulo) está dentre os dados de entrada. Esse tipo de entrada geralmente está relacionado a problemas em que o foco é a predição ou classificação dos valores de entrada.

Enquanto o aprendizado não supervisionado não contém o rótulo do dado de entrada, ou seja, o algoritmo não supervisionado em sua maioria aprende por similaridade entre os dados. Uma utilização para aprendizado não supervisionado é encontrar propriedades interessantes ou de destaque em uma base de dados não rotulada popularmente por meio de *clusters* [53].

O aprendizado por reforço tem o seu aprendizado de maneira diferente dos apresentados acima, pois ele baseia-se em estímulos do ambiente e em recompensas ou punições resultantes de "experimentações" [53]. O aprendizado por reforço em sua maioria tem como objetivo a maximização das recompensas fornecidas pelo ambiente o qual ele está inserido, buscando um "caminho ótimo". Em alguns casos, os algoritmos utilizam Processos de Markov [76] como auxilio para distribuição de recompensas acumuladas [74].

Dentre os algoritmos utilizados em *ML*, as árvores de decisão são especialmente importantes para a explicação do algoritmo utilizado no presente trabalho. Elas têm como inspiração a árvore natural porém vista ao contrário, ou seja, sua raiz é seu início está em sua maioria representada na parte superior, em seguida são os nós os quais podem ser divididos em: nós de decisão ou nós folhas. Os nós de decisão contêm, como o nome indica, critérios que servem para a decisão dos caminhos tomados para alcançar os nós folha. Os nós folha por sua vez contém os valores finais da árvore. O caminho realizado pelo algoritmo árvore de decisão é a saída por meio da raiz, seguindo os nós de decisão, obedecendo os critérios impostos por eles com o intuito de chegar a um nó folha.

A Figura 2.4 apresenta esse ciclo tendo inicio no nó raiz, como o primeiro nó da árvore. Seguido o caminho da esquerda, o algoritmo segue para um nó de decisão. O nó de decisão é ilustrado com o critério de decisão $X > 2,3$. Por fim é realizado mais uma seleção à esquerda, finalizado em um nó folha gerando um caminho para a determinada árvore. Esse caminho é responsável por determinar a predição realizada pelo algoritmo.

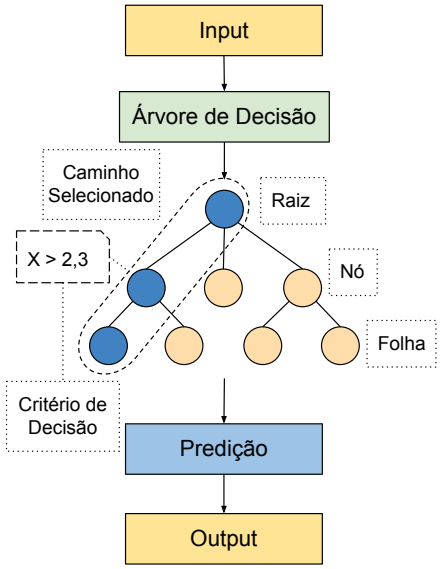


Figura 2.4: Imagem representativa do algoritmo de Árvore de Decisão.

A popularidade das árvores de decisão pode ser justificada devido a sua simplicidade e quatro vantagens quando comparadas com outros algoritmos [46]: Primeiramente, as árvores de decisão são não paramétricas e, portanto, o modelo pode aprender tarefas complexas caso dados suficientes sejam fornecidos. A segunda vantagem é permitir como entradas vetores heterogêneos, ou seja, com diferentes tipos de dados como por exemplo números e letras. A terceira vantagem é sua robustez com relação a ruídos, dados irrelevantes e atributos errados. Por fim, a quarta vantagem é o uso das árvores de decisão como parte fundamental ou de apoio para modelos de predição considerados estados da arte, como por exemplo o algoritmo *Random Forest* [11], técnicas de *boosting* como *eXtreme Gradient Boosting* (*XGBoost*) [16], dentre outros.

Técnicas ou algoritmos de *Ensable* utilizam uma combinação de múltiplos algoritmos simples, chamados modelos base, em um único modelo de predição fazendo com que o modelo produza múltiplas saídas internamente utilizando somente uma entrada [46]. Como técnicas de *Ensable* utilizam uma combinação de vários algoritmos para sua predição, a variância entre os resultados tende a ser menor do que múltiplas predições individuais, além de melhorar a performance e qualidade de predição do modelo utilizado.

Essa ideia é popularmente relacionada ao teorema do júri de Condorcet [22] do campo da ciência política [46]. O teorema afirma que se um grupo de eleitores independentes que têm

uma maior probabilidade de estarem corretos do que errados, deseja chegar a uma decisão por maioria de votos, então convidar mais eleitores aumenta a probabilidade da decisão estar correta. Computacionalmente, dado que algoritmos baseados em modelos tem uma maior acurácia com relação aos aleatórios, e que os erros são distribuídos de forma independente, um algoritmo de *ensamble*, em média, supera a acurácia de seus modelos constituintes [38]. Este conceito pode ser analisada utilizando a ideia de *Bias–variance trade-off*.

O *Bias–variance trade-off* decompõe o erro cometido por um modelo de predição em três partes: *bias*, vairância, (do inglês: *variance*) e um ruído não redutível [46]. O *bias* de um modelo está relacionado a erros sistemáticos, devido a suposições inerentes sobre o limite de decisão que um classificador produz. Por exemplo, um modelo linear tem alta chance de não capturar limites de decisão não lineares e, devido a isso, tem um *bias* inerente. Um modelo com um alto *bias* portanto, realizará previsões consistentemente incorretas. Por outro lado, nos casos em que variações na escolha dos dados de treinamento levam a diferentes modelos, os modelos induzidos farão previsões inconsistentes para a mesma instância de teste sobre diferentes hipóteses e, portanto, são variáveis. Dessa forma, a compensação *bias–variance* fornece uma ferramenta útil para diagnosticar e entender o desempenho de predição de modelos.

Um princípio por trás dos métodos de *ensemble* é introduzir perturbações aleatórias no procedimento de aprendizado e, assim, gerar vários modelos ligeiramente diferentes a partir de uma única entrada para treinamento [46]. As previsões desses modelos são então combinadas para formar a previsão do conjunto, que no qual é esperada uma redução da variância, sem aumentar o *bias*. Muitas estratégias para construir *ensembles* foram desenvolvidas e estudadas para aumentar o desempenho de predição, reduzindo o *bias*, a variância ou ambos. Geralmente, os métodos introduzem explicitamente aleatoriedade no algoritmo de aprendizado ou exploram versões aleatórias da entrada em cada execução do algoritmo de aprendizado para produzir modelos preditivos diversificados. Como resultado, a variância é introduzida tanto na entrada quanto na randomização do modelo, resultando em modelos base mais variáveis e, dependendo da quantidade de aleatoriedade, um *bias* aumentado.

O algoritmo *Random Forest*, é um *ensamble* de Árvores de Decisão, ou seja, é uma combinação de múltiplas Árvores de Decisão as quais realizam as predições e em seguida, seus resultados são comparados entre si e o resultado majoritário ou médio é escolhido como resultado do *RF* [62]. As árvores geradas internamente pelo *RF* podem ser diferentes entre si, tomando caminhos totalmente diferentes.

A Figura 2.5 mostra de maneira geral o funcionamento desse algoritmo, tendo início com a definição da quantidade de Árvores de Decisão geradas pelo *RF*. No caso do exemplo, são três árvores. O passo seguinte é a execução das três Árvores de Decisão de maneira independente. Em seguida, cada árvore seleciona um caminho de decisão, no caso do exemplo, esses caminhos foram diferentes entre as três árvores, resultando em predições distintas. Por fim o algoritmo *RF* faz uma média das predições realizadas por suas Árvores de Decisão e utiliza essa média como saída.

Desde a sua criação, variantes do algoritmo *RF* têm sido investigadas para melhorar o desempenho de predição [46]. Para reduzir o custo computacional, Cutler e Zhao [23] propõem uma variante de *RF*, as árvores perfeitamente aleatórias (do inglês: *Perfectly Random Tree Ensemble*, *PERT*), no qual as árvores são totalmente crescidas sem levar em conta a variável alvo, usando atributos aleatórios e pontos de divisão numéricos. Uma investigação empírica mostra que o *PERT* funciona de forma semelhante à formulação original proposta por Breieman [11] em termos de desempenho de predição, sendo significativamente mais rápido [46]. Relacionadas a conjuntos de *PERT*, Geurts et al. [33] introduziram árvores extremamente aleatórias (do inglês: *Extremely Randomized Trees*, *ETs*). As *ET*s, trabalham de maneira semelhante ao *PERT*,

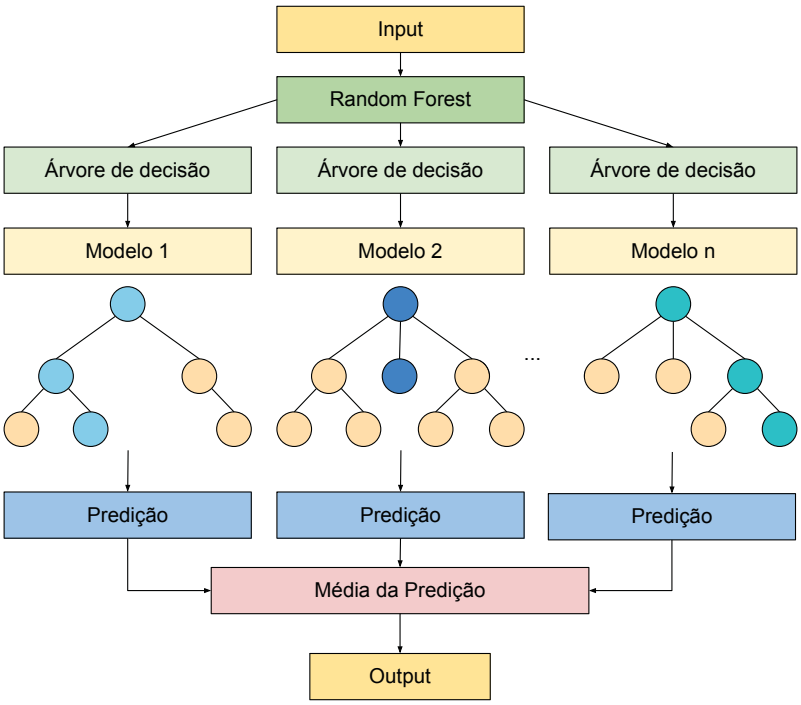


Figura 2.5: Imagem representativa do algoritmo *Random Forest*.

combinando seleção aleatória de atributos com limites de discretização extraídos aleatoriamente de uma distribuição uniforme. A versão *ET* é a versão utilizada no trabalho.

2.4 APRENDIZADO PROFUNDO

Os modelos de *Deep Learning* (*DL*) necessitam de uma grande quantidade de dados e um alto poder computacional para serem treinados [61]. Os dados podem vir de coletas de serviços de pesquisa e de redes sociais por exemplo [106]. Para auxílio no poder computacional necessário são utilizadas GPUs (do inglês: *Graphics Processing Unit*). GPUs são *hardwares* desenvolvidos para processamento de imagens e vídeos. Devido a isso, as GPUs possuem um alto desempenho para cálculos matemáticos. A utilização de *frameworks* que utilizam GPU para aumento na velocidade dos cálculos reduz o tempo de processamento, em alguns casos, de semanas para dias [106].

Diferentemente do aprendizado tradicional, em *DL* o aprendizado acontece por meio da separação de características dos dados, utilizando-se pesos internos na rede, permitindo uma maneira autônoma de treinamento. Enquanto algoritmos de *ML* tradicional requerem um tratamento dos dados para o aprendizado, ou seja, eles necessitam de dados rotulados para seu treinamento [48]. A Figura 2.6 ilustra essa diferença. Outra diferença das redes de *DL* é o uso de neurônios, sendo eles, nós responsáveis pelo controle das informações que transitam pela rede, funcionando de maneira similar aos neurônios humanos.

Na literatura existem múltiplos algoritmos de *DL* [78] como: redes neurais recursivas [88, 35], redes neurais recorrentes [43], redes neurais convolucionais [47, 51], redes neurais de crença profunda [40], e *Autoencoder* variacional [50]. As redes neurais recorrentes são utilizadas com frequência nas áreas de processamento de linguagem natural e processamento de fala [19]. Elas têm seu diferencial por utilizarem informações sequenciais, sendo adequadas para problemas em que sequências de dados são aproveitadas para o aprendizado [78]. Portanto, uma rede neural

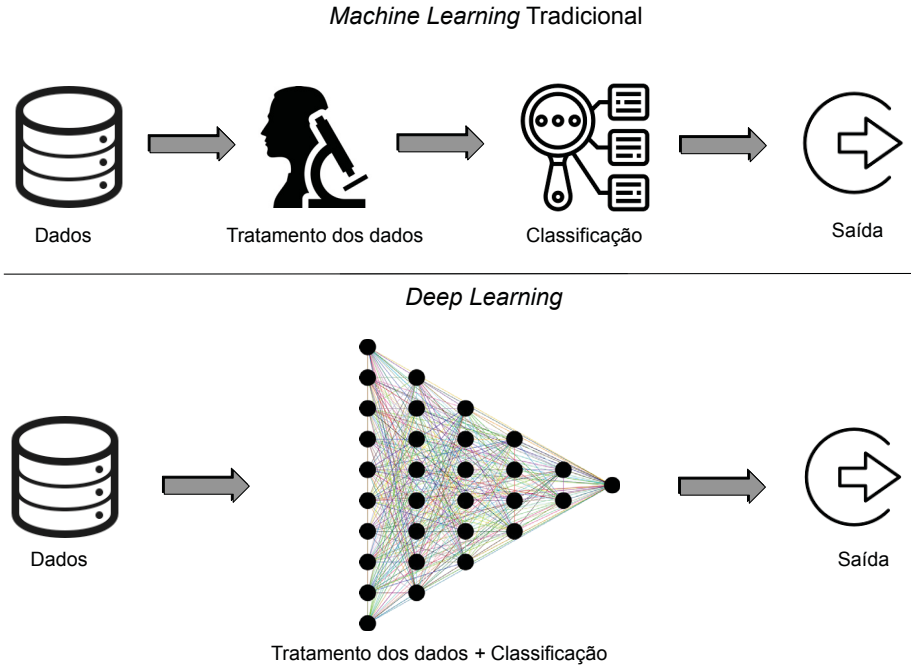


Figura 2.6: *Deep Learning* vs *Machine Learning*.

recorrente pode ser vista como unidades de "memória de curto prazo" que incluem a camada de entrada x, a camada oculta (estado) s e a camada de saída y [78].

Um dos principais problemas com as redes neurais recorrentes é a sua sensibilidade a explosões no gradiente ou desaparecimento do gradiente, ou seja, os gradientes aumentam ou diminuem muito devido a várias multiplicações durante a fase de treinamento [34], causando um esquecimento das entradas iniciais para aprendizado de novas entradas.

As redes *Long Short-Term Memory* (*LSTM*) são utilizadas para tratar esse problema. A estrutura dessa rede apresenta blocos de memória nas suas conexões recorrentes e cada bloco armazena o estado no tempo atual no qual a rede está, além de apresentar unidades para controle do fluxo de informações (*gates*) [78]. Considerando x_t como as entradas da célula, c_t como o estado da célula (*cell state*) e h_t como o estado oculto (*hidden state*). A Figura 2.7 ilustra a diferença entre as redes neurais recorrentes e as *LSTM*s.

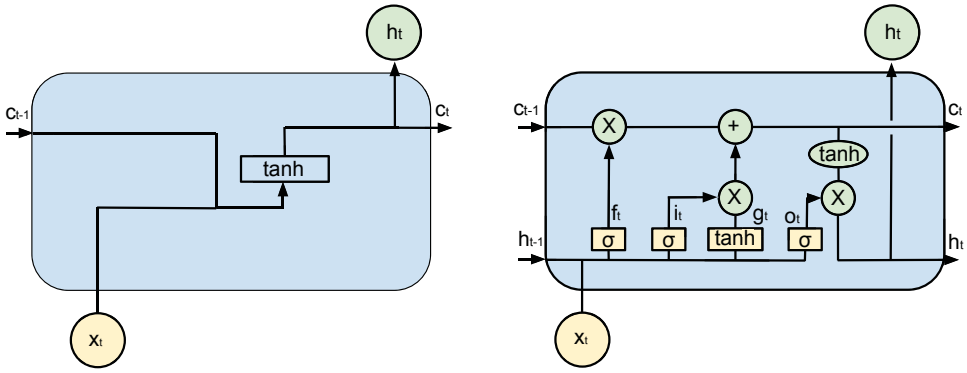


Figura 2.7: Comparação entre as unidades das redes neurais recorrentes e *LSTM* (adaptada de Mamandipoor et al. [64]).

O portão de esquecimento (*forget gate*), representado por f_t na Figura 2.7, é responsável por controlar quais informações de camadas anteriores são relevantes o suficiente para serem

transferidas para as camadas subsequentes. Considerando φ como função de ativação sigmoide da rede, H como a quantidade de neurônios e X a quantidade de entradas da rede, a Equação 2.4 demonstra matematicamente o funcionamento do portão.

$$f_t = \varphi(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \tag{2.4}$$

em que x_t é a entrada da célula no tempo t, $h_{(t-1)}$ é o valor do *hidden state* no instante de tempo anterior ao avaliado, $W_{if} \in \mathbf{R}^{HxX}$, é o peso da entrada do portão de esquecimento e $W_{hf} \in \mathbf{R}^{HxH}$ é o peso da entrada do estado oculto, sendo if *input forget* e hf *hidden forget*. A variável $b_{if} \in \mathbf{R}^X$ é o *bias* da camada de entrada, enquanto $b_{hf} \in \mathbf{R}^H$ é o bias da camada de esquecimento, por fim f_t é a saída do *forget gate*.

O portão de entrada (*input gate*), representado por i_t e o *cell gate* representado por g_t na Figura 2.7, são responsáveis pela avaliação de novas informações presentes nas entradas da *LSTM*, integrando informações relevantes à rede. A Equação 2.5 demonstra de maneira matemática seu funcionamento.

$$g_t = tanh(W_{ig}x_t) + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}) \tag{2.5}$$

no qual $tanh$ trata-se da Tangente Hiperbólica, $W_{ig} \in \mathbf{R}^H xX$ é o peso da entrada do *cell gate* (*input cell*, ig), $W_{hg} \in \mathbf{R}^H xH$ é o peso da entrada do estado oculto (*hidden cell*, hg), $b_{ig} \in \mathbf{R}X$ é o bias da entrada do *cell gate* e $b_{hg} \in \mathbf{R}^H$ é o bias para a entrada do estado oculto.

$$i_t = \varphi(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \tag{2.6}$$

em que $W_{ii} \in \mathbf{R}^{HxX}$, é o peso do portão de entrada (*input input*), $W_{hi} \in \mathbf{R}^{HxH}$ é o peso da entrada do estado oculto (*hidden input*), $b_{ii} \in \mathbf{R}^X$ é o *bias* do portão de entrada e $b_{hi} \in \mathbf{R}^H$ é o *bias* da entrada do estado oculto.

A atualização da célula *LSTM*, ou c_t, modifica o valor do estado da célula, o qual contém o valor resultante das operações já realizadas sem a utilização do portão de saída. O resultado dessa operação é utilizado no cálculo de h_t (estado oculto) no portão de saída. Sua descrição matemática é dada pela Equação 2.7:

$$c_t = f_t * c_(t-1) + i_g * g_t \tag{2.7}$$

na qual o símbolo * representa o produto de Hadamard [3], dado por uma multiplicação elemento a elemento de duas matrizes de dimensões equivalentes e resultando em uma matriz de mesma dimensão.

O portão de saída (*output gate*), representado por o_t na Figura 2.7 é responsável por processar e controlar a saída y_t da rede e h_t para a propagação de informações para a próxima camada. A Equação 2.8 é uma representação matemática do cálculo de o_t, enquanto a Equação 2.9 demonstra o cálculo de h_t.

$$o_t = \varphi(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \tag{2.8}$$

em que $W_{io} \in \mathbf{R}^H xX$ e $b_{io} \in R^X$ são respectivamente o peso da entrada do portão de saída e *bias* do portão de saída (*input output*, io), enquanto $W_{ho} \in \mathbf{R}^H xH$ e $b_{ho} \in \mathbf{R}^H$ são respectivamente os pesos e *bias* das entradas de *hidden state* anterior (*hidden output ho*).

$$h_t = o_t * tanh(c_t) \tag{2.9}$$

no qual o_t é a entrada do portão de saída e c_t é a atualização da célula.

As redes são alimentadas com dados para seu aprendizado, e definem pesos para classificação ou identificação de padrões. Ao executar uma rede, esta interpreta as características dos dados por meio de camadas de neurônios onde, após a classificação, são atualizados os pesos dessas camadas de acordo com os erros de classificação. Pode-se dizer que são esses pesos presentes na rede neural que são responsáveis pelo aprendizado da rede. É importante lembrar que durante o funcionamento da rede, a ativação de uma camada, é responsável pela ativação da camada subsequente.

2.5 APRENDIZADO POR REFORÇO

Como mencionado anteriormente, os algoritmos de aprendizado por reforço têm como principal característica, interagir com o ambiente e reagir a "estimulos" sejam eles recompensas ou punições com o objetivo de maximizar as recompensas recebidas, ou seja, suas entradas não são rotulados como os presentes nos algoritmos de aprendizado supervisionado [53].

Existem duas principais maneiras para resolver os problemas de aprendizado por reforço [45]. A primeira consiste na busca de um comportamento específico que seja bom para o atual ambiente. Enquanto a segunda utiliza programação dinâmica e técnicas estatísticas para estimar a qualidade e utilidade das ações tomadas de acordo com o estado do ambiente. A segunda estratégia em geral apresenta melhor utilização da estrutura especial dos problemas de aprendizado por reforço, os quais não estão presentes em problemas de otimização em geral [45].

2.5.1 *Multi-Armed Bandit*

Problemas *MAB* (do inglês: *Multi-Armed Bandit*) [82] são problemas de decisão sequenciais relacionados ao dilema *Exploration versus Exploitation* (*EvE*) [54]. Para tais problemas, são desejadas soluções com o melhor desempenho (*Exploitation*), mas também é importante garantir a diversidade (*Exploration*).

O problema *MAB* está relacionado ao cenário em que um jogador joga em um conjunto K de máquinas caça-níqueis (ou braços/ações) que mesmo sendo idênticas, resultam em ganhos diferentes. Depois que um jogador puxa um dos braços a_i, $\forall\ i \in K$, em um turno t, uma recompensa ($q_{i,t}$) é obtida de acordo com alguma distribuição, tendo então como objetivo maximizar a soma dessas recompensas.

Uma política γ é uma estratégia que escolhe, a cada momento t, o próximo braço a ser puxado com base em recompensas e decisões observadas anteriormente. O problema *MAB* busca determinar a política que maximiza a recompensa cumulativa esperada [12] sobre o dilema *EvE*. Na literatura foram realizadas revisões das principais políticas do *MAB* propostas [54]. Dentre elas, vale destacar a política ϵ-greedy [98], é amplamente utilizada devido à sua simplicidade. A cada vez, tal política avalia os braços e define uma estimativa de qualidade empírica $\hat{q}_{i,t}$ com base em execuções anteriores. O valor $\hat{q}_{i,t}$ de um braço i no tempo t é baseado na soma de suas recompensas anteriores divididas pelo número de vezes que i foi puxado. Depois, a política seleciona, com probabilidade $1 - \epsilon$, o braço com o maior valor $\hat{q}_{i,t}$ (*Exploration*), ou com uma probabilidade ϵ, seleciona aleatoriamente um braço (*Exploitation*). O parâmetro ϵ é utilizado para equilibrar o *EvE* na política ϵ-greedy.

Outra política *MAB* é a *Fitness-Rate-Rank based on Multi-Armed Bandit* (*FRRMAB*), uma política que tem apresentado bons resultados no contexto de Seleção Adaptativa de Operadores [57]. Nesta política, o melhor braço é escolhido de acordo com a Equação 2.10:

$$\text{Selecione } a_t = \underset{i \in K}{\operatorname{argmax}} \left(FRR_{i,t} + C \times \sqrt{\frac{2 \times ln \sum_{j=1}^{K} n_{j,t}}{n_{i,t}}} \right) \tag{2.10}$$

onde o objetivo é, para cada momento, selecionar o melhor braço de um conjunto de braços que tem um valor estimado empiricamente ($FRR_{i,t}$) em um intervalo que depende do número de vezes ($n_{i,t}$) que o braço foi aplicado anteriormente. O parâmetro C controla o *trade-off* entre *Exploration* e *E xploitation*.

Essa política inclui dois procedimentos: atribuição de crédito e seleção de operadora (braço/ação). Na atribuição de crédito, a política *FRRMAB* utiliza um método baseado em classificação que usa a taxa de melhoria do *fitness* (do inglês: *Fitness Improvement Rate*, FIR). Neste procedimento, primeiro é calculada a FIR para cada braço, não utilizando os valores brutos das recompensas, pois seu uso pode deteriorar a eficiência do algoritmo [57, 30]. Depois, os valores FIR são armazenados em uma determinada *Sliding Window* (*SW*) organizada como uma fila, que é usada para avaliar as aplicações recentes de acordo com o seu tamanho W. A *SW* tem como função armazenar uma determinada quantidade de dados W de acordo com o instante de tempo, para a utilização em futuras avaliações. O segundo procedimento, seleciona aleatoriamente um braço até que todos os braços sejam selecionados, então usa a política *FRRMAB* para avaliar cada braço e selecionar o melhor, dando assim a chance de todos serem igualmente selecionados.

2.5.2 *Contextual Multi-Armed Bandit*

Em *CMAB* (do inglês: *Contextul Multi-Armed Bandit*) [59][1], antes de puxar um braço, a política observa o contexto (estado) para cada braço de um espaço de recurso d-dimensional $x_{i,c} \in \mathcal{X}^d$, onde $\mathcal{X}$ é um espaço de contexto limitado, associado ao tempo atual. Depois, um braço é escolhido e sua recompensa é recebida.

Nesse sentido, para cada momento c, o ambiente "cria" um contexto e recompensa de uma distribuição desconhecida $(x_c, q_c) \sim \mathcal{D}$. O contexto (estado) consiste em uma descrição do ambiente que a política pode usar para realizar ações mais informadas. Então, uma política $\gamma : \mathcal{X} \rightarrow K$ escolhe um braço $\gamma(x_c)$ de acordo com o contexto atual. Finalmente, uma recompensa é recebida pela escolha, $r_{c,\gamma(x_c)}$. Em casos com contextos diferentes, o braço que tem a maior recompensa pode variar [107], e um contexto pode incluir tudo o que sabemos no tempo c.

Assim como nos problemas *MAB*, políticas particulares são projetadas para lidar com *CMAB*, dentre elas destaca-se o Limite de Confiança Superior Linear (do inglês: *Linear Upper Confidence Bound*, *LinUCB*) [58]. Essa política foi proposta para recomendar notícias aos usuários e pressupõe que as relações entre os contextos e as recompensas sejam lineares. *LinUCB* é uma política genérica e adaptável para ser utilizada em outras aplicações. A politica decompõe o vetor de características da rodada atual em uma combinação linear de vetores de características vistos nas rodadas anteriores e usa os coeficientes e recompensas, calculados nas rodadas

[1]Na literatura, *bandits* contextuais são às vezes chamados de *bandits* com informações secundárias [18], *bandits* com covariáveis [101], *bandits* com conselhos de especialistas [5], *bandits* associativos [92] e aprendizado de reforço associativo [44].

anteriores para calcular a recompensa esperada na rodada atual. A seleção de um braço acontece de acordo com a Equação 2.11:

$$\text{Selecionar } a_c = \underset{i \in K}{\text{argmax}} \left(\hat{\theta}_i^{\mathsf{T}} x_{i,c} + \alpha \times \sqrt{x_{i,c}^{\mathsf{T}} \mathbf{A}_i^{-1} x_{i,c}} \right) \tag{2.11}$$

onde o primeiro termo é a recompensa de uma regressão descrita e o segundo termo o desvio padrão da recompensa. Em cada ponto de tempo c, o melhor braço a_c é escolhido calculando quais braços dão a maior recompensa prevista de acordo com o contexto observado (primeiro termo).

A cada momento c, o procedimento *LinUCB* recebe os braços com seu respectivo contexto observado, a constante α que equilibra o dilema *EvE* (um valor alto de α favorece *Exploration*), e uma dimensão d com Informação contextual. A constante α é o único parâmetro a ser definido pelo usuário e é dada por $\alpha = 1 + \sqrt{ln(2/\rho)/2}$, com uma probabilidade de confiança (1 - ρ). Em seguida, o procedimento retorna o braço com o maior valor para o limite de confiança (a_c).

2.5.3 SW-LinUCB

Assim como nos problemas de *MAB*, políticas particulares são projetadas para lidar com *CMAB*, Gutowski et al. [37] propuseram o *Sliding Window - Linear Upper Confidence Bound* (*SW-LinUCB*). Esta política visa a mitigar os efeitos da não estacionaridade utilizando os benefícios de uma *SW*. A politica *SW-LinUCB* permite mudanças na qualidade do braço ao longo do processo de busca, e sua avaliação não é prejudicada pelo seu desempenho em um estágio muito inicial, o que pode ser irrelevante para o seu desempenho atual. *SW-LinUCB* seleciona um braço de acordo com a Equação 2.12.

$$\text{Selecionar } a_t = \underset{i \in K}{\text{argmax}} \left(\left(1 - \frac{\boldsymbol{Occ}_{SW}(i,t)}{|SW|} \right) \times \hat{\theta}_i^{\mathsf{T}} x_{i,t} + \alpha \times \sqrt{x_{i,t}^{\mathsf{T}} \mathbf{A}_i^{-1} x_{i,t}} \right) \tag{2.12}$$

onde $\boldsymbol{Occ}_{SW}(i,t)$ representa o número de vezes que um braço i foi selecionado nas últimas aplicações $|SW|$.

Esta política é uma adaptação da política do *LinUCB*, portanto, essas políticas compartilham funcionalidades semelhantes. Para observar as diferenças entre eles, a política *SW-LinUCB* é detalhada no Algoritmo 2.1 e as partes que divergem da política *LinUCB* são destacadas em cinza. Neste algoritmo, a *SW* é introduzida para armazenar o histórico de seleções de cada braço i, e é definido como um conjunto de subsequências binárias $SW_{t,i} = \{0..(2^{(|SW|+1)} - 1)\}$. Por exemplo, para um SW com tamanho = 6 e $SW_{t,i} = 101001$ significa que o braço i foi selecionado nas tentativas $t = 6$, $t = 4$ e $t = 1$.

Desta forma, para cada novo braço i, $\boldsymbol{Occ}_{SW}(i,t)$ é zerado (linha 12). Então, se a *SW* foi preenchida, é contabilizado o histórico de seleção associado a um braço i (linhas 15 a 17). Tal informação é usada durante o cálculo do limite de confiança superior $\hat{q}_{t,i}$ (linha 18). Então, o braço com o maior $\hat{q}_{t,i}$ é selecionado (linha 20). Em seguida, a *SW* é atualizada, para cada braço, somando 1 ou 0 para seu braço associado, representando a seleção ou não do braço durante o processo (linhas 21 e 22). As informações dentro do SW são truncadas para preservar o tamanho do SW (linhas 23 a 25). Finalmente, o melhor braço selecionado a_t é retornado (linha 26).

Algorithm 2.1: Procedimento do SW-LinUCB para escolher um braço (adaptado de [37]).

Input: conjunto de braços (K)
Contexto atual (x_t)
$\alpha \in \mathbb{R}_+$
Dimensão $d \in \mathbb{N}$
Janela deslizante (SW)
Output: Melhor braço (a_t)
begin
  foreach $i \in K$ **do**
    if *i é novo* **then**
      $\mathbf{A}_i \leftarrow \mathbf{I}_d$ (dxd matriz identidade)
      $\mathbf{b}_i \leftarrow \mathbf{0}_{dx1}$ ($dx1$ vetor zero)
      $\boldsymbol{Occ}_{SW}(i,t) \leftarrow \mathbf{0}$
    end
    $\hat{\theta}_i \leftarrow \mathbf{A}_i^{-1}\mathbf{b}_i$ ▷ recompensa acumulada
    if $t > |SW|$ **then**
      /* Contagem de seleção de um braço i nas últimas seleções de $|SW|$ */
      $\boldsymbol{Occ}_{SW}(i,t) \leftarrow \#_1(SW_{t,i})$
    end
    // limite superior de confiança
    $\hat{q}_{i,t} \leftarrow \left(1 - \frac{\boldsymbol{Occ}_{SW}(i,t)}{|SW|}\right) \times \hat{\theta}_i^{\top} x_{t,i} + \alpha \times \sqrt{x_{t,i}^{\top} \mathbf{A}_i^{-1} x_{t,i}}$
  end
  Select $a_t \leftarrow \underset{i \in K}{\operatorname{argmax}}(\hat{q}_{i,t})$ ▷ com laços quebrados arbitrariamente
  // Atualizar a SW
  $\forall i \in K \neq a_t$ Atualize as subsequências $SW_{t,i}$ adicionando um bit 0
  Atualize a subsequência SW_{t,a_t} adicionando um bit 1
  if $t > |SW|$ **then**
    Calcular um deslocamento lógico para a esquerda para todos os $SW_{t,i}$ e SW_{t,a_t}
  end
  return a_t
end

2.6 CONSIDERAÇÕES FINAIS

Nesse capitulo foram apresentados conceitos relacionados a esse trabalho, sendo eles: teste de software, ambientes de integração contínua e aprendizado de máquina. Os algoritmos a serem utilizados no trabalho foram apresentados com mais detalhe, sendo eles: *RF*, *MAB* e *CMAB*.

Dentro da área de estudos de aprendizado por reforço, o *MAB* faz parte de uma subclasse de problemas um pouco mais simples de serem trabalhados. Enquanto o problema *CMAB* é intermediário entre o problema *MAB* (Figura 2.8), no qual cada ação afeta apenas a recompensa imediata, e o aprendizado por reforço completo, no qual as ações podem afetar a próxima situação bem como a recompensa [93]. Normalmente, é raro quando o contexto não está disponível [55], assim como o *CMAB* é mais aplicável do que as variantes não contextuais, como em ensaios clínicos [97] e sistemas de recomendação [59][2].

Enquanto o intuito do ambiente de integração contínua é otimizar a entrega de modificações realizadas pelos desenvolvedores, há uma crescente necessidade da otimização de recursos utilizados nesse ambiente. Como os custos dos testes de regressão em sua maioria são elevados,

[2]Netflix usa *CMAB* para recomendar filmes e séries [14].

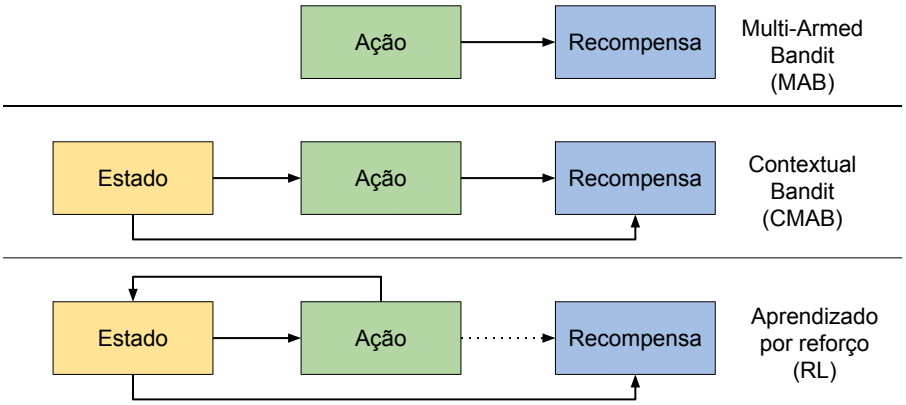


Figura 2.8: Principais diferenças entre *MAB*, *CMAB* e aprendizado por reforço.

técnicas de priorização desses casos de teste têm sido propostas, dentre elas, destaca as técnicas que utilizam algoritmos de *ML*.

O próximo capítulo apresenta estudos que propõem abordagens relacionadas a esse trabalho, tanto em utilização de aprendizado profundo em aplicações de engenharia de *software*, quanto utilização de abordagens de aprendizado de máquina para a priorização de casos de teste.

3 TRABALHOS RELACIONADOS

Nesse capítulo são apresentados estudos relacionados ao trabalho sendo proposto. O capítulo está dividido em seções relacionadas aos respectivos temas: *Machine Learning* (*ML*) e *DL* em teste de software e trabalhos de priorização de casos de teste em ambientes de integração contínua.

3.1 USO DE ML E DL NA ATIVIDADE DE TESTE

Para o levantamento de trabalhos que aplicam DL em teste foi realizada uma busca utilizando a *string* de busca a seguir, a qual foi derivada tendo como base o *survey* realizado por Pouyanfar et al. [77].

(“software test”) AND (“deep learn*” OR “Long Short Term Memory” OR “lstm” OR “Convolutional neural network” OR “cnn” OR “recurrent neural network” OR “rnn” OR “Multilayer perceptron” OR “MLP” OR “Deep Belief Network*” OR “DBN” OR “Generative Adversarial Network” OR “GAN” OR “Self Organizing Map” OR “SOM” OR “Restricted Boltzmann Machine” OR “RBM”)*

Foram utilizadas três bases escolhidas devido a sua popularidade em engenharia de software: IEEExplore, Scopus e ACM. Para a seleção dos estudos relacionados, primeiramente foram elaborados alguns critérios para a seleção, sendo eles: trabalhos desenvolvidos em inglês ou português, tema dos estudos relacionados a DL e serem estudos relacionados com teste de software. Essa pesquisa foi desenvolvida em abril de 2021.

Após a verificação desses critérios, foram separados os estudos em que eram aplicadas técnicas de DL em engenharia de software e os estudos que aplicavam técnicas da engenharia de software em sistemas baseados em DL. Para descrição nesta seção, foram selecionados somente os com maior semelhança com o trabalho desenvolvido, ou seja aqueles que utilizam redes *LSTM*, Mapas auto-organizados (do inglês: *Self-Organizing Maps*, SOM) e *Perceptron* de multi camadas (do inglês: *Multi Layer Percptron, MLP* para resolverem problemas no contexto de teste de software.

Medhat et al. [70] propõem um *framework* para teste de regressão e teste de integração em ambientes de internet das coisas (do inglês: *Internet of things*, IoT), com o objetivo fornecer um conjunto priorizado auto-adaptativo de casos de testes. Essa abordagem utiliza *LSTM* para extração de características e classificação dos exemplos. Para a seleção e priorização dos casos de teste utiliza um algoritmo genético e meta heurística recozimento simulado (do inglês: *Simulated Annealing*). A *LSTM* extrai os atributos significativos e os utiliza para a classificação dos conjuntos de casos de teste de acordo com os componentes de IoT presentes. As características gerais avaliadas foram: Identificador (ID), nome do caso de teste, e descrição [70].

O trabalho desenvolvido por Dobuneh et al. [81] busca utilizar os dados gerados na sessão de usuário em sistemas online como dados de entrada para o teste de regressão, por meio de clusterização utilizando SOM. É proposta a utilização de uma combinação entre a priorização dos casos de teste e a *clusterização* (agrupamento) dos casos de teste. Salvando as ações do usuário durante a sua sessão, o sistema busca gerar os casos de teste simulando um sistema caixa-preta, uma vez que os usuários realizam suas ações sem ter conhecimento da estrutura do sistema. Em casos em que há dependências entre as páginas acessadas pelo usuário, são utilizadas técnicas para aumentar esse caso de teste na tentativa de solucionar tal dependência [81].

As técnicas de clusterização geram um conjunto reduzido de casos de teste em comparação com o original. Contudo, mesmo com tamanho reduzido, em alguns casos o conjunto de teste

ainda não é executado em sua integridade devido a limitações de tempo e de recursos. Buscando uma melhora na acurácia e efetividade dos testes, os conjuntos são ordenados de acordo com um critério de ordenação. Com o intuito de avaliar a técnica desenvolvida, foi aplicado um estudo de teste onde foram aplicados o SOM. Os resultados de avaliação da técnica mostraram que executando aproximadamente 50% dos casos de teste, o algoritmo detectou 28 de um conjunto de 30 falhas totais.

Momotaz e Dohi [72] utilizam redes profundas MLP para predição de defeitos com o objetivo de manter uma alta confiabilidade do software. A rede desenvolvida utiliza o método *Delta* [42, 96], o qual tem objetivo de encontrar amostras para regressões não lineares. Para o treinamento da base, foram utilizados os métodos de transformações de dados estatísticos: Bartlett [7], Fisz [31] e Anscombe [4], bem como um treinamento sem a utilização de tais métodos. Os resultados obtidos mostram que as redes alimentadas com transformações obtiveram melhor desempenho.

Czibula et al. [24] utilizam redes *fuzzy* em conjunto com SOM para o problema de predição de defeitos. Antes do treinamento da rede, os dados são normalizados com o objetivo de encontrar sub-conjuntos de métricas de software que são relevantes para a detecção de defeitos. O algoritmo busca agrupar as métricas por meio das matrizes de associação *fuzzy* criadas, de modo a maximizar os graus de associação entre os exemplos. A comparação realizada no trabalho indica que o algoritmo proposto tem resultados melhores ou equivalentes que algoritmos já existentes. Contudo, existem técnicas que têm uma performance melhor do que a apresentada utilizando métodos *fuzzy*.

Ayon [6] utiliza várias redes neurais em conjunto com um algoritmo genético e um otimizador de partículas (do inglês: *Particle Swarm Optimization*, PSO) para o problema de predição de defeitos. Nos quais o GA é utilizado para a seleção de atributos enquanto o PSO é responsável por agrupá-los. Esses dados são utilizados para o treinamento das redes: recorrente, *Feedforward*, profunda e artificial. Como método de avaliação, também são realizadas comparações com técnicas existentes na literatura, destacando que o algoritmo desenvolvido atinge uma acurácia de 98,47% usando a rede neural profunda, e 98,39% utilizando redes neurais recorrentes, enquanto algoritmos como árvore de decisão, utilizados no trabalho relacionado de Chug et al. [21] atingem uma acurácia de 99.36%.

O trabalho de Busjaeger e Xie [13] cria um modelo de previsão de falhas para os casos de teste usando SVM^{map}. Tal modelo é utilizado na priorização, levando em consideração cinco atributos: cobertura de teste do código modificado, similaridade textual entre testes e alterações, falha de teste recente ou histórico de falhas e idade do teste. Mas esses atributos precisam de informações adicionais e dependem de instrumentação de código.

As principais limitações dos trabalhos mencionados acima é que eles exigiram um esforço e custo adicional para calcular a cobertura do teste ou outras informações adicionais. Além disso, eles não abordam as principais características dos ambientes de CI - volatilidade dos casos de teste. Não são adaptativos, ou seja, não aprendem com as prioridades passadas.

3.2 PRIORIZAÇÃO DE CASOS DE TESTE EM AMBIENTES DE INTEGRAÇÃO CONTÍNUA

Marijan et al. [68, 66, 69, 67] introduziram diferentes abordagens *TCPCI*. A primeira, *ROCKET* [68], considera a distância do status de falha de uma execução atual de um caso de teste até seu tempo de execução. Um trabalho posterior dos autores [66] considera, dado um orçamento de teste, outros fatores como detecção de falhas, regras de negócio, desempenho e perspectivas técnicas. Uma ferramenta, chamada *TITAN* [69], que é baseada em programação de restrições

(*constraint programming*), minimiza o número de casos de teste que cobrem alguns requisitos que os casos de teste originais cobrem. Então, o conjunto minimizado é priorizado usando o *ROCKET*. Para abordar a priorização de casos de teste no contexto de Sistemas Altamente Configuráveis, Marijan et al. [67] usa um algoritmo de aprendizado para classificar casos de teste considerando a cobertura de recursos e descobrir casos de teste redundantes. A prioridade é calculada com base em sua eficácia histórica de detecção de falhas e considerando um orçamento de teste.

Xiao et al. [102] propõem o uso de uma *LSTM* baseada em séries temporais para a resolução do problema de priorização de casos de teste em ambientes de integração contínua. Essa abordagem busca uma melhor predição considerando todos os históricos de testes ambiente. A abordagem considera os testes como: testes com falhas (1) e testes sem falhas (0). Com isso ela busca os casos de teste os quais apresentam classificações únicas dentro de n ciclos anteriores, para realizar a predição para o ciclo $n + 1$. O algoritmo trabalha com a hipótese de que casos de teste que conseguem detectar falhas em ciclos anteriores têm maior probabilidade de detectar falhas nos próximos ciclos [49]. Entretanto, em casos em que o histórico de casos de teste apresente somente os mesmos valores para todos os ciclos de execuções, a *LSTM* apresenta dificuldades para apresentar o resultado ideal para a predição. Buscando resolver esse problema, foram realizados cálculos da probabilidade dos casos de teste apresentarem os mesmos valores para todos os ciclos, buscando encontrar casos de teste que não apresentem informações o suficiente. Em seguida, são removidos os casos de teste considerados como ruídos. Por fim, a priorização é realizada.

COLEMAN formula o problema *TCPCI* como um problema *Multi-Armed Bandit*. Para o algoritmo cada caso de teste é um braço, para considerar a natureza dinâmica do problema *TCPCI*. Para isso, *COLEMAN* incorpora duas variantes de *MAB*: volátil e combinatória. Na primeira variação, a abordagem seleciona vários braços em cada turno (*commit*), para produzir um conjunto ordenado. Enquanto na segunda, apenas os casos de teste disponíveis em cada *commit* são considerados para priorização. A segunda variação visa a lidar com a volatilidade dos casos de teste. Ao final, uma função de recompensa é utilizada para obter *feedback* (recompensa) a partir da priorização proposta pela abordagem. Com base nesse *feedback*, a abordagem visa incorporar o aprendizado da aplicação do conjunto de casos de teste priorizado.

A abordagem *COLEMAN* trabalha com os ambientes de CI de acordo com as seguintes etapas: Após uma *build* bem-sucedida, durante a fase de teste, *COLEMAN* começa a agir antes da execução do teste. Neste momento, a abordagem usa uma política *MAB* para priorizar um conjunto de casos de teste disponível (T) do *commit* atual (c). Assim, os casos de teste do conjunto de casos de teste priorizados (T') são executados até que o orçamento de teste disponível seja atingido. O *feedback* sobre o (T') aplicado é obtido e utilizado pela política do *MAB* para adaptar sua experiência para futuras predições. Ao final, um relatório é gerado e os desenvolvedores são informados. Além disso *COLEMAN* pode usar diferentes funções de recompensa e políticas *MAB*, sendo que a política *FRRMAB* apresentou os melhores resultados nos experimentos conduzidos [28].

A abordagem *Reinforced Test Case Selection* (*RETECS*) [90] usa aprendizado por reforço para priorizar e selecionar casos de teste. *RETECS* considera como entrada a duração do caso de teste, dados históricos de falha e última execução. Ele é guiado por uma função de recompensa que aprende ao longo dos ciclos e pode se adaptar às mudanças. Os autores compararam diferentes variantes aprendizado por reforço e a variante que utiliza redes neurais apresentou os melhores resultados.

Essas últimas abordagens, *COLEMAN* e *RETECS*, apresentam algumas vantagens porque lidam adequadamente com a volatilidade dos casos de teste e abordam o dilema *EvE* [82].

Para resolver esse dilema, uma abordagem precisa equilibrar a diversidade de casos de teste e a quantidade de novos casos de teste e casos de teste que são propensos a erros ou que possuem alta capacidade de falhar. Este problema está relacionado ao orçamento de teste, pois se apenas os casos de teste propensos a erros forem considerados sem diversidade, alguns casos de teste nunca poderão ser executados. Em experimentos relatados na literatura [28], *COLEMAN* superou *RETECS*, por apresentar melhor desempenho, em relação ao *NAPFD* e outros indicadores.

Mas *RETECS* e *COLEMAN* apresentam algumas desvantagens, eles não consideram o estado atual do sistema para a priorização. Seu desempenho diminui para sistemas com um grande conjunto de casos de teste, nos quais muitas falhas estão distribuídas por muitos casos de teste.

O trabalho de Bertolino et al. [10] apresenta resultados comparando duas estratégias de *ML* em integração contínua e avalia o desempenho de diferentes algoritmos de *ML*. A primeira estratégia avaliada, denominada *Learning-to-Rank*, utiliza aprendizado supervisionado para treinar um modelo baseado em alguns recursos de teste. O modelo é então usado para classificar conjuntos de teste em confirmações futuras. O problema com esta estratégia é que ela requer retreinamento do modelo, porque muitas vezes o modelo pode não ser mais representativo quando o contexto de confirmação muda. A segunda estratégia avaliada, denominada *Ranking-to-Learn*, é mais adequada ao contexto dinâmico de integração contínua. Esse tipo de estratégia é baseada em aprendizado por reforço, e é o foco do nosso trabalho. Os autores concluem que as estratégias *Ranking-to-Learn* são mais robustas em relação à volatilidade dos casos de teste, alterações de código e número de testes com falha.

3.3 CONSIDERAÇÕES FINAIS

Neste capitulo foram apresentados os trabalhos que utilizam *ML* e *DL* na atividade de teste em geral e mais particularmente para priorizar casos de teste no ambiente de integração contínua.

Alguns trabalhos têm como objetivo a predição de defeitos utilizando DL [72, 24, 6], entretanto os mais relacionados são os que realizam priorização de casos de teste. A rede *LSTM* desenvolvida por Xiao et al. [102] utiliza o histórico de testes disponível tendo como base a hipótese de que testes que falharam no passado tendem a falhar novamente no futuro. Contudo, esse trabalho não considera o tempo gasto para a priorização e custo dentro do ambiente durante a execução da priorização. A arquitetura da *LSTM* desenvolvida não ficou disponível para a realização de comparativos de desempenho com variações de camadas *LSTM*.

A abordagem *COLEMAN* [28] obteve bons resultados de priorização utilizando *MAB*, buscando resolver o dilema de *EvE*. Essa abordagem também utiliza o histórico de falhas como principal fonte de dados para sua predição, além de utilizar uma *SW* como meio de melhorar o aprendizado considerando a volatilidade de casos de teste.

O trabalho de Bertolino et al. [10] mostra que o algoritmo *Random Forest* apresentou os melhores resultados mas, os autores não compararam esse algoritmo com a abordagem *COLEMAN* a qual apresentou melhores resultados do que o *RETECS* [28]. A Tabela 3.1 apresenta um breve resumo dos trabalhos relacionados, apresentando seu contexto de desenvolvimento e alguns pontos positivos e negativos.

Por isso, a proposta do capítulo seguinte é o uso da rede *LSTM* e do algoritmo *RF* para priorização de casos de teste em ambientes de integração continua, o que pode trazer contribuições para o estado-da-arte atual em *TCPCI*.

Tabela 3.1: Comparativo entre os trabalhos relacionados.

Referencia	**Tecnica**	**Contexto**	**Positivos**	**Negativos**
Medhat et al. [70]	LSTM/GA	Priorização de casos de testes no contexto de IoT	Tem alta acurácia na predição	Pré processamento realizado pela LSTM para alimentação do GA. O GA exige um alto tempo de processamento.
Dobuneh et al. [81]	SOM	Combinação entre clusterização e priorização de casos de teste em sistemas online	Utiliza os dados do usuário para geração dos casos de teste	Exige um número de usuários para utilização do sistema. Algumas áreas do sistema de raro acesso podem ficar sem cobertura.
Xiao et al. [102]	LSTM	Priorização de casos de teste em ambientes de IC	Obtém altos valores para APFD	Não leva em consideração, tempo de execução e de priorização
RETECS [90]	Rede neural	Priorização de casos de teste em ambientes de IC	Obtém altos valores para APFD e NAPFD; tem um tempo de priorização baixo	Superado por *COLEMAN* quando comparado diretamente
COLEMAN [28]	MAB	Priorização de casos de teste em ambientes de IC	Obtém altas acurácias; tem um tempo de priorização baixo	Não considera o estado presente para priorização
Bertolino et al. [10]	Múltiplos algoritmos	Priorização de casos de teste em ambientes de IC	Avaliação de vários algoritmos e definição de 2 estratégias	Não avalia a abordagem *COLEMAN* baseada em *MAB*

4 ABORDAGEM PROPOSTA

Como mencionado no capitulo anterior, com a crescente necessidade de uma priorização de baixo custo e de alta qualidade, várias técnicas foram propostas para a priorização de casos de teste em ambientes de integração contínua, dentre elas destacam-se as abordagens baseadas em aprendizado por reforço e outras baseadas em *ML* que não consideram a dependência temporal entre as instâncias do treinamento.

Por isso, neste capítulo é introduzida uma abordagem para aplicar diferentes algoritmos de regressão, utilizando alguns conceitos da estratégia *Ranking-to-Learn*, definida por Bertolino et al. [10]. A abordagem tem como objetivo realizar a priorização de casos de teste e satisfazer a necessidade dos ambientes de integração contínua, utilizando o método de *SW*, similarmente às abordagens existentes na literatura baseadas em aprendizado por reforço [28, 90], permitindo o uso do aprendizado online. Para o desenvolvimento deste trabalho, foi utilizado o algoritmo *Random Forest* e a rede neural *LSTM*.

4.1 UMA VISÃO GERAL DA ABORDAGEM

Tendo em vista as definições apresentadas no Capitulo 2 e após a avaliação de trabalhos relacionados, apresentados no Capitulo 3, esse trabalho visa a solucionar o problema de *TCPCI* por meio da utilização dos algoritmos *RF* e *LSTM*. A abordagem baseia-se em regressão, buscando primeiramente de maneira numérica, a predição da probabilidade de falhas dos casos de teste e em seguida ordenando os casos de teste de maneira com que os casos de teste com maior probabilidade de falhar sejam executados primeiro na fila do ambiente de integração contínua. A abordagem também utiliza o método de *SW* para seu funcionamento, com o objetivo de facilitar a utilização dos dados de histórico dos casos de teste.

Figura 4.1 ilustra como a proposta de uso de *ML* se insere no ambiente de integração contínua. Primeiramente, o código é construído a partir do *commit* mais recente c. Se a compilação for bem-sucedida, o conjunto de casos de testes presentes no *commit* e os presentes no histórico, de acordo com o tamanho da *SW*, são utilizados como entrada para os algoritmos para previsão de propensão a falhas. A próxima etapa filtra os casos de teste $t \in Tc$ da saída, classifica-os em ordem decrescente, transformando-os no conjunto priorizado $T'c$, que é executado pelo ambiente de integração contínua. Finalmente, $T'c$ é avaliado e seus resultados são salvos como conjunto de dados históricos a serem utilizados na próxima iteração do ambiente.

O Algoritmo 4.1 representa de maneira computacional o fluxo descrito anteriormente. A linha um representa as entradas do algoritmo: conjunto C de *commits*, o tamanho W da *SW* a ser utilizado e o percentual de orçamento de teste disponível TB. A linha cinco refere-se à inicialização do contador de *commits i* em zero, o qual é utilizado para o algoritmo reconhecer que o conjunto de testes a ser avaliado iniciará o ciclo de execuções do algoritmo.

O algoritmo recebe como entrada um conjunto de *commits* C, o tamanho da janela deslizante W e o percentual de orçamento de teste disponível TB, e em seguida é inicializado o contador de *commits i* com zero (linha cinco). Para cada *commit c* presente no conjunto C (linha seis), o conjunto de testes T_c presente no *commit*, o tempo de teste total disponível A_c e o valor de orçamento de teste do *commit* TB_c são avaliados pelo algoritmo.

Para as primeiras avaliações $i < 2$ (linha dez), o algoritmo não tem histórico de falhas o suficiente para realizar a predição, sendo assim, é realizado uma priorização de maneira aleatória (linha onze). Para os outros casos, o algoritmo utiliza do histórico de testes H de acordo com

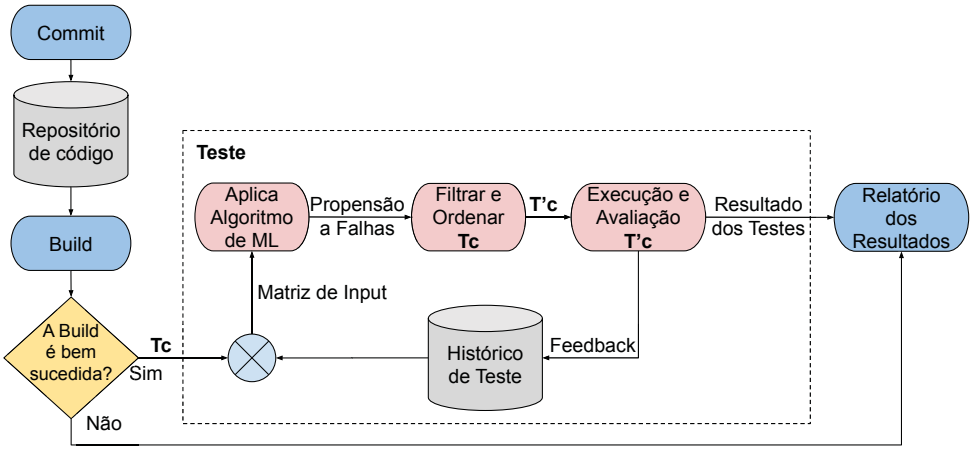


Figura 4.1: Visão geral da abordagem aplicada ao ambiente de integração contínua.

Algorithm 4.1: Pseudocódigo do funcionamento do algoritmo.

```
1  Input: Conjunto de Commits (C)
2         Tamanho da Janela Deslizante (W)
3         Percentual de orçamento de teste disponível (TB)
4  begin
5      i ← 0 ;                                   ▷ contador de commits
6      foreach c ∈ C do
7          T_c ← Conjunto de testes do sistema no commit atual;
8          A_c ← Tempo total para executar T_c;
9          TB_c ← Valor de orçamento de teste em relação a A_c;
10         if i < 2 then
11             T'_c ← T_c priorizado aleatoriamente ; ▷ Não há histórico suficiente
12         else
13             H ← Histórico de dados dos últimos W commits;
14             T'_c ← T_c priorizado com o algoritmo de ML
15         end
16         Avalia T'_c considerando TB_c (Ex.: NAPFD)
17         i ← i + 1;
18     end
19 end
```

o tamanho da *SW W*. Após a execução do algoritmo de *ML*, é gerado o conjunto de testes priorizado T' (linha quatorze) que em seguida é avaliado de acordo com alguma métrica de qualidade selecionada, por exemplo, no Algoritmo 4.1 foi utilizada a métrica *NAPFD* (linha dezesseis). Por fim é realizado um incremento no contador de *commits* i, seguindo para o próximo ciclo de execuções (linha dezessete).

Em busca de uma melhoria na qualidade de predição, na literatura é utilizada a técnica de janela deslizante (do inglês: *Sliding Window*, *SW*), principalmente em problemas de séries temporais, no caso dos testes de regressão, ela é utilizada em conjunto com o histórico testes. Essa técnica tem como objetivo guardar alguns dados do histórico de testes para reutilização em próximas predições.

A Figura 4.2 ilustra o funcionamento do algoritmo para uma janela deslizante de tamanho $W = 2$. O algoritmo começa com a análise do *commit* $i = 1$, para o qual não há histórico

de teste, portanto a janela deslizante ainda não recebeu dados. Na análise do *commit* $i = 2$, a janela deslizante é preenchida com informação do caso de teste avaliado anteriormente, no caso do exemplo, o caso de teste t_1. A figura representa a avaliação do *commit* $i = 3$, no qual a janela deslizante está preenchida com os valores de t_1 e t_2. Como o valor representado é $W = 2$, a janela está totalmente preenchida. Para o momento de análise do *commit* $i = 4$, a janela deslizante é preenchida por t_3. Com a adição desse caso de teste, o caso de teste t_1 é descartado da janela deslizante, dando lugar para o conjunto formado por t_2 e t_3. É importante notar que o método *SW* ilustrado é o de janela fixa que adota apenas os dados mais recentes, ou seja, apenas os W *commits* mais recentes para serem usados como entrada para os algoritmos.

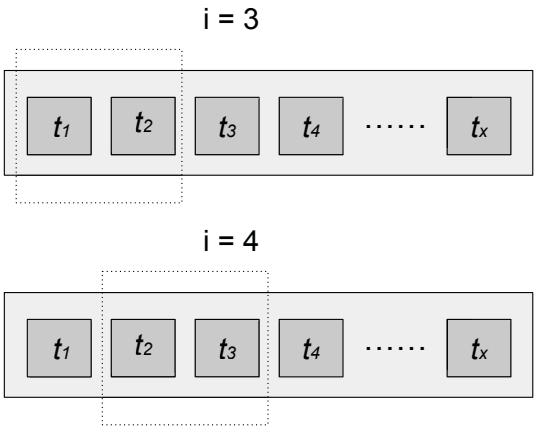


Figura 4.2: Exemplo do funcionamento da janela deslizante.

A Figura 4.3 ilustra de maneira mais detalhada a execução para a predição, na qual a matriz de entrada é definida por dados dos casos de teste presentes no histórico, de acordo com o tamanho da *SW*, o último *commit* presente no histórico, ou seja, o *commit* anterior ao analisado. Essa matriz então, é dada como entrada para os algoritmos de *ML*, os quais geram um vetor de previsão contendo a propensão a falhas de todos os casos de testes presentes na matriz de entrada. Em seguida os casos de testes são ordenados de acordo com a propensão a falha e mapeados para que a saída contenha somente os casos de testes presentes no *commit* analisado. Por fim, um novo vetor com os casos de testes ordenados e mapeados é gerado.

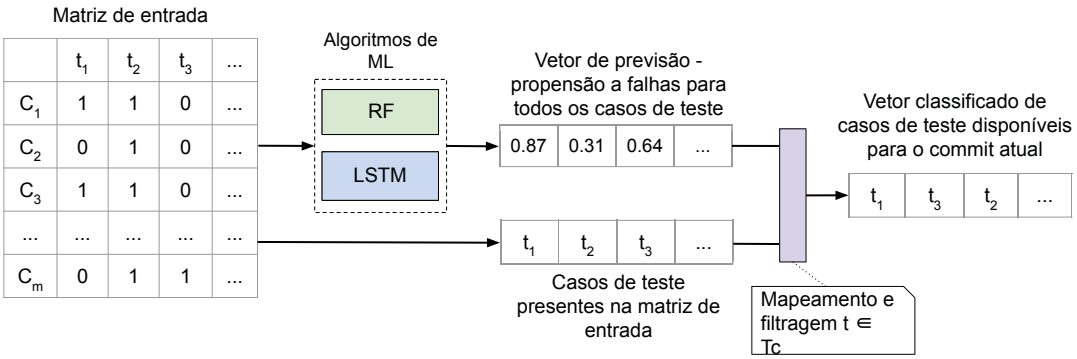


Figura 4.3: Visão geral de como a abordagem realiza a priorização dos casos de teste.

4.2 AVALIAÇÃO

Para avaliar a proposta de uso do método *SW*, dois algoritmos de *ML* foram utilizados. O primeiro é o *Random Forest*, descrito na Seção 2.3. Esse algoritmo utiliza como base várias árvores de decisão combinadas para formular sua predição. Este algoritmo foi escolhido por apresentar os melhores resultados segundo diferentes critérios avaliados para estratégias de *Ranking-to-Learn* no trabalho de Bertolino et al. [10].

O segundo algoritmo escolhido é a rede neural *LSTM*, cujas camadas são ilustradas na Figura 4.4. Primeiramente, os dados vão para a camada de entrada, em seguida são processados pela primeira camada de ativação *tanh* subsequente de uma camada de *dropout*; após, uma segunda camada de ativação *tanh* leva a uma camada densa e, finalmente, à camada de saída. Foi escolhida a rede *LSTM* devido à sua capacidade de lembrar as coisas. Ela é capaz de simular uma memória e tem apresentado resultados promissores em diferentes domínios e para previsão de séries temporais [78]. Além disso, não foi avaliada no trabalho de Bertolino et al., e não foi comparada com a abordagem baseada em *MAB*.

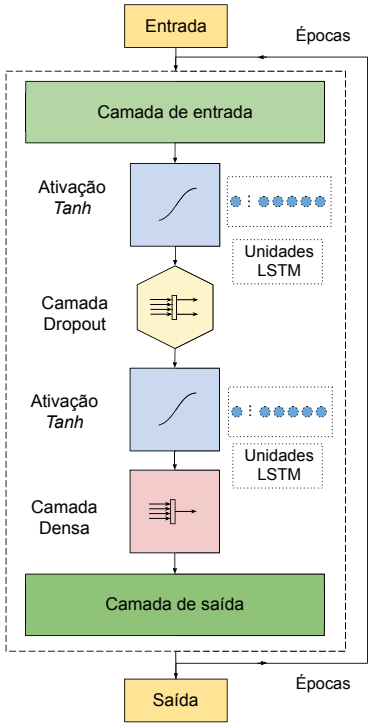


Figura 4.4: Visão geral da rede *LSTM*.

As métricas consideradas para avaliação do desempenho da priorização realizadas pelos algoritmos, tanto durante a execução do algoritmo para a avaliação dos casos priorizados, quanto durante a fase experimental são as mesmas avaliadas e utilizadas por Prado Lima e Vergilio [28]; sendo elas: *NAPFD*, *APFDc*, *NTR* e *PT*.

Foram aplicados os testes estatísticos Kruskal-Wallis [52], Mann-Whitney [65] e Friedman [32] com um nível de confiança de 95%. Kruskal-Wallis foi utilizado para avaliar o desempenho das abordagens em cada sistema; Mann-Whitney para avaliar um grupo de desempenhos no mesmo sistema ou para análise post-hoc; Friedman para avaliar o comportamento da abordagem em diferentes sistemas. Para tal, cada sistema torna-se uma variável dependente, na qual são aplicadas múltiplas abordagens.

Além disso, para calcular a magnitude do tamanho do efeito da diferença entre os grupos, foi utilizada a métrica $\hat{A}_{12}$ [95] de Vargha e Delaney. Essa medida varia de 0 a 1 e define

a probabilidade de um valor, retirado aleatoriamente da primeira amostra, ser superior a um valor retirado aleatoriamente da segunda amostra. A magnitude $desprezvel$ ($\hat{A}_{12} < 0,56$) geralmente não apresenta uma diferença estatística, as magnitudes $pequena$ ($0,56 \leq \hat{A}_{12} < 0,64$) e $média$ ($0,64 \leq \hat{A}_{12} < 0,71$) podem ou não gerar diferenças estatísticas, já uma $grande$ magnitude ($0,71 \leq \hat{A}_{12}$) representa uma diferença estatística significativa, que normalmente pode ser vista nos números sem muito esforço.

4.3 QUESTÕES DE PESQUISA

A principal hipótese do experimento conduzido é que o uso de *RF* e *LSTM* juntamente com o método de janela deslizante permite reduzir o esforço e tempo do *TCPCI*, melhorando o desempenho em relação aos trabalhos relacionados. Nesta seção apresenta-se a configuração do experimento conduzido para avaliar tal hipótese. De acordo com os objetivos, as seguintes questões de pesquisa foram formuladas:

QP1: *A abordagem usando LSTM e RF é aplicável nos ambientes de integração contínua?* Esta questão investiga a aplicabilidade da abordagem com ambos os algoritmos. Consideramos o tempo para realizar os ciclos de integração contínua e o tempo que ambos os algoritmos gastam para realizar a priorização.

QP2: *Qual é o desempenho da rede LSTM e do algoritmo RF comparado com as abordagens COLEMAN e RETECS?* Esta questão compara ambos os algoritmos com as abordagens baseadas em aprendizado por reforço *COLEMAN* e *RETECS* que podem ser consideradas abordagens estado-da-arte para *TCPCI*. Os resultados são avaliados de acordo com algumas medidas comuns usadas na literatura de teste de regressão.

4.4 SISTEMAS UTILIZADOS

Para permitir a comparação com abordagens existentes foi adotado um conjunto de sistemas (*datasets*) já utilizado na literatura [28, 90, 105]. Foram utilizados os resultados das abordagens *COLEMAN* e *RETECS* reportados no trabalho de Prado Lima e Vergilio[1] [28]. A abordagem *COLEMAN* foi executada com a política *FRRMAB*, e a abordagem *RETECS* com uma rede neural. Os mesmos valores para os orçamentos disponíveis foram também adotados: 10%, 50% e 80%.

Os sistemas utilizados são detalhados na Tabela 4.1 que contém: o nome do sistema, o período dos *logs* de compilação analisados, o total de compilações identificadas e, entre parênteses, o número de compilações incluídas na análise. *Logs* de *build* com alguns problemas foram descartados, por exemplo, extração de informações (*log* de *build* inválido), e aqueles em que os casos de teste não executaram. A quarta coluna mostra o total de falhas encontradas; entre parênteses, o número de compilações nas quais pelo menos um teste falhou. A quinta coluna mostra o número de casos de teste únicos identificados nos *logs* de compilação; entre parênteses, o intervalo entre maior e menor quantidade de casos de teste executados nas compilações. A sexta e sétima colunas apresentam a duração média e o desvio padrão em minutos dos ciclos de integração contínua e o intervalo entre eles.

`Druid`, desenvolvido pela Alibaba, é um *pool* de conexão de banco de dados escrito em Java. `Fastjson`, criado pelo Alibaba, é uma biblioteca Java que pode ser usada como um analisador/gerador JSON rápido para Java. `Deeplearning4j` é uma biblioteca de aprendizado profundo para Java Virtual Machine. `DSpace` é um software de código aberto que oferece

[1]Disponível em: https://osf.io/epnuk

Tabela 4.1: Sistemas utilizados.

Nome	Período	Builds	Falhas	Casos de teste	Duração (min)	Intervalo (min)
Druid	2016/04/24-2016/11/08	286 (168)	270 (71)	2391 (1778-1910)	4.97 ± 10.66	384.76 ± 468.86
Fastjson	2016/04/15-2018/12/04	2710 (2371)	940 (323)	2416 (900-2102)	1.97 ± 0.89	233.22 ± 401.26
Deeplearning4j	2014/02/22-2016/01/01	3410 (483)	777 (323)	117 (1-52)	12.33 ± 14.91	306.05 ± 442.55
DSpace	2013/10/16-2019/01/08	6309 (5673)	13413 (387)	211 (16-136)	11.78 ± 7.03	291.29 ± 411.19
Guava	2014/11/06-2018/12/02	2011 (1689)	7659 (112)	568 (308-512)	62.53 ± 80.31	435.55 ± 464.52
IOF/ROL	2015/02/13-2016/10/25	2392 (2392)	9289 (1627)	1941 (1-707)	1537.27 ± 2018.73	1324.36 ± 291.78
LexisNexis	2018/09/27-2018/11/15	54 (54)	21189 (54)	2662 (2007-2377)	0.8668 ± 0.808	900.367 ± 305.125
OkHttp	2013/03/26-2018/05/30	9919 (6215)	9586 (1408)	289 (2-75)	7.64 ± 5.64	220.17 ± 405.93
Paint Control	2016/01/12-2016/12/20	20711 (20711)	4956 (1980)	1980 (1-74)	424.46 ± 275.90	1417.86 ± 144.97
Retrofit	2013/02/17-2018/11/26	3719 (2711)	611 (125)	206 (5-75)	2.40 ± 1.60	270.86 ± 449.41
ZXing	2014/01/17-2017/04/16	961 (605)	68 (11)	124 (81-123)	13.14 ± 12.37	411.10 ± 465.53

facilidades para o gerenciamento de acervos digitais, utilizados para a implementação de repositórios institucionais. Guava, desenvolvido pela Google, é um conjunto de bibliotecas principais para Java que inclui novos tipos de coleção, APIs/utilitários para simultaneidade, E/S e outros. OkHttp, desenvolvido pela *Square*, é um cliente HTTP e HTTP/2 para aplicativos *Android* e Java. Retrofit, também desenvolvido pela *Square*, é um cliente HTTP de tipo seguro para *Android* e Java. ZXing (*Zebra Crossing*) é uma biblioteca de leitura de código de barras para Java e *Android*. Os sistemas IOF/ROL e Paint Control são sistemas industriais para testar robôs industriais complexos da *ABB Robotic* [90]. LexisNexis é um conjunto de dados industrial para testar sistemas web complexos na empresa LexisNexis [105].

Mais detalhes sobre esses sistemas estão disponíveis no pacote de replicação disponibilizado por Prado Lima e Vergilio [79]. Este pacote contém algumas figuras que ilustram o número de falhas por ciclo para cada *dataset*, o que permite observar a volatilidade dos casos de teste.

4.5 IMPLEMENTAÇÃO E CONFIGURAÇÃO

Para a implementação da rede *LSTM* foi utilizada a biblioteca cuDNN [17], que por meio dos *frameworks* Tensorflow [1] e Keras [20], permite que a rede utilize GPUs. Alguns parâmetros foram definidos com valores padrão, conforme descrito no guia Keras[2] para desenvolvimento CuDNNLSTM. São eles: função de ativação: *tanh*; função de ativação recorrente: sigmóide; *dropout* recorrente: zero; *unroll*: falso; uso de *bias*: verdadeiro; Se os dados de entrada contiverem uma máscara, eles devem ser preenchidos rigorosamente à direita; e a execução rápida é habilitada no contexto externo.

Para a configurar a rede, foram realizados testes usando o sistema Deeplearning4j, treinando diferentes configurações. Primeiro foi definido o modelo como sequencial e realizados experimentos variando o número de épocas no intervalo [10, 100] e o número de neurônios no intervalo [25, 256]. Após esses testes, foram utilizados 100 neurônios para cada camada e 100 épocas para o modelo. Com esses parâmetros selecionados, diferentes configurações de camada foram avaliadas gerando um total de 5 configurações, usando-se em todas elas a camada de ativação *tanh*. As configurações realizadas alternaram além das camadas de ativação *tanh* seguida de uma camada de *dropout* com valor 0,2, considerando como configuração base uma configuração com 3 camadas sendo elas: primeiramente uma camada de ativação *tanh*, seguida de uma *dropout* e outra camada de ativação *tanh*. Seguindo essa configuração, os testes foram realizados usando as configurações para 3, 5, 7, 9 e 11 camadas. No final, a configuração base (3 camadas) foi selecionada devido ao seu menor tempo de priorização e precisão comparável a outras configurações. Os parâmetros da rede: otimizador e *loss*, foram

[2]https://keras.io/api/layers/recurrent_layers/lstm

selecionados o otimizador Adam e a função *loss* erro quadrado médio, para todos os experimentos realizados. Os experimentos foram realizados em 2 plataformas que possibilitaram o uso de GPUs: Google Colab Pro[3] e Fed4Fire[4] (modelo: GV100GL [Tesla V100 PCIe 32 GB]).

Para implementar o algoritmo de regressão *Random Forest*, foi utilizada a ferramenta scikit-learn [75] que utiliza a média das previsões das árvores geradas. Os experimentos com *Random Forest* foram conduzidos na plataforma GENI [9] com a seguinte configuração: Intel(R) Xeon(R) E5-2640 v3 com CPU de 2,60 GHz, 94GB de RAM, rodando Linux Ubuntu 18.04.1 LTS. Para definir os parâmetros, número de estimadores (número de árvores geradas) e W, foram selecionados os sistemas `DeepLearning4J` e `FastJson`, e realizados 10 execuções independentes.

Foram testadas variações de 100, 250 e 500 estimadores e uma variação de 0, 1, 4, 100 para W, sendo que o tamanho 0 representa uma abordagem cumulativa, ou seja, foram utilizados todos os dados históricos de teste. Além disso, antes foram selecionados tamanhos com base na literatura para permitir a comparação com abordagens existentes onde: $W = 1$ (não há dados históricos de teste disponíveis); $W = 4$; e onde $W = 100$ (adotado por Prado Lima e Vergilio [28]). Em relação ao número de estimadores, 500 é considerado o valor padrão na literatura [36]; 100 estimado por Hartshorn [39] como um ótimo começo na fase de ajuste; e 250 para marcar um valor intermediário.

Durante a fase de ajuste, observou-se que $W = 0$ era inviável, pois em alguns sistemas com alto número de casos de teste, o tempo de priorização ultrapassava 60 segundos. O melhor desempenho geral foi obtido usando $W = 4$; Um valor para W igual a 100 teve um bom desempenho apenas para `FastJson`, W igual a 1 teve um bom desempenho apenas para `DeepLearning4J`. Então $W = 4$ foi escolhido. Observou-se que 500 estimadores tiveram o melhor desempenho geral, pontuando muito em ambos os sistemas, especialmente com $W = 4$.

Foram usados os resultados de 10 execuções independentes para o algoritmo *RF* e para a rede *LSTM* para comparativo com as abordagens *COLEMAN* e *RETECS*. Com relação as execuções das abordagens, foram selecionados as 10 primeiras execuções das abordagens de acordo com os resultados obtidos por Prado Lima e Vergilio [28].

4.6 RESULTADOS E ANÁLISE

Nesta seção, os resultados são apresentados e analisados, com o objetivo de responder às nossas questões de pesquisa.

4.6.1 QP1: Aplicabilidade considerando os ciclos de integração contínua

Para responder à QP1, o tempo de *build* de cada conjunto de dados, apresentado na Tabela 4.1, foi comparado com o tempo de execução dos algoritmos *RF* e *LSTM*, apresentado na Tabela 4.2. Nessa tabela, os valores foram obtidos das 10 execuções independentes com cada orçamento de teste de 10%, 50% e 80%. As células em negrito contêm os melhores valores. Com essas informações pode-se concluir se eles conseguem resolver o problema *TCPCI* de forma que o tempo de execução dos testes seja justificável para utilizar a abordagem proposta. Pode-se observar que ambos os algoritmos são aplicáveis a todos os sistemas; o tempo gasto por eles não se sobrepõe ao tempo de execução do ciclo. Ambos possuem um tempo de execução menor que o tempo de construção e intervalo entre os *commits* dos sistemas.

[3]https://colab.research.google.com

[4]https://portal.fed4fire.eu

Observando-se a Tabela 4.2 vê-se que ambos os algoritmos levam apenas alguns segundos para serem executados. Na pior das hipóteses, considerando o desvio padrão, demoram cerca de 7 segundos (*RF* para `Lexisnexis` e *LSTM* para `Druid`). Exceto para `Lexisnexis`, *RF* apresenta os melhores valores de *PT*. A rede *LSTM* demora cerca de 4 vezes mais para ser executada do que o algoritmo *Random Forest*. No entanto, analisando os valores de *NTR* (Tabela 4.3), pode-se ver que este fato não impacta o tempo gasto no ciclo de integração contínua. Os valores *NTR* representam o quanto o tempo de compilação do ambiente de integração continua melhora devido à previsão de falhas antecipada como efeito da priorização dos casos de teste e pela duração das compilações do conjunto de dados. O desempenho de ambos os algoritmos é semelhante. Considerando os 33 casos (11 sistemas x 3 orçamentos), *LSTM* apresenta valores melhores que *RF* em 19 casos (58%), e *RF* apresenta valores melhores que *LSTM* em 14 (42%).

Tabela 4.2: Tempo de priorização (em segundos) e desvio padrão para *LSTM* e *RF*.

	Orçamento de tempo 10%		**Orçamento de tempo 50%**		**Orçamento de tempo 80%**	
Dataset	*LSTM*	*RF*	LSTM	RF	LSTM	RF
`DSpace`	4.8784 ± 0.4940	**1.1088 ± 0.0180**	4.8824 ± 0.4820	**1.1263 ± 0.0190**	5.0206 ± 0.6560	**1.1281 ± 0.0190**
`Druid`	7.1356 ± 0.0160	**1.3113 ± 0.0670**	7.1388 ± 0.0100	**1.3332 ± 0.0400**	7.1390 ± 0.0100	**1.3333 ± 0.0530**
`Fastjson`	5.2117 ± 0.5710	**1.4338 ± 0.0330**	5.2106 ± 0.5510	**1.4378 ± 0.0310**	5.0499 ± 0.5550	**1.3958 ± 0.0280**
`Deeplearning4j`	3.7395 ± 0.0400	**1.0768 ± 0.0440**	3.7532 ± 0.0560	**1.1057 ± 0.0170**	3.7532 ± 0.0640	**1.0945 ± 0.0260**
`Guava`	4.9526 ± 0.0640	**1.1529 ± 0.0220**	4.9923 ± 0.1810	**1.1605 ± 0.0150**	4.9718 ± 0.0830	**1.1450 ± 0.0310**
`IOF/ROL`	4.9038 ± 0.6070	**1.1240 ± 0.0090**	4.8968 ± 0.6010	**1.1177 ± 0.0170**	4.8276 ± 0.5860	**1.1200 ± 0.0090**
`Lexisnexis`	**4.5934 ± 0.1230**	6.8368 ± 0.1420	**4.5309 ± 0.0940**	6.5438 ± 0.6570	**4.5282 ± 0.0910**	6.6674 ± 0.3180
`PaintControl`	4.6137 ± 0.0070	**1.0999 ± 0.0070**	4.6135 ± 0.0070	**1.1114 ± 0.0100**	4.6135 ± 0.0070	**1.1122 ± 0.0110**
`Okhttp`	4.6087 ± 0.0070	**1.1090 ± 0.0130**	4.6081 ± 0.0070	**1.1247 ± 0.0140**	4.6081 ± 0.0070	**1.1257 ± 0.0160**
`Retrofit`	4.5799 ± 0.0080	**1.1019 ± 0.0100**	4.5812 ± 0.0060	**1.1038 ± 0.0190**	4.5821 ± 0.0040	**1.1097 ± 0.0060**
`Zxing`	4.6012 ± 0.5700	**1.1023 ± 0.0280**	4.5854 ± 0.5510	**1.1014 ± 0.0120**	4.6304 ± 0.5930	**1.1026 ± 0.0170**

Nos casos em que o tempo de execução é alto, por exemplo para o conjunto de dados `IOF/ROL`, há um alto impacto no tempo de *build*, como no qual a abordagem pode reduzir esse tempo em aproximadamente 52% com o orçamento de teste de 10% em 68% com o orçamento de 80%, o que resultaria em um tempo de compilação de 1537,27 (≈ 25,6 horas) cair para aproximadamente 768,635 minutos (≈ 12,8 horas) no primeiro caso e aproximadamente 492 minutos (≈ 8,2 horas).

Resposta à QP1: pode-se concluir que *LSTM* e *RF* são aplicáveis para resolver o problema *TCPCI*. Eles exigem, na maioria dos casos, alguns segundos para serem executados, e a priorização realizada é viável em comparação com o tempo de construção. Mas a rede *LSTM* demora quase 4 vezes mais que o *RF* para realizar a priorização, contudo isso não é significativo no tempo de execução do ciclo de integração contínua.

4.6.2 QP2: Comparação com abordagens de aprendizado por reforço da literatura

Para responder à QP2 foram avaliados os valores para *NTR* (Tabela 4.3), *NAPFD* e *APFDc* (Tabela 4.4). Conforme mencionado anteriormente, esses valores (médias ± desvio padrão) foram obtidos para os três orçamentos de teste a partir de 10 execuções, devido a restrições de tempo de treinamento das redes *LSTM*. Devido aos diferentes ambientes de execução, os tempos de priorização não puderam ser utilizados em nossa análise, mas estão presentes na Tabela 4.2 como referência.

Na Tabela 4.4, os valores destacados em negrito com um símbolo "★" denotam que o algoritmo obteve a melhor pontuação dentro do grupo. Nos casos em que os resultados são estatisticamente equivalentes ao melhor, a célula é destacada em cinza. Se não houver diferença

Tabela 4.3: Valores médios e desvio padrão NTR para: *COLEMAN*, *RETECS*, *LSTM* e *RF*.

SUT	NTR			
	RETECS	COLEMAN	*LSTM*	*RF*
	Orçamento de tempo: 10%			
DSpace	0.0102 ± 0.001	**0.0251 ± 0.004**	0.0188 ± 0.001	0.0160 ± 0.000
Druid	0.1973 ± 0.107	0.1599 ± 0.100	0.3772 ± 0.007	**0.3841 ± 0.002**
Fastjson	0.0213 ± 0.008	0.0604 ± 0.024	0.1027 ± 0.002	**0.1077 ± 0.000**
Deeplearning4j	**0.5432 ± 0.019**	0.4663 ± 0.000	0.4624 ± 0.011	0.4748 ± 0.000
Guava	0.0388 ± 0.004	0.0330 ± 0.002	**0.0524 ± 0.002**	0.0509 ± 0.000
IOF/ROL	0.5634 ± 0.012	**0.5710 ± 0.003**	0.5218 ± 0.152	0.4751 ± 0.000
Lexisnexis	0.9861 ± 0.010	0.9961 ± 0.000	**0.9988 ± 0.000**	0.9981 ± 0.000
Paint Control	**0.1139 ± 0.000**	0.1131 ± 0.000	0.0967 ± 0.001	0.0955 ± 0.000
OkHttp	0.0541 ± 0.007	0.0702 ± 0.000	0.0675 ± 0.002	**0.0776 ± 0.000**
Retrofit	0.0074 ± 0.001	0.0073 ± 0.000	0.0074 ± 0.000	**0.0076 ± 0.000**
ZXing	**0.0123 ± 0.000**	0.0037 ± 0.000	0.0106 ± 0.002	0.0079 ± 0.000
	Orçamento de tempo: 50%			
DSpace	0.0219 ± 0.002	**0.0499 ± 0.006**	0.0366 ± 0.002	0.0211 ± 0.000
Druid	0.0982 ± 0.020	**0.4212 ± 0.008**	0.3906 ± 0.005	0.3922 ± 0.002
Fastjson	0.0666 ± 0.026	0.0768 ± 0.028	**0.1171 ± 0.001**	0.1155 ± 0.000
Deeplearning4j	**0.5462 ± 0.010**	0.4624 ± 0.000	0.4795 ± 0.012	0.4863 ± 0.000
Guava	0.6760 ± 0.018	**0.7185 ± 0.003**	0.6658 ± 0.004	0.6641 ± 0.000
IOF/ROL	0.9908 ± 0.006	0.9961 ± 0.000	**0.9988 ± 0.000**	0.9980 ± 0.000
Lexisnexis	0.0410 ± 0.005	0.0419 ± 0.004	**0.0576 ± 0.001**	0.0568 ± 0.000
Paint Control	0.1138 ± 0.001	**0.1223 ± 0.000**	0.0967 ± 0.001	0.0956 ± 0.000
OkHttp	0.1052 ± 0.003	**0.1486 ± 0.000**	0.0712 ± 0.002	0.0792 ± 0.000
Retrofit	0.0138 ± 0.000	**0.0172 ± 0.000**	0.0139 ± 0.001	0.0140 ± 0.000
ZXing	**0.0199 ± 0.001**	0.0110 ± 0.000	0.0171 ± 0.001	0.0184 ± 0.000
	Orçamento de tempo: 80%			
DSpace	0.0266 ± 0.001	**0.0526 ± 0.005**	0.0400 ± 0.001	0.0238 ± 0.000
Druid	0.1313 ± 0.114	**0.4289 ± 0.005**	0.3955 ± 0.010	0.4014 ± 0.002
Fastjson	0.0553 ± 0.014	0.0902 ± 0.026	**0.1207 ± 0.001**	0.1190 ± 0.000
Deeplearning4j	**0.5491 ± 0.006**	0.4047 ± 0.000	0.4828 ± 0.007	0.4880 ± 0.000
Guava	0.6891 ± 0.006	**0.7329 ± 0.002**	0.6842 ± 0.005	0.6830 ± 0.000
IOF/ROL	0.9880 ± 0.006	0.9961 ± 0.000	**0.9988 ± 0.000**	0.9981 ± 0.000
Lexisnexis	0.0336 ± 0.011	0.0557 ± 0.011	**0.0590 ± 0.001**	0.0590 ± 0.000
Paint Control	0.1158 ± 0.001	**0.1203 ± 0.000**	0.0968 ± 0.001	0.0956 ± 0.000
OkHttp	0.1157 ± 0.002	**0.1551 ± 0.000**	0.0721 ± 0.002	0.0800 ± 0.000
Retrofit	0.0145 ± 0.001	**0.0179 ± 0.000**	0.0145 ± 0.000	0.0147 ± 0.000
ZXing	0.0187 ± 0.003	**0.0227 ± 0.000**	0.0175 ± 0.001	0.0184 ± 0.000

estatística, ambos os algoritmos são contados como os melhores em nossa análise. Um "▼" indica que o tamanho do efeito foi insignificante em relação ao melhor valor, enquanto "▽" denota uma magnitude pequena, "△" uma magnitude média e "▲" uma grande magnitude [28]. O tamanho do efeito foi realizado durante os testes *post-hoc*, ou seja, quando há diferença estatística. Pode-se ver em ambas as tabelas que a maioria dos valores tem a grande magnitude (símbolo "▲").

Analisando-se os valores do *NAPFD* (Tabela 4.4), tanto *RF* quanto *LSTM* apresentam bom desempenho no orçamento de teste de 10%, atingindo, os melhores valores ou estatisticamente equivalentes, respectivamente em 9 e 7 sistemas (de 11), superando *COLEMAN* e *RETECS*, que são os melhores, respectivamente, em 5 e 3 sistemas.

Ao analisar o orçamento de teste de 50%, nota-se que *RF* apresenta os melhores valores em 8 sistemas, *COLEMAN* e *LSTM* têm a mesma pontuação, atingindo os melhores valores em 6 sistemas, e *RETECS* apenas em 2. Esses resultados mostram que para orçamentos médios e baixos, considerando apenas os valores do *NAPFD*, o algoritmo de *RF* supera os demais. Por outro lado, para o orçamento de 80%, *COLEMAN* tem o melhor desempenho em 9 sistemas, enquanto *RF* em 6, *LSTM* em 5 e *RETECS* em 3. Considerando o *NAPFD*, todos os sistemas e orçamentos de teste, *RF* atingiu os melhores valores, ou estatisticamente equivalentes aos melhores, em 23 casos (de 33, ≈70%), enquanto *COLEMAN* e *LSTM* apresentaram os melhores valores respectivamente, em 20 (≈61%) e 18 (≈55%) casos, e *RETECS* apenas em 8 (≈24%).

Em relação aos valores de *APFDc* (Tabela 4.4), e considerando o orçamento de teste 10%, *RF* e *LSTM* têm o melhor desempenho em 7 sistemas, seguidos de *RETECS* em 6, e *COLEMAN* em 4. *RF* também apresentou o melhor desempenho para o orçamento de teste 50% em 7 sistemas, seguido por *RETECS* e *COLEMAN* que foram os melhores em 6 sistemas. *LSTM* foi o melhor em 5. Isso mostra que *RETECS* tem melhor desempenho quando o custo dos casos

de teste é considerado. Novamente, para o orçamento de teste 80%, *COLEMAN* obtém a melhor pontuação em 9 sistemas, seguido por *RF* em 5, *LSTM* e *RETECS* em 4. Para *APFDc*, todos os sistemas e orçamentos, *RF* e *COLEMAN* atingiu os melhores valores, ou estatisticamente equivalentes aos melhores, em 19 casos (de 33, ≈58%), enquanto *RETECS* em 18 (≈55%) e *LSTM* em 16 (≈48%). Os algoritmos têm um desempenho geral semelhante.

Tabela 4.4: Valores médios e desvio padrão NAPFD e APFDc para: *RETECS*, *COLEMAN*, *LSTM* e *RF*.

Dataset	NAPFD				APFDc			
	RETECS	*COLEMAN*	*LSTM*	*RF*	*RETECS*	*COLEMAN*	*LSTM*	*RF*
				Orçamento de tempo: 10%				
DSpace	0.9408 ± 0.0000 ▲	**0.9496 ± 0.0040** ★	0.9473 ± 0.0010 ▼	0.9485 ± 0.0000 ▼	0.9456 ± 0.0000 ▲	**0.9513 ± 0.0040** ★	0.9469 ± 0.0010 ▲	0.9473 ± 0.0000 ▼
Druid	0.6919 ± 0.1080 ▲	0.6718 ± 0.0540 ▲	0.9161 ± 0.0050 ▲	**0.9284 ± 0.0020** ★	0.8089 ± 0.0710 ▲	0.6754 ± 0.0560 ▲	0.9170 ± 0.0040 ▲	**0.9307 ± 0.0010** ★
Fastjson	0.8716 ± 0.0050 ▲	0.9014 ± 0.0220 ▲	0.9491 ± 0.0010 ▲	**0.9572 ± 0.0000** ★	0.8806 ± 0.0040 ▲	0.9009 ± 0.0220 ▲	0.9493 ± 0.0010 ▲	**0.9572 ± 0.0000** ★
Deeplearning4j	0.6729 ± 0.0150 ▲	0.7716 ± 0.0000 ▲	0.7652 ± 0.0040 ▲	**0.7771 ± 0.0000** ★	**0.8109 ± 0.0100** ★	0.7774 ± 0.0010 ▲	0.7657 ± 0.0030 ▲	0.7784 ± 0.0000 ▲
Guava	0.9578 ± 0.0040 ▲	0.9584 ± 0.0010 ▲	**0.9800 ± 0.0010** ★	0.9796 ± 0.0000 ▽	0.9770 ± 0.0030 ▲	0.9580 ± 0.0010 ▲	**0.9803 ± 0.0010** ★	0.9798 ± 0.0000 ▽
IOF/ROL	**0.3779 ± 0.0030** ★	0.3671 ± 0.0010 ▲	0.3557 ± 0.0010 ▲	0.3577 ± 0.0000 ▲	**0.3820 ± 0.0030** ★	0.3701 ± 0.0010 ▲	0.3558 ± 0.0010 ▲	0.3576 ± 0.0000 ▲
Lexisnexis	0.0938 ± 0.0600 ▲	0.1437 ± 0.0010 ▲	0.3457 ± 0.0040 ▲	**0.3686 ± 0.0000** ★	0.0959 ± 0.0580 ▲	0.1431 ± 0.0010 ▲	0.3424 ± 0.0040 ▲	**0.3650 ± 0.0000** ★
Paint Control	**0.9077 ± 0.0000** ★	0.9076 ± 0.0000 ▲	0.9074 ± 0.0000 ▲	0.9074 ± 0.0000 ▲	**0.9081 ± 0.0000** ★	0.9080 ± 0.0000 ▲	0.9074 ± 0.0000 ▲	0.9074 ± 0.0000 ▲
OkHttp	0.8072 ± 0.0060 ▲	0.8407 ± 0.0000 ▲	0.8356 ± 0.0010 ▲	**0.8513 ± 0.0000** ★	0.8268 ± 0.0060 ▲	0.8425 ± 0.0000 ▲	0.8347 ± 0.0010 ▲	**0.8482 ± 0.0000** ★
Retrofit	0.9619 ± 0.0020 ▲	0.9642 ± 0.0000 ▲	0.9656 ± 0.0000 ▲	**0.9659 ± 0.0000** ★	**0.9672 ± 0.001** ★	0.9646 ± 0.000 ▲	**0.9655 ± 0.001** ★	0.9648 ± 0.000 △
ZXing	0.9855 ± 0.0000 ▽	0.9828 ± 0.0000 ▲	**0.9864 ± 0.0020** ★	0.9860 ± 0.0000 ▽	**0.9893 ± 0.0000** ★	0.9835 ± 0.0000 ▲	0.9860 ± 0.0010 ▲	0.9855 ± 0.0000 ▲
				Orçamento de tempo: 50%				
DSpace	0.9487 ± 0.0010 ▲	**0.9766 ± 0.0080** ★	0.9665 ± 0.0010 ▲	0.9495 ± 0.0000 ▲	0.9615 ± 0.0010 ▲	**0.9767 ± 0.0090** ★	0.9666 ± 0.0010 ▲	0.9485 ± 0.0000 ▲
Druid	0.6052 ± 0.0080 ▲	**0.9691 ± 0.0080** ★	0.9359 ± 0.0060 ▲	0.9417 ± 0.0020 ▲	0.7426 ± 0.0360 ▲	**0.9770 ± 0.0090** ★	0.9370 ± 0.0060 ▲	0.9412 ± 0.0010 ▲
Fastjson	0.8936 ± 0.0170 ▲	0.9171 ± 0.0340 ▲	**0.9636 ± 0.0010** ★	0.9632 ± 0.0000 ▲	0.9423 ± 0.0210 ▲	0.9191 ± 0.0330 ▲	0.9644 ± 0.0010 ▲	**0.9654 ± 0.0000** ★
Deeplearning4j	0.6559 ± 0.0160 ▲	0.8200 ± 0.0000 ▲	0.8403 ± 0.0050 ▲	**0.8501 ± 0.0000** ★	**0.8432 ± 0.0100** ★	0.8135 ± 0.0010 ▲	0.8295 ± 0.0060 ▲	0.8372 ± 0.0000 △
Guava	0.4995 ± 0.0070 ▲	**0.5192 ± 0.0020** ★	0.4794 ± 0.0030 ▲	0.4816 ± 0.0000 ▲	0.5016 ± 0.0070 ▲	**0.5226 ± 0.0020** ★	0.4797 ± 0.0030 ▲	0.4815 ± 0.0000 ▲
IOF/ROL	0.2706 ± 0.1340 ▲	0.5617 ± 0.0080 ▲	0.6251 ± 0.0020 ▲	**0.6487 ± 0.0000** ★	0.2807 ± 0.1250 ▲	0.5487 ± 0.0080 ▲	0.6105 ± 0.0020 ▲	**0.6328 ± 0.0000** ★
Lexisnexis	0.9595 ± 0.0040 ▲	0.9668 ± 0.0050 ▲	**0.9849 ± 0.0010** ★	0.9834 ± 0.0000 ▲	0.9828 ± 0.0020 ▲	0.9665 ± 0.0060 ▲	**0.9851 ± 0.0010** ★	0.9840 ± 0.0000 ▲
Paint Control	0.9138 ± 0.0000 ▲	**0.9150 ± 0.0000** ★	0.9074 ± 0.0000 ▲	0.9074 ± 0.0000 ▲	0.9139 ± 0.0000 ▲	**0.9162 ± 0.0000** ★	0.9075 ± 0.0000 ▲	0.9074 ± 0.0000 ▲
OkHttp	0.8433 ± 0.0030 ▲	**0.9316 ± 0.0000** ★	0.8395 ± 0.0020 ▲	0.8534 ± 0.0000 ▲	0.8863 ± 0.0020 ▲	**0.9246 ± 0.0000** ★	0.8386 ± 0.0020 ▲	0.8500 ± 0.0000 ▲
Retrofit	0.9714 ± 0.0010 ▲	**0.9893 ± 0.0000** ★	0.9844 ± 0.0010 ▲	0.9847 ± 0.0000 ▲	0.9775 ± 0.0010 ▲	**0.9885 ± 0.0000** ★	0.9833 ± 0.0010 ▲	0.9838 ± 0.0000 ▲
ZXing	0.9880 ± 0.0010 ▲	0.9857 ± 0.0000 ▲	0.9920 ± 0.0010 ▲	**0.9946 ± 0.0000** ★	**0.9954 ± 0.0010** ★	0.9869 ± 0.0000 ▲	0.9919 ± 0.0010 ▲	0.9946 ± 0.0000 ▲
				Orçamento de tempo: 80%				
DSpace	0.9505 ± 0.0010 ▲	**0.9825 ± 0.0070** ★	0.9717 ± 0.0010 ▲	0.9501 ± 0.0000 ▲	0.9636 ± 0.0010 ▲	**0.9810 ± 0.0080** ★	0.9714 ± 0.0010 ▲	0.9494 ± 0.0000 ▲
Druid	0.6683 ± 0.1100 ▲	**0.9820 ± 0.0040** ★	0.9431 ± 0.0110 ▲	0.9584 ± 0.0020 ▲	0.6957 ± 0.1080 ▲	**0.9902 ± 0.0050** ★	0.9447 ± 0.0110 ▲	0.9512 ± 0.0010 ▲
Fastjson	0.8925 ± 0.0110 ▲	0.9296 ± 0.0340 ▲	**0.9687 ± 0.0010** ★	0.9659 ± 0.0000 ▲	0.9155 ± 0.0110 ▲	0.9320 ± 0.0320 ▲	0.9690 ± 0.0010 △	**0.9693 ± 0.0000** ★
Deeplearning4j	0.6620 ± 0.0190 ▲	0.8641 ± 0.0010 ▲	0.8896 ± 0.0020 ▲	**0.8964 ± 0.0010** ★	0.8563 ± 0.0120 ▲	0.7991 ± 0.0010 ▲	0.8576 ± 0.0040 ▲	**0.8613 ± 0.0010** ★
Guava	0.5316 ± 0.0050 ▲	**0.5679 ± 0.0010** ★	0.5224 ± 0.0030 ▲	0.5218 ± 0.0000 ▲	0.5334 ± 0.0040 ▲	**0.5700 ± 0.0010** ★	0.5225 ± 0.0030 ▲	0.5217 ± 0.0000 ▲
IOF/ROL	0.3043 ± 0.1170 ▲	0.7036 ± 0.0030 ▲	0.6898 ± 0.0030 ▲	**0.7142 ± 0.0000** ★	0.3181 ± 0.0920 ▲	0.6795 ± 0.0030 ▲	0.6712 ± 0.0030 ▲	**0.6939 ± 0.0000** ★
Lexisnexis	0.9571 ± 0.0080 ▲	0.9842 ± 0.0150 ▼	0.9862 ± 0.0010 △	**0.9868 ± 0.0000** ★	0.9683 ± 0.0110 ▲	0.9826 ± 0.0150 ▼	0.9864 ± 0.0010 △	**0.9871 ± 0.0000** ★
Paint Control	0.9160 ± 0.0000 ▲	**0.9172 ± 0.0000** ★	0.9074 ± 0.0000 ▲	0.9074 ± 0.0000 ▲	0.9157 ± 0.0000 ▲	**0.9177 ± 0.0000** ★	0.9074 ± 0.0000 ▲	0.9074 ± 0.0000 ▲
OkHttp	0.8564 ± 0.0050 ▲	**0.9478 ± 0.0000** ★	0.8407 ± 0.0020 ▲	0.8545 ± 0.0000 ▲	0.8979 ± 0.0030 ▲	**0.9362 ± 0.0000** ★	0.8397 ± 0.0020 ▲	0.8509 ± 0.0000 ▲
Retrofit	0.9747 ± 0.0030 ▲	**0.9916 ± 0.0000** ★	0.9873 ± 0.0010 ▲	0.9873 ± 0.0000 ▲	0.9809 ± 0.0030 ▲	**0.9903 ± 0.0000** ★	0.9858 ± 0.0010 ▲	0.9862 ± 0.0000 ▲
ZXing	0.9879 ± 0.0000 ▲	**0.9996 ± 0.0000** ★	0.9927 ± 0.0010 ▲	0.9953 ± 0.0000 ▲	0.9945 ± 0.0020 ▲	**0.9996 ± 0.0000** ★	0.9926 ± 0.0010 ▲	0.9952 ± 0.0000 ▲

Por fim, analisando-se os valores de *NTR* (Tabela 4.3), pode-se observar que *RF* possui os melhores valores em 4 sistemas para o orçamento de teste 10% e o pior desempenho para os orçamentos de teste de 50% e 80 % (0 sistemas). *COLEMAN* atingiu os melhores valores para os orçamentos de teste de 50% e 80%, respectivamente, em 6 e 7 sistemas. Mas apresenta um baixo desempenho no orçamento de 10% (2 casos).

Resposta à QP2: Pode-se concluir que o algoritmo *RF* supera os demais algoritmos para orçamentos de teste mais restritivos e médios, apresentando os melhores valores de *NAPFD* e *APFDc*. *COLEMAN* supera os outros algoritmos para o orçamento de teste menos restritivo. A rede *LSTM* tem bom desempenho para o orçamento de 10%, e *RETECS* quando se considera *APFDc*, para orçamentos menores e médios.

4.6.3 Discussão dos resultados

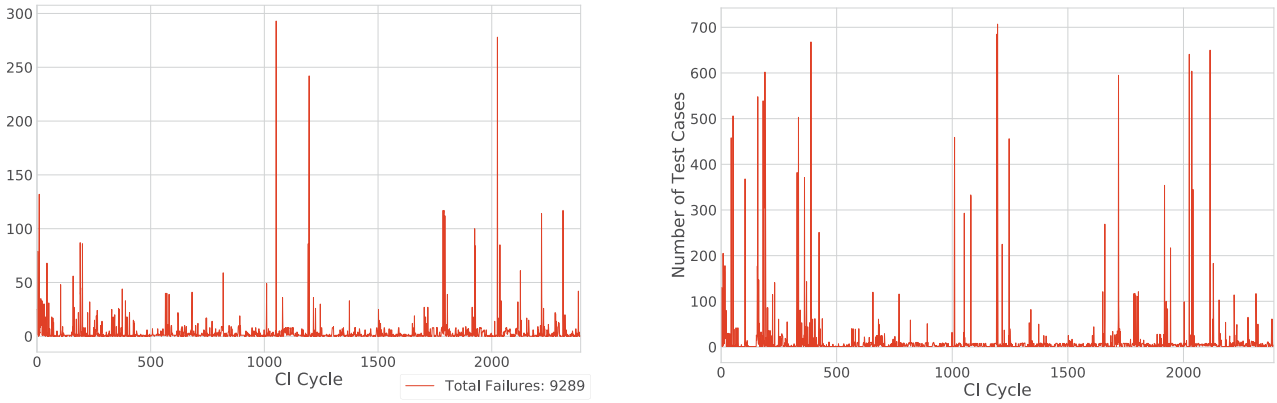
Uso da rede *LSTM*: analisando os resultados descritos na última seção observa-se que a rede *LSTM* é mais adequada para ambientes de desenvolvimento com um orçamento de teste pequeno. Para orçamentos de teste menos restritivos, *COLEMAN* e *RF* superam a rede *LSTM* com exceção de 3 sistemas: `Fastjson`, `DSpace` e `LexisNexis`; para estes a rede *LSTM* apresentou os melhores resultados ou equivalentes para predições em todos os orçamento de teste. Em relação

ao `Fastjson` e ao `DSpace`, ambos os sistemas apresentam baixa distribuição de falhas, e a volatilidade dos casos de teste não tem influência ou correlação com os casos de teste com falha. Considerando o conjunto de dados `LexisNexis`, os casos de teste com falha são distribuídos em muitos testes, e isso contribui para um melhor desempenho na parcela de orçamento de teste maiores (menos restritivos).

Por outro lado, a rede *LSTM* e os demais algoritmos tiveram baixo desempenho no conjunto de dados `IOF/ROL`. A Figura 4.5 ilustra, para este sistema, a volatilidade do caso de teste e as falhas por ciclo juntamente com o ciclos de integração contínua[5].

Como pode-se observar, a volatilidade dos casos de teste é alta, assim como o número de testes reprovados. A alta volatilidade dos casos de teste pode estar associada a uma estratégia de pré-commit, por exemplo, seleção de casos de teste. Consequentemente, os algoritmos não possuem informações históricas suficientes para fazer previsões precisas; o baixo desempenho da rede *LSTM* pode estar relacionado com uma situação de *esquecimento catastrófico* (do inglês: *catastrophic forgetting*).

Outra consideração é que um número baixo de casos de teste por ciclo pode ser a causa de uma pontuação baixa no orçamento de teste de 10%, por exemplo, no conjunto de dados `OkHttp`. Nos casos em que há baixa volatilidade e baixo número de casos de teste, como no conjunto de dados `ZXing`, uma pontuação alta foi alcançada provavelmente devido à relevância dos dados históricos à medida que ocorrem os ciclos de integração contínua.



(a) Falhas por ciclo.

(b) Volatilidade de casos de teste.

Figura 4.5: Visão geral do dataset IOF/ROL (retirado de Prado Lima e Vergilio [79]).

Uso do algoritmo *RF*: a análise na fase de treinamento mostra que o tamanho da janela deslizante W tem um impacto significativo no desempenho geral do modelo. Outra observação é que o tempo de priorização está diretamente correlacionado com o número de estimadores dados. O tempo de execução dobra para cada aproximadamente 250 estimadores.

RF obteve altos valores de *NAPFD* e *APFDc* para todos os orçamentos de teste nos sistemas: `Fastjson`, `Druid`, `LexisNexis` e `Deeplearning4j`. Para `Deeplearning4j`, *RF* provavelmente obteve uma boa pontuação devido à baixa quantidade de casos de teste. Mas *RF* não obteve valores elevados de *NTR*. Na maioria dos casos a diferença para os melhores valores de *NTR* obtidos pelos outros algoritmos é pequena, assim como para o conjunto de dados `Druid`. Mas para o conjunto de dados `Okhttp` esta diferença é significativa. Em conclusão, o *RF* é mais adequado para orçamentos de teste baixos e médios (10% e 50%).

O *RF* tem o melhor desempenho para orçamentos de teste mais restritivos e possui menos tempo de priorização. Essa suposição pode diferir no caso de um conjunto de dados

[5]Veja as características dos conjuntos de dados em nosso material complementar [86]

maior com dados de teste históricos mais ricos, no qual a rede neural teria dados suficientes para aprender com mais precisão.

Uso de *COLEMAN* e *RETECS*: *COLEMAN* apresentou o melhor desempenho para orçamentos de teste menos restritivos. Isso provavelmente acontece devido a uma melhor adaptação nestes casos. *RETECS* teve bom desempenho apenas para *APFDc* para orçamentos baixos e médios.

4.7 AMEAÇAS À VALIDADE

Abaixo são apresentadas possíveis ameaças à validade dos resultados, de acordo com a taxonomia de Wohlin et al. [100].

Validade Interna: os ambientes de execução e configurações de parâmetros podem ser considerados uma ameaça. Como a configuração do algoritmo foi selecionada por experimentação, nem todas as possibilidades foram contempladas, portanto, uma melhor configuração poderia levar a melhores resultados. O uso da estrutura CuDNNLSTM restringe a configuração das funções de ativação da camada oculta. Nos casos em que seu uso não é obrigatório e os recursos de hardware são abundantes, configurações mais personalizadas das funções de ativação podem gerar resultados melhores.

Validade Externa: apenas onze sistemas foram usados. Assim, os resultados não podem ser generalizados pois o conjunto pode não ser representativo o suficiente para avaliar a escalabilidade da abordagem. Experimentos com outros sistemas devem ser realizados. Para tanto, acreditamos que o estudo pode ser replicado, utilizando os dados presentes em nosso repositório [86] e utilizando a descrição de configuração para *LSTM* e *RF*.

Validade de Conclusão: a aleatoriedade dos algoritmos de *ML* é uma ameaça. Para minimizar essa ameaça, foram selecionados para a análise os resultados de 10 execuções. Outra ameaça está relacionada aos testes estatísticos utilizados. Para minimizar essa ameaça, foram utilizados os testes: Kruskal-Wallis, Mann-Whitney e Friedman, estes comumente adotados para algoritmos não determinísticos em problemas de engenharia de software. Por fim, os múltiplos ambientes para execuções também podem induzir alguma aleatoriedade devido a diferentes configurações de hardware.

4.8 CONSIDERAÇÕES FINAIS

Este capítulo introduziu uma abordagem para *TCPCI* baseada no histórico de falhas dos casos de teste. A abordagem utiliza o método da janela deslizante (*SW*) para aplicação das técnicas de regressão baseadas em *ML*, que permite lidar adequadamente com algumas particularidades dos ambientes de integração contínua: dilema *EvE* e volatilidade dos casos de teste. A abordagem é leve, requer apenas os resultados dos testes anteriores; é livre de modelos e independente da linguagem de programação; e não requer análise de código.

A abordagem foi avaliada usando o algoritmo de regressão *RF* e uma rede *LSTM* com onze sistemas e três orçamentos de teste. Os resultados mostram que ambos, se aplicam ao problema *TCPCI* em relação ao tempo de construção e aos ciclos de integração contínua. Foi realizada uma comparação com as abordagens de aprendizado por reforço *COLEMAN* e *RETECS*, utilizando-se algumas medidas comuns do teste de regressão. Em relação à métrica *NAPFD*, todos os sistemas e orçamentos de teste, *RF* atingiu os melhores valores, ou estatisticamente equivalentes, em 23 casos (de 33, ≈70%), enquanto *COLEMAN* e *LSTM* apresentaram os melhores valores respectivamente, em 20 (≈61%) e 18 (≈55%) casos, e *RETECS* em apenas 8

(≈24%). Para *APFDc*, *RF* e *COLEMAN* alcançaram os melhores valores, ou equivalentes, em 19 casos (de 33, ≈58%), enquanto *RETECS* em 18 (≈55%) e *LSTM* em 16 (≈48%).

Foram apresentadas algumas implicações práticas para o uso dos algoritmos e abordagens avaliadas. O *RF* supera os outros algoritmos para orçamentos de teste baixos e médios (orçamentos de teste 10% e 50%). *COLEMAN* supera os outros algoritmos para o orçamento menos restritivo (de 80%). A rede *LSTM* tem bom desempenho para o orçamento de 10%, e *RETECS* quando a função custo (*APFDc*) é considerada, para orçamentos baixos e médios.

No próximo capítulo são descritos os resultados de um experimento que avalia a abordagem proposta utilizando outras informações associadas aos casos de teste. A ideia é verificar se o uso de informações de contexto contribui para aumentar a performance dos algoritmos.

5 INVESTIGANDO O USO DE INFORMAÇÕES DE CONTEXTO

Como demonstrado no capitulo anterior, as abordagens baseadas em histórico são ferramentas que conseguem resolver o problema de *TCPCI* com sucesso considerando os desafios e peculiaridades do ambiente, sendo o algoritmo *RF* melhor para casos com orçamentos de teste baixo e médio, enquanto a abordagem *COLEMAN* é mais adequada para casos em que o orçamento de teste é menos restritivo (de 80%). Contudo o experimento descrito no capítulo anterior para avaliar a abordagem proposta não utilizou informações de contexto e atributos do sistema.

Neste capítulo a abordagem proposta é avaliada com informações de contexto, sendo estas informações utilizadas como o estado dos algoritmos contextuais. Estas informações são utilizadas na tentativa de melhoria da predição da priorização. Para isso foi implementada uma versão do algoritmo *RF* (apontado como o melhor nos experimentos descritos no capítulo anterior), chamada *CRF* (*Contextual Random Forest*). Os resultados obtidos com esta nova versão são comparados com uma versão da abordagem *COLEMAN* que também utiliza informações do contexto de teste e a técnica de *Contextul Multi-Armed Bandit* (*CMAB*).

5.1 INFORMAÇÃO DE CONTEXTO UTILIZADA

Existem muitas informações relacionadas ao teste de software, ao produto sendo testado, e ao ambiente de desenvolvimento que podem impactar na probabilidade de um caso de teste falhar. Neste trabalho foram consideradas algumas características (do inglês: *features*) dos casos de teste. Abaixo são apresentadas as *features* utilizadas.

1. **Duração**: O tempo gasto por um caso de teste para executar.

2. **Número de erros**: O número de assertivas de teste que detectaram erros durante o teste. Assume-se que um grande número de assertivas de teste com erros tem maior probabilidade de detectar falhas diferentes.

3. **Número de assertivas executadas**: É o número de assertivas executadas durante o teste, considerando que algumas assertivas não são executados devido a anteriores terem falhado.

4. **Linhas do código fonte**: Este recurso conta a quantidade de linhas de código excluindo comentários e linhas vazias.

5. **Complexidade ciclomática de *McCabe***: Esse recurso considera a complexidade de *McCabe* dos casos de teste avaliados. Alta complexidade pode estar relacionada a um caso de teste mais elaborado, aumentando as chances de falhar.

6. **Idade do caso de teste**: Este recurso mede quanto tempo o caso de teste existe e é dado por um número que é incrementado para cada novo ciclo de integração contínua em que o caso de teste aparece.

7. **Alteração no caso de teste**: Considera se um caso de teste foi alterado. Se um caso de teste for alterado de um *commit* para outro, há uma alta probabilidade de que a alteração tenha sido realizada porque alguma parte do código fonte do *software* precisa ser testada.

Para identificar as *features* mais relevantes, para cada sistema, foi executada a Análise de Componentes Principais (do inglês: *Principal Component Analysis*, PCA) [94] sobre as *features*, reduzindo os dados até o número de componentes que explicam 95% da variância como descrito na Figura 5.1. O PCA é um dos métodos populares usados para redução de dimensionalidade de um conjunto de *features*. As variáveis foram padronizadas pois não são medidas na mesma escala e o método PCA é sensível a mudanças de escala [63]. Para todos os sistemas, foi realizada uma comparação por pares. No entanto, para cada sistema, o número de componentes escolhidos foi diferente.

Por esta razão, foram determinados pares de *features* colineares com base no coeficiente de correlação de Pearson. Para cada par acima do limite especificado (em valor absoluto), identificou-se uma das variáveis a serem removidas, privilegiando aquelas escolhidas pelo método de PCA. As *features* escolhidas são: idade do caso de teste, linhas do código fonte, número de erros, alteração no caso de teste.

Como foram selecionados diferentes componentes em cada sistema, todas as *features* foram organizadas em grupos para avaliar um sistema em diferentes aspectos, principalmente em relação à seleção de *features*. Os grupos de *features* são apresentados na Tabela 5.1.

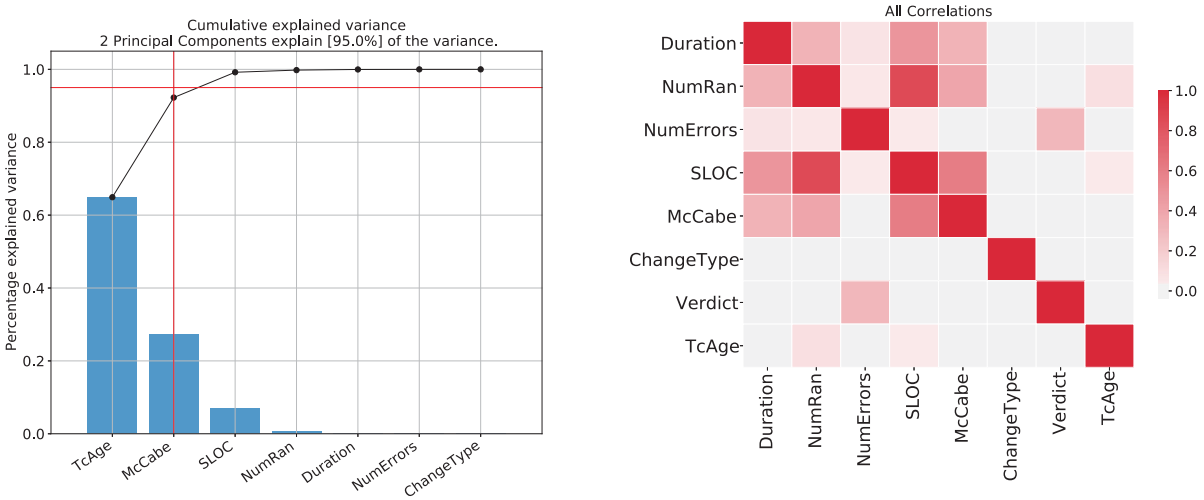


Figura 5.1: Análise PCA (esquerda) e coeficiente de correlação para as características do sistema `DSpace`.

As *features* foram agrupadas em 6 grupos de acordo com suas características, sendo esses grupos: Todas as *Features* (TF), Tempo de Execução (TE), Tamanho do Programa (TP), Complexidade do caso de teste (CCT), Evolução do caso de teste (ECT) e Seleção de *Features* (SF), descritos na Tabela 5.1. O grupo SF considera apenas as *features* escolhidas usando PCA e coeficiente de correlação de Pearson.

5.2 AVALIAÇÃO

As métricas de avaliação do desempenho da priorização dos algoritmos selecionados, são as mesmas métricas no Capítulo 4, são elas: *NAPFD*, *APFDc*, *NTR* e *PT*, com a adição das métricas: *Fault-Detection Effectiveness* (*FDE*) e *RFTC*. Os testes estatísticos aplicados foram também os mesmos, sendo eles: Kruskal-Wallis, Mann-Whitney e Friedman. Para avaliação da diferença entre os grupos, foi utilizada a métrica de $\hat{A}_{12}$ Vargha e Delaney.

Buscando uma melhor avaliação dos algoritmos baseados em contexto e uma melhor comparação com sua contraparte não contextual, é proposta a métrica: Porcentagem do conjunto de casos de teste programada (do inglês: Test Case Set Percentage Scheduled, TCPS)

Tabela 5.1: Grupos de *features*.

Grupo	***Features* presentes**
TF	Duração, Número de erros, Número de assertivas executadas, Linhas do código fonte, Complexidade ciclomática de *McCabe*, Idade do caso de teste e Alteração no caso de teste.
TE	Duração e Número de erros.
TP	Linhas do código fonte.
CCT	Número de assertivas executadas e Complexidade ciclomática de *McCabe*.
ECT	Idade do caso de teste e Alteração no caso de teste.
SF	Número de erros, Linhas do código fonte, Idade do caso de teste e Alteração no caso de teste.

Para ser aplicável em ambientes de integração contínua, uma abordagem precisa fornecer soluções com alta qualidade em um tempo viável e utilizando adequadamente o orçamento de teste definido. Quando um orçamento de teste é definido, deseja-se que o máximo de testes de T' possa ser executado no tempo disponível. TCPS avalia a porcentagem do conjunto de testes executada em relação ao tamanho total do conjunto de testes. Uma porcentagem maior de casos de teste executados tem mais chance de revelar falhas.

5.3 QUESTÕES DE PESQUISA

O principal objetivo desse experimento é avaliar o impacto das variáveis de contexto quanto utilizadas em algoritmos já conceituados na literatura como o *COLEMAN* e *RF*, utilizando métricas de qualidade descritas na Seção 5.2. De acordo com os objetivos estabelecidos, as seguintes questões de pesquisa foram formuladas:

QP1: *Qual é o melhor grupo de informações de contexto para os algoritmos selecionados?* Esta questão de pesquisa investiga os grupos de informações de contexto (*features*) descritos na Tabela 5.1 a fim de avaliar se há um grupo o qual o desempenho da priorização se destaca.

QP2: *Qual é o desempenho dos algoritmos que utilizam informação de contexto?* Esta questão de pesquisa busca avaliar o desempenho de priorização de ambos algoritmos contextuais: *CRF* e *CMAB*.

QP3: *Qual é o desempenho dos algoritmos contextuais com relação às abordagens não contextuais?* Esta questão tem como objetivo comparar os algoritmos contextuais com as abordagens não contextuais e por fim avaliar se os algoritmos contextuais utilizando o melhor grupo de *features* obtêm uma melhoria de desempenho para priorização.

QP4: *Qual o desempenho de todos os algoritmos analisados?* Esta questão tem como objetivo analisar o desempenho de priorização dos algoritmos *CRF* e *CMAB*, e também os algoritmos utilizados nos experimentos do capítulo anterior.

5.4 SISTEMAS UTILIZADOS

Com o objetivo de proporcionar uma comparação justa e manter a reprodutibilidade da pesquisa, foram selecionados os sistemas já utilizados por Prado Lima e Vergilio [28], mencionados no

capítulo anterior. Para extrair as *features*, foi desenvolvida uma ferramenta[1] que itera sobre os *commits* sob análise para obter o contexto circundante (*features*) de cada caso de teste. Durante a extração de *features*, alguns ruídos foram identificados tais qual problemas no *git checkout* para alguns *commits* ocorreu erro fatal, e com *commits* de *merge* que não possuem alterações, nessas situações os casos de teste foram preenchidos com informações anteriores. Foi realizada uma validação manual para garantir a integridade dos dados. Além dessas dificuldades, o código fonte de alguns sistemas utilizados por Prado Lima e Vergilio deixou de ser público. Por esses motivos, foi possível extrair os recursos de apenas seis sistemas, sendo eles: `Dspace`, `Druid`, `Fastjson`, `OkHttp`, `Retrofit` e `ZXing` (mais informações presentes na Tabela 4.1).

5.5 IMPLEMENTAÇÃO E CONFIGURAÇÃO

Para dar inicio a exploração do uso de *features* no contexto de *TCPCI*, foi realizada a implementação de uma versão do algoritmo *RF* utilizando informações de contexto, que será chamada de *Contextual Random Forest* (*CRF*).

Os dados extraídos são compilados em uma matriz contendo as informações contextuais separadas em colunas enquanto os casos de teste foram separados nas linhas. Para cada *commit*, foram armazenados os casos de teste presentes no histórico assim como na versão não contextual da abordagem.

5.5.1 *Contextual Random Forest*

O input para o algoritmo *CRF* se dá por meio da matriz com o grupo de *features* escolhido, separando primeiramente as *features* a serem analisadas de seu "veredicto", ou seja, se o caso de teste falha. Em seguida as *features* são separadas de acordo com o tamanho da *SW*, utilizando até o penúltimo *commit* anterior para o treinamento da rede e o ultimo *commit* para a predição do *commit* atual. A Figura 5.2 mostra um exemplo de separação dos dados utilizando o grupo de *features* TE para uma $SW = 4$, durante a análise do *commit* $c = 5$, sendo a parte superior da imagem a representação da matriz com todos os dados e a parte inferior a separação realizada.

Durante a realização do *tunning* do algoritmo, foi realizada a execução de múltiplas configurações com os grupos de *features* extraídas e utilizando os sistemas: `Druid` e `Fastjson`. As configurações variaram de modificações no tamanho da *SW*, variando a mesma com valores entre 4 a 100, bem como variando os estimadores do *CRF* de 250 a 500. A configuração que apresentou melhores resultados foi a configuração com a *SW* de tamanho 40 com 500 estimadores.

Após a definição da melhor configuração para o algoritmo, o mesmo foi executado para todos os *datasets* disponíveis com o objetivo de extrair o melhor grupo de *features* para a comparação com a abordagem não contextual do algoritmo *RF*.

5.5.2 *Contextual Multi-Armed Bandit*

Com relação aos experimentos conduzidos para o algoritmo *CMAB*, foram avaliadas duas políticas encontradas na literatura: a *Linear Upper Confidence Bound* (*LinUCB*) e a *SW-LinUCB*. Foram realizadas comparações empíricas com ambas, utilizando os grupos apresentados na Seção 5.1, a fim de selecionar a política a ser utilizada nas comparações com as abordagens não contextuais e com o algoritmo *CRF*.

[1] https://github.com/jacksonpradolima/mining-repositories.

Entrada considerando uma SW de tamanho 4 e analisando o commit 5

	Duração	N erros	Veredicto
T1c1	12	2	1
T2c1	31	0	0
T3c1	2	5	0
…	…	…	…
Txc4	3	2	1

Entrada para fase de treinamento

Entrada usada para o treinamento do modelo

	Duração	N erros
T1c1	12	2
T2c1	31	0
T3c1	2	5
…	…	…
Tnc3	40	12

Entrada usada para treinamento do modelo

	Veredicto
T1c1	1
T2c1	0
T3c1	0
…	…
Tnc3	0

Entrada para predição

Input usado para predição

	Duration	N erros
T1c4	11	0
T2c4	45	1
T3c4	2	1
…	…	…
Txc4	3	2

Figura 5.2: Exemplo de entrada utilizada no algoritmo *CRF*.

Como resultado, foi gerada a Tabela 5.2, a qual destaca a quantidade de melhores valores encontrados pelas politicas avaliadas. Nota-se que a política *LinUCB* possui o menor *PT* para todos os casos e para todos os orçamentos. Outro destaque é sua performance nos baixos orçamentos de teste, obtendo a maior quantidade de melhores valores neste orçamento. Contudo, em uma análise geral, a política *SW-LinUCB* apresenta a maior quantidade de melhores resultados, devido aos resultados apresentados nos orçamentos de 50% e 80%.

Após a seleção da política *SW-LinUCB* como a mais adequada, ela foi utilizada para realizar os experimentos de comparação com as abordagens não contextuais e com o algoritmo *CRF*.

5.6 RESULTADOS E ANÁLISE

Nesta seção, são apresentados e analisados os resultados, com o objetivo de responder às nossas questões de pesquisa.

5.6.1 QP1: Seleção do grupo de *features*

Após a realização do *tunning* de parâmetros, foi analisado o resultado da melhor configuração para selecionar um grupo para comparação com o algoritmo não contextual.

Para o algoritmo *CRF*, os grupos Todas as *features* (TF) e Tempo de execução (TE) se destacaram por obterem a maior quantidade de melhores valores na maioria das métricas. Já para o algoritmo *CMAB*, destacaram-se os grupos Complexidade do Caso de Teste (CCT) e TE, os quais obtiveram a maior parte dos maiores valores e na maioria dos casos obtiveram valores próximos, quando comparado entre eles, como pode ser observado na Tabela 5.3 e na Tabela 5.4.

Contudo, o grupo TE mostrou-se superior com relação a métrica *PT*, principalmente para o algoritmo *CRF*, sendo superior em todos os casos quando comparado com o grupo TF.

Tabela 5.2: Comparativo entre os melhores valores adquiridos pelas políticas *LinUCB* e *SW-LinUCB* em relação aos seis sistemas utilizados

Métrica	Políticas		
	Equivalente	*LinUCB*	*SW-LinUCB*
	Orçamento de tempo: 10%		
APFDc	0	3	3
NAPFD	0	3	3
NTR	0	4	2
FDE	1	3	2
RFTC	0	2	4
PT	0	6	0
	Orçamento de tempo: 50%		
APFDc	0	2	4
NAPFD	0	2	4
NTR	0	1	5
FDE	0	2	4
RFTC	0	4	2
PT	0	6	0
	Orçamento de tempo: 80%		
APFDc	0	1	5
NAPFD	0	2	4
NTR	0	1	5
FDE	0	2	4
RFTC	0	4	2
PT	0	6	0

Isso ocorre devido à simplicidade desse grupo, que contém somente 2 *features*: Duração e Número de erros, enquanto o grupo TF utiliza 7 *features*. Considerando os pontos acima, foi selecionado o grupo TE como o grupo que apresenta os melhores resultados e este foi utilizado para a comparativo com as abordagens não contextuais.

Vale a pena observar que no algoritmo *CRF* os grupos TP, CCT e ETC, obtiveram um resultado muito semelhantes, obtendo pouca variação entre os resultados. Isso pode mostrar que esses grupos possuem uma semelhança para o aprendizado do algoritmo. Levando em consideração o grupo SF para o algoritmo *CMAB*, nota-se que nos poucos casos em que ele se destacou, foi quase exclusivamente para o orçamentos de teste de 10%.

Resposta à QP1: Pode-se concluir que para o algoritmo *CRF* o grupo de todas as *features* (TF) apresenta bons resultados, assim como o grupo Seleção de *Features* (SF) para o *CMAB*. Contudo, o grupo Tempo de Execução (TE) apresenta o melhor desempenho no geral, obtendo resultados similares ou melhores nas métricas: *NAPFD* e *APFDc* e obtendo melhores resultados na métrica *PT*.

5.6.2 QP2: Comparativo entre os algoritmos contextuais

Após ter selecionado o melhor grupo de *features* foram comparados ambos os algoritmos contextuais. Foram realizadas 30 execuções independentes, com o objetivo de minimizar a aleatoriedade presente nos algoritmos de *ML*. A análise dos resultados mostra que na maioria dos casos, o algoritmo *Contextual Random Forest*, como apresentado na Tabela 5.5, é o melhor para casos com orçamento de teste restritivo (10%), contudo, em orçamentos de teste médios ou maiores, ambos algoritmos demonstram estar mais adaptados para alguns *datasets*. Analisando as métricas *NAPFD* e *APFDc*, pode-se notar que há um equilíbrio de melhores valores nos

Tabela 5.3: Quantidade de melhores valores para métricas de qualidade por orçamento de tempo para cada grupo de *feature* para o algoritmo *CRF* para os seis sistemas avaliados

Métrica	Grupos de *features*					
	TF	TE	TP	CCT	ECT	SF
	Orçamento de tempo: 10%					
APFDc	2	3	1	0	0	0
NAPFD	2	4	2	1	1	1
NTR	2	3	0	0	0	1
FDE	2	4	1	1	2	1
RFTC	3	3	1	1	1	2
PT	0	4	0	2	0	0
	Orçamento de tempo: 50%					
APFDc	2	3	0	0	0	1
NAPFD	2	3	1	1	1	1
NTR	1	2	2	2	2	2
FDE	3	2	0	0	0	1
RFTC	1	3	0	1	0	1
PT	0	4	0	2	0	0
	Orçamento de tempo: 80%					
APFDc	1	2	3	3	3	0
NAPFD	3	2	1	1	1	0
NTR	2	4	2	2	2	0
FDE	2	3	0	0	0	1
RFTC	1	3	2	1	1	0
PT	0	4	0	2	0	0

Tabela 5.4: Quantidade de melhores e valores equivalentes para métricas de qualidade por orçamento de tempo para cada grupo de *feature* para o algoritmo *CMAB*.

Métrica	TF	TE	TP	CCT	ECT	SF
	Orçamento de tempo: 10%					
APFDc	3	2	0	2	0	4
NAPFD	3	2	0	2	0	5
NTR	1	1	0	1	0	3
FDE	1	1	0	1	0	3
RFTC	1	5	2	1	0	2
PT	0	0	6	0	0	0
	Orçamento de tempo: 50%					
APFDc	2	2	3	3	0	2
NAPFD	1	5	2	3	0	1
NTR	0	3	0	3	0	0
FDE	1	1	1	3	0	0
RFTC	2	5	2	1	1	1
PT	0	0	6	0	0	0
	Orçamento de tempo: 80%					
APFDc	0	2	1	2	3	4
NAPFD	0	6	0	3	2	1
NTR	1	2	1	2	0	0
FDE	2	2	0	2	0	0
RFTC	2	6	1	3	0	0
PT	0	0	6	0	0	0

orçamentos de teste de 50% e 80%. O algoritmo *CMAB* obteve os melhores resultados nos *datasets*: `Dspace`, `Fastjson` e `Retrofit`, enquanto o algoritmo *CRF* obteve nos *datasets* `Druid`, `OkHttp` e `ZXing` como demonstrado na Tabela 5.6. Contudo o algoritmo *CRF* foi o melhor no orçamento de teste de 10%, não obtendo o melhor valor somente no dataset `DSpace`.

Tabela 5.5: Comparativo entre os algoritmos *Contextual Random Forest* e *Contextual Multi-Armed Bandit* com relação aos seis sistemas utilizados

Métrica	**Algoritmos Contextuais**		
	Equivalente	*CRF*	*CMAB*
	Orçamento de tempo: 10%		
APFDc	0	6	0
NAPFD	0	5	1
NTR	0	4	2
FDE	0	5	1
RFTC	0	3	3
PT	0	0	6
TCPS	0	5	1
	Orçamento de tempo: 50%		
APFDc	0	3	3
NAPFD	0	3	3
NTR	0	1	5
FDE	0	4	2
RFTC	0	4	2
PT	0	0	6
TCPS	0	5	1
	Orçamento de tempo: 80%		
APFDc	0	3	3
NAPFD	0	3	3
NTR	0	1	5
FDE	0	4	2
RFTC	0	3	3
PT	0	0	6
TCPS	0	4	2

Considerando o *PT*, o *CMAB* foi muito superior em todos os casos, levando em média no melhor caso 0.05 segundos para realização da priorização no dataset `Retrofit` e 1 segundo para o pior caso no dataset `Fastjson`. Por outro lado, o *CRF* obteve valores muito maiores de *PT*, entre 1.3 segundos para o melhor caso no dataset `DSpace` e 7.7 segundos no dataset `Druid`.

Os resultados da análise da métrica *NTR* apresenta o mesmo padrão obtido analisando as métricas *NAPFD* e *APFDc*: o algoritmo *CRF* foi o melhor no orçamento de teste de 10%, não obtendo o melhor valor somente no dataset `DSpace`, enquanto para outros orçamentos, o mesmo padrão de *datasets* foi observado.

Analisando o *RFTC*, observa-se que os algoritmos obtiveram um desempenho equivalente para o orçamento de teste de 10% e 80%, ambos obtendo 50% (3 de 6 casos) dos melhores resultados. Contudo, o algoritmo *CRF* obteve 67% (4) melhores resultados, superando o *CMAB*.

Avaliando o *FDE*, nota-se que o algoritmo *CRF* foi o melhor no orçamento de teste de 10% (5 de 6 casos), para o orçamento de teste de 50% e 80% (4 de 6 casos), demonstrando superioridade quando avaliando falhas não repetidas no sistema.

Por fim, os resultados da métrica TCPS apresentam um comportamento semelhante ao das métricas anteriores com relação ao orçamento de teste de 10%, no qual o *CRF* obteve 5 de 6 melhores casos, bem como no orçamento de 50% (5) e 80% (4).

Um fato curioso que foi possível notar durante a análise dos resultados, é que o algoritmo *CMAB* atingiu os melhores valores em quase todos as métricas (exceto TCPS) para

o *dataset* DSpace, o qual apresenta uma volatilidade relativamente alta, demonstrando a alta adaptabilidade a volatilidade deste algoritmo.

Tabela 5.6: Valores médios e desvio padrão NAPFD e APFDc para: *CRF* e *CMAB*.

Dataset	NAPFD		APFDc	
	CRF	*SW-LinUCB*	*CRF*	*SW-LinUCB*
	Orçamento de tempo: 10%			
DSpace	0.9492 ± 0.0000 ▲	**0.9498 ± 0.0000** ★	**0.9501 ± 0.0000** ★	0.9491 ± 0.0000 ▲
Druid	**0.9168 ± 0.0020** ★	0.8986 ± 0.0000 ▲	**0.9182 ± 0.0010** ★	0.8967 ± 0.0000 ▲
Fastjson	**0.9462 ± 0.0000** ★	0.9239 ± 0.0000 ▲	**0.9459 ± 0.0000** ★	0.9221 ± 0.0000 ▲
OkHttp	**0.8576 ± 0.0000** ★	0.8147 ± 0.0000 ▲	**0.8545 ± 0.0000** ★	0.8152 ± 0.0000 ▲
Retrofit	**0.9659 ± 0.0000** ★	0.9652 ± 0.0000 ▲	**0.9664 ± 0.0000** ★	0.9654 ± 0.0000 ▲
ZXing	**0.9873 ± 0.0000** ★	0.9835 ± 0.0000 ▲	**0.9867 ± 0.0000** ★	0.9837 ± 0.0000 ▲
	Orçamento de tempo: 50%			
DSpace	0.9652 ± 0.0000 ▲	**0.9720 ± 0.0000** ★	0.9662 ± 0.0000 ▲	**0.9704 ± 0.0000** ★
Druid	**0.9344 ± 0.0020** ★	0.9048 ± 0.0000 ▲	**0.9318 ± 0.0020** ★	0.9048 ± 0.0000 ▲
Fastjson	0.9471 ± 0.0000 ▲	**0.9552 ± 0.0000** ★	0.9473 ± 0.0000 ▲	**0.9504 ± 0.0000** ★
OkHttp	**0.9232 ± 0.0000** ★	0.8862 ± 0.0000 ▲	**0.9136 ± 0.0000** ★	0.8793 ± 0.0000 ▲
Retrofit	0.9838 ± 0.0000 ▲	**0.9854 ± 0.0000** ★	0.9829 ± 0.0000 ▲	**0.9846 ± 0.0000** ★
ZXing	**0.9934 ± 0.0000** ★	0.9877 ± 0.0000 ▲	**0.9930 ± 0.0000** ★	0.9883 ± 0.0000 ▲
	Orçamento de tempo: 80%			
DSpace	0.9692 ± 0.0000 ▲	**0.9776 ± 0.0000** ★	0.9702 ± 0.0000 ▲	**0.9746 ± 0.0000** ★
Druid	**0.9548 ± 0.0020** ★	0.9247 ± 0.0000 ▲	**0.9433 ± 0.0010** ★	0.9192 ± 0.0000 ▲
Fastjson	0.9473 ± 0.0000 ▲	**0.9675 ± 0.0000** ★	0.9477 ± 0.0000 ▲	**0.9575 ± 0.0000** ★
OkHttp	**0.9308 ± 0.0000** ★	0.8928 ± 0.0000 ▲	**0.9194 ± 0.0000** ★	0.8826 ± 0.0000 ▲
Retrofit	0.9874 ± 0.0000 ▲	**0.9884 ± 0.0000** ★	0.9857 ± 0.0000 ▲	**0.9866 ± 0.0000** ★
ZXing	**0.9948 ± 0.0000** ★	0.9904 ± 0.0000 ▲	**0.9945 ± 0.0000** ★	0.9905 ± 0.0000 ▲

Resposta à QP2: Na maioria dos casos, o algoritmo *CRF* é o melhor, principalmente para casos com orçamento de teste restritivo (10%), contudo, ambos algoritmos demonstram estar mais adaptados para alguns *datasets*. Especificamente, o *CMAB* foi melhor em quase todas as métricas para o *dataset* DSpace.

5.6.3 QP3: Comparativo entre os algoritmos contextuais e não contextuais

Para a realização dos experimentos foram selecionados alguns *datasets* os quais eram possíveis a extração da quantidade de casos de teste e determinação de sua volatilidade.

Com o objetivo de analisar o impacto das variáveis de contexto, foi realizada uma comparação entre os algoritmos: *RF* e *CRF*; *FRRMAB* e *SW-LinUCB*. Os algoritmos contextuais foram utilizados com o grupo de *features* TE que apresentou os melhores resultados.

A última etapa antes da análise foi a execução de todos os algoritmos mencionados, incluindo os não contextuais, tanto *FRRMAB* quanto *RF*. Dito isto, alguns valores podem diferir de seus respectivos artigos de apresentação. Após a execução, foi realizada a análise estatística dos resultados utilizando os testes: Kruskal-Wallis, Mann-Whitney e Friedman.

Os resultados mostram que, em geral, os algoritmos não contextuais apresentam melhores resultados (tanto *FRRMAB* quanto *RF*). Isso pode ser devido à independência da qualidade das *features* extraídas, bem como da qualidade dos grupos utilizados. No entanto, os algoritmos contextuais obtiveram melhores resultados ao considerar orçamentos baixos e médios. No caso do *SW-LinUCB*, caso a métrica *PT* for desconsiderada, pode-se até argumentar que o algoritmo contextual é um pouco melhor que o *FRRMAB*.

Em primeiro lugar, analisa-se o *PT*, e como era esperado, os algoritmos não contextuais foram as melhores, especialmente *FRRMAB*. Contudo, não por uma grande margem, pois

Tabela 5.7: Valores para métricas melhores ou equivalentes por orçamento de tempo comparando os algoritmos: *FRRMAB* e *SW-LinUCB*; *Random Forest* e *Contextual Random Forest*.

Métrica	**Algoritmos baseados em *MAB***			**Algoritmos baseados em *RF***		
	Equivalente	*FRRMAB*	*SW-LinUCB*	Equivalente	*RF*	*CRF*
	Orçamento de tempo: 10%					
APFDc	1	2	3	0	3	3
NAPFD	0	1	5	0	3	3
NTR	0	2	4	0	2	4
FDE	0	2	4	1	1	4
RFTC	1	1	4	0	3	3
TCPS	0	3	3	0	4	2
PT	0	6	0	0	5	1
	Orçamento de tempo: 50%					
APFDc	1	4	1	0	4	2
NAPFD	1	2	3	0	5	1
NTR	0	4	2	0	4	2
FDE	0	3	3	0	5	1
RFTC	1	2	3	0	1	5
TCPS	0	3	3	0	5	1
PT	0	6	0	0	5	1
	Orçamento de tempo: 80%					
APFDc	1	4	1	0	5	1
NAPFD	1	3	2	0	4	2
NTR	0	5	1	0	4	2
FDE	0	4	2	0	4	2
RFTC	1	2	3	0	2	4
TCPS	0	1	5	0	4	2
PT	0	6	0	0	5	1

o pior caso para os algoritmos baseados em *MAB* foi de 0,5 segundos para *FRRMAB* e 1 segundo para *SW-LinUCB*. Quanto aos algoritmos baseados em *RF*, o pior caso para o *RF* foi de aproximadamente 1,8 segundos, já para o *CRF* foi aproximadamente 7,7 segundos. Surpreendentemente, no conjunto de dados Dspace, o *CRF* superou o algoritmo *RF* em alguns milissegundos (0,2 segundos).

Analisando a métrica *APFDc*, pode-se notar que a maioria dos melhores resultados ainda foram para os algoritmos não contextuais. O algoritmo *RF* obteve 67% dos melhores valores (12 de 18) e *FRRMAB* obteve 72% (13 de 18). Ou seja, as informações de contexto afetaram negativamente a previsão da métrica *APFDc*.

Considerando a métrica *NAPFD*, houve um cenário semelhante ao do *APFDc* ao considerar os algoritmos baseados em *RF*. O algoritmo *RF* obteve melhor pontuação em 67% (12) dos casos. No entanto a mesma observação não é aplicada para a abordagem *SW-LinUCB*, no seu caso a abordagem *SW-LinUCB* alcançou 61% (11) das melhores pontuações, mostrando que houve um impacto positivo para esta métrica, especialmente para o baixo orçamento de teste onde a porcentagem é de 83% (5 de 6).

Para o *FDE*, os algoritmos baseados em *MAB* empataram entre suas melhores pontuações, enquanto a *RF* obteve a melhor pontuação (61%) (11 casos) em relação ao algoritmo *CRF*. Isso mostra que os recursos do sistema não tiveram um impacto bom em nenhum dos algoritmos, o que reduziu o desempenho do algoritmo *CRF* em geral. No entanto, para cenários de baixo orçamento de teste, o *CRF* obteve 83% (5 casos) dos melhores resultados, o que significa que especificamente para cenários de baixo orçamento de teste a utilização de variáveis de contexto pode ser interessante.

A métrica *RFTC* mostra um resultado interessante, nos quais os algoritmos contextuais foram realmente melhores, tanto *CRF* quanto *SW-LinUCB* alcançando 67% (12) com *SW-LinUCB* alcançando 1 resultado igual e 2 resultados estatisticamente iguais que *FRRMAB*. Isso significa que, para a detecção precoce de falhas, os recursos do sistema têm um impacto positivo.

Analisando os valores de *NTR*, as melhores ainda foram os algoritmos não contextuais, mas não por uma grande margem, *RF* alcançando 56%(10) dos melhores resultados e *FRRMAB* alcançou 61% (11). Uma observação interessante é que ambos os algoritmos contextuais alcançaram 67% (4) para orçamentos baixos.

Em relação ao TCPS, a abordagem *RF* foi a melhor para 72% (13), contrariando a crença de que as *features* do sistema ajudariam o algoritmo contextual a organizar melhor os casos de teste. No entanto, o contrário é observado ao analisar o *SW-LinUCB*, pois obteve 61% (11) melhores resultados.

Uma observação interessante sobre os resultados é que o algoritmo *SW-LinUCB* realmente foi a melhor para a maioria das métricas, em todos os orçamentos de teste para o conjunto de dados `Retrofit`. A mesma afirmação não é verdadeira ao analisar o *CRF*, onde os resultados foram mais equilibrados para cada abordagem em função de orçamento de teste.

Resposta à QP3: No geral, as informações de contexto apresentaram um aumento na qualidade de predição no orçamento de teste de 10%, principalmente para o *CMAB*. Contudo, para o algoritmo *CRF*, a utilização das variáveis de contexto não auxiliaram a melhorar os resultados, a abordagem não contextual obteve melhores resultados. Avaliando o algoritmo *CMAB*, a utilização das *features* melhorou a qualidade da predição do algoritmo, contudo, no geral a sua contra parte ainda é a melhor.

5.6.4 QP4: Comparativo entre todos algoritmos analisados

Para responder a esta questão de pesquisa, os quatro algoritmos utilizados foram comparados utilizando suas respectivas melhores configurações. No geral, pode-se avaliar que os algoritmos não contextuais *COLEMAN* (*FRRMAB*) e *RF* obtêm os melhores resultados para a maioria das métricas analisadas, sendo a abordagem *COLEMAN* o algoritmo analisado com menor *PT* e a abordagem *RF* a mais indicada para orçamentos de teste baixos e médios. Já a avaliação dos algoritmos contextuais sugere que o algoritmo *CRF* apresenta melhores resultados para orçamentos de teste baixos, enquanto os resultados algoritmo *CMAB* não se destacaram dentre os algoritmos avaliados.

Com base na Tabela 5.8 e analisando o orçamento de teste de 10%, pode-se analisar que em sua maioria, os algoritmos *RF* e *CRF* apresentaram os melhores resultados, principalmente considerando as métricas *APFDc*, *NAPFD*, *NTR* e *FDE*.

Para as métricas *RFTC* e TCPS, um comportamento diferente foi observado: o algoritmo *RF* apresenta destaque nos resultados quando considerando o TCPS em conjunto com o algoritmo *FRRMAB*. Contudo para a métrica *RFTC*, o algoritmo *SW-LinUCB* apresenta os melhores resultados em 50% dos casos (3 casos de 6) seguido do *RF* com 33% (2 de 6 casos).

Considerando um ponto de vista geral, para orçamentos de teste baixos, o algoritmo *RF* ainda é o mais indicado, contudo, o algoritmo *CRF* apresenta certa melhoria na qualidade de predição, demonstrando potencial para os algoritmos contextuais.

Analisando o orçamento de teste de 50%, pode-se notar um equilíbrio melhor entre os melhores algoritmos não contextuais. As métricas para as quais observa-se um comportamento diferente são *NAPFD*, *NTR* e *RFTC*. Para as métricas *NAPFD* e *NTR*, observa-se um empate

entre os algoritmos *FRRMAB*, *RF* e *SW-LinUCB*, obtendo 2 melhores valores. Diferentemente para a métrica *RFTC* outro padrão foi observado; Há a dominância dos algoritmos contextuais (5 de 6 casos), sendo 3 dos melhores casos obtidos pelo algoritmo *SW-LinUCB*.

Considerando orçamentos de teste médios, a diferença de qualidade de predição entre os algoritmos não contextuais com os contextuais se torna mais aparente, com a exceção do *RFTC* os algoritmos contextuais se demonstraram superiores obtendo 5 melhores resultados de 6 *datasets*.

Por fim, para o orçamento de teste de 80%, pode-se notar uma dominância do algoritmo *FRRMAB*, obtendo em sua maioria 4 melhores valores (exceto *RFTC* com 3 melhores casos e TCPS com melhor 1 caso). Nesse orçamento, o segundo algoritmo com maior quantidade de melhor valores é o *SW-LinUCB*. Contudo, para o TCPS, o algoritmo *RF* ainda demonstra um bom desempenho obtendo 3 de 6 melhores casos.

Levando em consideração orçamentos de teste maiores, o algoritmo *FRRMAB* continua sendo o mais recomendado. Como demonstrado pela Tabela 5.8, ele o que apresenta a melhor performance na maioria dos casos, além de ser o algoritmo analisado com menor *PT*.

Tabela 5.8: Quantidade de melhores valores para métricas de qualidade por Orçamento de tempo.

Métrica	**Algoritmos avaliados**			
	FRRMAB	*RF*	*CRF*	*SW-LinUCB*
	Orçamento de tempo: 10%			
APFDc	1	3	2	1
NAPFD	0	3	2	1
NTR	0	2	3	1
FDE	0	3	2	1
RFTC	0	2	1	3
TCPS	2	3	1	0
PT	6	0	0	0
	Orçamento de tempo: 50%			
APFDc	3	2	0	1
NAPFD	2	2	0	2
NTR	2	2	0	2
FDE	3	3	0	0
RFTC	1	0	3	2
TCPS	1	4	1	0
PT	6	0	0	0
	Orçamento de tempo: 80%			
APFDc	4	1	0	1
NAPFD	4	0	0	2
NTR	4	1	0	1
FDE	4	1	1	1
RFTC	3	0	1	2
TCPS	1	3	1	1
PT	6	0	0	0

Resposta à QP4: Nota-se que os algoritmos não contextuais ainda são os melhores no geral, sendo os algoritmos equivalentes quando considerando orçamentos de teste médios. Ainda há uma necessidade de estudos para a melhoria da performance dos algoritmos contextuais, uma vez que apresentaram melhoria nas predições em certas situações.

5.6.5 Discussão dos resultados

Uso de algoritmos contextuais: Esses algoritmos apresentaram melhores resultados em orçamentos mais restritivos (10%), embora não superem o algoritmo não contextual *RF*.

O algoritmo *CRF* obteve melhor desempenho somente para o orçamento de tempo mais restritivo, obtendo uma baixa quantidade de melhores resultados nos outros dois orçamentos de tempo analisados (50% e 80%) quando comparado com os outros 3 algoritmos (*FRRMAB*, *RF* e *CMAB*). Contudo, entre os dois algoritmos contextuais, o *CRF* apresentou os melhores resultados no geral.

O algoritmo *SW-LinUCB* foi superado em todas as métricas quando comparado com os outros 3 algoritmos, exceto para a métrica *RFTC*, na qual apresentou melhores resultados no orçamento de 10%. Quando comparado com o algoritmo *FRRMAB*, o algoritmo *SW-LinUCB* também foi melhor em orçamentos mais restritivos. Contudo, para a métrica TCPS, este algoritmo apresentou os melhores resultados, principalmente nos orçamentos menos restritivos (80%).

Para os 6 grupos de *features* analisados, somente o grupo Tempo de Execução (TE) se destacou no geral. Para o algoritmo *CRF*, o grupo Todas as *Features* (AF) apresentou bons resultados, enquanto para o algoritmo *SW-LinUCB*, foi o grupo Complexidade do Caso de Teste (CCT), embora ambos tenham sido superados pelo grupo TE para os dois algoritmos.

Uso de algoritmos não contextuais: Os algoritmos não contextuais (*RF* e *FRRMAB*) foram os melhores no geral, obtendo a maior quantidade de melhores resultados quando avaliando os três orçamentos de teste analisados. A Figura 5.3 apresenta de forma resumida estes resultados. O algoritmo *RF* apresentou melhores resultados em orçamentos de 10%, enquanto o *FRRMAB* nos de 80%, empatando os resultados nos orçamentos de 50%. Ambos algoritmos possuem menor tempo de priorização quando comparados com os algoritmos contextuais, o que pode ser justificado pela ausência das *features*, sendo o algoritmo *FRRMAB* o qual possui menor tempo de priorização de todos, executando em frações de segundos em alguns casos.

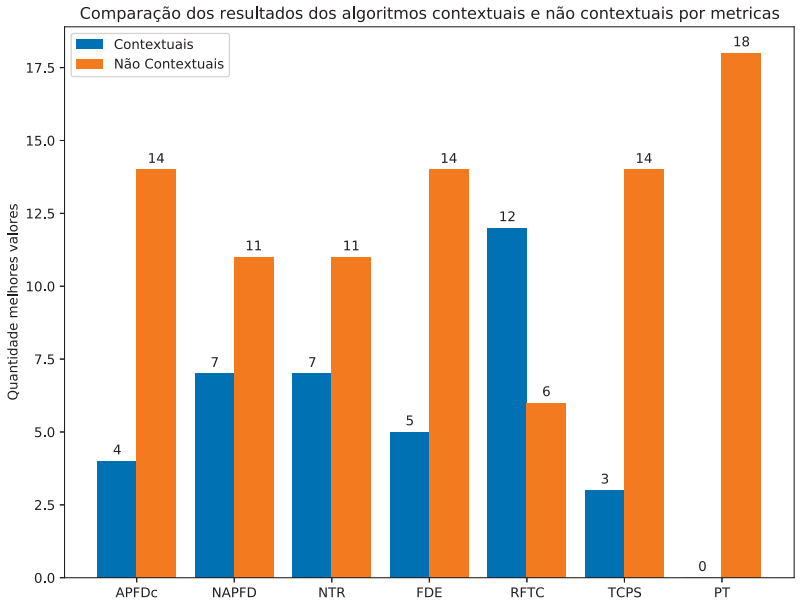


Figura 5.3: Quantidade de melhores resultados (0 a 16) dos algoritmos não contextuais (*MAB* e *RF*) e contextuais (*CRF* e *CMAB*) de acordo com as métricas utilizadas.

O algoritmo *RF* apresentou os melhores resultados em todos os orçamentos para a métrica TCPS, já quando comparado com o *CRF*, foi superado na métrica *NTR* para o orçamento de 10% e na métrica *RFTC* para o orçamento de 50%. Enquanto o algoritmo *FRRMAB*, no orçamento de 80%, apresentou um desempenho em 66% dos melhores casos (4 de 6 casos) para as métricas *APFDc*, *NAPFD*, *NTR* e *FDE*, além de um desempenho melhor em 50% (3 de 6) dos casos para a métrica RFTC.

5.7 AMEAÇAS À VALIDADE

Alguns fatores podem ser considerados ameaças a validade dos resultados apresentados.

Validade Interna: apenas um grupo selecionado de *features* foi analisado, existe a possibilidade de que uma única *feature* ou uma combinação diferente de *features* possa obter melhores resultados gerais e atender melhor às particularidades do ambiente de integração contínua. Quanto aos algoritmos de *ML*, apenas algumas configurações foram testadas, devido à natureza empírica do experimento. Considerando isso, assume-se que elas não são as melhores possíveis. Ocorreram casos em que um algoritmo teve um desempenho geral melhor. Contudo, na maioria das vezes, não houve desempenho supremo ou absoluto em todos os conjuntos de dados. Com uma configuração melhor, melhores resultados podem ser alcançados.

Validade Externa: devido aos problemas experimentados na extração de *features* de alguns *datasets* devido à mesclagem de *commits*, um número baixo de *datasets* foi usado. Para garantir os resultados e eliminar possíveis viés, mais *datasets* devem ser analisados para que os achados possam ser consolidados ou para que se possa chegar a uma conclusão diferente.

Validade de Conclusão: a natureza aleatória dos algoritmos de *ML* é uma ameaça. Para minimizar o viés de aleatoriedade dos algoritmos de *ML*, 30 execuções independentes foram usadas. No entanto, um número maior de execuções pode alcançar resultados mais sólidos. O hardware usado para realizar os experimentos não foi o mesmo em todos os algoritmos. O hardware pode ter alguma influência no desempenho, principalmente ao analisar o tempo de execução. Por fim, os métodos de avaliação dos algoritmos podem ser uma ameaça, para minimizar essa ameaça, foram utilizadas métricas amplamente utilizados na área de estudo de *TCPCI* e uma nova métricas foi proposta, a métrica *TCPS*.

5.8 CONSIDERAÇÕES FINAIS

Neste capítulo foi realizada uma avaliação do impacto do uso de informações contextuais para algoritmos de *ML* em *TCPCI* utilizando como base algoritmos baseados em *MAB* (*COLEMAN* [28] e *CMAB*) e em *Random Forest* (*RF* e *CRF*), avaliando 7 variáveis de contexto, separadas em 6 grupos diferentes.

O algoritmo *CMAB* (*SW-LinUCB*) e o algoritmo *Contextual Random Forest* (*CRF*) foram avaliados, utilizando cada grupo de *features* para selecionar o melhor grupo, em seguida, os resultados da utilização de variáveis contextuais foram comparados com as abordagens de *ML* encontradas na literatura. Com esta avaliação, pode-se considerar que para a variação do *CMAB*, o uso de variáveis de contexto do sistema ajudou um pouco na análise de cenários de baixo orçamento de teste (10%) e que no geral ajudou no resultado das previsões. No entanto, ao analisar a variação do *CRF*, percebe-se que o impacto obtido não atingiu as expectativas, não havendo melhoria na performance do algoritmo *RF*.

6 CONCLUSÃO

O trabalho realizado teve como objetivo principal a introdução de uma abordagem baseada no método de janela deslizante que permite a aplicação de diferentes algoritmos de aprendizado de máquina a fim de priorizar casos de teste no ambiente de integração contínua. A abordagem apresentada é utilizada na fase de teste do ciclo de integração contínua da seguinte forma: o código é construído a partir do *commit* mais recente c após uma compilação bem-sucedida. O conjunto de testes no *commit* a ser analisado, o caso de teste é integrado ao histórico e usado pelo algoritmo para previsão de propensão a falhas, é realizado então um filtro para os casos de teste t avaliados por $t \in Tc$ e os ordena em ordem decrescente, resultado no conjunto priorizado $T'c$, que é executado pelo ambiente de integração contínua. Finalmente, $T'c$ é avaliado e seus resultados são atualizados no histórico.

O uso do algoritmo *RF* e da rede *LSTM* foi avaliado utilizando 11 *datasets*. Após a etapa de análise dos algoritmos, conclui-se que ambos algoritmos são aplicáveis para o problema de *TCPCI* por apresentarem baixo tempo de priorização. Quando comparados os algoritmos propostos com as abordagens *COLEMAN* e *RETECS*, notou-se que os algoritmos propostos foram superiores principalmente nos casos de baixo e médio (de 10% e 50%) orçamento de teste, sendo superados pelo algoritmo *FRRMAB* (*COLEMAN*) para orçamentos menos restritivos (de 80%). Vale destacar que o algoritmo *RF* obteve o melhor desempenho geral dentre os analisados, superando a abordagem *COLEMAN*.

A avaliação descrita no Capitulo 5 apresenta uma análise empírica da utilização de variáveis de contexto (*features* do sistema) para solução do problema de *TCPCI*, utilizando 6 dos datasets previamente utilizados. Após a extração das informações de contexto, as mesmas foram então separadas em 6 grupos para avaliação, são eles: Todas as *Features*, Tempo de Execução, Tamanho do Programa, Complexidade do Caso de Teste, Evolução do Caso de Teste e Seleção de *Features* (*features* do sistema selecionados por meio da análise de PCA).

A etapa seguinte foi a análise desses grupos entre si, concluindo-se que o grupo Tempo de Execução obteve melhor desempenho, obtendo uma bons resultados de predição e um tempo de priorização mais baixo que os demais devido a menor quantidade de *features* presente neste grupo. Após a seleção do melhor grupo de *features*, foram executados os algoritmos contextuais *CMAB* e *CRF* utilizando o grupo Tempo de Execução e re-executadas as respectivas contra-partes não contextuais, ou seja, os algoritmos *FRRMAB* e *RF*.

Em seguida, os algoritmos foram analisados da seguinte maneira: Primeiramente foram comparados os algoritmos contextuais entre si, em seguida os algoritmos contextuais e suas contra-partes não contextuais e por fim foi realizada uma comparação geral dos 4 algoritmos analisados.

A primeira análise mostrou que o algoritmo *CRF* é melhor principalmente no orçamento de teste de 10%, assim como o algoritmo não contextual *RF*, enquanto para outros orçamentos, a performance dos algoritmos é bastante similar quanto à qualidade de predição, ambos obtendo os melhores resultados para 50% (3 de 6) dos *datasets* avaliados.

A segunda análise mostrou que os algoritmos contextuais não conseguiram superar suas contra-partes não contextuais. Contudo, apresentaram resultados promissores, demonstrando que talvez a análise de diferentes grupos ou utilização de diferentes *features* podem aprimorar os algoritmos a fim de superar sua contra-parte não contextual.

A analise final mostrou que ambos algoritmos não contextuais são equivalentes em questão de performance geral, sendo o algoritmo *RF* o melhor para orçamentos de teste mais baixos e o *FRRMAB* melhor para orçamentos menos restritivos.

6.1 LIMITAÇÕES

Esta seção apresenta as limitações da abordagem proposta e da avaliação realizada. Essas limitações devem ser abordadas em trabalhos futuros Tanto para a avaliação apresentada no Capitulo 4 quanto para a apresentada no Capitulo 5, foi utilizada uma pequena quantidade de *datasets*, principalmente para o segundo caso. Uma quantidade maior de *datasets* seria ideal para consolidar os resultados encontrados nesses capítulos.

Durante a realização dos estudos, foram utilizados ambientes variados para a execução dos algoritmos analisados. Isso pode gerar um problema para a análise de alguns fatores como por exemplo o tempo de priorização. A configuração dos algoritmos foi selecionada de maneira empírica, ou seja, provavelmente não é a configuração ideal para o problema.

A análise apresentada no Capitulo 5 limitou-se a alguns grupos de *features*. Esses grupos podem ter limitado o desempenho dos algoritmos contextuais. Outros grupos com diferentes combinações de *features* poderiam ter atingido melhores resultados.

6.2 CONTRIBUIÇÕES

As contribuições deste trabalho são:

- Uma abordagem baseada no método de janela deslizante e no histórico de falhas para o problema de *TCPCI*. Está abordagem é genérica e pode ser utilizada com diferentes algoritmos de *ML* e foi avaliada utilizando informações de contexto de teste;
- Avaliação da abordagem proposta com algoritmos de *ML* baseados em regressão para o problema de *TCPCI*;
- Implementação e avaliação de algoritmos de regressão genéricos baseados em variáveis de contexto: *CRF* e de uma abordagem baseada em *CMAB*.
- Publicações:
  - Enrique A. Da Roza, Jackson A. Prado Lima, Rogério C. Silva, and Silvia Regina Vergilio. Machine learning regression techniques for test case prioritization in continuous integration environment. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 196–206, 2022 [25].
  - Paulo Roberto Farah, Thainá Mariani, Enrique A. da Roza, Rogério C. Silva, and Silvia Regina Vergilio. Unsupervised learning for refactoring pattern detection. In *2021 IEEE Congress on Evolutionary Computation (CEC)*, pages 2377–2384, 2021 [83].

6.3 TRABALHOS FUTUROS

Após a finalização do trabalho, nota-se que os algoritmos contextuais apresentam resultados promissores, mesmo que não tenham superado suas contra-partes não contextuais. Para trabalhos

futuros, pretende-se realizar um estudo mais aprofundado sobre a utilização de informação de contexto para os algoritmos analisados, utilizando grupos diferentes e possíveis novas *features* para avaliação, bem como uma maior quantidade de *datasets* para serem analisados.

Para aprofundar o estudo, pretende-se avaliar outros algoritmos de *ML* promissores, como por exemplo o algoritmo XGBoost [16], o qual também é baseado na construção de árvores de decisão.

REFERÊNCIAS

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, 2016.

[3] Tsuyoshi Ando. Majorization relations for hadamard products. *Linear Algebra and its Applications*, 223-224:57–64, 1995. Honoring Miroslav Fiedler and Vlastimil Ptak.

[4] Francis John Anscombe. The transformation of poisson, binomial and negative-binomial data. *Biometrika*, 35(3/4):246–254, 1948.

[5] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM Journal on Computing*, 32(1):48–77, January 2003.

[6] Safial Islam Ayon. Neural network based software defect prediction using genetic algorithm and particle swarm optimization. In *2019 1st International Conference on Advances in Science, Engineering and Robotics Technology (ICASERT)*, pages 1–4, 2019.

[7] Maurice Stevenson Bartlett. The square root transformation in analysis of variance. *Supplement to the Journal of the Royal Statistical Society*, 3(1):68–78, 1936.

[8] Kent Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999.

[9] Mark Berman, Jeffrey S. Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. Geni: A federated testbed for innovative network experiments. *Computer Networks*, 61:5 – 23, 2014. Special issue on Future Internet Testbeds Part I.

[10] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1–12, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[12] Sébastien Bubeck and Nicolò et al. Cesa-Bianchi. Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems. *Foundations and Trends® in Machine Learning*, 5(1):1–122, 2012.

[13] Benjamin Busjaeger and Tao Xie. Learning for Test Prioritization: An Industrial Case Study. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 975–980, New York, NY, USA, 2016. ACM.

[14] Ashok Chandrashekar, Fernando Amat, Justin Basilico, and Tony Jebara. Artwork personalization at netflix. `https://medium.com/netflix-techblog/artwork-personalization-c589f074ad76`. Accessed: 2019-07-16.

[15] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. Optimizing test prioritization via test distribution analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 656–667, New York, NY, USA, 2018. Association for Computing Machinery.

[16] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[17] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning, 2014. arXiv:1410.0759.

[18] Chih-Chun Wang, Sanjeev R. Kulkarni, and H. Vicent Poor. Bandit problems with side observations. *Transactions on Automatic Control*, 50(3):338–355, March 2005.

[19] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[20] François Chollet et al. Keras. `https://keras.io`, 2015.

[21] Anuradha Chug and Shafali Dhall. Software defect prediction using supervised learning algorithm and unsupervised learning algorithm. In *Confluence 2013: The Next Generation Information Technology Summit (4th International Conference)*, pages 173–179, 2013.

[22] Nicolas de Condorcet. *Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix*. Cambridge Library Collection - Mathematics. Cambridge University Press, 2014.

[23] Adele Cutler and Guohua Zhao. Pert-perfect random tree ensembles. *Computing Science and Statistics*, 33, 01 2001.

[24] Istvan Czibula, Gabriela Czibula, Zsuzsanna Onet-Marian, and Vlad-Sebastian Ionescu. A novel approach using fuzzy self-organizing maps for detecting software faults. *Studies in Informatics and Control*, 25, 06 2016.

[25] Enrique A. Da Roza, Jackson A. Prado Lima, Rogério C. Silva, and Silvia Regina Vergilio. Machine learning regression techniques for test case prioritization in continuous integration

environment. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 196–206, 2022.

[26] Márcio Eduardo Delamaro, José Carlos Maldonado, and Mário Jino. *Introdução ao teste de software, Cap - 1*. Elsevier, 2016.

[27] Thomas G. Dietterich. Machine learning for sequential data: A review. In Terry Caelli, Adnan Amin, Robert P. W. Duin, Dick de Ridder, and Mohamed Kamel, editors, *Structural, Syntactic, and Statistical Pattern Recognition*, pages 15–30, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[28] Jackson Antonio do Prado Lima and Silvia Regina Vergilio. A multi-armed bandit approach for test case prioritization in continuous integration environments. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[29] Jackson Antonio do Prado Lima and Silvia Regina Vergilio. Test case prioritization in continuous integration environments: A systematic mapping study. *Information and Software Technology*, 121:106268, 2020.

[30] Álvaro Fialho. *Adaptive operator selection for optimization*. PhD thesis, Université Paris Sud-Paris XI, 2010.

[31] Marek Fisz. The limiting distribution of a function of two independent random variables and its statistical application. *Colloquium Mathematicae*, 3(2):138–146, 1955.

[32] M. Friedman. A comparison of alternative tests of significance for the problem of m rankings. *The Annals of Mathematical Statistics*, 11(1):86–92, 1940.

[33] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Mach. Learn.*, 63(1):3–42, apr 2006.

[34] Xavier Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research - Proceedings Track*, 9:249–256, 01 2010.

[35] Christoph Goller and Andreas Kuchler. Learning task-dependent distributed representations by backpropagation through structure. In *Proceedings of International Conference on Neural Networks (ICNN 96)*, volume 1, pages 347–352 vol.1, 1996.

[36] L. Guo, Y. Ma, B. Cukic, and Harshinder Singh. Robust prediction of fault-proneness by random forests. In *15th International Symposium on Software Reliability Engineering*, pages 417–428, 2004.

[37] Nicolas Gutowski, Tassadit Amghar, Olivier Camp, and Fabien Chhel. Global versus Individual Accuracy in Contextual Multi-Armed Bandit. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC'19, page 1647–1654, New York, NY, USA, 2019. Association for Computing Machinery.

[38] L.K. Hansen and P. Salamon. Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(10):993–1001, 1990.

[39] Scott Hartshorn. Machine learning with random forests and decision trees. *Kindle edition*, 2016.

[40] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, July 2006.

[41] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[42] Jiunn Tzon Gene Hwang and Aidong Adam Ding. Prediction intervals for artificial neural networks. *Journal of the American Statistical Association*, 92(438):748–757, 1997.

[43] Michael Irwin Jordan. Serial order: A parallel distributed processing approach. Technical Report ICS Report 8604, Institute for Cognitive Science, University of California, San Diego, 1986.

[44] Leslie Pack Kaelbling. Associative reinforcement learning: Functions in k-dnf. *Machine Learning*, 15(3):279–298, Jun 1994.

[45] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *J. Artif. Int. Res.*, 4(1):237–285, may 1996.

[46] Isak Karlsson. *Order in the random forest*. PhD thesis, Stockholm University, Department of Computer and Systems Sciences, 2017.

[47] Andrei Karpathy, George Toderici, Sanketh Shetty, Thomas. Leung, Rahul. Sukthankar, and Li. Fei-Fei. Large-scale video classification with convolutional neural networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[48] Eda Kavlakoglu. Ai vs. machine learning vs. deep learning vs. neural networks: What's the difference?, 2020. URL https://www.ibm.com/cloud/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks.

[49] Jung-Min Kim and Adam Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, page 119–129, New York, NY, USA, 2002. Association for Computing Machinery.

[50] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013. cite arxiv:1312.6114.

[51] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Everest Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[52] William H. Kruskal and W. Allen Wallis. Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association*, 47(260):583–621, 1952.

[53] Miroslav Kubat. *An Introduction to Machine Learning*. Springer Publishing Company, Incorporated, 1st edition, 2015.

[54] Volodymyr Kuleshov and Doina Precup. Algorithms for multi-armed bandit problems. *arXiv preprint arXiv:1402.6028*, 2014.

[55] John Langford and Tong Zhang. The Epoch-Greedy Algorithm for Multi-armed Bandits with Side Information. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Proceedings of the Advances in Neural Information Processing Systems*, pages 817–824. Curran Associates, Inc., 2008.

[56] Marko Leppänen, Simo Mäkinen, Max Pagels, Veli-Pekka Eloranta, Juha Itkonen, Mika V. Mäntylä, and Tomi Männistö. The highways and country roads to continuous deployment. *IEEE Software*, 32(2):64–72, 2015.

[57] Ke Li, Álvaro Fialho, Sam Kwong, and Qingfu Zhang. Adaptive operator selection with bandits for a multiobjective evolutionary algorithm based on decomposition. *Evolutionary Computation, IEEE Transactions on*, 18(1):114–130, 2014.

[58] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. A Contextual-bandit Approach to Personalized News Article Recommendation. In *Proceedings of the 19th International Conference on World Wide Web*, WWW'10, pages 661–670, New York, NY, USA, 2010. ACM.

[59] Lihong Li, Wei Chu, John Langford, and Xuanhui Wang. Unbiased Offline Evaluation of Contextual-bandit-based News Article Recommendation Algorithms. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, WSDM'11, pages 297–306, New York, NY, USA, 2011. ACM.

[60] Zheng Li, Mark Harman, and Robert M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, 2007.

[61] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E. Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

[62] Frederick Livingston. Implementation of breiman's random forest machine learning algorithm. *Machine Learning Journal Paper*, 01 2005.

[63] Haroon Malik, Hadi Hemmati, and Ahmed E. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1012–1021, 2013.

[64] Behrooz Mamandipoor, Mahshid Majd, Mostafa Sheikhalishahi, Claudio Modena, and Venet Osmani. Monitoring and detecting faults in wastewater treatment plants using deep learning. *Environmental Monitoring and Assessment*, 192, 02 2020.

[65] Henry B. Mann and Donald R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[66] Dusica Marijan. Multi-perspective Regression Test Prioritization for Time-Constrained Environments. In *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security*, QRS'15, pages 157–162, Washington, DC, USA, 2015. IEEE Computer Society.

[67] Dusica Marijan, Arnaud Gotlieb, and Marius Liaaen. A learning algorithm for optimizing continuous integration development and testing practice. *Software: Practice and Experience*, 49(2):192–213, 2019.

[68] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study. In *IEEE International Conference on Software Maintenance*, pages 540–543. IEEE, September 2013.

[69] Dusica Marijan, Marius Liaaen, Arnaud Gotlieb, Sagar Sen, and Carlos Ieva. TITAN: Test Suite Optimization for Highly Configurable Software. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, ICST, pages 524–531. IEEE, March 2017.

[70] Noha Medhat, Sherin M. Moussa, Nagwa Lotfy Badr, and Mohamed F. Tolba. A framework for continuous regression and integration testing in IoT systems based on deep learning and search-based techniques. *IEEE Access*, 8:215716–215726, 2020.

[71] Harlan Ducan Mills. The management of software engineering, Part I: Principles of software engineering. *IBM Systems Journal*, 19(4):414–420, 1980.

[72] Begum Momotaz and Tadashi Dohi. Prediction interval of cumulative number of software faults using multilayer perceptron. In Roger Lee, editor, *Applied Computing & Information Technology*, pages 43–58, Cham, 2016. Springer International Publishing.

[73] Glenford James. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011.

[74] Qi Pang, Yuanyuan Yuan, and Shuai Wang. Mdpfuzz: Testing models solving markov decision processes. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 378–390, New York, NY, USA, 2022. Association for Computing Machinery.

[75] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[76] Jerônimo Pellegrini and Jacques Wainer. Processos de decisão de markov: um tutorial. *Revista de Informática Teórica e Aplicada*, 14(2):133–179, Dec. 2007.

[77] Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and S. S. Iyengar. A survey on deep learning: Algorithms, techniques, and applications. *ACM Comput. Surv.*, 51(5), September 2018.

[78] Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and Sundaraja Sitharama Iyengar. A survey on deep learning: Algorithms, techniques, and applications. *ACM Comput. Surv.*, 51(5), September 2018.

[79] Jackson A. Prado Lima and Silvia R. Vergilio. Supplementary material - a multi-armed bandit approach for test case prioritization in continuous integration environments, 2019. URL https://osf.io/epnuk/.

[80] Roger Pressman. *Engenharia de Software - 7.ed, cap - 1.* McGraw Hill Brasil, 2009.

[81] Mojtaba Raeisi Nejad Dobuneh, Dayang Jawawi, Mojtaba Vahidi-Asl, and Mohammad Malakooti. Clustering test cases in web application regression testing using self-organizing maps. *International Journal of Advances in Soft Computing and its Applications*, 7:1–14, 01 2015.

[82] Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527 – 535, 1952.

[83] Paulo Roberto Farah, Thainá Mariani, Enrique A. da Roza, Rogério C. Silva, and Silvia Regina Vergilio. Unsupervised learning for refactoring pattern detection. In *2021 IEEE Congress on Evolutionary Computation (CEC)*, pages 2377–2384, 2021.

[84] Gregg Rothermel and Mary J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[85] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

[86] Enrique A. Roza, Jackson A. Prado Lima, Rogério C. Silva, and Silvia R. Vergilio. Supplementary Material - Machine Learning Regression Techniques for Test Case Prioritization in Continuous Integration Environment, January 2022. URL https://osf.io/bek98/?view_only=4e9ea3b608924998aeb3f5fb80baa5f2.

[87] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, USA, 1st edition, 2001.

[88] Richard Socher, Cliff Chiung-Yu Lin, Andrew Y. Ng, and Christopher D. Manning. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, page 129–136, Madison, WI, USA, 2011. Omnipress.

[89] Ian Sommerville. *Engenharia de software, cap - 8*. Pearson Brasil, 2011.

[90] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 12–22, New York, NY, USA, 2017. Association for Computing Machinery.

[91] Jennifer Stapleton. *DSDM, dynamic systems development method*. Addison-Wesley, Harlow, England, 1997.

[92] Alexander L. Strehl, Chris Mesterharm, Michael L. Littman, and Haym Hirsh. Experience-efficient learning in associative bandit problems. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML'06, pages 889–896, New York, NY, USA, 2006. ACM.

[93] Richard S. Sutton and Andrew G. Barto. Reinforcement learning: An introduction. *The MIT Press*, 2011.

[94] Michael E. Tipping and Christopher M. Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 61(3):611–622, 1999.

[95] Andras Vargha and Harold D. Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, January 2000.

[96] Richard D. De Vieaux, Jennifer Schumi, Jason Schweinsberg, and Lyle H. Ungar. Prediction intervals for neural networks via nonlinear regression. *Technometrics*, 40(4):273–282, 1998.

[97] Sofía S. Villar, Jack Bowden, and James Wason. Multi-armed Bandit Models for the Optimal Design of Clinical Trials: Benefits and Challenges. *Statistical Science*, 30(2):199–215, 05 2015.

[98] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.

[99] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. A systematic literature review on the use of deep learning in software engineering research, 2020.

[100] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.

[101] Michael Woodroofe. A One-Armed Bandit Problem with a Concomitant Variable. *Journal of the American Statistical Association*, 74(368):799–806, 1979.

[102] Le Xiao, Huaikou Miao, Tingting Shi, and Yu Hong. LSTM-based deep learning for spatial–temporal software testing. *Distributed and Parallel Databases*, 38:687–712, 2020.

[103] Wang Xin, Wu Ji, Liu Chao, Yang Haiyan, Du Yanli, and Niu Wen-sheng. Exploring LSTM based recurrent neural network for failure time series prediction. *Beijing Hangkong Hangtian Daxue Xuebao/Journal of Beijing University of Aeronautics and Astronautics*, 44:772–784, 04 2018.

[104] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012.

[105] Zhe Yu, Fahmid Fahid, Tim Menzies, Gregg Rothermel, Kyle Patrick, and Snehit Cherian. TERMINATOR: better automated UI test case prioritization. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, FSE, pages 883–894. ACM, 2019.

[106] Qingchen Zhang, Laurence T. Yang, Zhikui Chen, and Peng Li. A survey on deep learning for big data. *Information Fusion*, 42:146–157, 2018.

[107] Li Zhou. A Survey on Contextual Multi-armed Bandits. *CoRR*, abs/1508.03326, 2015.