UNIVERSIDADE FEDERAL DE JUIZ DE FORA
INSTITUTO DE CIENCIAS EXATAS
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Camila Acácio de Paiva

UMA INFRAESTRUTURA PARA APOIAR TESTES DE REGRESSÃO ATRAVÉS DE PROVENIÊNCIA E PREVISÃO DOS RESULTADOS DE TESTES DE UNIDADE

Juiz de Fora
2018

Camila Acácio de Paiva

UMA INFRAESTRUTURA PARA APOIAR O PROCESSO DE TESTE DE SOFTWARE ATRAVÉS DE PROVENIÊNCIA E PREVISÃO DOS RESULTADOS DE TESTES DE UNIDADE

Dissertação apresentada ao Programa de Pós-graduação em Ciência da Computação, da Universidade Federal de Juiz de Fora como requisito parcial a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. D.Sc. Marco Antônio Pereira Araújo

Juiz de Fora
2018

Ficha catalográfica elaborada através do programa de geração
automática da Biblioteca Universitária da UFJF,
com os dados fornecidos pelo(a) autor(a)

Acácio de Paiva, Camila.
Uma infraestrutura para apoiar testes de regressão através de proveniência e previsão dos resultados de testes de unidade / Camila Acácio de Paiva. -- 2018.
128 f.

Orientador: Marco Antônio Pereira Araújo
Dissertação (mestrado acadêmico) - Universidade Federal de Juiz de Fora, ICE/Engenharia. Programa de Pós-Graduação em Ciência da Computação, 2018.

1. Engenharia de Software Contínua. 2. Testes de Regressão. 3. Proveniência de dados. 4. Algoritmos de Previsão. I. Pereira Araújo, Marco Antônio, orient. II. Título.

Camila Acácio de Paiva

UMA INFRAESTRUTURA PARA APOIAR O PROCESSO DE TESTE DE SOFTWARE ATRAVÉS DE PROVENIÊNCIA E PREVISÃO DOS RESULTADOS DE TESTES DE UNIDADE

Dissertação apresentada ao Programa de Pós-graduação em Ciência da Computação, da Universidade Federal de Juiz de Fora como requisito parcial a obtenção do grau de Mestre em Ciência da Computação.

Aprovada em 27 de Junho de 2018

BANCA EXAMINADORA

_____

Prof. D.Sc. Marco Antônio Pereira Araújo - Orientador
Universidade Federal de Juiz de Fora

_____

Prof. D.Sc. José Maria Nazar David
Universidade Federal de Juiz de Fora

_____

Profa. D.Sc. Vera Maria Benjamim Werneck
Universidade Estadual do Rio de Janeiro

AGRADECIMENTOS

A Deus por permitir alcançar este objetivo.

À minha família, em especial, aos meus pais Edmar e Kátia, e meu irmão Felipe, por me apoiarem nos momentos de dificuldades e estarem sempre ao meu lado.

Ao meu namorado Sérgio, que acompanhou tudo de perto, sempre me apoiando e me incentivando a melhorar.

Ao meu orientador, Marco Antônio, por todo o apoio, por todas as colaborações, pela paciência, dedicação, aprendizado, incentivo e auxílio no decorrer de toda a Graduação e Mestrado.

Ao professor José Maria, pela dedicação, colaboração, incentivo e aprendizado propiciados.

À professor Vera Maria e ao professor José Maria pelas considerações e contribuições para a conclusão desse trabalho.

A todos os professores do Departamento de Ciência da Computação da UFJF, em especial aos professores do núcleo de Engenharia de Software, por todo conhecimento adquirido durante os períodos do mestrado.

Aos meus amigos de longa data, Heleno e Maria Luiza, agradeço pelo apoio nos momentos mais difíceis e pelas contribuições para a realização deste trabalho.

Aos amigos do NEnC, André Abdalla, Claudio Lélis, Heitor Magaldi, Hugo Guércio, Iuri Carvalho, Lenita Ambrósio, Leonardo Pereira, Pedro Ivo e Phillipe Marques. Agradeço pelos bons momentos vividos em laboratório e pelas discussões que contribuíram com este trabalho.

"O que é seu encontrará um
caminho para chegar até você."

Caio Fernando de Abreu

RESUMO

O software está cada vez mais presente no cotidiano das pessoas. Vários setores ou aspectos do ambiente são influenciados por ele. Desta forma, o desenvolvimento de software torna-se uma atividade crítica. Assim, o processo de teste se torna crucialmente importante, pois qualquer negligência pode refletir na qualidade do produto.

Contudo, o cenário de desenvolvimento de software vem sofrendo mudanças a partir da necessidade de suprir demandas com maior agilidade e as exigências do mercado. É fundamental haver uma visão holística dos processos de desenvolvimento do software com o objetivo de gerar um ciclo de melhoria contínua. Tal visão é denominada como Engenharia de Software Contínua.

A Engenharia de Software Contínua é caracterizada pelo uso do *feedback* de execuções para alcançar uma melhoria contínua e pela realização das atividades de maneira contínua. *Feedback* esse que pode ser fornecido através da proveniência de dados: descrição das origens de um dado e os processos pelos quais passou. Diante disso, este trabalho apresenta uma infraestrutura que tem como foco a captura e o armazenamento do histórico dos dados de execução do projeto, a previsão dos resultados dos testes de unidade através de algoritmos de previsão *Logistic Regression, Naive Bayes e C4.5 Algorithm,* e a disponibilização dos dados para aplicações externas. Além disso, oferece elementos de visualização que auxiliam na compreensão dos dados. Um experimento com dados de um projeto real foi realizado com o intuito de identificar a acurácia das previsões.

Palavras-chave: Engenharia de Software Contínua, Testes de Regressão, Proveniência de dados, Algoritmos de Previsão.

ABSTRACT

Software is increasingly present in people's daily lives. Various sectors and the environment are influenced by them. In this way, the development of software becomes a critical activity. Thus, the testing process becomes crucially important because any negligence can affect the quality of the product and the insecurity related to the use of the software.

However, the software development scenario has undergone changes from the need to meet demands with greater agility and the demands of the market. It is fundamental to have a holistic view of the software development processes in order to generate a cycle of continuous improvement. Such a view is referred to as Continuous Software Engineering.

Continuous Software Engineering is characterized by the use of feedback from executions to achieve continuous improvement and by performing activities on an ongoing basis. Feedback this can be provided through the provenance of data: description of the origins of a given and the processes by which it passed. This work presents an infrastructure that focuses on the capture and storage of the project execution data history, the prediction of the results of the unit tests through prediction algorithms: Logistic Regression, Naive Bayes and C4.5 Algorithm, and the provision of data for external applications. It also provides preview elements which help in understanding the data. An experiment with data from a real project was carried out in order to identify the accuracy of the prediction.

Keywords: Continuous Software Engineering, Regression Tests, Data Provenance, Prediction Algorithms.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API – *Application Programming Interface*

ARFF - *Attribute-Relation File Format*

ESC - Engenharia de Software Contínua

GQM – *Goal/Question/Metrics*

IA - Inteligência Artificial

IC - Integração Contínua

IEEE - *Institute of Electrical and Electronics Engineers*

OPM - *Open Provenance Model*

OWL - *Web Ontology Language*

PICOC – *Population/Intervention/Comparison/Outcome/Context*

PROV-DM – PROV Data Model

PROV-O - PROV *Ontology*

ST - S-Transform

UFJF - Universidade Federal de Juiz de Fora

V&V - Validação e Verificação

W3C - *World Wide Web Consortium*

XML - *Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

O software está cada vez mais presente no cotidiano das pessoas. Vários setores ou aspectos do ambiente são influenciados por ele. Dessa forma, o desenvolvimento de software torna-se uma atividade crítica que precisa ser cuidadosamente estudada, compreendida, melhorada e apoiada.

Segundo SOMMERVILLE (2011), existem vários processos de software diferentes, mas todos devem incluir quatro fundamentais atividades: especificação de software, projeto e implementação do software, validação de software e evolução de software. Tais atividades incluem subatividades como, por exemplo, os testes.

Existem duas categorizações para os processos de software: dirigido a planos, que são aqueles em que as atividades são planejadas com antecedência e seguem planejamento inicial e os processos ágeis, onde o processo é gradativo e se adapta às necessidades do cliente (SOMMERVILLE, 2011).

O cenário de desenvolvimento de software vem sofrendo mudanças a partir da necessidade de suprir demandas com maior agilidade e as exigências do mercado. Segundo (FITZGERALD; STOL, 2017), em resposta a esse fenômeno, empresas têm adotado, de forma oblíqua, práticas ágeis de desenvolvimento de software. Segundo SOMMERVILLE (2011), desenvolvimento ágil é caracterizado por produzir software rapidamente e pelo desenvolvimento incremental em que pequenas funcionalidades são entregues aos clientes assim que possível.

Contudo, práticas adotadas apenas a nível de desenvolvimento e operacional não são suficientes, é necessário integrar ao nível estratégico das empresas. Dessa forma, é fundamental haver uma visão holística dos processos de desenvolvimento do software com o objetivo de gerar um ciclo de melhora contínua. Tal visão é denominada como Engenharia de Software Contínua (ESC) (FITZGERALD; STOL, 2017).

Caracterizam a Engenharia de Software Contínua o uso do *feedback* de execuções para obter uma melhoria contínua e a realização das atividades de maneira contínua. Dentre essas atividades, tem-se a integração contínua. Nesse processo, cada mudança realizada no software é integrada à versão principal através de ferramentas que automatizam o processo, como o Jenkins[1]. A cada integração, os testes de regressão

[1] https://jenkins-ci.org/

devem ser executados visando garantir que, ao adicionar novos componentes ou alterações, o conjunto continue funcionando como planejado, ou como anteriormente à adição.

Durante o processo, essas tarefas geram uma série de integrações e execuções. Em cada uma dessas execuções, por exemplo, diferentes configurações podem ser utilizadas, incluindo o uso de processamento paralelo, diferentes técnicas de priorização, seleção, ou minimização de casos de teste. Logo, são gerados dados que poderiam ser armazenados e utilizados como *feedback* dos processos.

Manter os dados gerados pelas execuções, descrever suas origens e os processos pelos quais ele passou, segundo (BUNEMAN; KHANNA; TAN, 2001) é denominado como proveniência de dados. Tal definição aplicada à engenharia de software pode auxiliar a melhoria dos processos. Um exemplo dessa aplicação pode ser observado no trabalho de DAPRA *et al.* (2015), no qual, cria-se uma abordagem denominada PROV-*Process*, responsável por analisar dados de execução de processos de desenvolvimento de software. Nesse mesmo sentido, este trabalho propõe o uso da proveniência para a captura de dados da execução de testes de regressão com o objetivo de auxiliar na melhoria do processo de teste.

Contudo, apenas capturar os dados não é suficiente. É necessário refinar os dados para que se possa obter informações que auxiliem na melhoria do processo. Com esse objetivo, a proveniência de dados pode ser utilizada através de algoritmos que se adaptam às caraterísticas do conjunto de dados. Estes algoritmos são chamados de Algoritmos de Aprendizagem de Máquina e que, segundo SIMON (2013), podem ser definidos como o "campo de estudo que dá aos computadores a habilidade de aprender sem serem explicitamente programados".

Os Algoritmos de Aprendizagem de Máquina são aplicados em diversas áreas e para diferentes fins. Um exemplo dessa aplicação pode ser constatado em (HADI *et al.*, 2010), o qual aplica a técnica *S-Transform (ST)* para extrair recursos de sons cardíacos. Esses recursos são utilizados como entrada para o classificador, *Multilayer Perceptron Network.* Como resultados, os autores obtiveram 98% de acerto na classificação. Nesse mesmo sentido, este trabalho propõe o uso dos algoritmos de predição *Logistic Regression, Naive Bayes e C4.5 Algorithm,* para prever os resultados dos testes de unidade através de um conjunto de treinamento. Tais algoritmos foram selecionados por serem os mais utilizados na área e pesquisa em questão, chegou-se a esta conclusão a partir de uma revisão de literatura realizada e abordada na Seção 3.

Mediante os conceitos acima relacionados, segue-se a motivação, bem como o problema levantado, o enfoque da solução encontrada e por fim, o objetivo.

1.1 MOTIVAÇÃO

Como a Engenharia de Software Contínua (ESC) se baseia no uso do *feedback* de execuções para alçar uma melhoria contínua, e pela realização das atividades de maneira contínua, um grande número de informações são geradas diariamente. Com a adoção, cada vez maior, da ESC, faz-se necessário medidas que auxiliem o *feedback.*

Como forma dar suporte ao *feedback*, desenvolver aplicações que mantenham o histórico dos dados contribuem de diversas maneiras. A seguir, algumas delas são apresentadas:

- Evitar problemas recorrentes. Diversos contratempos, por exemplo, falhas de casos de teste, ou não execução de algum teste, podem ocorrer durante a execução dos testes de regressão de uma versão de um software. A coleta de dados sobre estas situações pode auxiliar a evitar que elas ocorram novamente no futuro;
- Diferentes técnicas de priorização/seleção/minimização de casos de teste, como por exemplo, executar somente os testes que possuem maior probabilidade de falha, podem ser utilizadas para otimizar a execução dos testes de regressão. A captura dos dados desse processo pode servir para a escolha futura de técnicas que garantam um melhor resultado, assegurando encontrar falhas mais rapidamente;
- Buscar por melhores configurações. Diversas configurações, como diferentes técnicas de priorização/seleção/minimização e diferentes ambientes, podem ser utilizados ao longo da vida de um projeto. A coleta de dados sobre estas configurações pode auxiliar a encontrar melhores e mais rápidas, composições para a execução do projeto.
- Realizar previsões sobre o software utilizando Algoritmos de Aprendizado de Máquina. Por exemplo, a captura de dados sobre as execuções do *build* e testes podem servir como entrada para o treinamento dos algoritmos, para então, realizar as previsões sobre falhas dos *builds* ou falhas de testes de unidade, evitando a espera pelo *feedback;*

Desse modo, estas informações podem auxiliar desenvolvedores e analistas de teste a entender e gerenciar como os próximos *builds* e testes devem ocorrer, visando sempre que haja melhorias no processo através da diminuição do número de *builds* diários, tempo de espera pelo *feedback* e execução e erros recorrentes.

1.2 PROBLEMA

Diante desse cenário, o problema tratado neste trabalho refere-se à forma (modelos ou algoritmos) pela qual seja possível apoiar a captura dos dados gerados na execução de testes de regressão para serem utilizados como *feedback* para melhoria contínua no contexto de engenharia contínua de software.

1.3 ENFOQUE DA SOLUÇÃO

A solução proposta neste trabalho possui enfoque no suporte aos envolvidos no *build* do projeto, sejam eles, desenvolvedores ou analistas de testes. Tem como foco a captura e o armazenamento do histórico dos dados de execução do projeto, a previsão dos resultados dos testes de unidade através de algoritmos de previsão *Logistic Regression, Naive Bayes e C4.5 Algorithm,* e a disponibilização dos dados para que aplicações externas possam utilizá-los visando melhoria do processo de acordo com as especificações de cada aplicação.

1.4 OBJETIVO

Este trabalho tem como objetivo melhorar o processo de testes através da captura e disponibilização de informações sobre as execuções dos *builds* de um projeto. Para isso, serão utilizados algoritmos que visam prever os resultados dos testes de unidade sem que sejam executados. Dessa forma, os envolvidos conseguem diminuir a quantidade de *builds* diários, reduzir o tempo de espera pelo *feedback,* obter ciclos de teste mais rápidos e, consequentemente, assegurar que as entregas sejam realizadas mais depressa e, por fim, fornecer recursos para que aplicações externas utilizem estas informações.

1.5 ESTRUTURA DA DISSERTAÇÃO

Este trabalho foi dividido em 5 capítulos. Segue a sua estrutura:

Capítulo 1 - Introdução: apresenta o contexto no qual a pesquisa está inserida, bem como a motivação, o problema, o enfoque da solução e o objetivo.

Capítulo 2 - Pressupostos Teóricos: são abordados os principais pontos inerentes à Engenharia de Software Contínua, Testes de Regressão, Ontologias e Algoritmos de Aprendizagem de Máquina.

Capítulo 3 – Uma revisão sistemática de literatura: apresenta uma revisão sistemática que objetiva averiguar os algoritmos ou modelos utilizados para a predição de falhas dos testes de software.

Capítulo 4 – Infraestrutura de Apoio ao Teste de Regressão: descreve a arquitetura proposta, bem como, suas funcionalidades, recursos e ferramentas integradas.

Capítulo 5 - Avaliação da infraestrutura: apresenta a avaliação realizada por meio de estudo experimental para analisar a infraestrutura proposta.

Capítulo 6 – Conclusões e Trabalhos Futuros: são apresentas as conclusões e contribuições do trabalho, além de trabalhos futuros para a continuação da pesquisa.

2 PRESSUPOSTOS TEÓRICOS

Este capítulo aborda os principais temas relacionados a esta dissertação, entre eles, verificação e validação de software, testes de regressão, teste de unidade, engenharia de software contínua, integração contínua, proveniência de dados, algoritmos de predição e visualização. Além de abordar estes temas, o capítulo também se destina a embasar o leitor para o entendimento dos próximos capítulos.

2.1 VERIFICAÇÃO E VALIDAÇÃO DE SOFTWARE

Segundo SOMMERVILLE (2011), um processo de software é um conjunto de atividades que se relacionam e levam à produção de um software. Não existe um processo ideal, cada empresa se adapta às suas necessidades, entretanto, existem quatro fundamentais atividades para a engenharia de software. São elas: (i) especificação de software: ocorre a definição das funcionalidades e limitações de um software; (ii) projeto e implementação do software: realiza-se a produção do software; (iii) validação de software: acontece a validação do software com relação às exigências do cliente; e (iv) evolução de software: ocorre a evolução para atender novas necessidades.

A validação e verificação (V&V) de software tem por objetivo mostrar que um software se adequa aos seus requisitos e, ao mesmo tempo, satisfaz às necessidades do cliente (SOMMERVILLE, 2011). Dentre as principais técnicas de validação, tem-se o teste de software.

A atividade de teste é complexa. São diversos fatores que podem contribuir para a ocorrência de defeitos. Por essa razão, a atividade de teste pode ser dívida em fases com objetivos distintos. Em geral, pode-se estabelecer as seguintes fases: teste de unidade, teste de integração, teste de sistema e teste de regressão.

Segundo DELAMARO; MALDONADO e JINO (2007), os testes de unidade são aqueles que focam nas menores unidades do programa: funções, procedimentos, métodos ou classes, a fim de garantir que os aspectos de implementação de cada uma estejam corretos. Tais testes são executados separadamente e à medida que ocorre a implementação das unidades pelos desenvolvedores, sem a necessidade de dispor-se do sistema completamente finalizado. Seu objetivo é identificar defeitos de lógica e de implementação.

Os testes de integração são realizados após os testes de unidade e dão ênfase na construção da estrutura do sistema. Conforme as partes do software vão se integrando, é preciso verificar se as partes funcionam de maneira adequada e não geram erros (DELAMARO; MALDONADO; JINO, 2007).

Após os testes de unidade e integração, inicia-se os testes de sistema. Nessa fase, o objetivo é verificar se as funcionalidades do sistema estão de acordo com o documento de requisitos.

Depois que o sistema está completo, com todas as partes funcionando, entra em processo de manutenção. A cada modificação realizada no sistema, devem-se executar os testes de regressão visando assegurar que novos defeitos não foram introduzidos e conjunto continua funcionando como planejado, ou como anteriormente à adição da modificação.

Como dito anteriormente, a atividade de teste é complexa e demanda tempo. Com o mercado mais exigente e com a maiores necessidades, os processos foram aperfeiçoados através engenharia de software contínua, tema que será discutido na próxima seção.

2.2 ENGENHARIA DE SOFTWARE CONTÍNUA

A indústria de software passou por transições (BOSCH, 2014). Até há alguns anos atrás, o ciclo de vida de desenvolvimento do software era caracterizado por seguir os estágios de análise e definição de requisitos, projeto de sistema e software, implementação e teste unitário. Modelo esse, denominado como cascata e publicado por ROYCE (1987).

Durante a últimas décadas, esse cenário começou a mudar consideravelmente. A frequência das entregas dos softwares cresceram desde o início dos anos 2000 e, 10 anos depois, várias organização estão entregando um novo software múltiplas vezes ao dia (BOSCH, 2014). Tais práticas oferecerem benefícios em termos de qualidade e consistência (MICHLMAYR *et al.*, 2015), como por exemplo, menos falhas e um software mais consistente.

Segundo FITZGERALD e STOL (2017) a adoção persuasiva dos métodos ágeis comprova a necessidade por flexibilidade e rápida adaptação do atual ambiente de desenvolvimento do software. Porém, para que haja uma rápida detecção e correção dos problemas, faz-se necessária uma estreita conexão entre o desenvolvedor e a execução,

o que melhora a qualidade e a resiliência do software. Tal fato se manifesta através da crescente adoção de práticas de integração contínua. Tais práticas, tiveram grande popularidade devido à sua recomendação explícita do método de Programação Extrema (BECK, 2000) e, de fato, a prática é compatível com as frequentes interações do software produzidas pelos processos ágeis.

Entretanto, essas práticas, adotadas somente em termos de desenvolvimento e operacional, não são suficientes para suprir às necessidades do mercado, sendo necessário integrar também o nível estratégico das empresas. Dessa forma, é preciso haver uma visão mais holística dos processos de desenvolvimento do software com o objetivo de gerar um ciclo de melhoria contínua. Essa visão contínua dos processos de desenvolvimento de software é chamada de Engenharia de Software Contínua (FITZGERALD; STOL 2017).

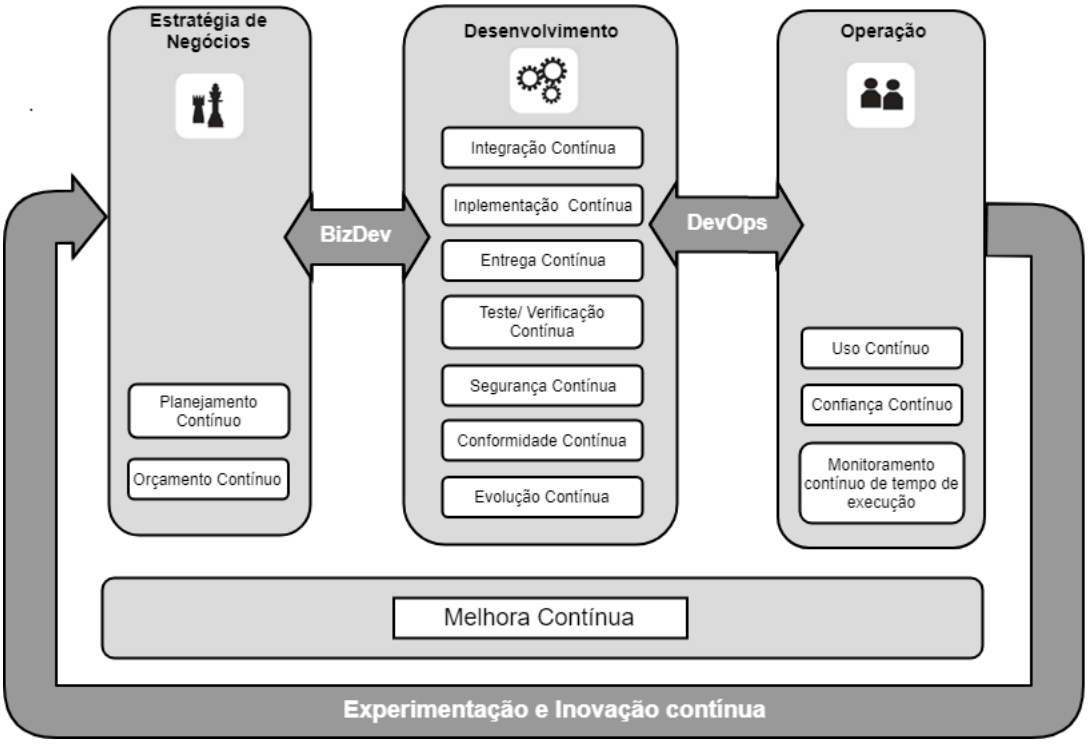
A Engenharia de Software Contínua considera todas as fases do ciclo de vida de um software sendo executadas de forma contínua. Como mostra a Figura 2-1 estas fases foram subdivididas em três subfases: Planejamento e Estratégia de negócios, Desenvolvimento e Operações, cada uma com suas atividades.

A subfase de Planejamento e Estratégia de negócios é composta pelo planejamento contínuo e orçamento contínuo. Durante o planejamento contínuo os planos são artefatos de forma dinâmica e abertos a evolução em resposta às mudanças ocorridas no ambiente de negócios. No orçamento contínuo as receitas e despesas se tornam uma atividade que facilita as mudanças ocorridas.

A subfase de Desenvolvimento e Operações é estabelecida pela: (i) integração contínua, processo o qual visa implantar o software imediatamente aos clientes assim que o novo código é desenvolvido, obtendo *feedback* mais rápido; (ii) entrega contínua, atividade que implementa softwares construídos automaticamente em algum ambiente de forma contínua; (iii) implementação contínua, garante que o software esteja pronto e entregue aos clientes; (iv) teste e verificação contínua, realiza inspeções durante todo o processo de desenvolvimento e testes automatizados que auxiliam a reduzir o tempo entre a introdução de erros e sua detecção, eliminando as causas de forma mais eficaz; (v) segurança contínua, transforma a segurança como uma preocupação fundamental em todas as fases do ciclo de vida de desenvolvimento, até mesmo após a implantação; conformidade contínua, visa buscar a satisfação dos padrões de conformidade de maneira contínua; e (vi) evolução contínua, que permite a evolução do software de maneira contínua.

A subfase de Operações é composta pelas fases de uso contínuo, que visa reter os clientes ao invés de procurar por novos; confiança contínua, confiança desenvolvida ao longo do tempo como resultado de interações baseadas na crença de que um fornecedor agirá de forma cooperativa para atender às expectativas do cliente sem explorar suas vulnerabilidades; e por fim, monitoramento contínuo, que visa prevenir problemas de qualidade de serviço através do monitoramento em tempo de execução.

Figura 2-1. Visão geral da engenharia de software contínua



Fonte: (Fitzgerald; Stol 2017)

Estas subfases, ao serem utilizadas em conjunto, podem prover a melhoria contínua nos processos de desenvolvimento de software de uma empresa.

No contexto deste trabalho, as principais atividades impactadas são integração contínua, implantação contínua, entrega contínua e teste contínuo. Tais atividades são detalhadas a seguir.

2.2.1 INTEGRAÇÃO CONTÍNUA

Dentre todas as atividades da Engenharia de Software Contínua, a Integração Contínua (IC) é a mais conhecida, isso, graças ao fato de a IC ser uma prática explícita no

método, muito popular, chamado Programação Extrema (XP) (BECK, 2000). A IC pode ser definida como o processo que normalmente é realizado automaticamente e assimila etapas interconectadas, tais como: detecção das mudanças, compilação de código, testes de unidade, testes de regressão e construção dos pacotes (FITZGERALD; STOL 2017).

Entretanto, segundo SMART (2011), a IC pode fazer muito mais. Auxilia os desenvolvedores a controlar a saúde do seu código; monitorar, automaticamente, a qualidade e as métricas de cobertura do código; reduzir a dívida técnica e manter os custos de manutenção baixos; e, simplificar e acelerar a entrega, ajudando na automatização do processo de implantação.

SMART (2011) afirma que, em essência, a IC visa reduzir riscos e fornecer o *feedback* mais rápido. Tal prática foi projetada para ajudar a identificar e corrigir problemas de integração e regressão mais rapidamente, resultando em entregas mais rápidas e com menos erros. Sendo assim, ao automatizar o processo de implantação, a IC auxilia a entregar o software aos analistas de teste e dos usuários finais de forma mais rápida, confiável e com menos esforço.

Para retirar o maior proveito dessa prática, a equipe deve adotar esse paradigma. Seus projetos devem ter um processo de compilação confiável, repetitivo e automatizado, sem nenhuma intervenção humana. Corrigir as falhas deve ser prioridade absoluta. O processo de implantação deve ser automatizado, sem etapas manuais e os testes precisam ser de alta qualidade. Para a realização de tais atividades, muitas organizações adotam a utilização de ferramentas para auxiliar o processo. Dentre estas ferramentas, tem-se o Jenkins[2], um dos servidores de integração contínua mais utilizados na atualidade. A seção seguinte descreve esta ferramenta e suas funcionalidades.

2.2.1.1 JENKINS

Jenkins[3], originalmente Hudson, é um Servidor de Integração Contínua escrito em Java. Em 2010, Hudson tornou-se a principal ferramenta de integração contínua do mercado, com percentual de aceitação acima de 70 (SMART, 2011).

Em 2009, a Oracle comprou a Sun. No final de 2010, surgiram desentendimentos entre a comunidade de desenvolvedores Hudson e a Oracle sobre

[2]https://jenkins-ci.org/
[3] https://jenkins-ci.org/

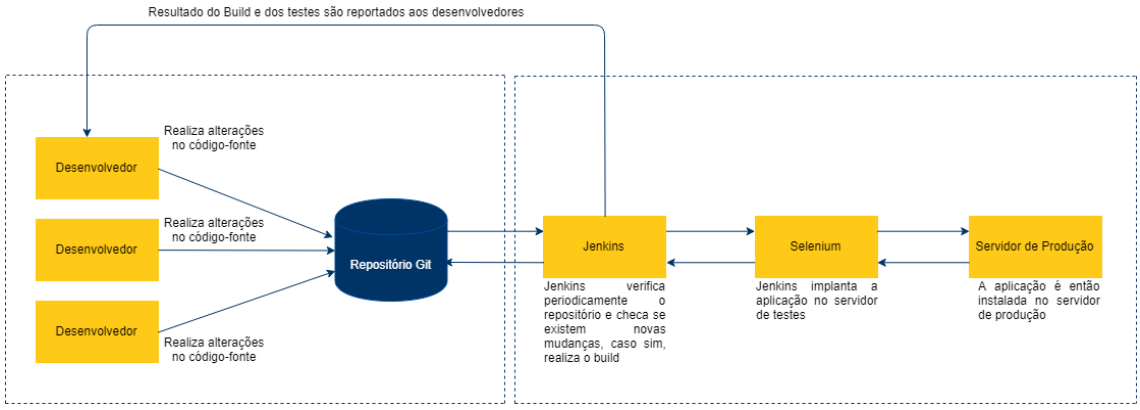
como avançar no desenvolvimento da ferramenta. Com isso, em 2011, a comunidade de desenvolvedores Hudson decidiu votar na renomeação do projeto para Jenkins. Em seguida, migraram a base original do código para um novo projeto no GitHub[4] e prosseguiram seu trabalho (SMART, 2011).

Após o ocorrido, a maioria dos usuários também seguiu os passos da comunidade Hudson. Em 2011, pesquisas mostraram que 75% dos usuários do Hudson mudaram para Jenkins, enquanto 13% ainda estavam usando Hudson e outros 12% estavam usando Hudson e Jenkins e estavam migrando para o Jenkins (SMART, 2011).

O servidor de integração contínua é utilizado para construir, testar, integrar mudanças, entregar e implantar softwares de maneira contínua e fácil. Com o Jenkins, as organizações podem acelerar o processo de desenvolvimento de software através da automação.

A ferramenta é adaptável às necessidades do usuário. Possui uma grande gama de *plug-ins*, e estes permitem a integração a vários estágios do desenvolvimento. Como exemplo, *plug-ins* são adicionados para cada fase de acordo com à necessidade do usuário: para o *build* do projeto, foi adicionado o Maven, para o controle de versões, o Git, para os testes, o Selenium.

Figura 2-2. Processo de Integração Contínua



Fonte: https://www.edureka.co/blog/what-is-jenkins/

A Figura 2-2 exemplifica o processo de Integração Contínua com o Jenkins. O processo começa assim que o desenvolvedor realiza o *commit* das mudanças no repositório. Após o servidor detecta as mudanças ocorridas, resgata estas mudanças e se prepara para o novo *build*. Caso o *build* falhe, o usuário será notificado. Caso o *build*

[4] https://github.com/jenkinsci

ocorra com sucesso, os testes são executados. Por fim, o Jenkins gera um *feedback* e notifica o desenvolvedor sobre o *build* e os resultados dos testes.

O Jenkins possui grandes vantagens, o que o torna, um dos mais utilizados servidores de integração contínua na atualidade, são elas: é uma ferramenta *open source*, possui mais de mil *plug-ins* que facilitam o trabalho e a personalização, permite que os usuários desenvolvam seus próprios *plug-ins*, é uma ferramenta gratuita e foi desenvolvido em Java, sendo portátil para outras plataformas.

Diante das suas características, o Jenkins foi selecionado, neste trabalho, como o servidor de integração contínua responsável por gerar os relatórios utilizados pela proposta na fase da extração de dados. Outra ferramenta responsável por gerar informações utilizadas é o SonarQube, descrito a seguir.

2.2.1.2 SONARQUBE

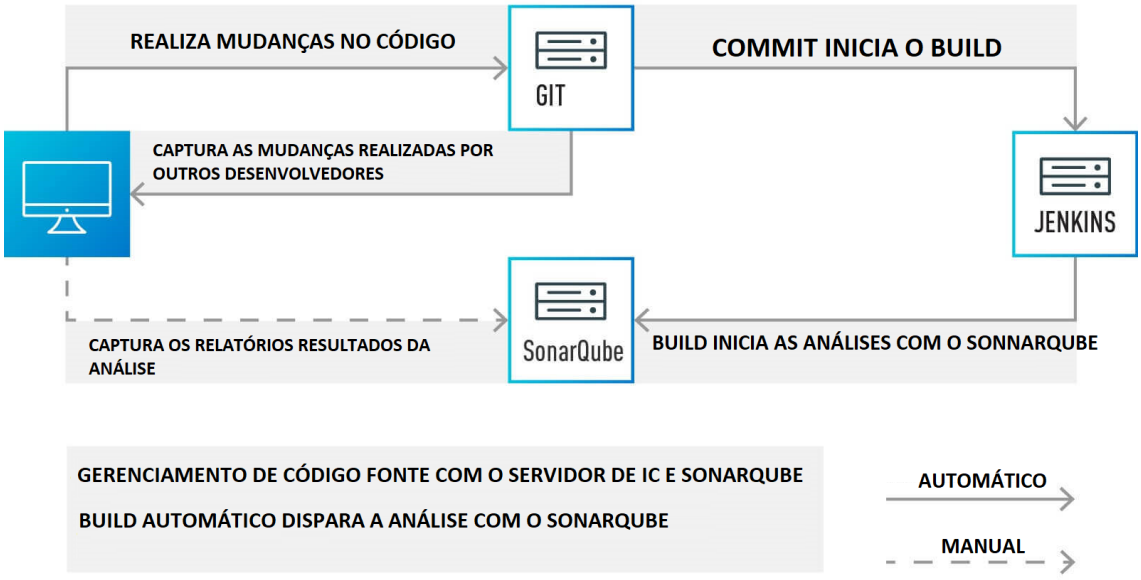
O SonarQube é uma plataforma de código aberto para realizar revisões automáticas com análise estática de código para detectar erros, mal cheiros de código e vulnerabilidades de segurança em mais de 20 linguagens de programação, incluindo Java, C #, JavaScript, TypeScript, C / C ++, COBOL e outras (BELLINGARD, 2018).

O projeto iniciou no ano de 2007 e seus fundadores tinham como objetivo oferecer a cada desenvolvedor a capacidade de medir a qualidade do código dos seus projetos. Seu lema era: “A inspeção contínua deve se tornar rotina assim como a integração contínua” (BELLINGARD, 2018).

Através da automação, o processo de análise estática do código gasta menos tempo e pode ser realizado diversas vezes por dia. Assim, se encaixa perfeitamente dentro da integração contínua. A Figura 2-3, exemplifica o processo de análises estáticas do código dentro de um servidor de integração contínua, no caso, o Jenkins. O processo começa assim que o desenvolvedor realiza o *commit* das mudanças no repositório. Após o *commit*, o servidor detecta as mudanças ocorridas, resgata estas mudanças e se prepara para o novo *build*. Durante o *build*, o servidor executa a SonarQube e disponibiliza as informações aos usuários (BELLINGARD, 2018).

O uso do SonarQube facilita o controle de qualidade do código e diminui o número de erros reais e potenciais. Além disso, mantém o histórico de como estas análises estáticas evoluíram, através da proveniência, tema da seção seguinte, podendo auxiliar melhorias no código e do processo (BELLINGARD, 2018).

Figura 2-3. Gerenciamento do código fonte com o Jenkins e SonarQube



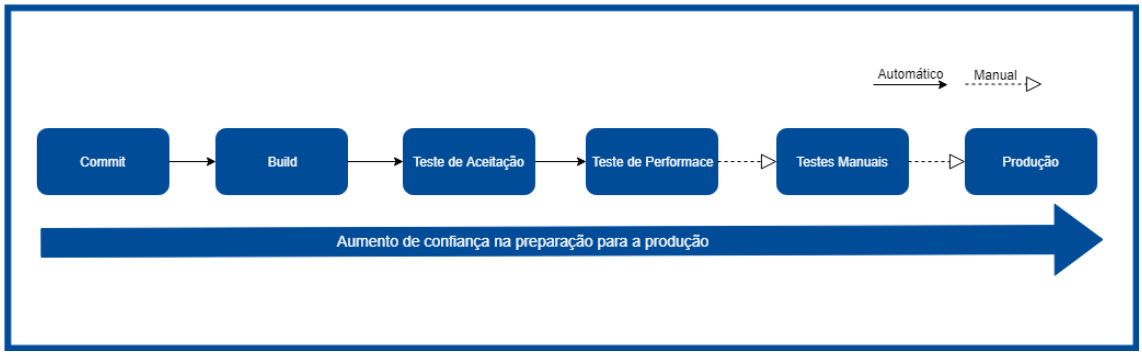
Fonte: https://www.infobip.com/pt/desenvolvedor/melhorando-a-qualidade-do-codigo-com-sonarqube

2.2.2 ENTREGA CONTÍNUA

A Entrega Contínua (EC) é um conjunto de práticas com o objetivo de garantir que um novo código esteja apto para ser disponibilizado em ambiente de produção. De acordo com NEELY e STOLT (2013), EC pode ser descrita como a capacidade de liberar o software sempre que a companhia desejar, podendo ser semanalmente ou diariamente.

A entrega contínua representa um passo adiante da integração contínua, e geralmente é implementada através da utilização de um *pipeline* de entrega, o qual consiste em fases necessárias que uma revisão do software deve passar para que seja finalmente entregue junto ao produto. CHEN e POWER (2010) adotou um pipeline com seis passos, ilustrado na Figura 2-4 .

Figura 2-4. Pipeline de entrega



Fonte: (CHEN; POWER, 2010)

O estágio de *commit* do código fonte consiste na submissão de uma revisão ao repositório de código fonte. Nesse estágio, que faz parte também da integração contínua, são executados os testes de regressão para garantir que o código fonte alterado seja funcional. Caso não sejam encontrados problemas nesse estágio, segue-se para o *build*. O *pipeline* instanciado pelo autor, executa novamente os testes de regressão, com o objetivo de gerar um relatório de cobertura. São executados também testes de integração e análises estáticas do código para garantir que o mesmo atenda aos padrões adotados pela empresa. As próximas etapas consistem em diferentes tipos de testes, até que se chegue à etapa de produção, onde a revisão é adicionada ao produto de software e entregue ao ambiente de produção. Embora no exemplo citado pelo autor o software seja entregue ao ambiente de produção, isso nem sempre é necessário. Para caracterizar o uso da prática da entrega contínua, a empresa deve ser capaz de entregar o software a qualquer momento em algum ambiente, como de testes, ou homologação, ou até mesmo de produção. A habilidade de entregar o software a qualquer momento e automaticamente para o ambiente de produção caracteriza a implantação contínua, que é detalhada a seguir.

2.2.3 IMPLANTAÇÃO CONTÍNUA

Dentre as atividades impactadas pelos testes de regressão, tem-se a implantação contínua. Para CLAPS *et al.*, 2015, o processo de implantação contínua visa implantar o software imediatamente aos clientes assim que o novo código é desenvolvido. Esta prática pode resultar em uma série de benefícios para as organizações, como por

exemplo, novas oportunidades de negócio, reduzir os erros em cada versão e evitar o desenvolvimento de código não utilizado.

O que diferencia a implantação contínua da implantação de software tradicional é a frequência em que ocorrem as implantações. A implantação contínua implanta o software para a produção com mais frequência do que a implantação de software tradicional. Alguns aspectos que diferenciam a implantação tradicional da implantação contínua serão descritos adiante.

Na implantação contínua as entregas são realizadas diariamente, enquanto na tradicional, as entregas são realizadas a cada 1-6 meses. Com relação à possibilidade de ocorrência de erros, a implantação tradicional é superior, uma vez que a implantação é um processo infrequente, e porque o software que será implantado contém muitas mudanças. Na contínua, os riscos são menores, visto que, pequenas mudanças são realizadas, logo criam-se pequenos problemas. Em relação ao ciclo de *feedback* do desenvolvedor e cliente, na implantação contínua os ciclos são muito curtos, logo, os clientes recebem constantemente atualizações. Já na implantação tradicional, os ciclos são longos pois dependem da frequência da implantação. Por fim, a implantação contínua ajuda a diminuir o desenvolvimento de recursos desnecessários devido os recursos serem desenvolvidos constantemente. Isso, por sua vez, evita o desenvolvimento de qualquer software desperdiçado. Já na tradicional, apenas 69% dos recursos necessários para os softwares são desenvolvidos (CLAPS *et al.*, 2015).

Como mostra a Figura 2-5, o *pipeline* de implantação contínua é semelhante à entrega contínua. Geralmente a diferença está no último estágio, onde a entrega para o ambiente de produção é feita de forma automatizada.

Figura 2-5. Diferença entre entrega continua e implantação contínua

Fonte: Adaptado de Sundman (2013)

2.2.4 TESTE CONTÍNUO

O teste contínuo representa um passo seguinte à integração contínua e à entrega contínua. O teste contínuo procura integrar as atividades de teste o mais próximo possível da codificação. É um processo tipicamente caracterizado por envolver alguma automação do processo de teste, ou priorização de casos de teste, para ajudar a reduzir o tempo entre a introdução do erro de areia a sua detecção, com o objetivo de eliminar os danos mais efetivos (FITZGERALD; STOL, 2017).

Segundo FITZGERALD e STOL (2017), semelhante à integração contínua, existem potenciais benefícios para a adoção deste processo. Primeiramente, os erros podem ser corrigidos rapidamente, isto porque o contexto está recente na memória dos desenvolvedores e, desta forma, as causas que levaram a problemas podem ser identificadas e eliminadas. Além disso, geralmente há algum nível de automação do processo de teste. SAFF e ERNST (2003) introduziram o conceito de "teste contínuo" e argumentaram que pode diminuir o tempo de desenvolvimento. O seu experimento mostrou que o teste contínuo pode ajudar a reduzir o tempo de desenvolvimento global em até 15%, sugerindo que o teste contínuo pode ser uma ferramenta eficaz para reduzir um tipo de desperdício, isto é, o tempo de espera.

2.3 PROVENIÊNCIA DE DADOS

BUNEMAN; KHANNA e TAN (2001) definem proveniência de dados como a descrição das origens dos dados e as atividades utilizadas para seu processamento. A proveniência de dados pode ser dividida em prospectiva e retrospectiva (LIM *et al.*, 2010). A proveniência prospectiva se concentra nos modelos para execuções futuras do dado. Já a proveniência retrospectiva tem como objetivo capturar informações de execuções passadas do dado, visando a melhoria em execuções futuras.

Os modelos mais utilizados para modelar a proveniência de dados são OPM (*Open Provenance Model*) e PROV. O modelo OPM tem como objetivo principal a interoperabilidade entre sistemas que o implementam (MOREAU *et al.*, 2008). Já o modelo PROV, que na verdade é uma família de documentos que descrevem padrões para utilização da proveniência de dados (GROTH; MOREAU, 2013). Dentre os documentos, tem-se o PROV-DM (modelo de captura de dados), o PROV-O (ontologia para o mapeamento do modelo de dados) e o PROV-CONSTRAINS (conjunto de restrições aplicáveis ao modelo de dados).

O PROV-DM possui uma separação entre tipos e relações no modelo, no qual seus tipos são:

- Entidade: é um tipo físico, digital, conceitual, ou algo com aspectos fixos. Entidades podem ser reais ou imaginárias;
- Atividade: algo que ocorre durante um período de tempo e atua sobre as entidades. Pode incluir consumo, processamento, transformação, modificação, realocação, uso ou geração de entidades.
- Agente: algo que possui algum tipo de responsabilidade por uma atividade, para a existência de uma entidade, ou para a atividade de outro agente.

Embora nem todas as relações PROV-DM sejam binárias, todas elas envolvem dois elementos principais. As relações primárias são apresentadas na Figura 2-6 e as relações secundárias na Figura 2-7. Com base nestas relações é composto o núcleo do PROV ilustrado pela Figura 2-8.

Figura 2-6. Relações primárias do PROV-DM

		Secondary Object		
		Entity	Activity	Agent
Subject	Entity	—	WasDerivedFrom (activity)	—
	Activity	WasAssociatedWith (plan)	WasStartedBy (starter) WasEndedBy (ender)	—
	Agent	—	ActedOnBehalfOf (activity)	—

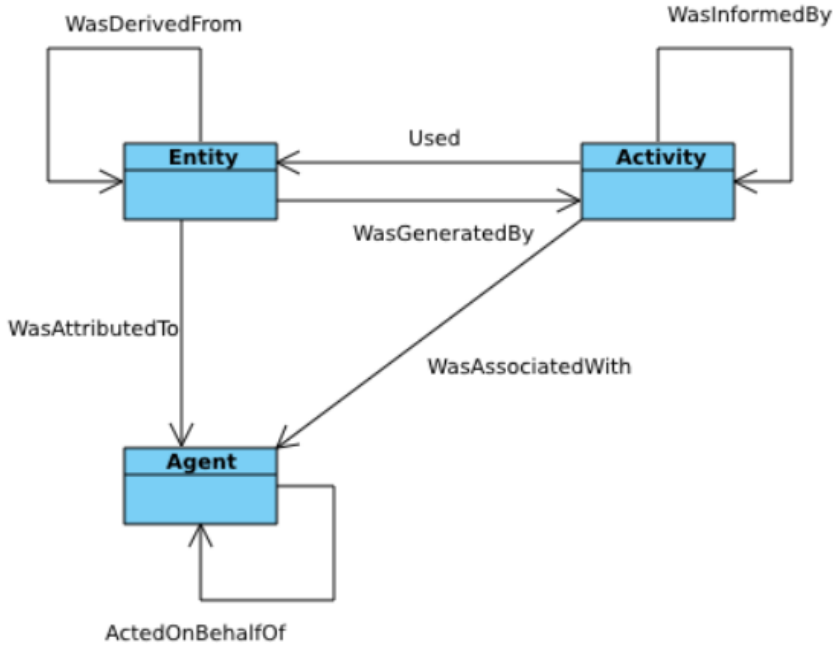
Fonte: PROV-O: The PROV Ontology

Figura 2-7. Relações secundárias do PROV-DM.

		Object		
		Entity	Activity	Agent
Subject	Entity	**WasDerivedFrom** Revision Quotation PrimarySource AlternateOf SpecializationOf HadMember	**WasGeneratedBy** WasInvalidatedBy *R T L*	**WasAttributedTo**
	Activity	**Used** WasStartedBy WasEndedBy *R T L*	**WasInformedBy**	**WasAssociatedWith** *R*
	Agent	—	—	**ActedOnBehalfOf**

Fonte: PROV-O: The PROV Ontology

Figura 2-8. Estrutura do Núcleo do PROV



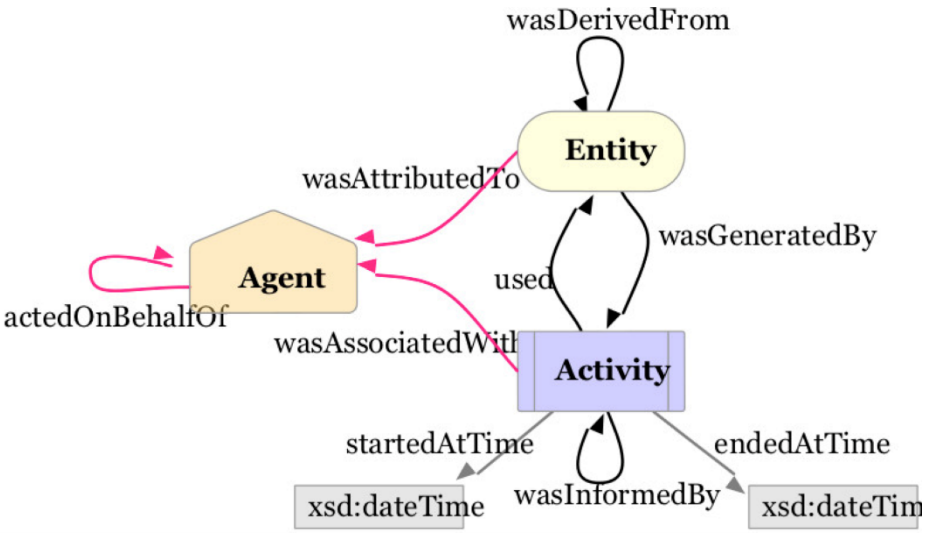
Fonte: PROV-O: The PROV Ontology

Outro documento importante da família PROV é o PROV-O, que trata de uso de ontologias. Segundo EDITION; ANTONIOU e HARMELEN (2008) o termo ontologia originou-se da filosofia e descreve, formalmente, um domínio de discurso. Normalmente, uma ontologia consiste em uma lista finita de termos e suas relações. Contudo, nos últimos anos, ontologia tornou-se um termo muito utilizado para a computação. Para GRUBER (1995) uma ontologia é descrita como uma especificação formal e explícita de uma conceptualização compartilhada. A utilização de ontologias possibilita o compartilhamento de conhecimento sobre os conceitos de um determinado domínio, a reutilização do conhecimento e o processamento de máquina (YU; LIYANG, 2014).

A ontologia PROV-O expressa o modelo de dados PROV usando a linguagem OWL[5]. Como mostra a Figura 2-9, fornece um conjunto de classes, propriedades, e restrições que podem ser utilizadas para representar e trocar informações de procedência gerada em diferentes sistemas e diferentes contextos.

Com base nas classes e relacionamentos existentes no modelo do PROV-O, é possível modelar uma ontologia representando o contexto de onde se quer capturar a proveniência de dados. A proveniência virá da consulta desses relacionamentos apresentados. Como exemplo dessa modelagem, uma ontologia que estende o PROV-O é apresentada no Capítulo 4.

Figura 2-9. Modelo base do PROV-O



Fonte: PROV-O: The PROV Ontology

[5] Web Ontology Language

Considerando a importância da captura da proveniência de dados para a melhoria de futuras execuções, este trabalho propõe a utilização de algoritmos de predição para prever os resultados dos testes de unidade através de um conjunto de treinamento. Para isso, aspectos dos algoritmos de predição são apresentados na próxima seção.

2.4 INTELIGÊNCIA ARTIFICIAL

Por milhares de anos, os seres humanos tentam entender como o cérebro é capaz de perceber, entender, prever e manipular um mundo muito maior e mais complicado do que ele. O campo da inteligência artificial, ou IA, vai ainda mais longe: a inteligência artificial visa entender e construir entidades inteligentes (RUSSELL; NORVIG, 2017).

A IA é um dos mais novos campos da ciência e engenharia. RUSSELL e NORVIG (2017) afirmam que seu desenvolvimento iniciou logo após a Segunda Guerra Mundial e foi nomeada por volta de 1956. Atualmente, tal campo abrange uma grande variedade de subcampos, tais como, provar teoremas matemáticos, diagnosticar doenças, dirigir carros e jogar xadrez. A IA é relevante para qualquer tarefa intelectual, é verdadeiramente um campo universal.

Existem diversas definições para a IA, dentre elas: o estudo dos cálculos que permitem perceber, argumentar e agir (CHARNIAK; MCDERMOTT, 1985) ou o estudo das faculdades mentais através do uso de modelos computacionais (WINSTON, 1992).

Nos últimos anos a IA ganhou cada vez mais popularidade. Com a globalização e a informatização dos processos, as organizações geram um grande número de dados diariamente, também chamado de *Big Data* (SIMON, 2013). Através de campos, como a IA, algoritmos podem ser utilizados para realizar previsões ou sugestões calculadas com base em um conjunto de dados. Alguns dos exemplos mais comuns de aprendizagem de máquinas são: os algoritmos utilizados na plataforma da *Netflix*, que sugerem filmes baseados em escolhas passadas, ou algoritmos da *Amazon* que recomendam livros com base naqueles comprados anteriormente.

Segundo RUSSELL e NORVIG (2017), os algoritmos de aprendizado de máquina podem ser divididos em 3 categorias: aprendizagem supervisionada, aprendizado não supervisionada e aprendizagem por reforço. Na aprendizagem supervisionada, são apresentados exemplos de entrada e saídas desejadas (pares *input-output*). Dessa forma, tais algoritmos aprendem uma função, ou regra geral, que prevê a

saída para novas entradas. Para a categoria aprendizado não supervisionado, nenhum tipo de saída é dado aos algoritmos, deixando-o sozinho encontrar um padrão nas entradas fornecidas. Por fim, na aprendizagem por tem como foco a criação de agentes capazes de tomar decisões acertadas em um ambiente sem que se tenha qualquer conhecimento prévio sobre o tal ambiente.

Neste trabalho foram utilizados os algoritmos de aprendizagem supervisionada, sendo eles: *Logistic Regression, Naive Bayes e C4.5 Algorithm.* Tais algoritmos foram selecionados a partir de uma revisão sistemática de literatura apresentado do Capítulo 3. Sendo estes, os algoritmos mais utilizados para previsões relacionadas ao software. Cada um deles será descrito nas subseções seguintes.

2.4.1 LOGISTIC REGRESSION

Os métodos de regressão tornaram-se um componente essencial para qualquer análise de dados que visa descrever a relação entre uma variável de resposta e uma ou mais variáveis exploratórias. São utilizados para variáveis com resultados discretos, ou seja, que assumem dois ou mais valores possíveis. Ao longo da última década, o modelo de regressão logística tornou-se, em muitos campos, os métodos padrão de análise nesta situação (HOSMER; LEMESHOW, 2000).

Para HOSMER e LEMESHOW (2000) é importante saber que o objetivo de uma análise usando esse método é o mesmo que qualquer técnica de modelo-construção utilizada na estatística: encontrar o modelo mais adequado e mais econômico, porém um modelo biologicamente razoável para descrever e receber um conjunto de variáveis independentes, ou covariáveis.

Existem dois modelos de regressão: regressão logística e a regressão linear. O que distingue um modelo de regressão logística do modelo de regressão linear é que a variável de resultado, na regressão logística, é dicotômica. Uma vez que essa diferença é contabilizada, os métodos empregados em uma análise usando regressão logística seguem os mesmos princípios gerais utilizados na regressão linear (HOSMER; LEMESHOW, 2000).

Para este trabalho, foi utilizado o modelo de regressão logística caracterizado por analisar os dados distribuídos binominalmente através da Equação 2.1.

$$\boldsymbol{f(x) = \frac{L}{1 + e^{-k(x-x_0)}}} \qquad (2.1)$$

Onde:

e = número de Euler;

x_0 = o valor x do ponto médio do sigmoide;

L = valor máximo da curva;

k = inclinação da curva.

2.4.2 NAIVE BAYES

Segundo RUSSELL e NORVIG (2017) o modelo de Rede Bayesiana mais comum usado na aprendizagem de máquinas é o *Naive Bayes*. Esse modelo tem sido amplamente estudado desde os anos de 1950. Pode ser descrito como: a variável C (que deve ser predita) é a raiz e as variáveis X, atributos, são as folhas. O modelo é "*Naive*", pois assume que os atributos X são condicionalmente independentes uns dos outros, dada a classe C. Assumindo o conjunto de atributos $x = (x_{1,} \dots, x_n)$, sendo n o número de atributos, os parâmetros são calculados através da Equação 2.2.

$$\boldsymbol{p(C_k|x_1, \dots, x_n)} \qquad (2.2)$$

Onde k são as possibilidades de resultados possíveis ou variáveis C_k.

Uma vez que o modelo foi treinado, pode ser usado para classificar novos exemplos para os quais a variável C não é observada. Assumindo o conjunto de atributos $x = (x_{1,} \dots, x_n)$, sendo n o número de atributos, a probabilidade de cada classe é dada pela Equação 2.3.

$$\boldsymbol{P(C|x_1, \dots, x_n) = \alpha\, P(C) \prod_i P(x_i|C)} \qquad (2.3)$$

Para RUSSELL E NORVIG (2017) o modelo *Naive Bayes* revela-se surpreendentemente bom em uma ampla gama de aplicações, além disso, possui alta escalabilidade em problemas muito grandes. Por fim, tal modelo de aprendizagem não tem dificuldade com dados ruidosos ou ausentes e podem fornecer previsões probabilísticas quando apropriado.

2.4.3 C4.5 ALGORITHM

As árvores de decisão surgiram do trabalho de Hoveland e Hunt no final dos anos 1950, culminando no livro "*Experiments in Introduction*" que descreve extensas experiências

com várias implementações de sistemas de aprendizagem de conceito (CLS). O algoritmo C4.5 foi criado por J. Ross Quinlan, que desenvolveu a ideia em 1978, por intermédio de um curso de pós-graduação na Universidade de Stanford (QUINLAN, JOHN ROSS, 1993).

De acordo com KOTSIANTIS (2007) o C4.5 é o algoritmo mais conhecido na literatura para construção de árvores de decisão. É uma extensão do algoritmo ID3. LIM; LOH; SHIH, 2000 realizaram um estudo que compara árvores de decisão com outros algoritmos de aprendizagem. Como resultado, o estudo mostra que o algoritmo C4.5 possui uma boa combinação entre taxa de erro e velocidade.

C4.5 constrói árvores de decisão a partir de um conjunto de dados de treinamento, utilizando o conceito de Entropia (QUINLAN, J ROSS, 1993) . Dado um conjunto de dados de treinamento com amostras já classificadas, $S = s_1, \dots, s_i$, onde i é o número de amostras. Cada amostra s_i consiste em um vetor p-dimensional $(x_{1,j}, x_{2,j}, \dots, x_{p,j})$ onde $x_j$representa os valores dos atributos da amostra.

Em cada nó da árvore, o algoritmo escolhe o atributo dos dados que mais particiona o seu conjunto de amostras em subconjuntos, sendo que estes tendem a uma categoria ou a outra. Esse critério de particionamento é a diferença em entropia, calculado pela Equação 2.4.

$$\boldsymbol{Entropia}\ (\boldsymbol{S}) = \sum_{\boldsymbol{i=1}}^{\boldsymbol{c}} -\boldsymbol{p_i log_2\, p_i} \qquad (2.4)$$

Onde c são os valores aceitos pela classe a ser classificada; p_ié a proporção de S pertencendo a classe i.

O atributo com maior entropia é escolhido para tomar a decisão. O algoritmo C4.5 então repete a etapa anterior nas partições menores.

Por fim, uma das características mais úteis das árvores de decisão é sua compreensão. Uma pessoa facilmente entende porque uma árvore de decisão classifica uma instância como pertencente à uma classe. Além disso, as árvores de decisão tendem a ser melhor quando se trata de características categóricas.

2.5 VISUALIZAÇÃO DE SOFTWARE

A visualização é o processo de transformar dados em uma forma visual, permitindo aos usuários a obtenção de informações a respeito de um tema (DIEHL, 2007) . Tal técnica

possui um grande papel na computação, auxiliando na compreensão humana. A visualização é uma importante forma de compreensão e é essencial para apoiar a construção de um modelo a respeito de uma determinada situação ou realidade (SPENCE, 2007).

Atualmente, independente se o conjunto de dados difere, representações visuais podem variar (BALDONADO; WOODRUFF; KUCHINSKY, 2000), como exemplo, uma visualização pode mostrar um gráfico de barras, enquanto outra mostra um gráfico de pizza. O ambiente interativo baseado em múltiplas visões (AIMV) (CARNEIRO; CONCEIÇÃO; DAVID, 2012) são ambientes que fornecem diversas visualizações para um mesmo conjunto de dados, ajudando a minimizar o surgimento de interpretações equivocadas.

2.6 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Este capítulo apresentou os pressupostos teóricos deste trabalho. Foram abordados os principais aspectos relativos à verificação e validação de software, testes de regressão, engenharia de software contínua, bem como suas subatividades: integração contínua, implantação contínua, entrega contínua e teste contínuo, proveniência de dados, assim como o modelo PROV, inteligência artificial e seus algoritmos *Logistic regression, Naive Bayes, C4.5 Algorithm,* por fim, visualização de software.

No capítulo seguinte são apresentados os trabalhos relacionados através de uma revisão sistemática de literatura.

3 TRABALHOS RELACIONADOS

Neste capítulo é apresentado uma revisão sistemática relacionada ao tema, o que permitiu identificar a relevância da pesquisa e os principais trabalhos relacionados.

3.1 REVISÃO SISTEMÁTICA DE LITERATURA

KITCHENHAM e CHARTERS (2007) definem revisão sistemática como um tipo de estudo secundário que são projetados para fornecer uma ampla visão de uma área de pesquisa. Segue um processo de pesquisa metodologicamente definido, para identificar, analisar e interpretar as evidências disponíveis relacionadas à uma questão de pesquisa definida de uma maneira não tendenciosa e repetível. É um meio de identificar, avaliar e interpretar pesquisas existentes disponíveis e relevantes para uma questão de pesquisa, tópico ou fenômeno de interesse, possibilitando novas pesquisas.

Dessa forma, conduziu-se uma revisão cujo objetivo foi construído através do método *Goal/Question/Model* (GQM), conforme (WOHLIN *et al.*, 2012), é*: **Analisar** algoritmos e modelos, **com o propósito de** caracterizá-los **em respeito a** predição de falha dos testes ou predição de falha do software **do ponto de vista dos** analistas de testes, **no contexto de** engenharia de software contínua.*

Partindo-se desse objetivo, foi elaborada a questão de pesquisa primária a seguir: **Quais tipos de algoritmos e/ou modelos são utilizados para a predição de falhas dos testes de software e/ou predição de falha do software com o propósito de melhoria do processo de teste?**

A partir da questão de pesquisa primária, citada anteriormente, derivou-se três questões secundárias de pesquisa, sendo que essas visam caracterizar a área. São elas:

Questão 1 – Quais tipos de informações são utilizados para a predição de falhas dos testes de software e/ou falha do software?

Questão 2 – Esses algoritmos e/ou modelos estão amparados por alguma ferramenta, mecanismo, infraestrutura, *framework*, ambiente, plataforma ou arquitetura?

Questão 3 – Qual o volume do conjunto de dados utilizados para a avaliação?

Questão 4 – De onde as informações utilizadas para realizar as previsões foram retiradas?

Questão 5 – Quais os percentuais de acerto dos algoritmos ou modelos propostos?

3.2 ESTUDOS PRELIMINARES

Durante a etapa do estudo preliminar, fontes de pesquisa foram selecionadas. Foram escolhidas somente bases digitais cujo conteúdo encontra-se disponível através do acesso disponibilizado pela UFJF (Universidade Federal de Juiz de Fora). Deste modo, foram selecionadas quatro bases. Sendo que estas, atendem ao critério de acesso pela UFJF e possuem grande utilização em pesquisas na área de Engenharia de Software. São elas: (i) *ACM Digital Library, (ii) IEEE Digital Library, (iii) Science Direct e (iv) Scopus*.

Definiu-se que os idiomas das pesquisas seriam o Inglês e o Português. O inglês por ser considerado como linguagem padrão internacional e o Português pelo fato de ser a língua oficial do país da pesquisa.

As buscas foram realizadas através de uma "*String*" definida por meio do método PICOC. PICOC é uma abreviação para "*Population, Intervention, Comparison, Outcome e Context*", e é um método usado para descrever uma pergunta pesquisável (DA SILVA *et al.*, 2010). A *String* de busca utilizada foi composta pelas palavras-chave apresentadas na Tabela 3.1 e executada sobre as bases mencionadas anteriormente, apenas adaptando-a às particularidades de cada uma. Ao final, os artigos retornados foram catalogados utilizando o Parsifal[6], uma ferramenta *online* para auxiliar pesquisadores a realizar revisões sistemáticas de literatura dentro do contexto da Engenharia de Software.

Tabela 3.1. PICOC

PICOC	**PALAVRAS-CHAVE**
P (POPULATION)	*Predict, Prediction*
I (INTERVENTION)	*Software test*
C (COMPARISON)	Não se aplica
O (OUTCOME)	*Model, Algorithm*
C (CONTEXT)	*Software engineering*

Fonte: Elaborado pelo próprio autor

[6] https://parsif.al/

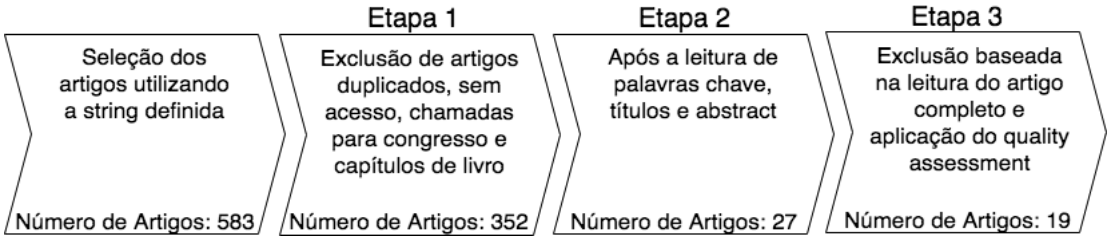
3.2.1 CRITÉRIOS DE REFINAMENTO DOS ESTUDOS

Após a catalogação, as apurações foram realizadas em três etapas, como mostra a Figura 3-1. A primeira delas contou com a exclusão dos artigos duplicados, sem acesso pela UFJF e chamadas para congresso, capítulos de livros e livros. Durante a segunda etapa, foram selecionados aqueles artigos que possuíam palavras-chave, título e *abstract* dentro do escopo estabelecido. Na terceira etapa, almejou-se ponderá-los em notas, de acordo com os critérios:

- Os autores descrevem explicitamente o algoritmo utilizado.
- Os autores descrevem explicitamente o conjunto de dados utilizados.
- Os autores descrevem explicitamente como a ferramenta, mecanismo, infraestrutura, *framework*, ambiente, plataforma ou arquitetura foi avaliada.
    - Os autores disponibilizam acesso à ferramenta, mecanismo, infraestrutura, *framework*, ambiente, plataforma ou arquitetura que dão suporte aos modelos ou algoritmos de predição.

As notas variavam entre 0 e 1, com intervalos de 0,5. Sendo 1 destinado ao artigo que está totalmente de acordo com a afirmativa, 0,5 ao parcialmente de acordo e 0, caso contrário, contabilizando um valor máximo igual a 4. Assim, classificou-se como úteis à pesquisa apenas aqueles trabalhos que obtiveram nota igual ou superior a 2.

Figura 3-1. Seleção de Artigos



Fonte: Elaborado pelo próprio autor

A Tabela 3.1 apresenta o PICOC com as respectivas palavras-chave utilizadas para a construção da *String* de busca. Todas estas foram definidas em inglês, pelo fato dos artigos possuírem título ou *abstract* em inglês.

Montou-se a *String* de busca através das palavras-chave, sendo estas intercaladas e agrupadas de acordo com o método PICOC.

"((predict) OR (prediction)) AND (("software test")) AND ((model) OR (algorithm)) AND (("software engineering"))"

Para a validação da *String* foram selecionados três artigos de controle, apresentados na Tabela 3.2, sendo que estes deveriam ser retornados nas buscas durante a execução.

O primeiro artigo, "*Improving fault prediction using Bayesian networks for the development of embedded software applications"* (ELENA, 2006), foi selecionado por apresentar uma técnica utilizada pela Motorola para desenvolver um modelo de Rede Bayesiana utilizado para a predição para falhas do software. Tal modelo, utiliza informações sobre seus produtos, uso e processo para realizar as previsões. O segundo artigo, *"Developing a Bayesian Network Model Based on a State and Transition Model for Software Defect Detection"* (JONGSAWAT; NIPAT; PREMCHAISWADI, 2012) foi selecionado por apresentar um modelo para diagnosticar causas-efeitos da detecção de defeitos de software no processo de teste. Tal modelo utiliza Redes Bayesianas para identificar módulos defeituosos, visando a eficácia dos testes e a melhoria da qualidade do sistema. Por fim, o terceiro artigo*, "Effective software fault localization using predicted execution results"* (GAO *et al.*, 2017), foi selecionado por apresentar um *framework* que objetiva reduzir o esforço na verificação dos resultados utilizando uma estratégia baseada em *Hamming Distance and K-means Clustering* para prever resultados das execuções dos testes.

Tabela 3.2. Artigos de Controle

ARTIGO	**ANO**	**LOCAL DE PUBLICAÇÃO**
Improving fault prediction using Bayesian networks for the development of embedded software applications	2006	Software Testing Verification and Reliability
Developing a Bayesian network model based on a state and transition model for software defect detection	2012	ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing

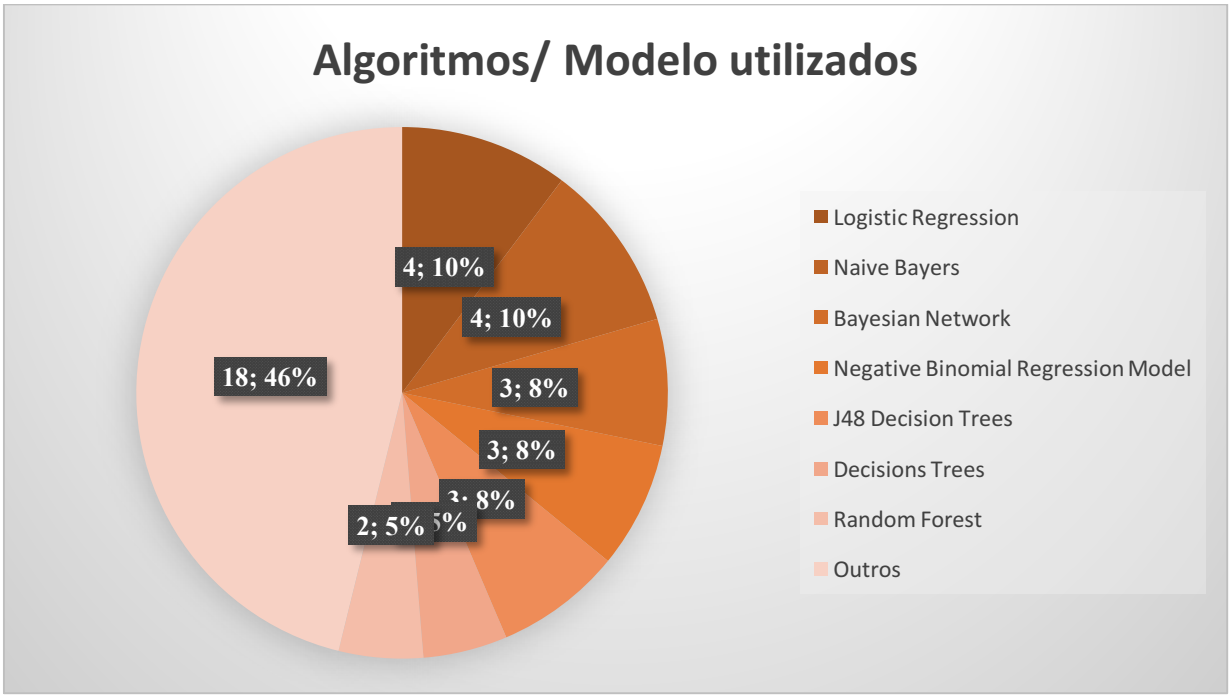
		(SNPD)
Effective software fault localization using predicted execution results	2017	Software Quality Journal

Fonte: Elaborado pelo próprio autor

3.2.2 RESULTADOS

Após o processo de seleção dos artigos, algumas observações foram realizadas visando responder às questões de pesquisa indicadas na Seção 3.1. A questão principal tem como objetivo apontar os tipos de algoritmos ou modelos utilizados para a predição das falhas dos testes de software e do software. A Figura 3-2 apresenta o gráfico com a distribuição dos algoritmos/modelos mais utilizados para a predição de falhas. Em primeiro lugar, empatados com 4 artigos, os algoritmos mais utilizados são *Logistic Regression e Naive Bayes*. Em segundo lugar, empatados com 3 artigos, os algoritmos são: J48 *Decision Tree, Random Forest e Bayesian Network*. Em terceiro lugar, empatados com 2 artigos, *Negative Binomial Regression Model e Decisions Trees*. Por fim, com 18 artigos, 18 algoritmos/modelos diferentes entre si e diferentes dos citados anteriormente.

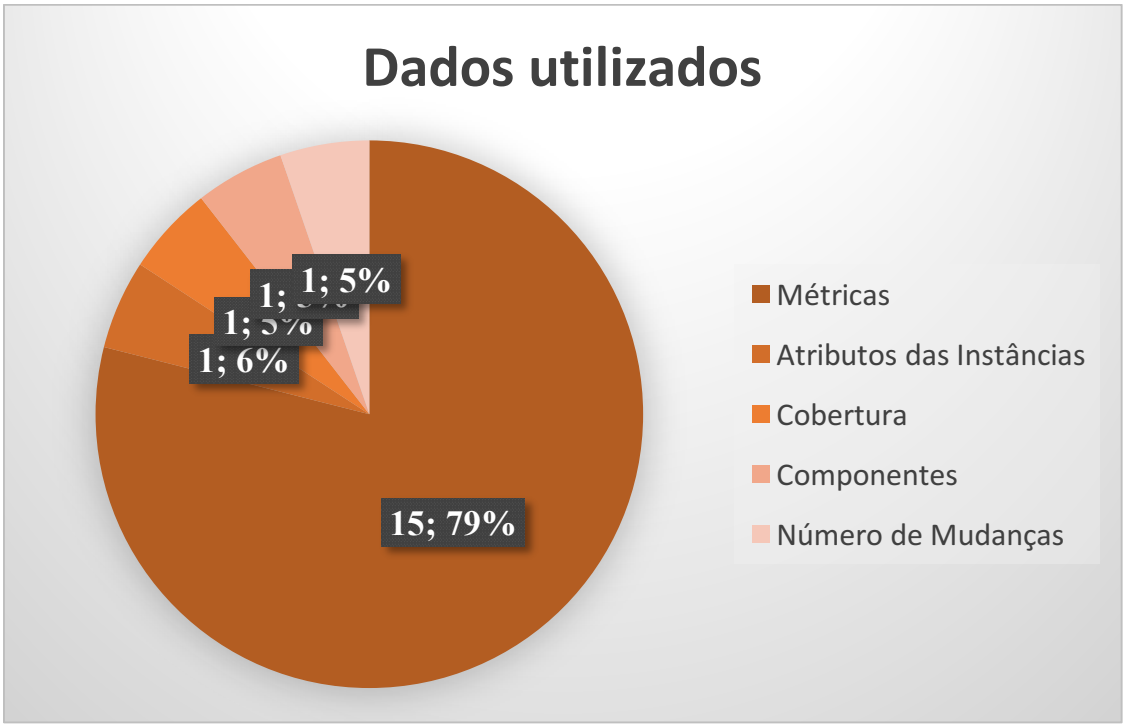
Figura 3-2. Algoritmos/Modelos utilizados



Fonte: Elaborado pelo próprio autor

A segunda questão de pesquisa tem como objetivo pontuar quais foram as informações mais utilizadas para a predição de falhas dos testes de software ou falha do software. Como mostra a Figura 3-3, dentre os 19 artigos aceitos no processo, 15 deles ou 78,9%, utilizaram métricas do(s) software(s) para realizar as previsões. Os artigos restantes, utilizaram componentes (5,6%), atributos das instâncias (5,6%), cobertura (5,6%) e número de mudanças (5,6%).

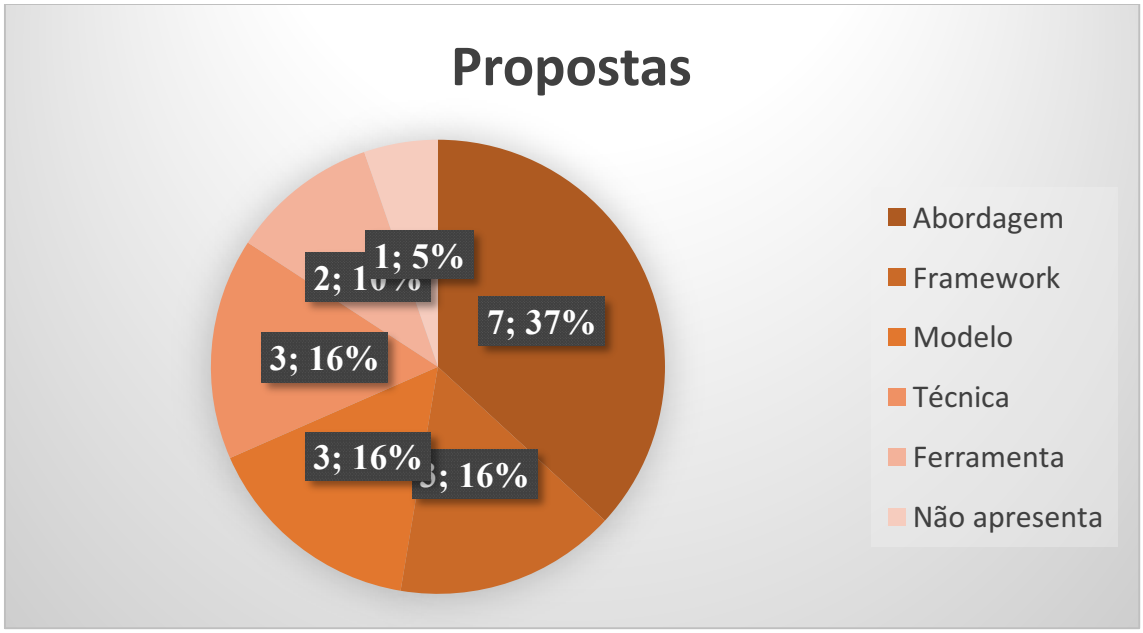
Figura 3-3. Informações utilizadas



Fonte: Elaborado pelo próprio autor

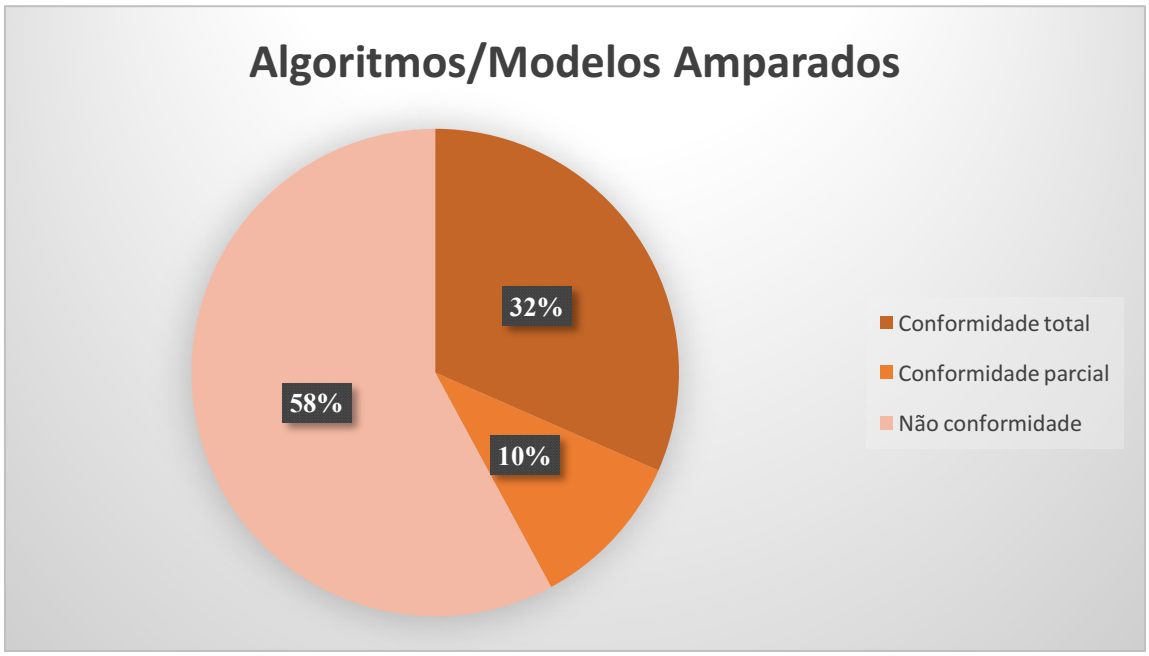
A terceira questão de pesquisa é a seguinte: Estes algoritmos ou modelos estão amparados por alguma ferramenta, mecanismo, infraestrutura, *framework*, ambiente, plataforma ou arquitetura? Tais elementos estão disponíveis para acesso? Como mostram a Figura 3-4 e Figura 3-5, 7 artigos apresentaram abordagens, 3 artigos apresentaram *frameworks*, 3 apresentaram modelos, 3 apresentaram técnicas, 2 apresentaram ferramentas e 1 realiza uma comparação entre diferentes técnicas. Desses artigos, apenas 32% (6 artigos) explicaram explicitamente a ferramenta que amparam os modelos ou algoritmos utilizados. Entretanto, apenas um, (SOHN, 2017), disponibilizou a técnica para utilização.

Figura 3-4. Arquiteturas propostas



Fonte: Elaborado pelo próprio autor

Figura 3-5. Algoritmos/Modelos amparados



Fonte: Elaborado pelo próprio autor

3.3 TRABALHOS RELACIONADOS

Nesta seção serão apresentados os resultados da revisão sistemática. Foram selecionados 19 artigos, que se enquadraram nos critérios anteriormente estabelecidos. Esses artigos serão, resumidamente, explicados a seguir.

JONGSAWAT e PREMCHAISWADI (2012) apresentam um modelo que visa diagnosticar possíveis causas e efeitos na detecção de defeitos do software no processo de teste. Os autores utilizam Redes Bayesianas para identificar módulos defeituosos, visando a eficácia dos testes e a melhoria da qualidade do sistema. Além disso, o modelo disponibiliza assistência aos desenvolvedores ao determinar níveis de prioridades no desenvolvimento do software.

SINGH; KAUR e MALHOTRA (2010) objetivam encontrar relações entre métricas orientadas a objetos e vários níveis de falha, visando priorizar aquelas que são mais graves. Para tal, são utilizados diferentes Modelos de Predição, Regressão e *Machine Learning.*

GHHS *et al.* (2017) propõem uma análise de acurácia de previsão dos resultados entre os mais populares e mais utilizados algoritmos de aprendizado de máquina, tais como: *Artificial Neural Networks (ANN), Particle Swarm Optimization (PSO), Decisions Trees (DT), Naive Bayes's (NB) e Linear Classifier (LC)*. Esses algoritmos foram analisados utilizando a ferramenta KEEL e validados a técnica de *K-Fold Cross Validation*. Como resultado, o modelo *Linear Classifier* se sobressaiu aos demais.

GAO *et al.* (2017) apresentam um *framework* composto pelos algoritmos *Hamming Distance and K-means Clustering* que visam localizar e prever se os resultados dos testes a partir de um vetor de cobertura. Seu principal objetivo é reduzir o esforço de verificação.

A ferramenta *GUI-Based* desenvolvida por OSTRAND, THOMAS *et al.* (2010) prevê a probabilidade de falhas para arquivos individuais de versões sucessivas em sistemas caracterizados pela extensão, duração e quantidade de desenvolvedores. A previsão se dá a partir de dois estágios: extração das informações históricas e atuais do sistema, e aplicação do modelo Negativo Binomial de Regressão. Como resultado, a ferramenta entrega uma lista ordenada pela propensão de falhas dos arquivos.

O *ToolKit* chamado *RaPiD* proposto por GUI; SI e YANG (2013) combina Teste de Hipóteses e verificação do Modelo Probabilístico, de modo a fornecer

"confiança" de qualidade do software e quantificar os erros. A ideia é aplicar testes de hipóteses para componentes determinísticos do sistema e usar técnicas de verificação de modelos probabilísticos para levantar os resultados através dos componentes não determinísticos.

Em (NAM; PAN; KIM, 2013) é proposta a abordagem TCA+, estendida de uma abordagem já conhecida, denominada TCA. Tal abordagem visa a transferência de conhecimento para a predição de testes em novos sistemas, ou seja, oferece um conjunto de regras para selecionar uma opção normalizada de dados para o treinamento da rede, para obter melhor desempenho de previsão.

ZHANG; WANG e JING (2016) propõem uma abordagem que utiliza *Graph Based Semi-Supervised Learning (GSSL)* e métricas dos softwares, McCabe, Halstead, linhas de código (LOC) e outras, para prever falhas nos softwares.

CANFORA *et al.* (2015) propõem uma abordagem, *Coined as Multiobjective Defect Predictor (MODEP)*, baseada em múltiplos objetivos. Tal abordagem permite ao usuário selecionar "preditores" que conseguem uma harmonização entre o número de classes que possivelmente falharão e linhas de código que devem ser analisadas ou testadas. MODEP é, ainda, composta por técnicas de *Machine Learning*, dentre estas: *Logic Regression e Decisions Trees*, que auxiliam na multiobjetividade de técnicas em *machine learning*.

WEYUKER *et al.* (2008) dispõem uma comparação entre modelos de previsão de falhas. Para tanto, foram utilizados: Negative *Binomial Regression Model e Recursive Partitioning Model,* aplicados através das mesmas variáveis "preditoras" e em sistemas de larga escala.

Em (OSTRAND; WEYUKER; BELL, 2007) um modelo de predição de testes é apresentado. Esse modelo produz previsões mais precisas e automatizadas do que o modelo anteriormente proposto pelos autores.

HERBOLD (2013) propõe uma estratégia baseada em distância para a seleção dos dados utilizados no treinamento dos algoritmos de predição. Essa estratégia, utiliza as características dos dados disponíveis. Como resultado, os pesquisadores observaram que a estratégia melhora a taxa de sucesso das previsões.

BOETTICHER (2005) propõe uma técnica de predição de falhas no software utilizando atributos não relacionados às classes. Foram conduzidos 20 estudos experimentais em 5 conjunto de dados da NASA, utilizando dois algoritmos de aprendizado diferentes: J48 e *Naive Bayes*. Cada conjunto de dados foi dividido em 3

grupos: um grupo de treinamento, um grupo de teste com vizinhos “agradáveis” e um grupo com vizinhos “desagradáveis". Nessa abordagem, "vizinhos agradáveis", consiste em instâncias de teste mais próximas das instâncias de treinamento da classe. Já os "vizinhos desagradáveis" são os mais próximos de instâncias de treinamento da classe oposta. Os estudos experimentais "agradáveis" obtiveram uma precisão de 94% e as experiências "desagradáveis" atingiram 20% de precisão. Com base nesses resultados, é proposta uma nova técnica de amostragem vizinha mais próxima.

Em (BISHNU, 2011) uma abordagem de Clusterização *K-Medoids* é aplicada à predição de falhas no software. Essa abordagem, primeiramente, aplica *K-Tree* para encontrar agentes sobre os dados que serão utilizados para a predição. Posteriormente, são identificados clusters nesses dados. O algoritmo K-PAM *(K-Medoids*) então é aplicado para a predição de falhas. A abordagem obteve um número menor de cálculos das distâncias, porém obteve resultados semelhantes às demais abordagens.

CARVALHO; POZO; VERGILIO (2010) propõem uma abordagem para predição de falhas. Tal abordagem é baseada no algoritmo *Multiobject Particle Swarm Optimization (MOPSO)*, o qual, através de conceitos de Domínio de Pareto, cria um modelo composto por regras com específicas propriedades. Estas regras podem ser utilizadas por modelos não ordenados de classificação, assim são mais intuitivas e compreensíveis.

ELENA (2006) descreve uma técnica utilizada pela Motorola para desenvolver uma modelo de Rede Bayesiana que prevê falhas em seus produtos. São utilizadas suas informações de uso dos produtos e processo dos mesmos, mais precisamente, métricas coletadas durante a fase de desenvolvimento para realizar as previsões.

Fluccs (Fault Localization Using Code and Change Metrics) é apresentado por (SOHN, 2017) e visa a localização de falha. Essa técnica aprende a classificar os elementos do programa com base em: *Spectrum Based Fault Localization (SBFL)* e métricas do software. Para a aplicação, foram utilizados os algoritmos: *Genetic Programming (GP)* e *Linear Rank Support Vector Machines (SVMs).*

BOWES *et al.*, 2016 apresentam uma previsão de falha de mutação baseada em contexto, que utiliza métricas e quatro algoritmos de previsão diferentes, são eles: *Naive Bayes, Logistic Regression, Random Forest, J48.*

Os resultados anteriormente apresentados foram sumarizados e são indicados na Tabela 3.3.

Tabela 3.3. Artigos aceitos

Título	Algoritmo	Informações	Ferramenta	Dados
(JONGSAWAT; PREMCHAISWADI, 2012)	*Bayesian Network*	Métricas do software	*Framework*	-
(SINGH; KAUR; MALHOTRA, 2010)	*Logistic regression*	Métricas do software	Modelo	NASA
(SINGH; CHUG, 2017)	*Artificial neural networks Artificial neural networks Articial Neural Networks, P Article Swarm Optimization, Decisions Trees, Naive Bayes, Linear Classifier*	Instâncias	-	NASA
(GAO *et al.*, 2017)	*Hamming Distance, K-Means Clustering*	Informações de Cobertura	*Framework*	Siemens suite, the Unix suite, gzip, grep, make, sed and Ant
(OSTRAND; PARK, 2010)	*Negative Binomial Regression Model*	Métricas do software	Ferramenta	6 Sistemas não especificados
(GUI; SI; YANG, 2013)	*Probabilistic Model Checking, Hypothesis Testing*	Componentes do Software	Ferramenta	CCS system
(NAM; PAN; KIM, 2013)	*TCA+*	Métricas do software	Abordagem	ReLink e AEEEM
(ZHANG; JING; WANG, 2017)	*Graph Based Semi-Supervised Learning*	Métricas do software	Abordagem	NASA
(CANFORA *et al.*, 2015)	*Logistic Regression Decision Trees*	Métricas do software	Abordagem	PROMISE Repository
(JING *et al.*, 2014)	*Cost-Sensitive Discriminative Dictionary Learning*	Métricas do software	Técnica	NASA

(WEYUKER; OSTRAND; BELL, 2008)	*Negative Binomial Regression, Recursive Partitioning Model*	Métricas do software	Modelo	Inventory, Provisioning, Voice Response
(OSTRAND; WEYUKER; BELL, 2007)	*Negative Binomial Regression*	Número de Mudanças	Abordagem	35 *releases* de um projeto
(HERBOLD, 2013)	*Logistic Regression, Naive Bayes, Bayesian Networks, SVM with RBF kernel, J48 Decision Trees, Random Forest, Multilayer Perceptron*	Métricas do software	Abordagem	PROMISE Repository
(BOETTICHER, 2005)	*J48, Naive Bayes*	Métricas do software	Técnica	NASA
(BISHNU; BHATTACHERJEE, 2011)	*K-PAM (Partitioning Around Medoids)*	Métricas do software	Abordagem	AR3, AR4, AR5 Datasets
(CARVALHO; POZO; VERGILIO, 2010)	MOPSO algorithm, named MOPSO-N.	Métricas do software	Abordagem	NASA
(PÉREZ-MIÑANA; GRAS, 2006)	Bayesian network	Métricas do software	Modelo	Three Projects
(SOHN, 2017)	Genetic Programming, Linear Rank Support Vector Machines	Métricas do software	Técnica	Defects4J Repository
(BOWES *et al.*, 2016)	Naive Bayes, Logistic Regression, Random Forest, J48	Métricas do software	*Framework*	Apache, Eclipse, TelCom

Fonte: Elaborado pelo próprio autor

Mediante a execução da revisão sistemática, apresentada nesta seção, conclui-se que os trabalhos apresentados se relacionam com a proposta desta dissertação no

sentido de proporem, de uma forma geral, abordagens que, através de informações históricas, realizam previsões de falhas dos testes ou *builds* do software. Entretanto, a proposta deste trabalho, detalhada na Seção 4, se difere destas abordagens nos seguintes aspectos: oferece uma infraestrutura que realiza a captura dos dados através da proveniência e os armazena, une os algoritmos de predição em um só local, oferece visualizações, permite que aplicações externas consigam utilizar os dados armazenados e, por fim, a solução tem como contexto a engenharia de software contínua. Desta forma, identifica-se a viabilidade de desenvolvimento da proposta deste trabalho, haja vista que não foram encontrados na literatura, trabalhos que aliem tecnologias distintas, como ontologias e algoritmos de previsão para obtenção de conhecimento sobre o processo de teste de software.

3.4 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Este capítulo apresentou uma revisão sistemática da literatura, na qual foi possível identificar evidências sobre a previsão de falhas, através de trabalhos publicados nesta área. Por meio das questões de pesquisa, foi possível a identificação de publicações que abordam a previsão dos resultados dos testes, seus modelos/algoritmos utilizados e suas ferramentas, e *frameworks* que auxiliam no processo de previsão. Bem como suas tecnologias utilizadas, formas de avaliação e resultados. Com isso, observa-se a viabilidade da proposta, que engloba o uso de captura de informações, ontologias, algoritmos de previsão e visualização de dados.

4 ISRET: UMA INFRAESTRUTURA PARA A PREVISÃO DOS RESULTADOS DE TESTES DE UNIDADE

Este capítulo tem como objetivo apresentar a solução para o problema apresentado. Relaciona-se ao desenvolvimento de uma infraestrutura que viabilize a captura, o armazenamento e a utilização das informações e métricas, relativas às execuções de um software no contexto de engenharia de software contínua. Além disso, esta infraestrutura realiza predições sobre os testes de software, provê elementos de visualização e a possibilidade de aplicações externas utilizarem suas informações.

4.1 INTRODUÇÃO

A ESC é caracterizada pelo uso do *feedback* de execuções para obter uma melhoria contínua e a realização das atividades contínua. Dentre essas atividades, tem-se a IC. Durante os processos, essas atividades geram uma série de dados que poderiam ser armazenados e utilizados como *feedback* dos processos.

Diante desse cenário, o problema tratado neste trabalho refere-se à elaboração de uma infraestrutura pela qual seja possível apoiar a captura dos dados gerados na execução de testes de regressão para serem utilizados como *feedback* para melhoria contínua. A seguir são apresentados, de forma geral, os requisitos não funcionais e funcionais.

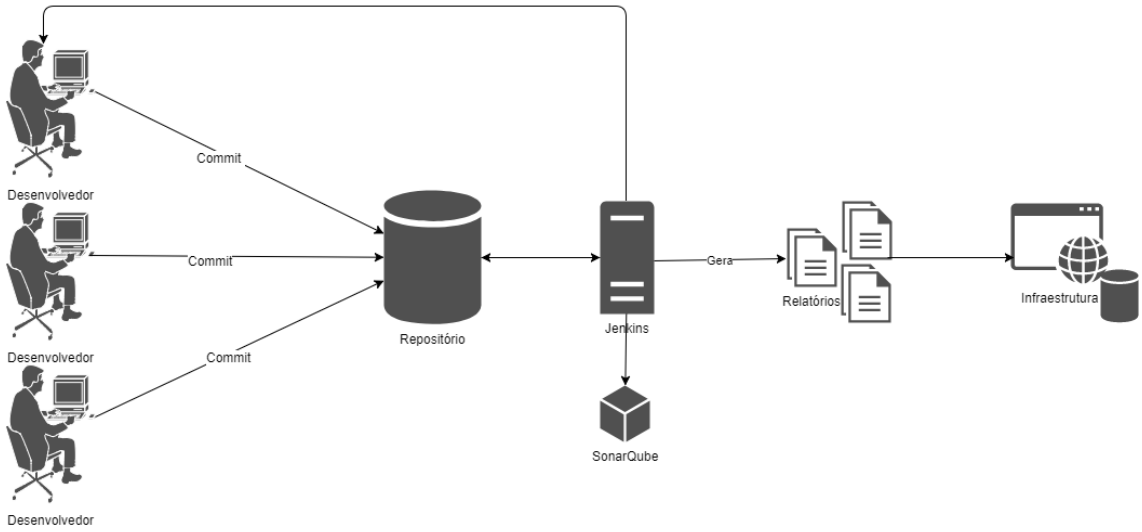
Dentre os requisitos não funcionais tem-se: usabilidade, permite que a infraestrutura funcione de forma simples aos usuários, por exemplo, sem a necessidade de compreender os algoritmos de previsão ou ontologias; portabilidade, permite que a infraestrutura construída funcione em diversos sistemas operacionais (Windows, Linux e MacOS); extensibilidade e independência das camadas construídas, o que permite suportar outras aplicações e adicionar novos módulos de maneira fácil.

Dos requisitos funcionais, pode-se destacar que a infraestrutura deve prover: coleta de dados gerados em arquivos XML através das ferramentas Jenkins e SonarQube após a execução dos *builds*; inferência sobre os dados coletados para descoberta de nova informação, como por exemplo a cobertura de um caso de teste; múltiplos algoritmos para a previsão dos testes de unidade; múltiplas visualizações para

auxiliar na compreensão do código; disponibilização dos dados coletados e inferidos para consulta, visando o apoio à melhoria dos processos.

A Figura 4-1 ilustra o processo de integração contínua adaptado pela inserção da infraestrutura proposta. A ISRET se encaixa na última etapa do processo de IC. O processo se inicia quando um desenvolvedor realiza um *commit* no repositório. Para este trabalho, foi adotado como servidor padrão de IC o Jenkins. Em seguida, o servidor detecta as alterações, captura essas mudanças e se prepara para o novo *build*. Por fim, o Jenkins gera relatórios e notifica o desenvolvedor sobre o resultado do *build*. Durante o *build*, o servidor executa a SonarQube, com o objetivo de realizar análise estática do código. Posteriormente à geração dos relatórios, a infraestrutura proposta começa a atuar. É responsável por ler, capturar e armazenar as informações contidas nesses relatórios, objetivando realizar as previsões dos resultados dos testes de unidade.

Figura 4-1. Fluxograma do processo

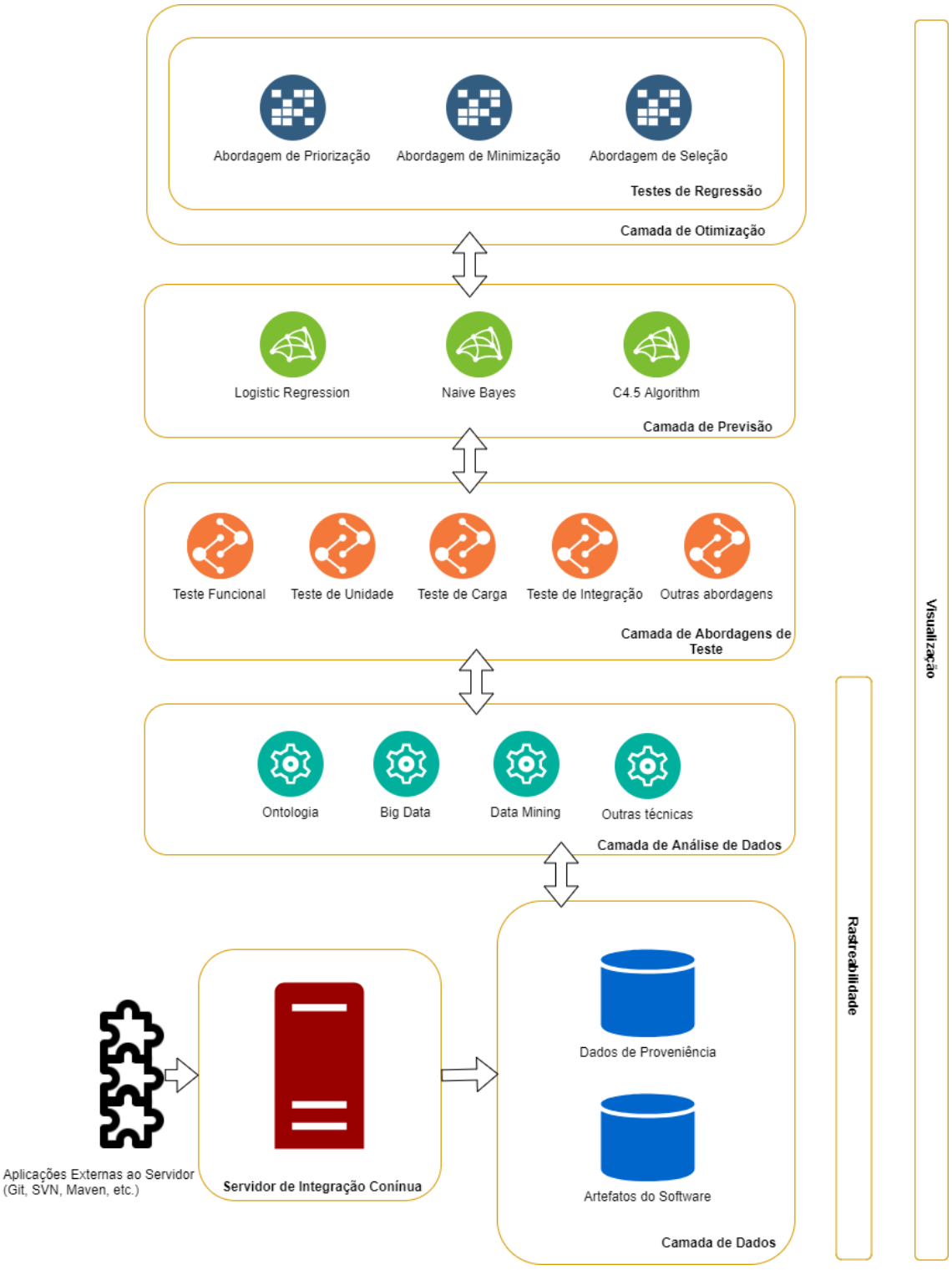


Fonte: Elaborado pelo próprio autor

A arquitetura da infraestrutura proposta é apresentada pela Figura 4-2. A infraestrutura foi fragmentada em 5 camadas: camada de dados, armazena os dados; camada de análise dos dados, uma vez que é necessário armazenar os dados; camada de abordagens de teste, uma vez que é necessário retirar informações dos dados; camada de previsão, uma vez que é necessário realizar o *feedback* através das previsões; e camada de otimização, uma vez que é necessário realizar o *feedback* através dos dados históricos; além de 2 módulos: rastreabilidade, realiza a rastreabilidade; e visualização,

auxilia na compreensão dos dados. Nas subseções seguintes, as camadas e os módulos são explicados de forma detalhada.

Figura 4-2. Arquitetura Infraestrutura



Fonte: Elaborado pelo próprio autor

4.1.1 CAMADA DE DADOS

A camada de dados tem como objetivo a captura e o armazenamento dos dados gerados pelo Jenkins e pelo SonarQube. Essa camada é composta por dois bancos de dados, Artefatos do Software e Dados de Proveniência. O banco Artefatos do Software é responsável por armazenar os artefatos do software, tais como: relatórios, o código fonte do projeto, e qualquer documento relacionado a ele. O banco Dados de Proveniência, tem como objetivo o armazenamento dos dados capturados através da leitura de arquivos em formato XML e inferências realizadas pela ontologia, que serão detalhadas na Subseção 4.1.2.1. A Figura 3-3 e a Tabela 4.1 são uma exemplificação desses arquivos e das informações retiradas dos mesmos.

Para realização do armazenamento, o banco Dados de Proveniência é um modelo relacional que utiliza como base o PROV-DM, modelo de dados do PROV, (BUNEMAN; KHANNA; TAN, 2001). O PROV-DM é um modelo conceitual de dados que constitui a base W3C de especificações de proveniência. O modelo possui *design* modular e está estruturado de acordo com seis componentes. São eles: entidades, atividades e o momento em que são criados, utilizados ou terminam; derivações entre entidades; agentes responsáveis por entidades que foram geradas e atividades que ocorreram; *bundles*, um mecanismo de suporte à proveniência de proveniência; propriedades para ligar entidades que se referem à mesma coisa; e, coleções que formam uma estrutura lógica para os seus membros (MOREAU; MISSIER, 2013).

O diagrama de tabelas relacionais, apresentado na Figura 4-4, apresenta os componentes mencionados anteriormente, vindos do PROV-DM, em branco. Os componentes em azul foram aqueles adicionados para atender as necessidades específicas do contexto da proposta. Com base no modelo detalhado, os dados de proveniência são armazenados nas tabelas propostas e podem, então, ser utilizados pela camada de análise, descrita a seguir.

Figura 4-3. Relatório JUnit Jenkins

```
<suites>
  <suite>
    <file>C:\Program Files (x86)\Jenkins\workspace\GameOfLife\gameoflife-core\target\surefire-reports\TEST-behavior.CountingThings.xml</file>
    <name>behavior.CountingThings</name>
    <duration>0.726</duration>
    <time>0.726</time>
    <enclosingBlocks/>
    <enclosingBlockNames/>
    <cases>
      <case>
        <duration>0.079</duration>
        <className>behavior.CountingThings</className>
        <testName>Adding two integers</testName>
        <skipped>false</skipped>
        <failedSince>0</failedSince>
      </case>
    </cases>
  </suite>
  <suite>
```

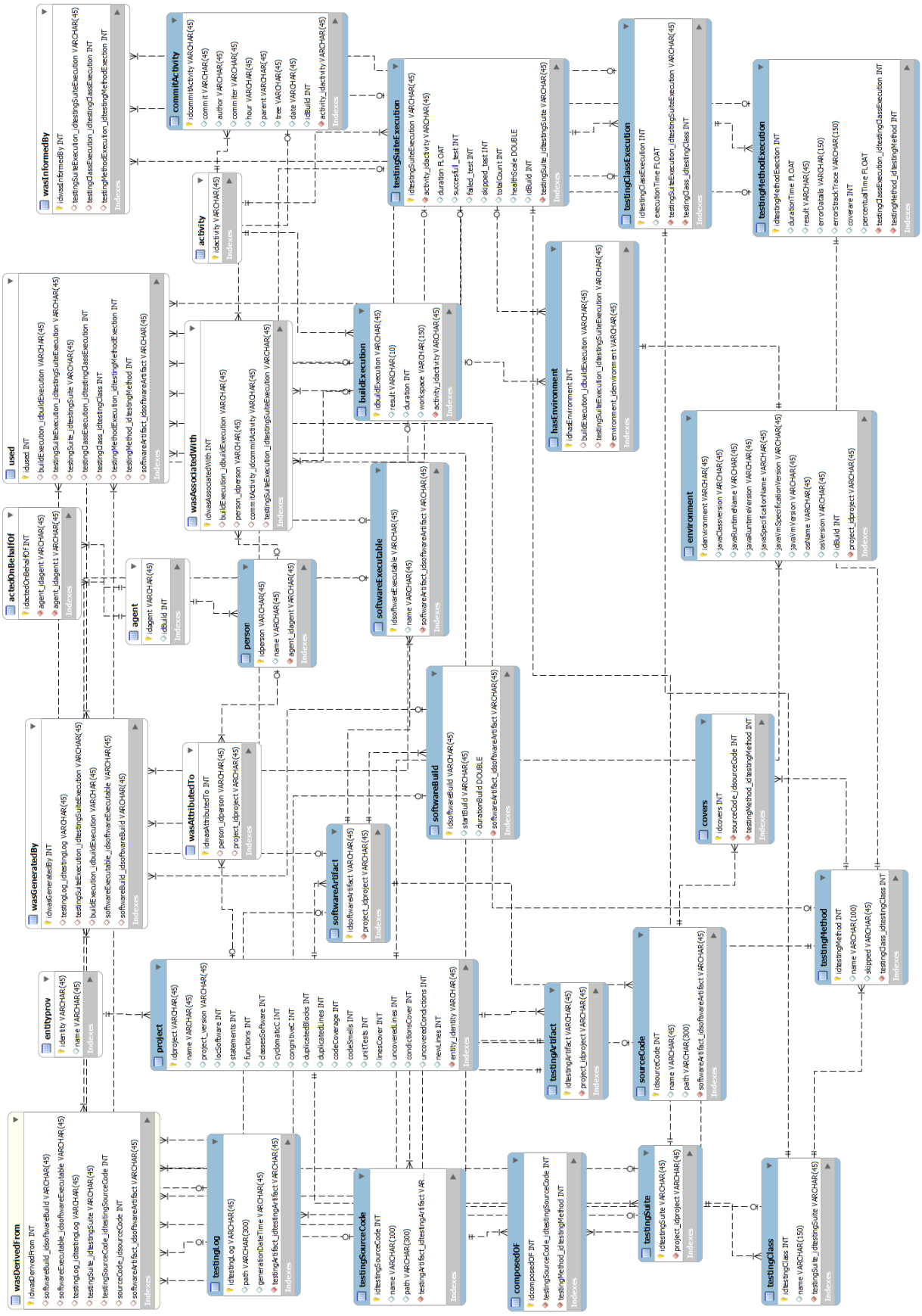
Fonte: Jenkins - Execução do buil

Tabela 4.1. Informações retiradas

Classe/ Tabela	Dado	Como o dado é extraído	Fracionamento do arquivo	Exemplo do dado
Person	name	Extraído do arquivo *build*.xml	<userId>camila</userId>	camila
TestingSuiteExecution	failing_tests		<failCount>0</failCount>	0
	skipped_tests		<skipCount>0</skipCount>	0
	successful_test	total_tests - failing_tests - skipped_tests	<totalCount>16069</totalCount>	16069
SoftwareBuild	*build*start	Extraído do arquivo *build*.xml e transformado de milisegundos em data	<startTime>1494285582531</startTime>	Mon 8 May 2017 20:19:42
	*build*Finish	*build*start+duration	<duration>128041</duration>	Mon 8 May 2017 20:21:50
TestingCaseExecution	executionTime	Extraído do arquivo junitResult.xml (tag duration)	<case> <duration>0.0</duration> <className>BagUtilsTest</className> <testName>testTransformedSortedBag</testName> <skipped>false</skipped> </case>	0.0
	result	Extraído do arquivo junitResult.xml. Caso o caso de teste tenha sido pulado, o valor na tag skipped será true. Caso o caso de teste falhe, uma tag chamada errorStackTrace aparecer, caso contrário, o caso de teste foi executado normalmente.		successful
	testCaseName	Extraído do arquivo junitResult.xml (tag testName		testTransformedSortedBag

Fonte: Elaborado pelo próprio autor

Figura 4-4. Esquema Relacional do Banco de dados



Fonte: Elaborado pelo próprio autor

4.1.2 CAMADA DE ANÁLISE DOS DADOS

A partir dos dados armazenados no banco de dados, é possível utilizar técnicas para extrair conhecimento e informações.

Atualmente existem várias abordagens que auxiliam no processo de extração de informações, tais como: *data mining*, segundo (HAND; SMYTH; MANNILA, 2001), é uma análise de um grande número de dados para encontrar relações e sumarizar os dados de novas maneiras para que possam ser entendidos e úteis para seus utilizadores; *big data*, segundo (WARD; BARKER, 2013), é o termo utilizado para descrever o processo de aplicações de abordagens computacionais, como inteligência artificial e *machine learning*, em um conjunto de dados brutos e complexos.

Além disso, existem abordagens que podem ser utilizadas para a representação do conhecimento, como as ontologias. Através delas, consegue-se estabelecer uma compreensão comum sobre objetos e os seus relacionamentos em um determinado domínio, através de um modelo formal e manipulável. As ontologias são utilizadas com o objetivo de enriquecer semanticamente os dados e de descobrir novas informações através das inferências.

Neste trabalho ambos tipos de abordagens são utilizados. Primeiramente é realizado o enriquecimento dos dados através das ontologias, e logo depois são aplicados algoritmos de inteligência artificial para a realização de previsões. As seções seguintes apresentam como essas tarefas são realizadas.

4.1.2.1 ONTOLOGIA

Conforme mencionado anteriormente, neste trabalho utilizou-se como base a ontologia denominada PROV-O. LEBO *et al.*, (2013) fornecem especificações básicas para implementar aplicações de proveniência em diferentes domínios. As classes e propriedades da ontologia PROV-O são definidas de forma que possam ser utilizadas diretamente para representar informações de proveniência em diversos domínios. Sendo assim, a Ontologia PROV-O possibilita seu uso em aplicações específicas, além de servir como um modelo de referência para a criação de ontologias de proveniência para um domínio específico (LEBO *et al.*, 2013).

Com o objetivo de identificar ontologias para testes na literatura, foi realizada um mapeamento sistemática de literatura, detalhada em:

https://github.com/CamilaAcacio/ISReT.git. Esse mapeamento seguiu o GQM: ***Analisar*** *ontologias,* ***com o propósito de*** *caracterizar,* ***em respeito aos*** *testes de software* ***do ponto de vista dos*** *analistas de testes,* ***no contexto de*** *engenharia de software.*

Como resultado, tem-se a Tabela 4.2. Nessa revisão, foram encontradas 15 ontologias, grande parte delas expressa conceitos gerais sobre testes de software. Nenhuma das ontologias encontradas utiliza o modelo PROV como base. Além disso, tratam de testes de uma forma geral, diferente da proposta, que trata sobre testes de regressão. Por fim, nenhuma delas se aplica ao contexto de engenharia de software contínua, como nesta proposta.

Dessa forma, foi desenvolvida uma ontologia formalizada em alguns conceitos do ambiente de execução de testes de regressão. A sua representação pode ser observada na Figura 4-5. As formas em cinza claro representam as classes nativas do PROV-O, já as formas em azul representam as classes inerentes ao ambiente dos testes de regressão. Essas classes são descritas na Tabela 4.3.

Tabela 4.2. Artigos Selecionados

Nome da Ontologia	**Artigo**	**Implementação nos projetos**	**Linguagem Utilizada**	**Conteúdo da Ontologia**	**PROV**
N/M	(ZHU; HUO, 2005)	Ambiente de software multiagente para suportar o desenvolvimento e a manutenção evolucionários de aplicativos baseados na web	XML	Conhecimento geral de testes de software	Não
OntoTest	(BARBOSA; NAKAGAWA; MALDONADO, 2006)	Construído para aquisição de suporte, organização, reutilização e compartilhamento de conhecimento de testes	OWL	Conhecimento geral de testes de software	Não

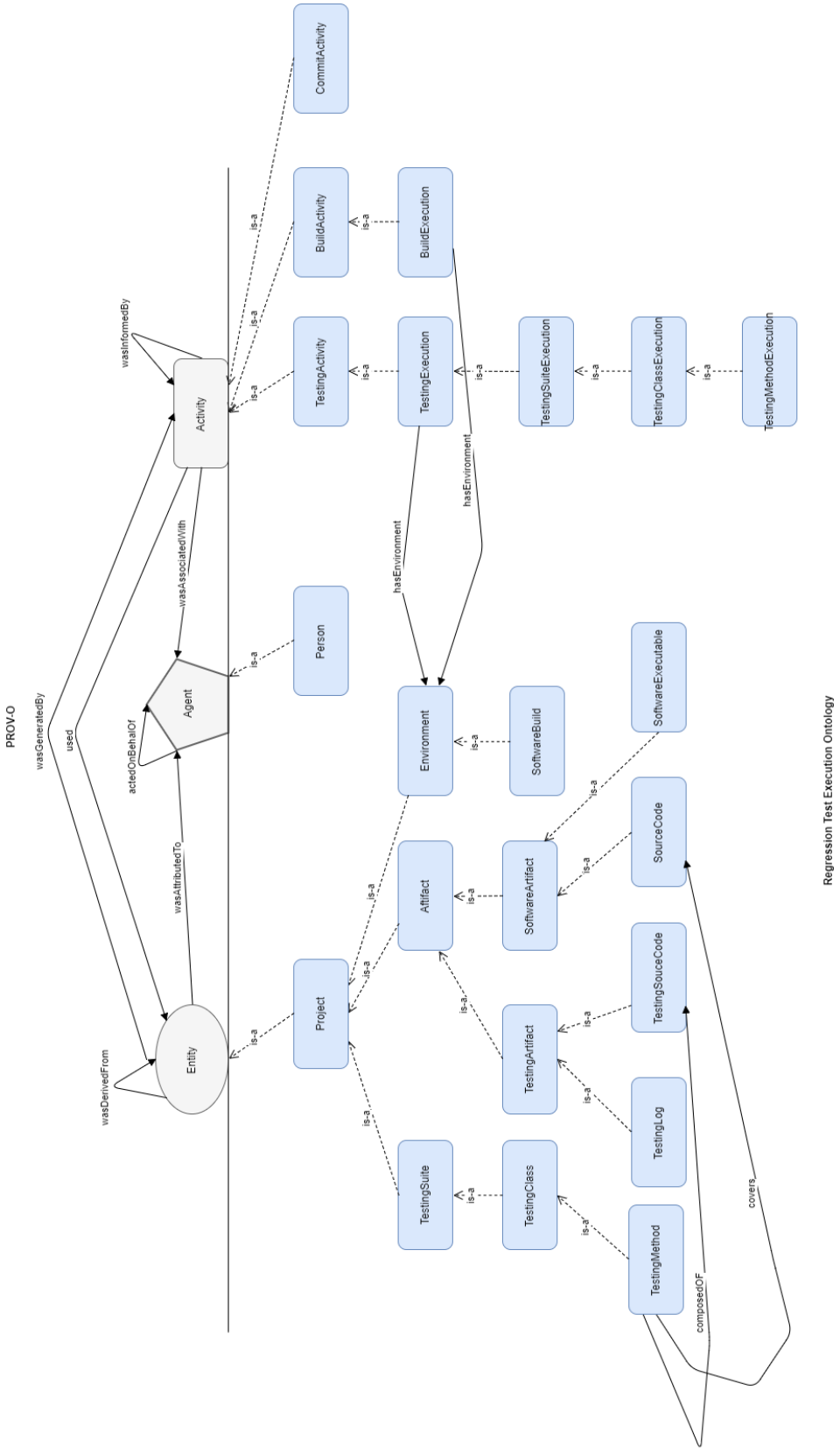
TOM (Test Modeling Ontology)	(BAI *et al.*, 2008)	Web Services	OWL-S	*Design* de teste e execução de teste	Não
STOWS	(ZHANG, Y; ZHU, 2008)	Web Services	OWL-S	Conhecimento geral de testes de software	Não
MND-TMM	(RYU; RYU; BAIK, 2008)	Desenvolvimento de sistema de software de armamento	OWL	Teste de software de acordo com os padrões militares	Não
OntoTest	(BARBOSA *et al.*, 2008)	Utilização da ontologia no desenvolvimento de ferramentas relacionadas a testes. Um aplicativo para organizar e pesquisar conceitos de teste; e para definir um conjunto comum de módulos funcionais que diferentes ferramentas de teste devem fornecer	OWL	Conhecimento geral de testes de software	Não
SWTOI	(BEZERRA; COSTA; OKADA, 2009)	Projeto de Teste do Linux	OWL	Conhecimento geral de testes de software	Não

Software Testing ontology	(CAI *et al.*, 2009)	Diretriz para construção de ontologia de teste de software baseada em SWEBOK e classificação dela baseada em modelo de qualidade de software	N/M	Teste de software com foco na reutilização do caso de teste	Não
TaaS	(YU, L *et al.*, 2009)	Melhorar a eficiência da garantia de qualidade de software	*Framework*	Quatro camadas de ontologia para *framework* Taas	Não
OntoTest	(NAKAGAWA; BARBOSA; MALDONADO, 2009)	Apresentar como as ontologias podem ser usadas no contexto de um processo para estabelecer arquiteturas de referência.	OWL	Conhecimento geral de testes de software	Não
N/M	(ANANDARAJ; PADMANABHAN; RAMESHKUMAR, 2013)	Classificação de programação e teste usando o Protege	OWL	Usando ontologia para ensinar testes de software	Não
N/M	(G. SAPNA; MOHANTY, 2011)	Capturar informações sobre as principais atividades no domínio e suas interações	UML	Gerenciamento do cenário de teste	Não
GUI Ontology	(LI, HAN *et al.*, 2011)	Construção de caso de teste	OWL	Conceito GUI, conceito não GUI e código fonte	Não

N/M	(LI, XUEXIANG; ZHANG, 2012)	Construído para suporte reutilização do caso de teste	N/M	Teste de software com foco na reutilização de casos de teste	
ROoST	(SOUZA; FALBO; VIJAYKUMAR, 2013)	Focando no processo de teste de software, atividades, artefatos que são usados e produzidos por eles, e técnicas de teste para design de caso de teste	N/M	Conhecimento geral de testes de software	Não

Fonte: Elaborado pelo próprio autor

Figura 4-5. Regression Test Execution Ontology



Fonte: Elaborado pelo próprio autor

Tabela 4.3. Classes adicionadas ao PROV-O

Classe Base PROV-O	**Classe Criada**	**Descrição**
Activity	*TestingActivity*	Responsável por agrupar atividades que são inerentes aos processos de teste
	TestingExecution	Representa uma execução dos testes
	TestingSuiteExecution	Representa uma execução da suíte de teste
	TestingClassExecution	Representa uma execução da classe de teste
	TestingMethodExecution	Representa uma execução do método de teste
	BuildActivity	Responsável por agrupar atividades que são inerentes ao processo de *build*
	BuildexEcution	Representa uma execução do *build*
	CommitActivity	Representa a realização de um *commit*
Agent	*Person*	Representa uma pessoa no ambiente de testes
Entity	*Project*	Responsável por agrupar entidades que são inerentes projeto
	TestingSuite	Representa um agrupamento de casos de teste
	TestingClass	Representa uma classe de teste
	TestingMethod	Representa um método de teste
	Artifact	Agrupa os artefatos em geral do projeto
	TestingArtifact	Agrupa artefatos relacionados aos testes do software
	TestingLog	Representa os *logs* gerados pelas execuções dos testes
	TestingSourceCode	Representa o código fonte dos casos de testes do software
	Software*Artifact*	Representa outros artefatos do software que não estão diretamente relacionados com os testes
	SourceCode	Representa os arquivos de código fonte do software
	Software*Executable*	Representa o arquivo binário utilizado para executar o software
	Environment	Representa o ambiente do projeto
	Software*Build*	Representa uma construção específica do software que está sendo testado

Fonte: Elaborado pelo próprio autor

A ontologia tem como objetivo o enriquecimento semântico dos dados coletados e, ainda, a descoberta de novos conhecimentos através de regras de inferência presentes. O modelo PROV-O possui alguns relacionamentos nativos, ao realizar a extensão do modelo, tais relacionamentos são herdados. Contudo, além dos relacionamentos nativos, foram criados novos relacionamentos apresentados na Tabela 4.4.

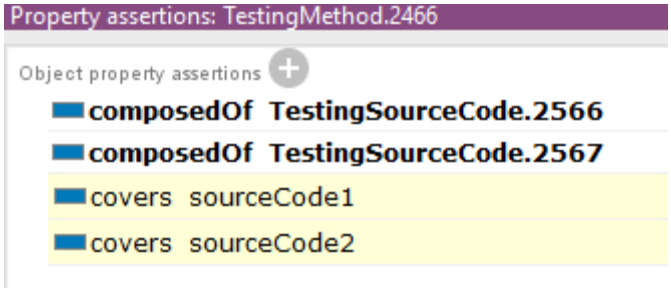
Tabela 4.4. Relacionamentos adicionados

Domínio (origem)	**Relacionamento**	**Alcance (destino)**	**Descrição**
TestingMethod	*composedOf*	*TestingSourceCode*	Utilizado para determinar os casos de teste que compõem um método de teste
TestingMethod	*covers*	*SourceCode*	Utilizado para determinar a cobertura de um método de teste com relação ao código fonte
TestingExecution BuildExecution	*hasEnvironment*	*Environment*	Utilizado para determinar quais os ambientes em que as execuções dos testes e do *build* foram realizadas

Fonte: Elaborado pelo próprio autor

Com o objetivo de exemplificar, é possível utilizar o relacionamento *covers* para a inferência da cobertura de um método de teste. Essa informação pode ser útil para os desenvolvedores avaliarem seus casos de teste. A inferência acontece a partir da proposição de que um caso de teste (*TestingSourceCode*) é derivado (*wasDerivatedFrom*) de um código fonte (*SourceCode*), e ainda, que um método de teste (*TestingMethod*) é composto (*composedOf*) por um caso de teste (*TestingSourceCode*). A inferência realizada pode ser observada na Figura 4-6 .

Figura 4-6. Inferência pela propriedade Covers



Fonte: Elaborado pelo próprio autor

Após o enriquecimento das informações, os dados estão prontos para serem utilizados pelas próximas camadas da arquitetura, explicadas nas seções seguintes.

4.1.3 CAMADA DE ABORDAGENS DE TESTE DE SOFTWARE

Esta camada é responsável por abranger as técnicas de teste. Existem diversos tipos com objetivos distintos. Um mesmo processo pode utilizar, às vezes, três, quatro, ou mais técnicas diferentes para testar um software.

O intuito da proposta é abranger o maior número de técnicas possível, visando sempre a melhoria do processo de teste de modo geral. Entretanto, inicialmente, foram selecionadas as técnicas de teste de unidade e teste de regressão. A proposta captura as informações dos testes de unidade executados durante os testes de regressão. Após a captura dos dados, ocorre o enriquecimento semântico. Com os dados prontos, são executados os algoritmos de predição, que serão explicados na subseção seguinte.

4.1.4 CAMADA DE PREVISÃO

A partir dos dados enriquecidos semanticamente, através da ontologia, é possível também utilizar técnicas de inteligência artificial para extrair conhecimento estratégico para os desenvolvedores e analistas de teste.

Existem vários algoritmos que auxiliam no processo de mineração de dados, tais como: *Logistic Regression, Naive Bayes's,* J48 *Decision Tree, Random Forest e Bayesian Network*, e outros. Após um mapeamento sistemático, apresentado na seção 3, foram selecionados três algoritmos mais utilizados na literatura: *Logistic Regression, Naive Bayes's,* J48 *Decision Tree*. Para a aplicação dos algoritmos foi utilizada uma

API do Weka (HALL *et al.*, 2009), por tratar-se de um sistema de código aberto, e conter um conjunto de algoritmos de aprendizado de máquina.

Como o objetivo da proposta é prever os resultados dos testes de unidade, foram utilizadas 20 métricas. Quatro delas, relativas à execução do teste de unidade. As dezesseis restantes, são métricas relativas ao software, como por exemplo, número de linhas de código, complexidade ciclomática e outros. Utilizar métricas relativas ao software uma prática muito comum em previsões, como apresentado na seção 3. A Tabela 4.5 exemplifica as métricas utilizadas.

Tabela 4.5. Métricas utilizadas

Métrica	**Origem**
durationTime	Execução do Teste
percentualTime	Execução do Teste
coverage	Execução do Teste
newLines	Código Fonte
LOC	Código Fonte
statements	Código Fonte
functions	Código Fonte
*classesOf*Software	Código Fonte
cyclomaticC	Código Fonte
cognitiveC	Código Fonte
duplicatedBlocks	Código Fonte
duplicatedLines	Código Fonte
codeCoverage	Código Fonte
codeSmells	Código Fonte
unitTests	Código Fonte
linesCover	Código Fonte
uncoveredLines	Código Fonte
condictionsCovers	Código Fonte
uncoveredCondictions	Código Fonte
result	Código Fonte

Fonte: Elaborado pelo próprio autor

Para que a mineração dos dados capturados seja executada, esses devem estar no formado padrão[7] especificado pela técnica de mineração de dados a ser utilizada no Weka (HALL *et al.*, 2009). Para tanto, um arquivo com a extensão ARRF é gerado pela infraestrutura, conforme a Figura 4-7.

Figura 4-7. Arquivo data.arrt

```
@RELATION result

@ATTRIBUTE durationTime REAL
@ATTRIBUTE percentualTime   REAL
@ATTRIBUTE coverage REAL
@ATTRIBUTE newLines REAL
@ATTRIBUTE locSoftware  REAL
@ATTRIBUTE statements   REAL
@ATTRIBUTE functions    REAL
@ATTRIBUTE classesSoftware  REAL
@ATTRIBUTE cyclomaticC  REAL
@ATTRIBUTE cognitiveC   REAL
@ATTRIBUTE duplicatedBlocks REAL
@ATTRIBUTE duplicatedLines  REAL
@ATTRIBUTE codeCoverage REAL
@ATTRIBUTE codeSmells   REAL
@ATTRIBUTE unitTests    REAL
@ATTRIBUTE linesCover   REAL
@ATTRIBUTE uncoveredLines   REAL
@ATTRIBUTE condictionsCovers    REAL
@ATTRIBUTE uncoveredCondictions REAL
@ATTRIBUTE class    {passed,fail}

@DATA
0.038,12.9693,1.0,1,1095,146,51,8,85,60,2,50,50.2,10,56,175,77,74,47,fail
0.039,5.14512,1.0,13,1097,147,51,8,85,60,2,50,43.0,11,56,175,91,74,51,passed
0.014,7.36842,1.0,13,1097,147,51,8,85,60,2,50,89.6,11,56,6,0,0,0,passed
0.408,71.4536,1.0,5,1095,146,51,8,85,60,2,50,91.6,10,56,5,0,0,0,passed
0.013,7.30337,1.0,6,1095,146,51,8,85,60,2,50,51.0,10,56,6,4,2,1,passed
0.11,26.57,1.0,72,1095,146,51,8,85,60,2,50,100.0,0,56,0,0,0,0,passed
0.408,71.7047,1.0,74,1016,159,52,8,92,71,2,50,86.1,11,56,188,23,86,15,passed
0.453,70.5607,1.0,0,1095,146,51,8,85,60,2,50,91.6,11,37,7,0,4,1,passed
0.466,62.973,1.0,427,1470,177,61,11,97,64,2,2,77.3,13,56,49,42,8,5,passed
```

Fonte: Elaborado pelo próprio autor

O arquivo gerado é, então, interpretado pelos algoritmos C45, *Logistic Regression* ou *Naive Bayes*. O usuário pode selecionar um, dois, ou os três algoritmos implementados da infraestrutura, através da API do Weka, como ilustrado na Figura 4-8.

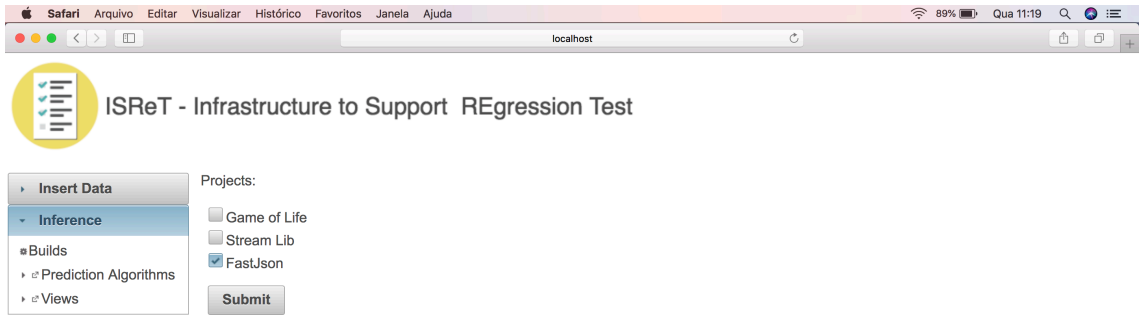
[7] Cada atributo é inserido no arquivo com extensão ARFF, precedido de "@attribute" e sucedido por todos os valores atribuídos ao mesmo, os quais são dispostos entre chaves ("}"). Já os dados preenchidos em cada linha da tabela do arquivo padrão de importação, são dispostos na ordem correspondente aos atributos definidos por "@attribute", sendo separados por vírgula e quando vazios representados por um ponto de interrogação (?). A especificação de que os valores constantes no arquivo correspondem aos dados referentes aos atributos, é dada por "@data", que precede os respectivos dados.

Figura 4-8. Passo 1: Seleção dos Algoritmos

Fonte. Infraestrutura proposta (PROV-RUT)

Após a seleção dos algoritmos, o usuário escolhe o projeto e os *builds* que deseja utilizar nas previsões. Essas etapas são ilustradas pelas Figura 4-9 e Figura 4-10. Nos exemplos aqui demonstrados, foi utilizado um projeto real chamado FastJson[8].

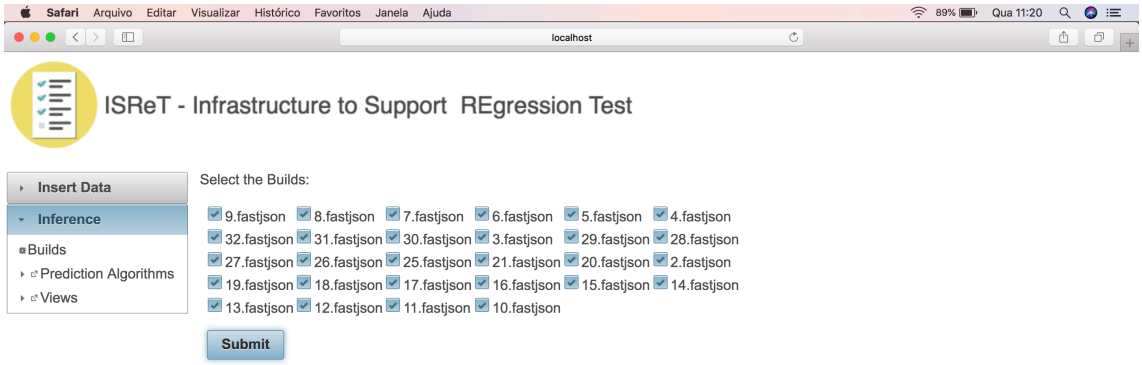
Figura 4-9. Passo 2: Seleção do Projeto



Fonte. Infraestrutura proposta (PROV-RUT)

[8] https://github.com/alibaba/fastjson

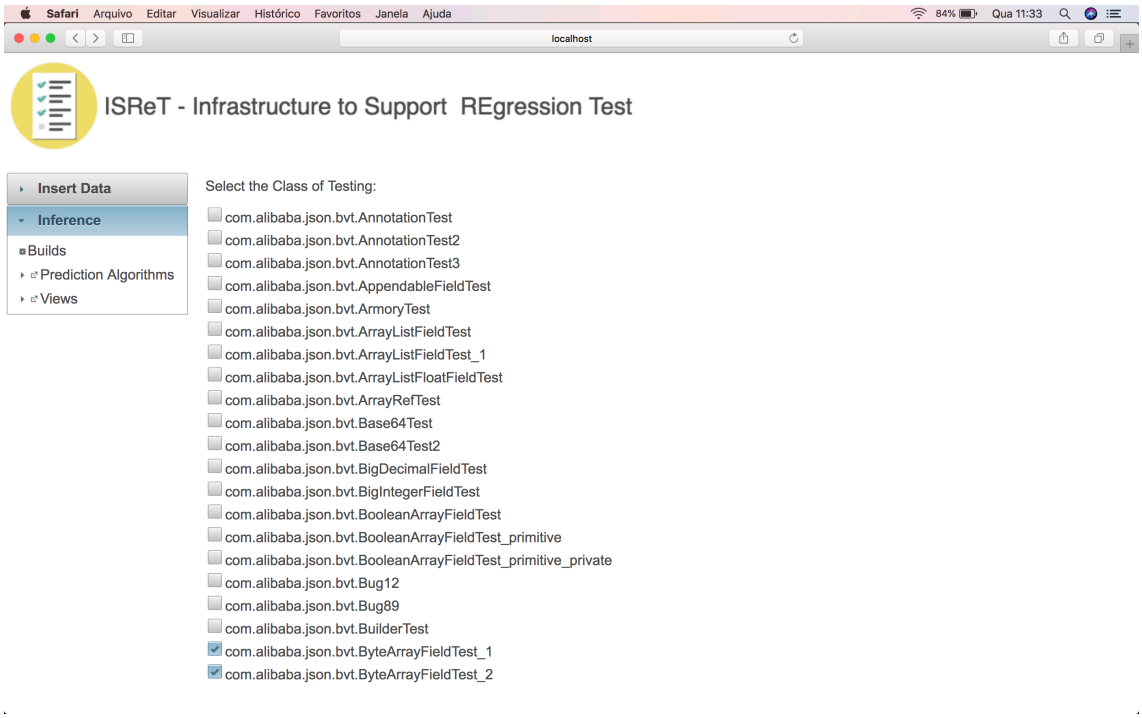
Figura 4-10. Passo 3: Seleção dos Builds



Fonte. Infraestrutura proposta (PROV-RUT)

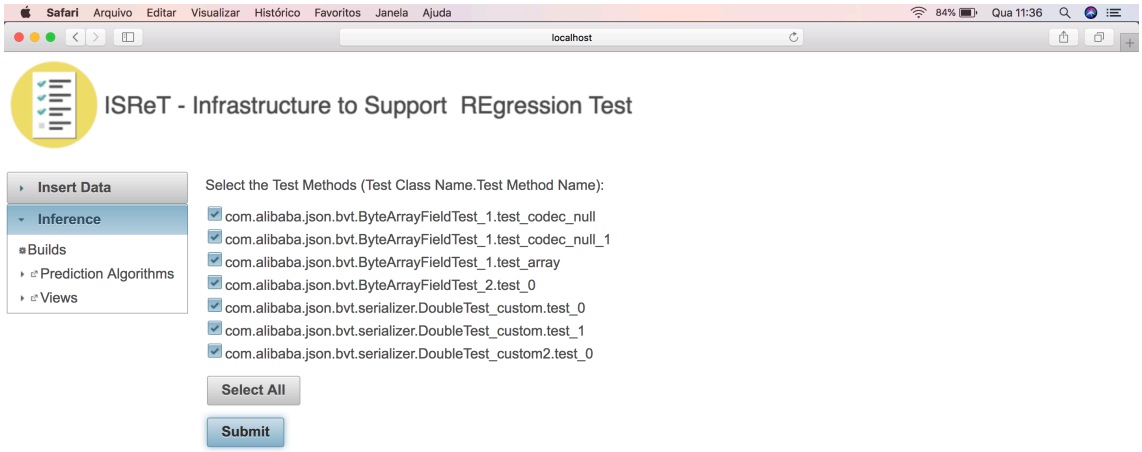
Após a escolha dos *builds*, o usuário seleciona as classes e os métodos de teste que deseja realizar as previsões. As Figuras 4-11 e 4-12 mostram o processo realizado para se obter os resultados.

Figura 4-11. Passo 4: Seleção das Classes de Teste



Fonte. Infraestrutura proposta (PROV-RUT)

Figura 4-12. Passo 5: Seleção dos Métodos de Teste

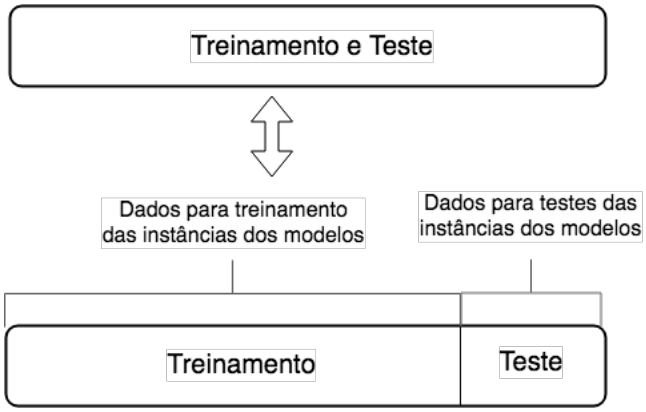


Fonte. Infraestrutura proposta (PROV-RUT)

Para a realização das previsões cada método de teste selecionado tem uma instância criada para os modelos (algoritmos) selecionados. Cada instância passa por duas etapas: (i) treinamento, no qual um conjunto de dados é utilizado para ajustar os parâmetros da instância do modelo, e (ii) teste, na qual a instância do modelo final é avaliada.

Conforme lustrado na Figura 4-13, a infraestrutura realiza a distribuição dos dados, treinamento e teste, da seguinte forma: o conjunto *Teste*, contém as informações do *build* mais recente e o conjunto de *Treinamento* é caracterizado por possuir as informações dos *builds* anteriores ao mais recente. Exemplificando, os *builds* 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, foram selecionados. O conjunto de Teste seria composto pelas informações geradas pelo *build* 10. E o conjunto Treinamento seria composto pelas informações dos builds 1, 2, 3, 4, 5, 6, 7, 8, 9. A finalidade das divisões é: o conjunto Treinamento é utilizado para o treinamento das instâncias dos modelos e, o conjunto *Teste* é responsável por realizar os testes nas instâncias dos modelos criadas. Os dados foram divididos dessa forma, visto que, no contexto de engenharia de software contínua, onde as mudanças são pequenas, o *build* mais atual é mais semelhante ao próximo do que qualquer outra instância dos dados. Desta forma, geram-se previsões mais coerentes ao contexto (BALDAUF; DUSTDAR; ROSENBERG, 2007).

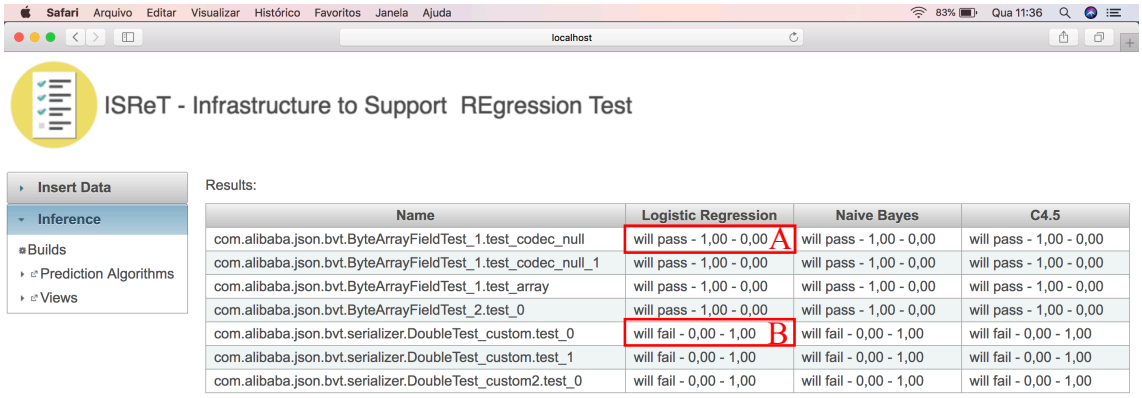
Figura 4-13. Divisão dos dados Históricos



Fonte. Elaborado pelo próprio autor

Após o treinamento e teste de todos os métodos de testes selecionados pelo usuário, os resultados são mostrados como mostra a Figura 4-14

Figura 4-14. Passo 6: Resultados das previsões



Fonte. Infraestrutura proposta (PROV-RUT)

Assinalados na Figura 4-14 pelas letras A e B em vermelho estão os resultados obtidos. A infraestrutura além de disponibilizar ao usuário a classe, *pass* ou *fail*, que aquele método de teste pertence, disponibiliza a probabilidade do resultado. A probabilidade varia entre 0,00 e 1,00 e é dividida entre duas classes, *pass* ou *fail.*

Nos exemplos destacados, o método de teste “test_codec_null” tem 1,00 ou 100% de probabilidade de pertencer a classe *pass*. E o método “test_0” tem 1,00 ou 100% de probabilidade de pertencer a classe *fail*. Essa probabilidade foi inserida para auxiliar os usuários na interpretação dos resultados.

Na próxima seção é apresentada a camada de otimização, responsável por conter as abordagens dos testes de regressão necessárias para a otimização da suíte de testes.

4.1.5 CAMADA DE OTIMIZAÇÃO

Uma suíte de testes de regressão tradicional é composta por todos os casos de teste desenvolvidos ao longo do processo para o software. Porém, essa técnica é muito custosa, necessita de um grande poder computacional e demanda muito tempo. Para resolver esse problema, foram criadas técnicas para minimizar esse custo (DSSOULI *et al.*, 2017).

Segundo DSSOULI *et al.* (2017), as técnicas de seleção de casos de teste de regressão têm como objetivo selecionar um subconjunto de casos de testes de uma suíte de testes de regressão. A seleção pode ser manual, caso o desenvolvedor conheça todo o contexto do projeto, ou pode ser automatizada, utilizando alguma técnica específica. As técnicas de priorização visam alterar a ordem na qual os casos de teste são executados, seguindo algum critério específico. Dessa forma, caso seja necessário parar a execução dos testes, é possível que os testes mais importantes já tenham sido executados, por terem sido priorizados. Por fim, as técnicas de minimização têm como objetivo diminuir a suíte de testes retirando testes obsoletos ou redundantes.

A proposta implementa uma técnica de priorização baseada no histórico de falhas dos testes de unidade. A técnica calcula o percentual de falhas daquele teste de unidade ao longo dos *builds* selecionados. Para isso, primeiramente, o usuário seleciona o projeto que deseja obter informações. Em seguida, seleciona os *builds* que deseja que entrem no cálculo. O cálculo é então realizado, através da Equação (4.1):

$$\boldsymbol{P(x) = 100 * \sum f \div \sum t} \qquad (4.1)$$

Onde:

x é o teste de unidade;

f é somatório de falhas que o teste x obteve nos *builds* selecionados;

t é o somatório de execuções (falhas ou sucessos) que o teste x obteve nos *builds* selecionados.

Após a realização do cálculo, um arquivo em formato XML é gerado com a lista dos testes e seus respectivos *scores*. A Figura 4-15 é uma exemplificação do

arquivo gerado pela técnica. A *Tag testName* contém o nome do método de teste e a *Tag score* contém o valor calculado, pela equação mencionada anteriormente, para aquele teste.

Figura 4-15. Exemplo de arquivo gerado pela técnica implementada

```
<case>
<testName>Adding two integers</testName>
<score>10.0</score>
</case>
<case>
<testName>Adding two integers</testName>
<case>
<testName>aDeadCellSymbolShouldBeADot</testName>
<score>0.0</score>
</case>
<case>
<testName>aLiveCellShouldBeRepresentedByAnAsterisk</testName>
<score>25.0</score>
</case>
<case>
<testName>aLiveCellShouldBeRepresentedByAnAsterisk</testName>
<score>33.333332</score>
</case>
<case>
<testName>shouldBeAbleToCountLiveNeighboursOfACell</testName>
<score>16.666666</score>
</case>
<case>
<testName>shouldBeAbleToReadTheStateOfADeadCell</testName>
<score>0.0</score>
</case>
```

Fonte: Elaborado pelo próprio autor

Esse arquivo gerado pode ser utilizado por qualquer ferramenta externa para executar os testes de software. Além disso, outras técnicas podem ser implementadas utilizando as informações disponíveis.

4.1.6 MÓDULO DE RASTREABILIDADE

Como dito anteriormente, uma das informações inferidas é a cobertura de código fonte. Porém, para haver essa inferência é necessário obter informações sobre o código fonte. Como os relatórios gerados pelo Jenkins não fornecem essas informações, foi necessária a criação de um módulo de rastreabilidade.

O módulo de rastreabilidade tem como objetivo realizar a rastreabilidade entre o caso de teste e o código fonte. Para isso, o usuário precisa informar quais são as classes de teste e seu local de armazenamento. Diante dessa informação, o sistema lê cada classe e retira as informações de código fonte. A Figura 4-16 ilustra esse processo que se dá, a partir de uma lista de métodos de teste:

- Procura por anotações que indicam o início de um método de teste, tais como @Test, sinalizado na Figura 4-16 pela letra A;
- Compara se o método encontrado pela anotação é o mesmo dado pela lista, indicado na Figura 4-16 pela letra B;
    - Caso seja o mesmo método, procura por construtores no método de teste, indicado na Figura 4-16 pela letra C;
        - Após encontrar os construtores, procura por *assert*(s) do método, indicado na Figura 4-16 pela letra D;
        - Tendo a lista dos construtores e *asserts*, o algoritmo realiza o merge dessas informações;
        - Por fim, ele cria uma lista de códigos que são atingidos por aquele teste, como ilustrado na
        - Figura 4-17.
    - Caso não seja, continua comparando com os métodos restantes.

Após a coleta dos dados sobre o código fonte, eles são armazenados no banco de dados.

Figura 4-16. Código fonte do Teste testPushPeekPop

```
66      @SuppressWarnings("unchecked")                                          A
67      public void testPushPeekPop() {                                         B
68          final ArrayStack<E> stack = makeObject();                           C
69
70          stack.push((E) "First Item");
71          assertTrue("Stack is not empty", !stack.empty());                   D
72          assertEquals("Stack size is one", 1, stack.size());
73          assertEquals("Top item is 'First Item'", "First Item", (String) stack.peek());
74          assertEquals("Stack size is one", 1, stack.size());
75
76          stack.push((E) "Second Item");
77          assertEquals("Stack size is two", 2, stack.size());
78          assertEquals("Top item is 'Second Item'", "Second Item", (String) stack.peek());
79          assertEquals("Stack size is two", 2, stack.size());
80
81          assertEquals("Popped item is 'Second Item'", "Second Item", (String) stack.pop());
82          assertEquals("Top item is 'First Item'", "First Item", (String) stack.peek());
83          assertEquals("Stack size is one", 1, stack.size());
84
85          assertEquals("Popped item is 'First Item'", "First Item", (String) stack.pop());
86          assertEquals("Stack size is zero", 0, stack.size());
87
88      }
89
```

Fonte: Projeto Apache CommonsCollections

Figura 4-17. Resultado obtido pelo módulo de rastreabilidade

```
1  testPushPeekPop
2  makeObject.empty
3  makeObject.size
4  makeObject.peek
5  makeObject.pop
6
```

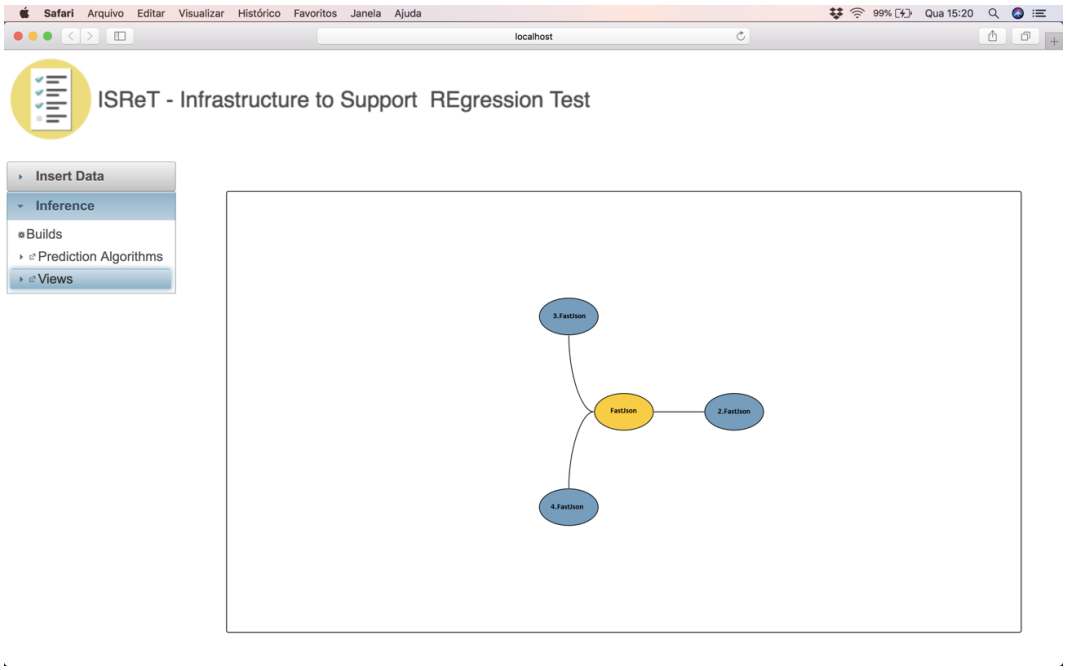
Fonte: Elaborado pelo próprio autor

4.1.7 MÓDULO DE VISUALIZAÇÃO

Uma aplicação web foi desenvolvida para facilitar a consulta aos dados por usuários, além de possibilitar a visualização das informações advindas da análise dos dados de execução do processo pela infraestrutura. Os dados são obtidos e exibidos na interface, a qual apresenta listagens distintas dos conteúdos, constantes nos registros de execução dos processos, e também permite a edição ou deleção desses.

A título de exemplo, a Figura 4-18 trás uma visualização disponível na arquitetura que tem como elemento central, em amarelo, o projeto selecionado e como objeto secundário, em azul, os *builds* daquele projeto.

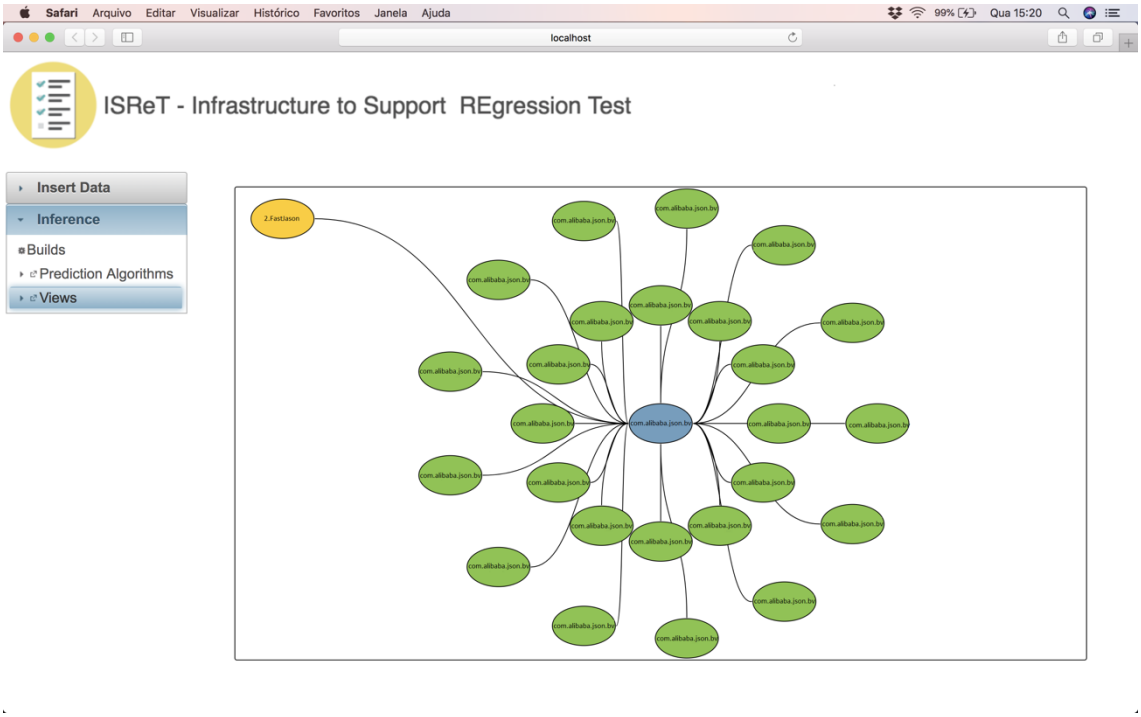
Figura 4-18. Visualização dos builds instanciados na ontologia



Fonte. Elaborado pelo próprio autor

A Figura 4-19 é uma visualização, estilo *mindmap*, das classes de testes, em verde, pertencentes a um *build*, em azul, de um projeto, em amarelo, FastJson.

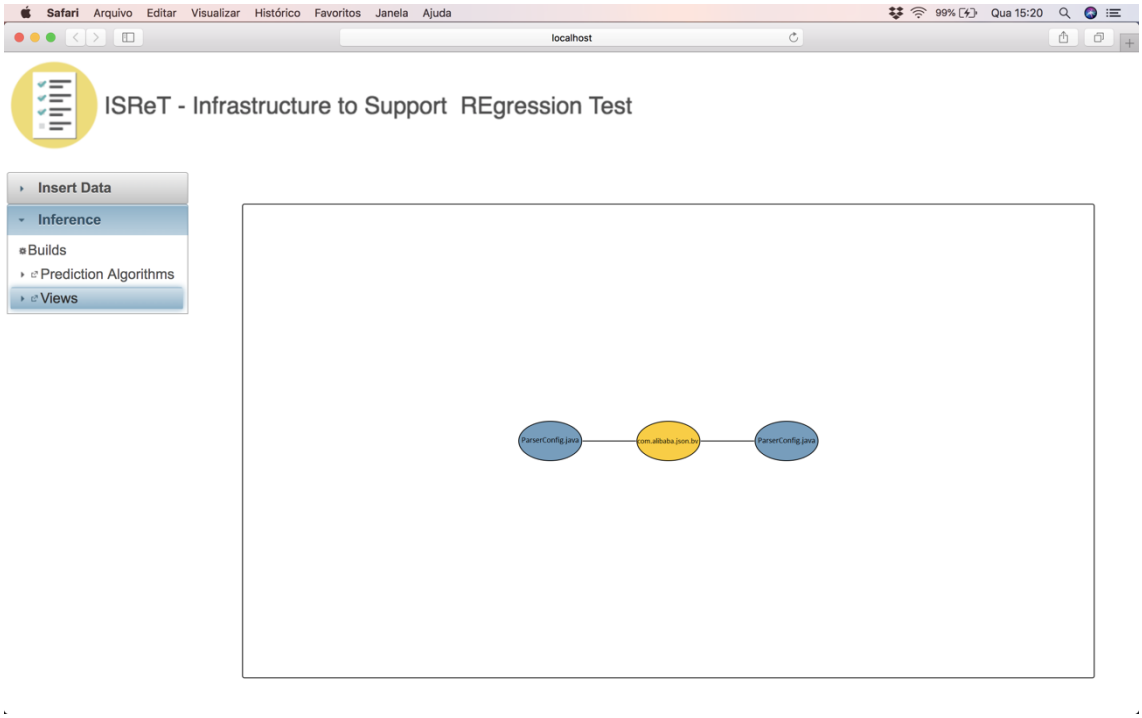
Figura 4-19. Mindmap das classes executadas no build número 2



Fonte. Elaborado pelo próprio autor

Por fim, tem-se a Figura 4-20. Tal figura exemplifica a inferência realizada pela ontologia. Essa visualização trata-se de um *mindmap* no qual o elemento central, em amarelo, é um caso de testes e os elementos secundários, em azul, são os códigos fonte cobertos por aquele caso de teste.

Figura 4-20. Mindmap da cobertura do teste



Fonte. Elaborado pelo próprio autor

4.2 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Este capítulo apresentou a infraestrutura desenvolvida. Através do uso da proveniência de dados aliado a ontologias e técnicas de inteligência computacional, é possível indicar ao desenvolvedor ou analista de testes possíveis resultados dos testes de unidade. Foram apresentadas as técnicas utilizadas e desenvolvidas para o funcionamento da infraestrutura, bem como a forma de visualização dos resultados apresentados.

O próximo capítulo apresenta a avaliação da infraestrutura, abrangendo a descrição das formas de avaliação e os resultados obtidos.

5 AVALIAÇÃO DA INFRAESTRUTURA DE PREVISÃO DOS RESULTADOS DE TESTES DE UNIDADE

Com o intuito de avaliar a infraestrutura proposta foi conduzido um estudo experimental. Esse estudo teve como objetivo avaliar a viabilidade das instâncias dos modelos de previsão dos resultados de testes de unidade e, através da coleta de dados em um projeto real, traçar um paralelo entre as previsões realizadas pelas instâncias dos modelos e os resultados obtidos na prática.

5.1 ESTUDO EXPERIMENTAL

Segundo WOHLIN *et al.* (2012) os estudos experimentais são realizados quando os responsáveis desejam controlar a situação e querem manipular o comportamento de forma direta, precisa e sistemática. Além disso, os estudos experimentais envolvem mais de um tratamento para comparar os resultados.

Estudos experimentais são apropriados para investigar diferentes aspectos, incluindo: confirmar teorias, confirmar algo amplamente aceito como verdadeiro, explorar relações, avaliar a acurácia de modelos (caso do experimento aqui realizado) e validar medidas. Além disso, envolvem diferentes etapas, tais como: escopo, planejamento, operação, síntese e resultados (WOHLIN *et al.*, 2012).

Os estudos experimentais podem ser orientados por humanos ou orientados por tecnologias. Nos estudos experimentais orientados por humanos, os participantes aplicam diferentes tratamentos em diferentes objetos como, por exemplo, dois métodos de inspeção de código são aplicados em pedaços de código. Já nos estudos experimentais orientados por tecnologias diferentes ferramentas são aplicadas em diferentes objetos, por exemplo, no caso experimento aqui realizado, duas ferramentas, Jenkins e a infraestrutura proposta, são aplicadas em um projeto para gerarem os resultados dos casos de teste (WOHLIN *et al.*, 2012). Quando o experimento é conduzido em laboratório e sob condições controladas, sem a interferência humana, e utiliza modelos computacionais, chama-se *in virtuo*, caso do experimento aqui descrito.

Para a realização do experimento foi necessária a seleção de um projeto. Este é um projeto real, chamado *FastJson* e disponibilizado no GitHub[9]. Foi selecionado pelos requisitos:

- conter testes de unidade e regressão, visto que são os testes que se deseja analisar;
- ser escrito em Java devido à necessidade de utilização do JUnit[10] e esse ser uma particularidade da linguagem;
- ser construído através de ferramentas de automação de *builds*, como *Maven* e *Ant*;
- possuir *commits* que realizam alterações que impactam no software, como adicionar métodos ou classes de teste, retirar métodos ou classes de teste;
- possuir integração com outro servidor de integração contínua como o Travis, o que possibilita ter resultados sem que o *build* seja executado, gerando uma visão sobre as modificações;
- estar entre os projetos mais populares do GitHub escritos em Java;

O projeto possui 69 contribuintes, 103 versões, cerca de 36 mil linhas de código, 4220 testes de unidade e 211 classes. É um dos projetos do site Alibaba, um site de vendas de produtos chineses.

FastJson é uma biblioteca Java que tem como objetivos: fornecer métodos simples que convertem objetos Java em JSON, e vice-versa; fornecer melhor desempenho ao servidor e ao cliente *Android*; permitir que objetos pré-existentes não modificáveis sejam convertidos em JSON; oferecer suporte extensivo ao Java *Generics*; permitir representações personalizadas para objetos e apoiar objetos arbitrariamente complexos (tipos genéricos e heranças profundas). Além disso, atende aos requisitos descritos.

As etapas que compõem o experimento, citadas anteriormente, são descritas nas subseções seguintes.

[9] https://github.com/alibaba/fastjson
[10] https://junit.org/junit5/

5.2 ESCOPO

Esta subseção apresenta o objetivo do estudo para avaliar a viabilidade da infraestrutura, conforme descrito na Tabela 5.1. Este estudo foi estabelecido de acordo com a abordagem GQM *(Goal, Question, Metric)* (WOHLIN *et al.*, 2012).

Tabela 5.1. Objetivo da Avaliação

Questões	**Respostas**
Objeto do estudo (o que será analisado?)	Infraestrutura de predição dos resultados de testes de unidade
Propósito (porquê / para o que o objeto vai ser analisado?)	Avaliar a performance da infraestrutura
Foco da qualidade (que propriedades do objeto serão analisadas?)	Acurácia das previsões realizadas pela infraestrutura a partir dos dados de proveniência
Perspectiva (quem irá utilizar os dados coletados)	Analistas de Teste
Contexto (em qual contexto a análise será realizada?)	Testes de Regressão na Eng. Software Contínua

Fonte. Elaborado pelo próprio autor

Assim, essa avaliação consiste em: Analisar a infraestrutura de previsão dos resultados de testes de unidade com a finalidade de avaliar sua performance com respeito à acurácia das previsões a partir dos dados de proveniência, do ponto de vista de analistas de testes, no contexto dos testes de regressão na Eng. Software Contínua.

5.3 PLANEJAMENTO

A etapa de planejamento é subdividida em: Contexto, Formulação das Hipóteses, Seleção de Variáveis, Seleção de Indivíduos, Instrumentalização e Validade da avaliação, que serão apresentados nas subseções seguintes: 5.3.1, 5.3.3, 5.3.4 e 5.4.

5.3.1 CONTEXTO

O contexto em que os estudos experimentais foram realizados são testes de regressão na Eng. Software Contínua. Os testes de regressão são aqueles executados a cada modificação realizada no sistema visando assegurar que novos defeitos não foram

introduzidos e o conjunto continua funcionando como planejado ou como anteriormente à adição das modificações.

A utilização dos testes de regressão como contexto permite que outros pesquisadores possam replicar os estudos experimentais realizados. Além disso, não é necessário um grande esforço para replicar o ambiente do experimento, tornando-o mais fácil de ser replicado.

Após definição do contexto, iniciou-se a etapa de formulação das hipóteses, descritas na subseção seguinte.

5.3.2 HIPÓTESES

O experimento foi realizado com o intuito de investigar se os resultados obtidos através das previsões realizadas pela infraestrutura são equivalentes aos resultados encontrados através da execução do *build*. A questão de pesquisa que dirigiu esse experimento foi:

"Existe diferença entre as previsões realizadas pela infraestrutura e os resultados encontrados através da execução do *build*?".

Assim, foi estabelecida a hipótese nula e alternativa:

$H_{0:}$ **Não** existe diferença entre as previsões realizadas pela infraestrutura e os resultados obtidos através do *build*.

$H_{1:}$ **Existe** diferença entre as previsões realizadas pela infraestrutura e os resultados obtidos através do *build*.

A partir dessa questão de pesquisa, derivou-se uma questão secundária:

"Existe diferença entre as previsões realizadas pelos três algoritmos: *Logistic Regression, Naive Bayes e C4.5 Algorithm*?".

$H_{0:}$ **Não** existe diferenças entres as previsões realizadas entres os três algoritmos.

$H_{1:}$ **Existe** diferenças entres as previsões realizadas entres os três algoritmos.

5.3.3 SELEÇÃO DE VARIÁVEIS

Para a realização do experimento, foram utilizadas 20 métricas, descritas anteriormente. Quatro delas, relativas à execução do teste de unidade. As dezesseis restantes, são

métricas relativas ao software. Estas, as 20 métricas especificadas na Tabela 5.2, são as variáveis independentes do experimento.

Tabela 5.2. Métricas utilizadas

Métrica	**Origem**
durationTime	Execução do Teste
percentualTime	Execução do Teste
coverage	Execução do Teste
newLines	Código Fonte
LOC	Código Fonte
statements	Código Fonte
functions	Código Fonte
*classesOf*Software	Código Fonte
cyclomaticC	Código Fonte
cognitiveC	Código Fonte
duplicatedBlocks	Código Fonte
duplicatedLines	Código Fonte
codeCoverage	Código Fonte
codeSmells	Código Fonte
unitTests	Código Fonte
linesCover	Código Fonte
uncoveredLines	Código Fonte
condictionsCovers	Código Fonte
uncoveredCondictions	Código Fonte
result	Código Fonte

Fonte: Elaborado pelo próprio autor

Como variáveis dependentes, tem-se os resultados obtidos pelas previsões, tais resultados se dividem em Passará (*Will pass*) e Falhará (*Will fail)*, os quais são as probabilidades daquele teste obter como resultado: sucesso ou falha no próximo *build*.

5.3.4 SELEÇÃO DE INDIVÍDUOS

Esta etapa tem como objetivo gerar indivíduos que serão utilizados como entrada pela infraestrutura para a realização das previsões. Após a seleção do projeto *FastJason* utilizou-se a última versão compartilhada no ano de 2017 com o intuito de reproduzir as modificações ocorridas até os dias atuais (fevereiro de 2018) e, consequentemente, gerar as informações necessárias.

Como mostra a Figura 5-1, esse processo é caracterizado por algumas etapas descritas a seguir. A primeira delas é definida pela replicação exata de cada *commit*, a Figura 5-2 exemplifica essa replicação. Destacados em vermelho tem-se o repositório original (alibaba/fastjson) e o repositório replicado (CamilaAcacio/fastjson), as inserções e deleções do *commit* foram replicadas no CamilaAcacio/fastjson exatamente como no original. A Tabela 5.3 apresenta cada modificação realizada e o número do *build* que executou tal modificação. Na segunda etapa, as alterações são enviadas ao GitHub. A terceira, envolve a execução do *build* no *Jenkins*, como exemplificado na Figura 5-3. Tal figura exemplifica a tela de *build* da ferramenta. Nela encontra-se as informações do *build* em questão, em destaque estão as alterações que entraram naquele *build*. Finalizado o processo, os relatórios necessários foram gerados. Por fim, a infraestrutura proposta inicia o processo de extração das informações. Os relatórios gerados pelo Jenkins são lidos, e os dados necessários são retirados e armazenados no banco de dados da infraestrutura.

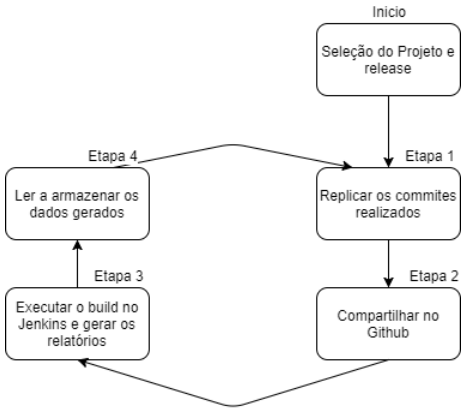
Tabela 5.3. Commits realizados e seus respectivos Builds

Commit	***Build***
add testcase for issue #1679	2
support Swagger UI 2.0 filter some types	3
Merge pull request #29 from alibaba/master ...	4
Merge pull request #1686 from VictorZeng/master ...	5
Slightly optimize code and remove redundant variables	6
bug fixed for issue #1699	7
bug fixed for issue #1662	8
fixed testcase.	9
add testcase for issue #1510	10
bug fixed for BeanToArray with PropertyFilters. for issue #1580	11
1.2.46-SNAPSHOT	12
add more denylist.	13

add more denylist.	14
bug fixed for toJavaObject. for issue #1727	15
add testcase for issue #1725	16
bug fixed for scanFloat. for issue #1723	17
bug fixed for enum deser. for issue #1582	18
fixed testcase.	19
add testcase for issue #569	20
1.2.47-SNAPSHOT	21
fixed compile warnings.	25
bug fixed for jsonpath	26
jsonpath support '&&' and '||', for issue #1733	28
add MappingFastJsonMessageConverter and testcase for spring-messaging	30
add testcase for issue #1739	31
bug fixed for float parse. for issue #1723	32

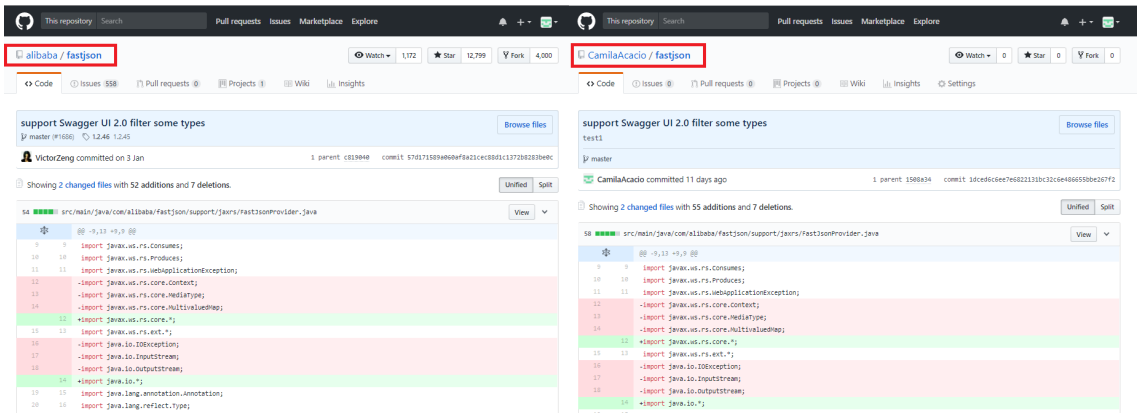
Fonte. Elaborado pelo próprio autor

Figura 5-1. Ciclo de elaboração dos dados



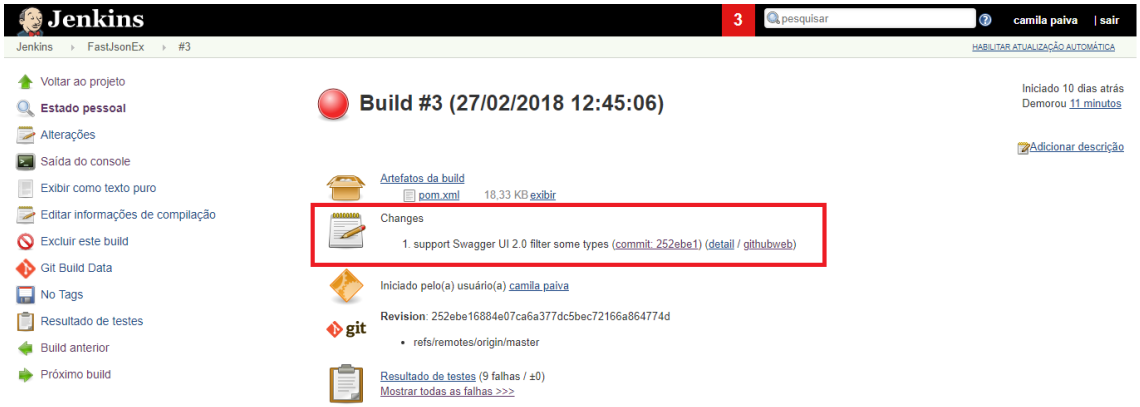
Fonte. Elaborado pelo próprio autor

Figura 5-2. Replicação do Commit



Fonte. Elaborado pelo próprio autor

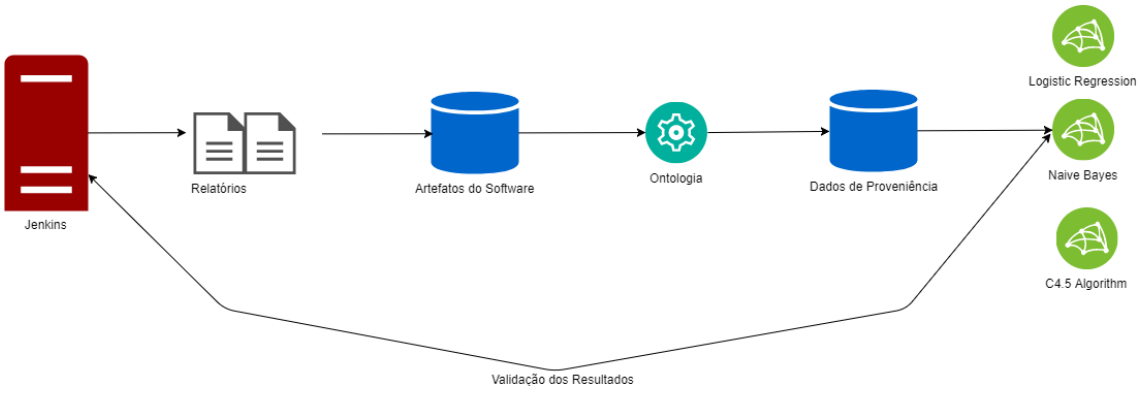
Figura 5-3. Execução do Build com as modificações realizadas



Fonte. Elaborado pelo próprio autor

Após os dados devidamente capturados iniciou-se o processo de avaliação da infraestrutura. Para isso, usou-se o fluxo representado na Figura 5-4. Conforme ilustrado, após a captura das informações (artefatos do software), essas são instanciadas na ontologia para a obtenção das informações de cobertura através das inferências (dados de proveniência). Geradas todas as informações necessárias, essas são selecionadas através de *SQLs*, Linguagem de Consulta Estruturada, para a realização da previsão através dos algoritmos: *Logistic Resgression, Naive Bayes e C45.*

Figura 5-4. Fluxo da avaliação



Fonte. Elaborado pelo próprio autor

Foram consideradas no processo de previsão as informações coletadas e instanciadas referentes a 26 *builds* (2 a 21, 25, 26, 28, 30 a 32) do projeto *FastJason*. Alguns *builds* foram descartados pois foram executados sem nenhuma modificação.

Essas informações foram utilizadas para a realização de 7 estudos experimentais diferentes, como mostra a Tabela 5.4.

Cada estudo experimental teve suas informações divididas. Tal divisão ocorreu como ilustrado na Figura 5-5.

A primeira divisão chamada de Validação é caracterizada por conter as informações do *build* mais recente realizado, como mostra a Tabela 5.4, coluna "**Validação**". Este conjunto servirá como validação dos resultados, averiguando se as previsões geradas são semelhantes ou divergem dos resultados reais. Caracterizam a segunda divisão, os dados chamados de Treinamento e Teste, compostos pelos *builds* anteriores ao mais recente, como mostra a Tabela 5.4, coluna "**Treinamento e Teste**".

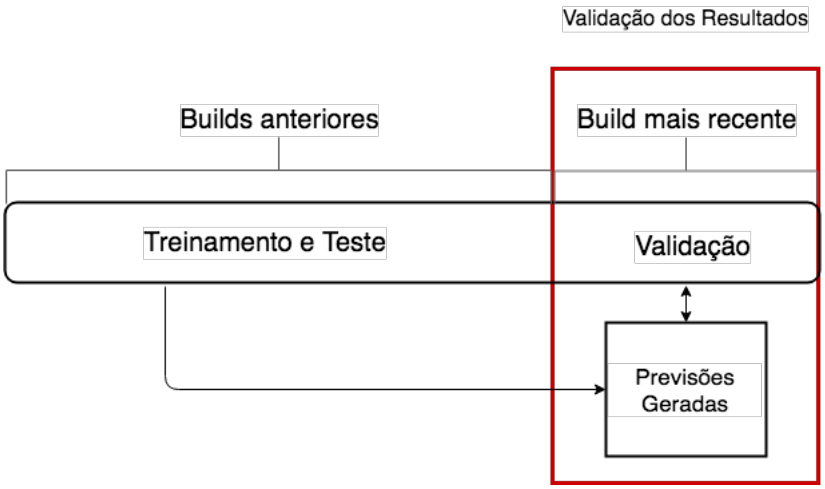
Tabela 5.4. Estudos experimentais e Builds utilizados

	Treinamento e Teste	**Validação**
Estudo experimental 1	2 a 20	21
Estudo experimental 2	2 a 21	25
Estudo experimental 3	2 a 21, 25	26
Estudo experimental 4	2 a 21, 25, 26	28
Estudo experimental 5	2 a 21, 25, 26, 28	30
Estudo experimental 6	2 a 21, 25, 26, 28, 30	31
Estudo experimental 7	2 a 21, 25, 26, 28, 30, 31	32

Fonte: Elaborado pelo próprio autor

A subdivisão entre os dados para treinamento e teste, explicada na seção 4.1.4, ocorre da seguinte forma: dos *builds* utilizados para Treinamento e Teste o mais atual, é utilizado como teste dos modelos contidos na infraestrutura e o restante são utilizados para a realização do treinamento dos algoritmos (*Logistic Regression*, C45, *Naive Bayes*). Como, por exemplo, no estudo experimental 1, foram utilizados os *Builds* 2 a 20. Para o teste, utilizou-se o *build* 20, para o treinamento, os *builds* 2 a 19.

Figura 5-5. Divisão dos dados para a realização do experimento



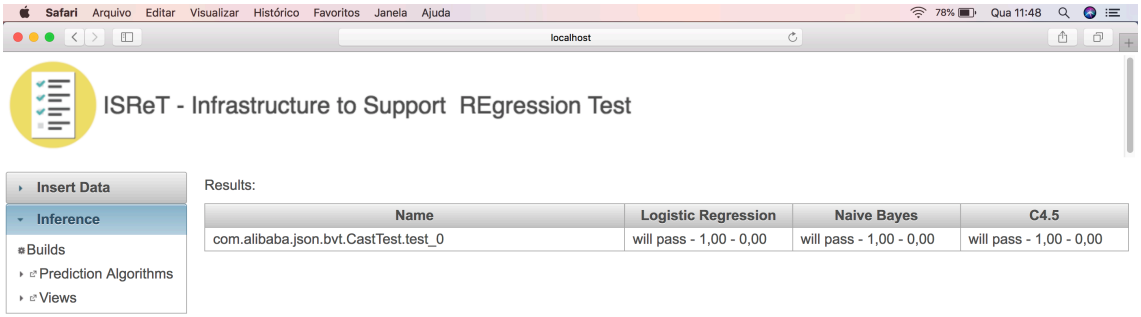
Fonte. Elaborado pelo próprio autor

5.4 AVALIAÇÃO DA INFRAESTRUTURA DE PREVISÃO DOS RESULTADOS DOS TESTES DE UNIDADE

Para a avaliação da infraestrutura foram realizados 7 estudos experimentais. Os resultados para os estudos experimentais se resumem em quatro diferentes tipos. São eles: verdadeiros positivos, verdadeiros negativos, falsos positivos e falsos negativos. Esses tipos são explicados a seguir.

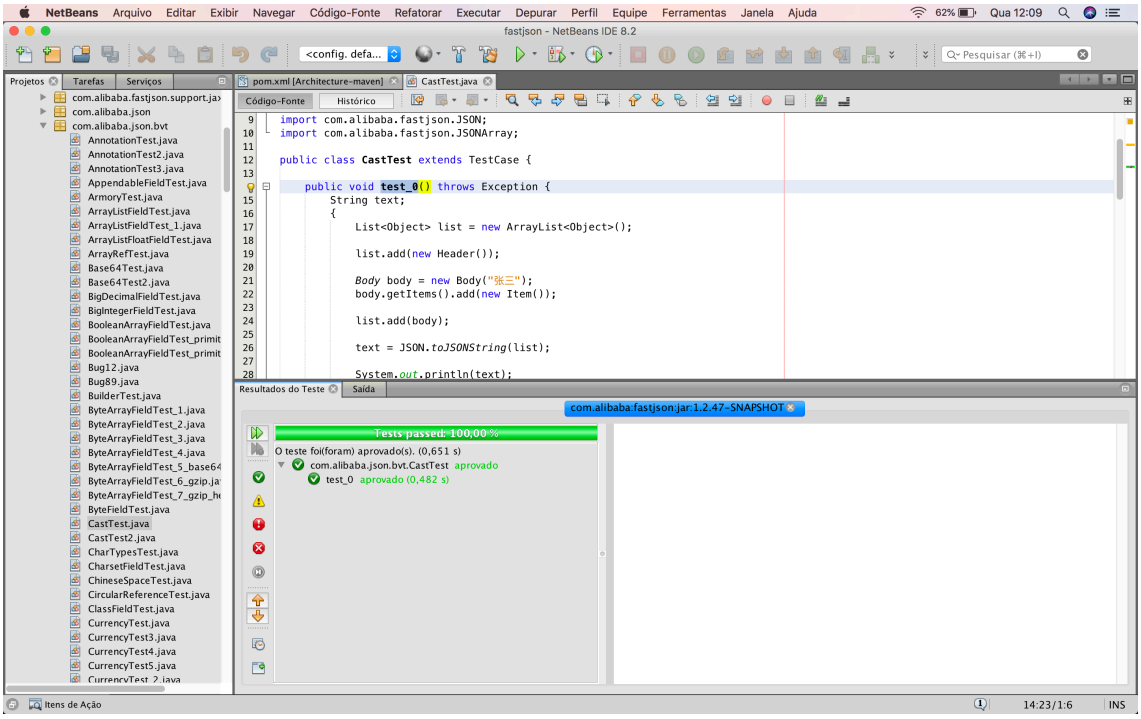
Os verdadeiros positivos são amostras que obtêm como resultado um valor positivo, no caso da infraestrutura, que o teste passará, e seu resultado é positivo, realmente passou. Como exemplo tem-se a Figura 5-6, que mostra a previsão do caso de teste “com.alibaba.json.bvt.CastTest.test_0” sendo positiva (passará) e a Figura 5-7 é a execução do caso de teste pela IDE Netbeans, evidenciando o resultado positivo (passou).

Figura 5-6. Exemplo de Verdadeiro Positivo pela Infraestrutura



Fonte. Elaborado pelo próprio autor

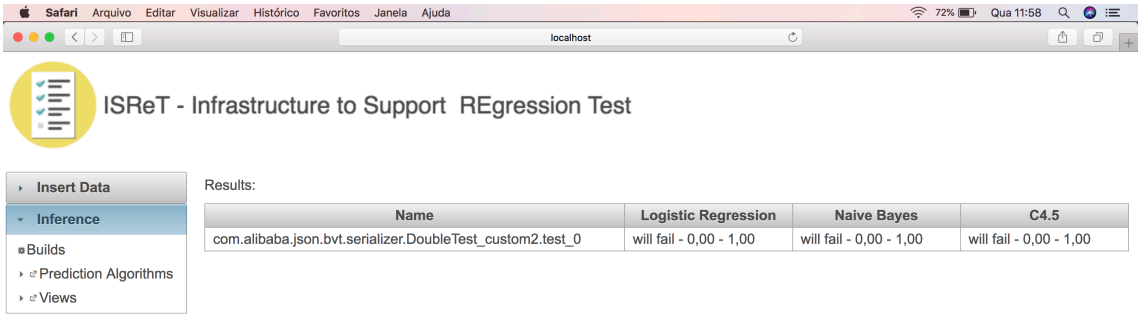
Figura 5-7. Exemplo de Verdadeiro Positivo pelo Netbeans



Fonte. Elaborado pelo próprio autor

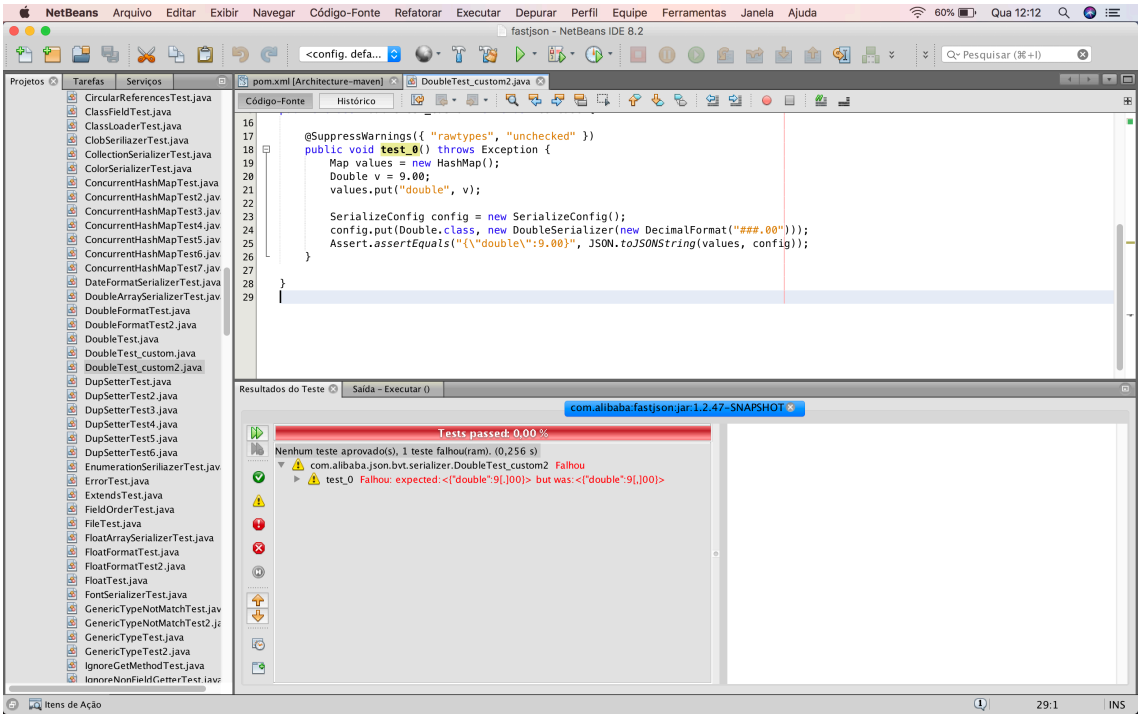
Os verdadeiros negativos que são aquelas amostras que obtêm como resultado um valor negativo, no caso da infraestrutura, que o teste falhará, e seu resultado é negativo, realmente falhou. A Figura 5-8, mostra a previsão do caso de teste "com.alibaba.json.bvt.serializer.DubleTest_custom2.test_0" sendo negativa (falhará) e a Figura 5-9 ilustra a execução do caso de teste pela IDE Netbeans, evidenciando o resultado negativo (falhou).

Figura 5-8. Exemplo de Verdadeiro Negativo pela Infraestrutura



Fonte. Elaborado pelo próprio autor

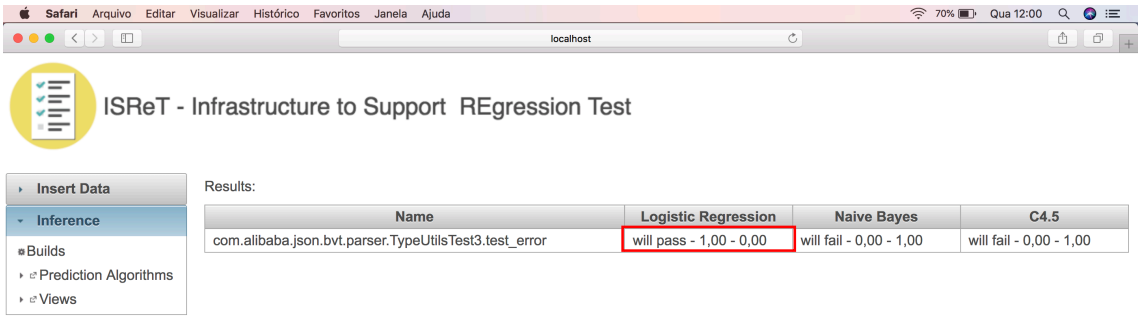
Figura 5-9. Exemplo de Verdadeiro Negativo pelo Netbeans



Fonte. Elaborado pelo próprio autor

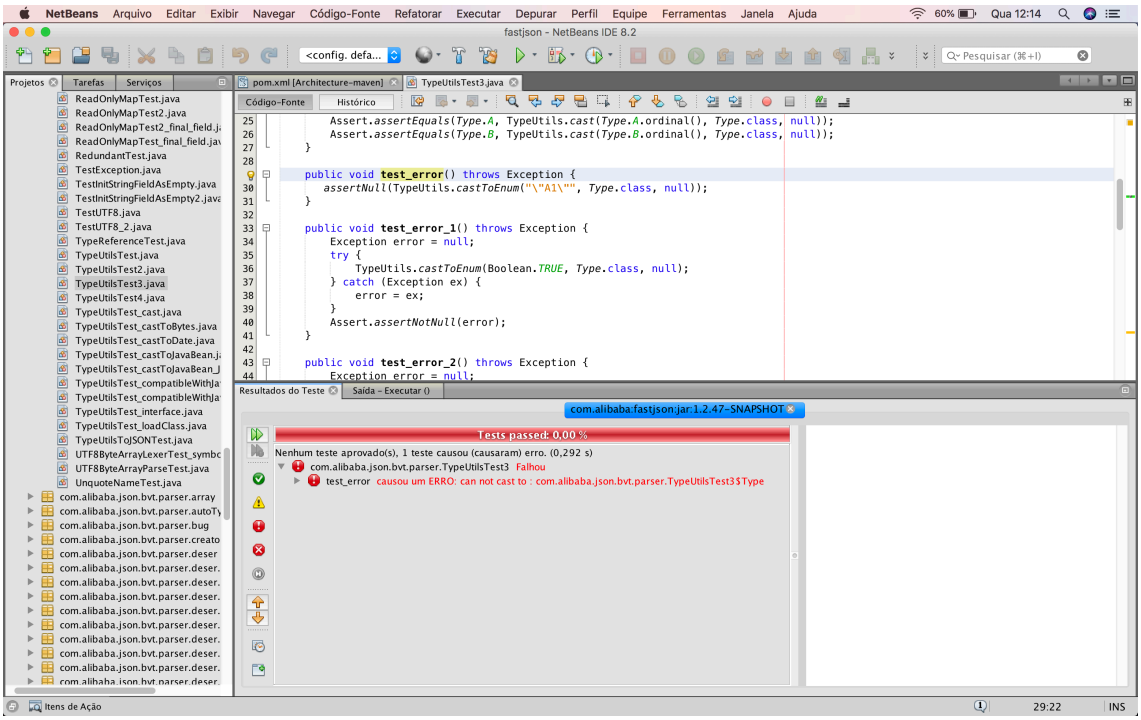
Existem ainda, os falsos positivos que são aquelas amostras que obtêm como resultado um valor positivo, no caso da infraestrutura, que o teste passará, entretanto, seu resultado é negativo, realmente falhou. Como exemplo, tem-se a Figura 5-10 que mostra a previsão do caso de teste "com.alibaba.json.bvt.parser.TypeUtils3..test_error" sendo positiva (passará) e a Figura 5-11 é a execução do caso de teste pela IDE Netbeans, evidenciando o resultado negativo (falhou).

Figura 5-10. Exemplo de Falso Positivo pela Infraestrutura



Fonte. Elaborado pelo próprio autor

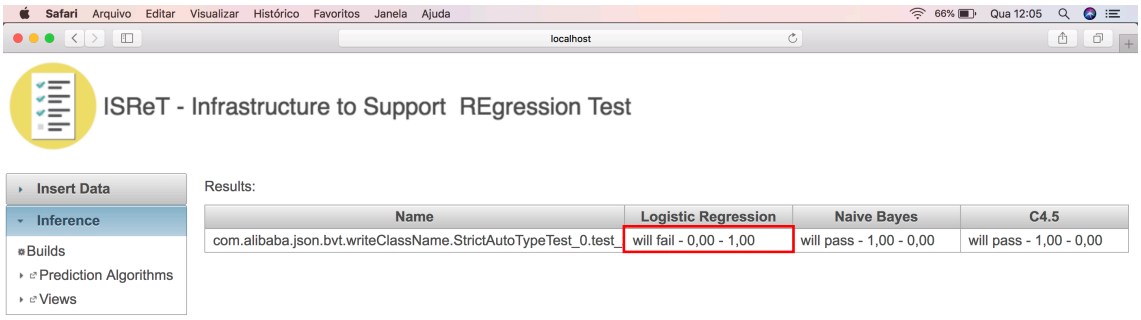
Figura 5-11. Exemplo de Falso Positivo pelo Netbeans



Fonte. Elaborado pelo próprio autor

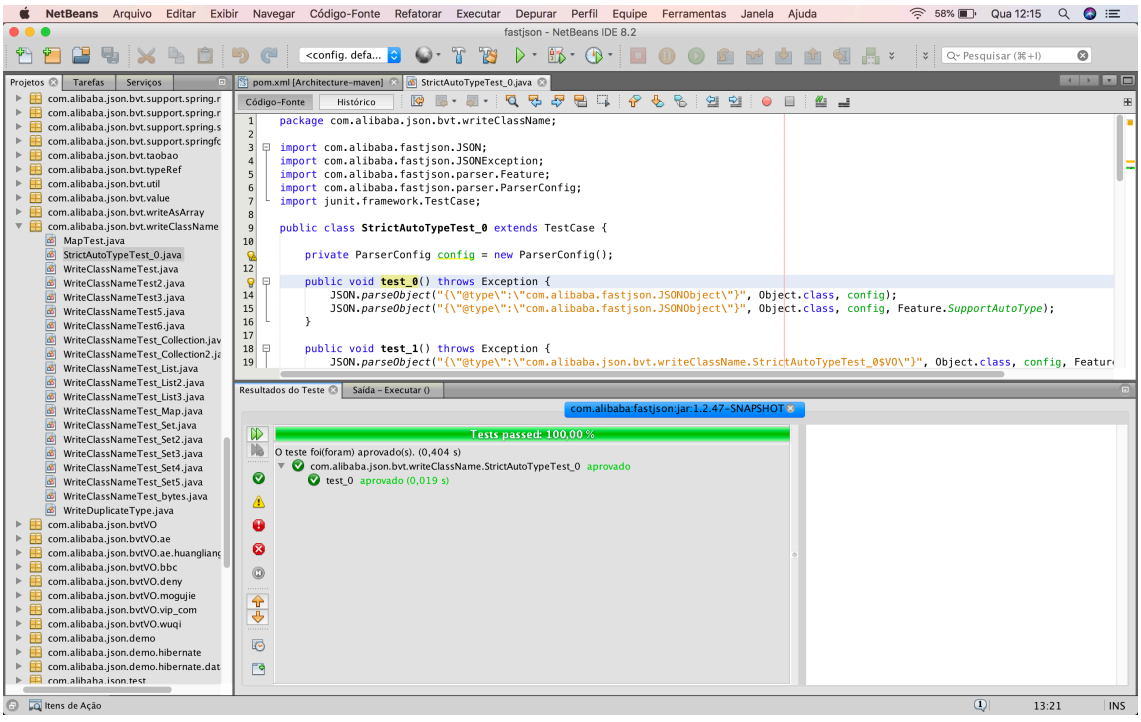
Os falsos negativos são aquelas amostras que obtêm como resultado um valor negativo, no caso da infraestrutura, que o teste falhará, entretanto, seu resultado é positivo, realmente passou. Como exemplo, tem-se a Figura 5-12 que mostra a previsão do caso de teste sendo negativa (falhará) e a Figura 5-13 é a execução do caso de teste pela IDE Netbeans, evidenciando o resultado positivo (passou).

Figura 5-12. Exemplo de Falso Negativo pela Infraestrutura



Fonte. Elaborado pelo próprio autor

Figura 5-13. Exemplo de Falso Negativo pelo Netbeans



Fonte. Elaborado pelo próprio autor

A Tabela 5.5 apresenta uma parte dos resultados, previstos e reais, do primeiro estudo experimental realizado para alguns testes de unidade selecionados do projeto em avaliação de forma aleatória. De forma resumida, os resultados serão apresentados, entretanto, o resultado por completo pode ser encontrado em

https://github.com/CamilaAcacio/ISReT.git. Aqueles que divergiram do resultado real foram destacados, chamam-se esses resultados de: falsos positivos ou falsos negativos.

Tabela 5.5. Parte dos Resultados obtidos – Experimento 1

Test Name	Logistic Regression	Naive Bayes	C45	Real Result
CurrencyTest5.test_0	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	fail
Issue1582.test_for_issue_1	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	fail
TypeUtilsTest3.test_error	will fail - 0,00 - 1,00	will pass - 0,99 - 0,01	will fail - 0,00 - 1,00	fail
DoubleFormatTest.test_format	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	fail
DoubleFormatTest2.test_format	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	fail
DoubleTest_custom.test_0	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	fail
DoubleTest_custom.test_1	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	fail
DoubleTest_custom2.test_0	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	fail
FloatFormatTest.test_format	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	fail
FloatFormatTest2.test_format	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	fail
WriteNonStringValueAsStringTestFloatField2.test_0	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	will fail - 0,00 - 1,00	fail
AnnotationTest.test_codec	will pass - 1,00 - 0,00	will pass - 1,00 - 0,00	will pass - 1,00 - 0,00	passed
AnnotationTest2.test_codec	will pass - 1,00 - 0,00	will pass - 1,00 - 0,00	will pass - 1,00 - 0,00	passed
AnnotationTest3.test_supperField	will pass - 1,00 - 0,00	will pass - 1,00 - 0,00	will pass - 1,00 - 0,00	passed
AppendableFieldTest.test_codec_null	will pass - 1,00 - 0,00	will pass - 1,00 - 0,00	will pass - 1,00 - 0,00	passed
AppendableFieldTest.test_codec_null_1	will pass - 1,00 - 0,00	will pass - 1,00 - 0,00	will pass - 1,00 - 0,00	passed
ArmoryTest.test_item	will pass - 1,00 - 0,00	will pass - 1,00 - 0,00	will pass - 1,00 - 0,00	passed

Fonte. Elaborado pelo próprio autor

Para o estudo experimental 1, foram utilizados 4229 casos de testes. Além disso, para treinamento e teste da infraestrutura foram utilizados os *builds* de 2 a 20 e para validação o *build* de número 21. Os resultados são apresentados nas Tabelas 5.6, 5.7 e 5.8. Para o algoritmo *Logistic Regression*, Tabela 5.6, 100% dos testes foram classificados de forma correta. Para o *Naive Bayes*, Tabela 5.7, 99,97% dos testes foram classificados de forma correta, apenas um deles foi classificado como falso positivo. Por fim, para o algoritmo C45, Tabela 5.8, 100% dos testes foram classificados de forma correta.

Tabela 5.6. Resultados para o Algoritmos Logistic Regression - Experimento 1

Logistic Regression		
	Positivos	Negativos
Verdadeiros	4218	11
Falsos	0	0

Fonte. Elaborado pelo próprio autor

Tabela 5.8. Resultados para o Algoritmos C45 - Experimento 1

C45		
	Positivos	Negativos
Verdadeiros	4218	11
Falsos	0	0

Fonte. Elaborado pelo próprio autor

Tabela 5.7. Resultados para o Algoritmos Naive Bayes- Experimento 1

Naive Bayes		
	Positivos	Negativos
Verdadeiros	4218	10
Falsos	1	0

Fonte. Elaborado pelo próprio autor

No segundo estudo foram utilizados 4229 casos de testes, a mesma quantidade do estudo anterior, já que não foram realizadas alterações que impactassem na quantidade de casos de teste. Os *builds* de 2 a 21 serviram para treinamento e teste da infraestrutura e, para validação, o *build* de número 25. Foram obtidos os resultados apresentados nas. Para *Logistic Regression*, *Tabela 5.9*, 100% dos testes foram classificados de forma correta. Já para o algoritmo *Naive Bayes, Tabela 5.10,* 99,97% dos testes foram classificados de forma correta, apenas um deles foi classificado como

falso positivo. Por fim, para C45, Tabela 5.11, 100% dos testes foi classificado de forma correta.

Tabela 5.9. Resultados para o Algoritmos Logistic Regression - Experimento 2

Logistic Regression		
	Positivos	Negativos
Verdadeiros	4218	11
Falsos	0	0

Fonte. Elaborado pelo próprio autor

Tabela 5.10. Resultados para o Algoritmos Naive Bayes- Experimento 2

Naive Bayes		
	Positivos	Negativos
Verdadeiros	4218	10
Falsos	1	0

Fonte. Elaborado pelo próprio autor

Tabela 5.11. Resultados para o Algoritmos C45 - Experimento 2

C45		
	Positivos	Negativos
Verdadeiros	4218	11
Falsos	0	0

Fonte. Elaborado pelo próprio autor

As Tabelas 5.12, 5.13, 5.14, apresentam o resultado do terceiro estudo experimental. Para esse estudo, foram utilizados 4229 casos de teste a mesma quantidade do estudo anterior, já que não foram realizadas alterações que impactassem na quantidade de casos de teste. Para treinamento e teste da infraestrutura utilizou-se os *builds* de 2 a 21, 25 e para validação o *build* de número 26. O algoritmo *Logistic Regression*, Tabela 5.12, obteve um percentual de 99,97% de acerto, apenas um deles foi classificado como falso positivo. Para o *Naive Bayes*, Tabela 5.13, 100% dos testes foram classificados de forma correta. Por fim, para o algoritmo C45, Tabela 5.14, 100% dos testes foram classificados de forma correta.

Tabela 5.12. Resultados para o Algoritmos Logistic Regression - Experimento 3

Logistic Regression		
	Positivos	Negativos
Verdadeiros	4218	10
Falsos	1	0

Fonte. Elaborado pelo próprio autor

Tabela 5.13. Resultados para o Algoritmos Naive Bayes - Experimento 3

Naive Bayes		
	Positivos	Negativos
Verdadeiros	4218	11
Falsos	0	0

Fonte. Elaborado pelo próprio autor

Tabela 5.14. Resultados para o Algoritmos C45 - Experimento 3

C45		
	Positivos	Negativos
Verdadeiros	4218	11
Falsos	0	0

Fonte. Elaborado pelo próprio autor

Para o Experimento 4, 4231 casos de teste foram analisados. Os *builds* de número 2 a 21, 25, 26 foram utilizados para a treinamento e teste da infraestrutura e o *build* de número 28 para a validação. Os resultados obtidos são apresentados na Tabela 5.15, Tabela 5.16 e Tabela 5.17. Para o algoritmo *Logistic Regression*, Tabela 5.15, 99,95% dos testes foram classificados de forma correta. Dois deles foram classificados de forma incorreta: um foi classificado como falso positivo e um como falso negativo. Para o *Naive Bayes*, Tabela 5.16, 99,97% dos testes foram classificados de forma correta, apenas um deles foi classificado como falso positivo. Por fim, para o algoritmo C45, Tabela 5.17, 100% dos testes foram classificados de forma correta.

Tabela 5.15. Resultados para o Algoritmos Logistic Regression - Experimento 4

Logistic Regression		
	Positivos	Negativos
Verdadeiros	4219	10
Falsos	1	1

Fonte. Elaborado pelo próprio autor

Tabela 5.16. Resultados para o Algoritmos Naive Bayes - Experimento 4

Naive Bayes		
	Positivos	Negativos
Verdadeiros	4220	10
Falsos	1	0

Fonte. Elaborado pelo próprio autor

Tabela 5.17. Resultados para o Algoritmos C45 - Experimento 4

C45		
	Positivos	Negativos
Verdadeiros	4220	11
Falsos	0	0

Fonte. Elaborado pelo próprio autor

No quinto experimento foram utilizados 4234 casos de testes. Os *builds* de 2 a 21, 25, 26 e 28 serviram para treinamento e teste da infraestrutura e, para validação, o *build* de número 30. Foram obtidos os resultados apresentados na Tabela 5.18, Tabela 5.19 e Tabela 5.20. Para o algoritmo *Logistic Regression*, Tabela 5.18 , 99,97% dos testes foram classificados de forma correta. Apenas um dos testes foi classificado de forma incorreta, como falso negativo. Já para *Naive Bayes* e C45, Tabela 5.19 e Tabela 5.20, respectivamente, 100% dos testes foram classificados de forma correta.

Tabela 5.18.Resultados para o Algoritmos Logistic Regression - Experimento 5

Logistic Regression		
	Positivos	Negativos
Verdadeiros	4222	11
Falsos	0	1

Fonte. Elaborado pelo próprio autor

Tabela 5.19.Resultados para o Algoritmos Naive Bayes - Experimento 5

Naive Bayes		
	Positivos	Negativos
Verdadeiros	4223	11
Falsos	0	0

Fonte. Elaborado pelo próprio

Tabela 5.20.Resultados para o Algoritmos C45 - Experimento 5

C45		
	Positivos	Negativos
Verdadeiros	4223	11
Falsos	0	0

Fonte. Elaborado pelo próprio autor

As Tabela 5.21, Tabela 5.22 e Tabela 5.23, apresentam o resultado do sexto estudo experimental. Para tal foram utilizados 4235 casos de teste. Como treinamento e

teste da infraestrutura foram utilizados os *builds* de 2 a 21, 25, 26, 28, 30 e, para validação, o *build* de número 31. O algoritmo *Logistic Regression*, Tabela 5.21, obteve um percentual de 99,97% de acerto, apena um deles foi classificado como falso positivo. Para o *Naive Bayes* e C45, Tabela 5.22 e Tabela 5.23, 100% dos testes foram consistentes.

Tabela 5.21. Resultados para o Algoritmos Logistic Regression - Experimento 6

Logistic Regression		
	Positivos	Negativos
Verdadeiros	4224	10
Falsos	1	0

Fonte. Elaborado pelo próprio autor

Tabela 5.22. Resultados para o Algoritmos Naive Bayes- Experimento 6

Naive Bayes		
	Positivos	Negativos
Verdadeiros	4224	11
Falsos	0	0

Fonte. Elaborado pelo próprio autor

Tabela 5.23. Resultados para o Algoritmos C45 - Experimento 6

C45		
	Positivos	Negativos
Verdadeiros	4224	11
Falsos	0	0

Fonte. Elaborado pelo próprio autor

No sétimo experimento foram utilizados 4237 casos de testes. Os *builds* de 2 a 21, 25, 26, 28 e 30 serviram para treinamento e teste da infraestrutura e, para validação, o *build* de número 31. Foram obtidos os resultados apresentados nas Tabela 5.24, Tabela 5.25 e Tabela 5.26. Para os três algoritmos, *Logistic Regression, Naive Bayes e C45*, 100% dos testes foram condizentes.

Tabela 5.24. Resultados para o Algoritmos Logistic Regression - Experimento 7

Logistic Regression		
	Positivos	Negativos
Verdadeiros	4226	11
Falsos	0	0

Fonte. Elaborado pelo próprio autor

Tabela 5.25. Resultados para o Algoritmos Naive Bayes - Experimento 7

Naive Bayes		
	Positivos	Negativos
Verdadeiros	4226	11
Falsos	0	0

Fonte. Elaborado pelo próprio autor

Tabela 5.26. Resultados para o Algoritmos C45 - Experimento 7

C45		
	Positivos	Negativos
Verdadeiros	4226	11
Falsos	0	0

Fonte. Elaborado pelo próprio autor

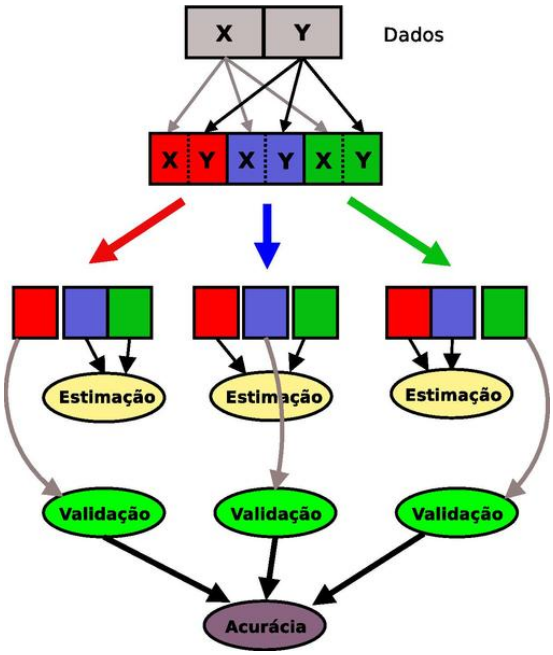
Após a realização dos estudos experimentais, pode-se observar que, de uma forma geral, foram obtidos bons resultados no contexto do estudo experimental. A acurácia dos modelos teve variações entre 99,95% e 100%. Entretanto, em métodos estatísticos existe um fenômeno chamado *overfitting* (HAWKINS, 2004), em que um modelo estatístico se ajusta muito bem ao conjunto de dados utilizado como treinamento e teste, mas se mostra ineficaz para prever novos resultados. Um modelo ajustado apresenta alta precisão quando testado com seu conjunto de dados, porém tal modelo não é uma boa representação da realidade, e por isso deve ser evitado.

Com o objetivo de averiguar se resultados obtidos eram um *overfitting*, foi realizado um novo experimento no qual, através da validação cruzada, testou-se os modelos. A validação cruzada (REFAEILZADEH; TANG; LIU, 2009) é uma técnica para avaliar a capacidade de generalização de um modelo. Busca-se estimar o quão preciso é o desempenho de um modelo para um novo conjunto de dados.

Para o oitavo experimento foi utilizado o método denominado K-Fold (REFAEILZADEH; TANG; LIU, 2009). Esse método consiste em dividir o conjunto de dados em K subconjuntos mutuamente exclusivos do mesmo tamanho e, a partir disso, um subconjunto é utilizado para teste e os K-1 restantes são utilizados para estimação dos parâmetros e calcula-se a acurácia do modelo. Esse processo é realizado K vezes alternando o subconjunto de teste. Ao final das K iterações calcula-se a acurácia sobre

os erros encontrados, obtendo assim uma medida mais confiável sobre a capacidade do modelo. A Figura 5-14 é um exemplo de como a validação cruzada pelo método K-Fold acontece.

Figura 5-14. Exemplificação de validação cruzada utilizando K=3

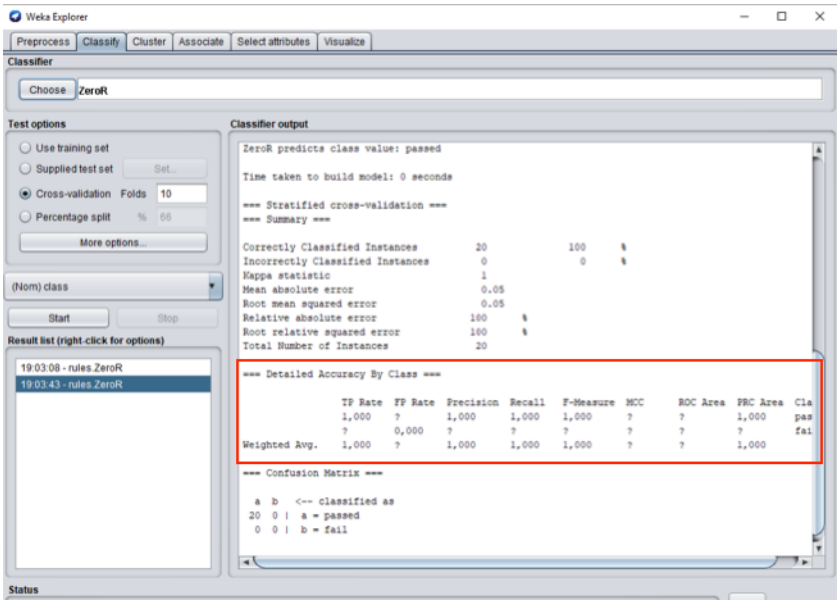


Como variáveis desse experimento foram utilizados os dados referentes ao Experimento 1: 4229 casos de testes e os dados referentes aos *build*s de 2 a 20. Para o método K-Fold, adotou-se como parâmetro, K=10, pois, segundo (REFAEILZADEH; TANG; LIU, 2009), esse valor de K é particularmente atraente porque faz previsões usando 90% dos dados, tornando mais provável que seja generalizável para os dados completos.

Como resultados, exemplificados pela Figura 5-15, obteve-se para o algoritmo *Logistic Regression*, precisões entre 77,8% e 100%. Para o algoritmo C45, entre 86,7% e 100% e, por fim, para Naive Bayes entre 77,8% e 100%. A partir desses valores constata-se que os resultados encontrados anteriormente, nos 7 estudos experimentais, não se caracterizam um *overfitting*, uma vez que os resultados obtidos na validação cruzada não foram baixos quando comparados com os resultados anteriores. Para a realização dessa averiguação foi utilizada uma ferramenta chamada Weka[11], a qual engloba vários algoritmos de aprendizado de máquina de forma gratuita.

[11] https://www.cs.waikato.ac.nz/~ml/weka

Figura 5-15. Exemplo de resultado para validação cruzada



Fonte. Elaborado pelo próprio autor

Outro fenômeno ocorrente em *machine learning* são conjuntos de dados desbalanceados. De acordo com (GALAR *et al.*, 2012) o problema de conjuntos de dados desbalanceados ocorre quando uma classe está sub representada no conjunto de dados; em outras palavras, o número de instâncias de uma classe ultrapassa a quantidade de instâncias de outra classe.

Nos Estudos experimentais realizados, houve classes desbalanceadas, portanto, há métricas mais apropriadas a serem consideradas em vez de precisão (GALAR *et al.*, 2012), são elas:

True positive rate/Taxa de Verdadeiros Positivos (TP_{rate}): percentual de instâncias positivas corretamente classificadas. Calculado através de: $\frac{TP}{TP+FN}$

True negative rate/Taxa de Verdadeiros Negativos (TN_{rate}): percentual de instâncias negativas corretamente classificadas. Calculado a partir: $\frac{TN}{FP+TN}$

False positive rate/Taxa de Falsos Positivos (FP_{rate}): percentual de instâncias positivas incorretamente classificadas. Calculado através de: $\frac{FP}{FP+TN}$

False negative rate/Taxa de Falsos Negativos (FN_{rate}): percentual de instâncias negativas incorretamente classificadas. Calculado a partir: $\frac{FN}{TP+FN}$

Para os estudos experimentais 1 e 2, os resultados podem ser observados na Tabela 5.27. Para os algoritmos *Logistic Regression* e C45 todas as instâncias positivas

e negativas foram classificadas corretamente. Para o algoritmo *Naive Bayes*, 90,90% das instâncias negativas foram classificadas de forma correta.

Tabela 5.27. Resultados para os estudos experimentais 1 e 2

	LG	NB	C45
TP_{rate}	100%	100%	100%
TN_{rate}	100%	90,90%	100%
FP_{rate}	0%	9,09%	0%
FN_{rate}	0%	0%	0%

Fonte. Elaborado pelo próprio autor

A Tabela 5.28 apresenta os resultados para os estudos experimentais 3 e 6. Para os algoritmos *Naive Bayes* e C45 todas as instâncias positivas e negativas foram classificadas corretamente. Para o *Logistic Regression*, 90,90% das instâncias negativas foram classificadas de forma correta.

Tabela 5.28. Resultados para os estudos experimentais 3 e 6

	LG	NB	C45
TP_{rate}	100%	100%	100%
TN_{rate}	90,90%	100%	100%
FP_{rate}	9,09%	0%	0%
FN_{rate}	0%	0%	0%

Fonte. Elaborado pelo próprio autor

Para o experimento 4 foram obtidos os resultados apresentados na Tabela 5.29. Os algoritmos três algoritmos obtiveram 100% das instâncias positivas classificadas corretamente. Entretanto, *Logistic Regression e Naive Bayes* obtiveram acertos de 90,90% para as instâncias negativas.

Tabela 5.29. Resultados para os estudos experimentais 4

	LG	NB	C45
TP_{rate}	100%	100%	100%
TN_{rate}	90,90%	90,90%	100%
FP_{rate}	9,09%	9,09%	0%
FN_{rate}	0%	0%	0%

Fonte. Elaborado pelo próprio autor

A Tabela 5.30 apresenta os resultados para o experimento 5. Para os algoritmos *Naive Bayes* e C45 todas as instâncias positivas e negativas foram classificadas corretamente. Para o *Logistic Regression*, 99,97% das instâncias negativas foram classificadas de forma correta.

Tabela 5.30. Resultados para os estudos experimentais 5

	LG	NB	C45
TP_{rate}	99,97%	100%	100%
TN_{rate}	100%	100 %	100%
FP_{rate}	0%	0%	0%
FN_{rate}	0,03%	0%	0%

Fonte. Elaborado pelo próprio autor

Para o experimento 7 foram obtidos os resultados apresentados na Tabela 5.31. Os três algoritmos obtiveram 100% das instâncias positivas classificadas corretamente. E, para as instâncias negativas, 100% foram classificadas corretamente.

Tabela 5.31. Resultados para os estudos experimentais 7

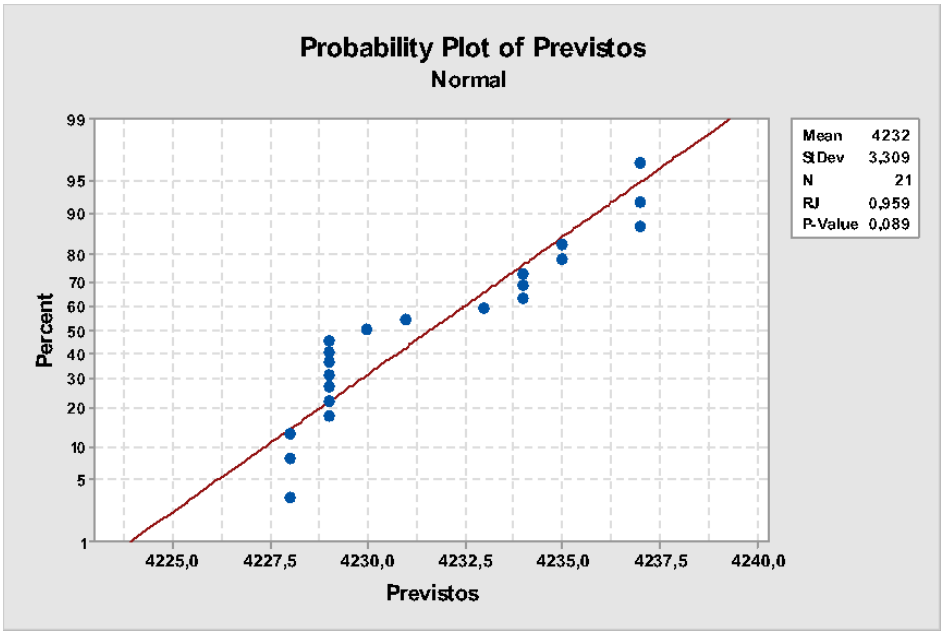
	LG	NB	C45
TP_{rate}	100%	100%	100%
TN_{rate}	100%	100 %	100%
FP_{rate}	0%	0%	0%
FN_{rate}	0%	0%	0%

Fonte. Elaborado pelo próprio autor

A partir dessas análises, observa-se que os modelos obtiveram percentuais assertivos acima de 77,8% para os estudos experimentais realizados. A menor acurácia obtida foi 99,95%, porém, vendo os resultados de um outro ângulo, o mínimo de acertos para as instâncias verdadeiras foi 90,90%. Diante destes resultados, respondendo à questão que dirigiu o estudo, realizou-se análises estatísticas para averiguar se os resultados previstos e reais são iguais.

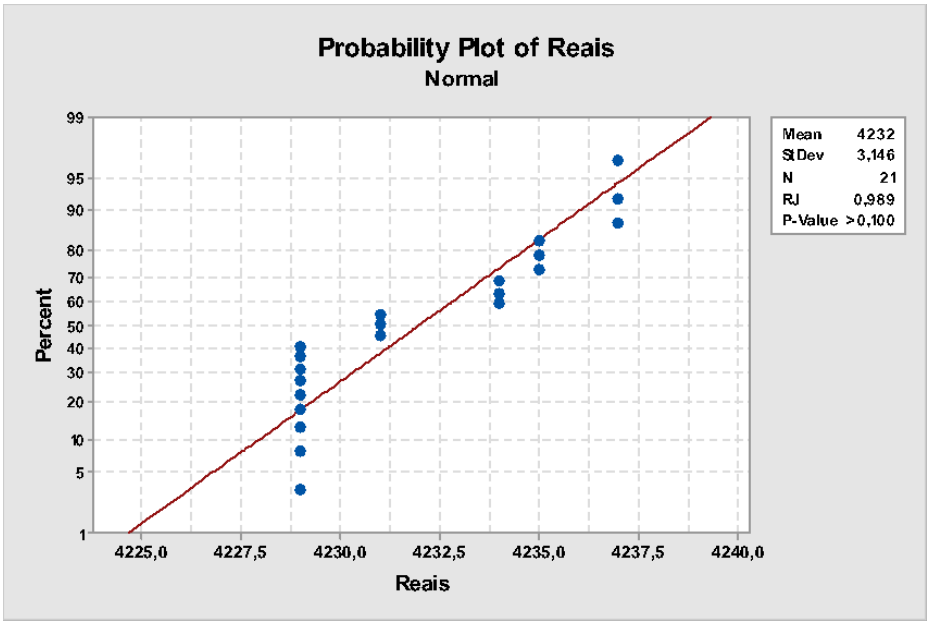
O primeiro teste a ser realizado é verificar a normalidade de cada variável. Como o número de amostras não é superior a 30, foi utilizado o teste Shapiro-Wilk. O gráfico de dispersão para a variável número de testes classificados corretamente pela previsão é observado pela Figura 5-16 e o gráfico de dispersão para a variável número de casos de teste é observado pela Figura 5-17.

Figura 5-16. Gráfico de dispersão para variável número de casos de testes classificados corretamente pela previsão



Fonte. Elaborado pelo próprio autor

Figura 5-17.Gráfico de dispersão para a variável casos de testes

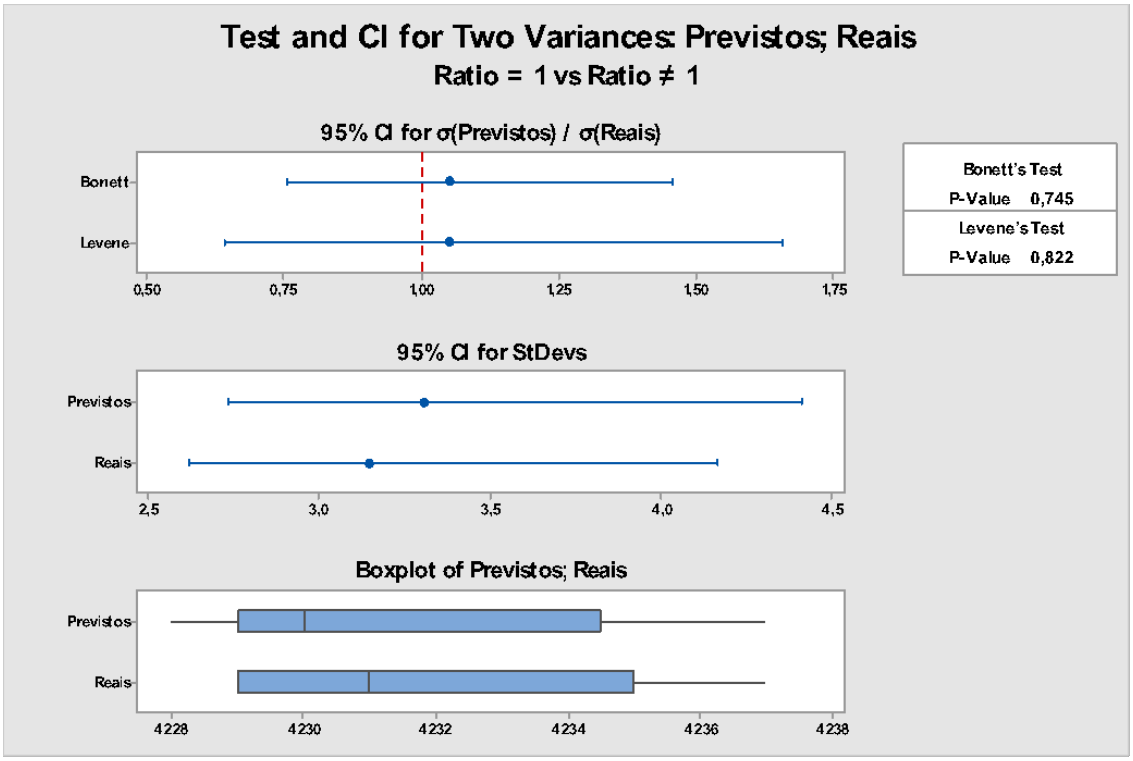


Fonte. Elaborado pelo próprio autor

Através do método de Shapiro-Wilk, pode-se observar que o p-value > 0,089 para cada uma das amostras é maior que o nível de significância estabelecido de 0,05. Então, não há indícios para aceitação da hipótese alternativa, aceitando-se a hipótese nula de que os dados são normais.

Após o teste de normalidade, aplicou-se o método de Levene para verificar a homocedasticidade dos dados. Como observa-se pela Figura 5-18, o p-value = 0,822 é maior que o nível de significância estabelecido de 0,05, desta forma os dados são homocedásticos.

Figura 5-18. Homocedasticidade dos dados



Fonte. Elaborado pelo próprio autor

Como os dados utilizados são normais e homocedásticos, foi utilizado o teste paramétrico Test-T. Como mostra a Figura 5-19, pode-se observar que o p-value = 0,704 é maior que o nível de significância estabelecido de 0,05. Então, não há indícios para aceitação da hipótese alternativa, aceitando-se a hipótese nula de que as médias são iguais, ou seja, não existe diferença entre as previsões realizadas pela infraestrutura e os dados reais.

Figura 5-19. Resultado Test-T

Two-Sample T-Test and CI: Previstos; Reais

Two-sample T for Previstos vs Reais

	N	Mean	StDev	SE Mean
Previstos	21	4231,62	3,31	0,72
Reais	21	4232,00	3,15	0,69

Difference = μ (Previstos) - μ (Reais)
Estimate for difference: -0,381
95% CI for difference: (-2,396; 1,634)
T-Test of difference = 0 (vs ≠): T-Value = -0,38 **P-Value = 0,704**
DF = 3

Fonte. Elaborado pelo próprio autor

A partir desta questão de pesquisa, derivou-se uma questão secundária, que objetiva identificar quais dos três algoritmos realizou as classificações de maneira mais eficaz. Diante dos resultados obtidos sumarizados na Tabela 5.32 e Tabela 5.33, observa-se que o algoritmo C45 obteve melhores classificações com relação aos demais algoritmos. Segundo GALAR *et al.* (2012) o algoritmo C45 é amplamente utilizado em casos de dados desbalanceados e apresenta melhores resultados: (BATISTA; PRATI; MONARD, 2004), (SU; HSIAO, 2007), (GARCIA; FERNÁNDEZ; HERRERA, 2009), (DROWN; KHOSHGOFTAAR; SELIYA, 2009). Contudo, foi realizada uma análise estatística para verificar o resultado.

Tabela 5.32. Resumo dos Resultados

	Logistic Regression	*Naive Bayes*	*C45*
Experimento 1	100%	99,97%	100%
Experimento 2	100%	99,97%	100%
Experimento 3	99,97%	100%	100%
Experimento 4	99,95%	99,97%	100%
Experimento 5	99,97%	100%	100%
Experimento 6	99,95%	100%	100%
Experimento 7	100%	100%	100%

Fonte: Elaborado pelo próprio autor

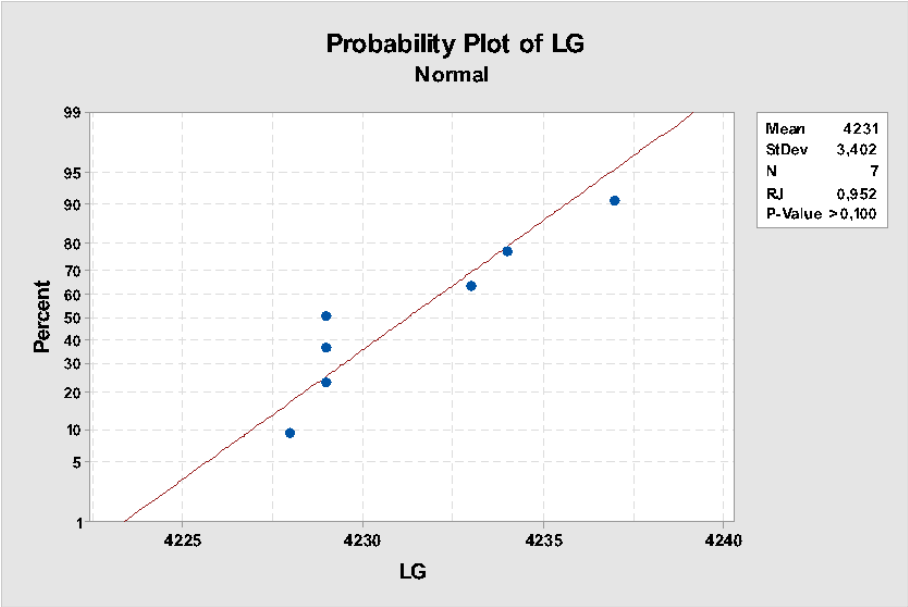
Tabela 5.33. Resumo dos Resultados 2

		Logistic Regression	*Naive Bayes*	*C45*
Experimento 1	TP_{rate}	100%	100%	100%
	TN_{rate}	100%	90,90%	100%
Experimento 2	TP_{rate}	100%	100%	100%
	TN_{rate}	100%	90,90%	100%
Experimento 3	TP_{rate}	100%	100%	100%
	TN_{rate}	90,90%	100%	100%
Experimento 4	TP_{rate}	100%	100%	100%
	TN_{rate}	90,90%	90,90%	100%
Experimento 5	TP_{rate}	99,97%	100%	100%
	TN_{rate}	100%	100 %	100%
Experimento 6	TP_{rate}	100%	100%	100%
	TN_{rate}	90,90%	100%	100%
Experimento 7	TP_{rate}	100%	100%	100%
	TN_{rate}	100%	100 %	100%

Fonte: Elaborado pelo próprio autor

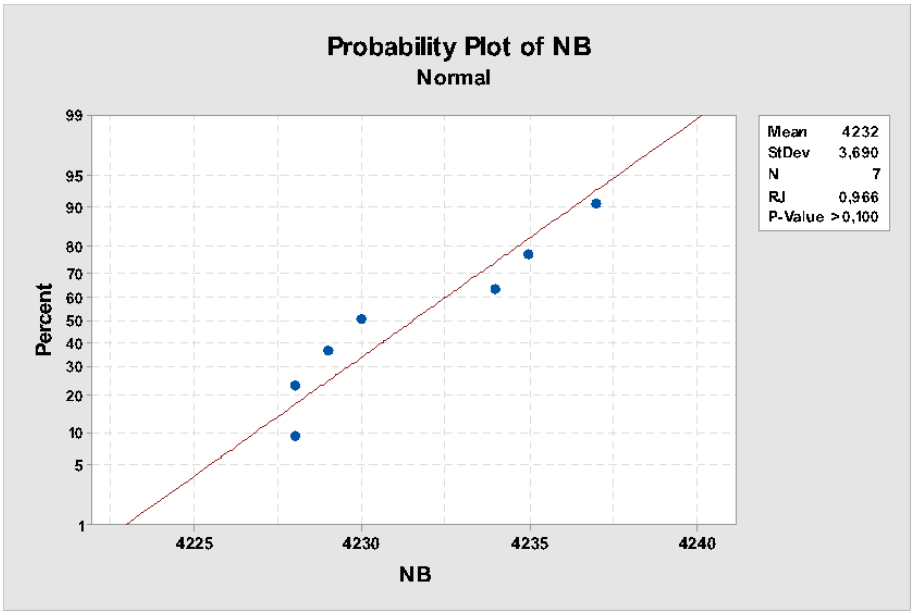
O primeiro teste a ser realizado é verificar a normalidade de cada variável. Como o número de amostras não é superior a 30, foi utilizado o teste Shapiro-Wilk. O gráfico de dispersão para a variável número de testes classificados corretamente pelo algoritmo *Logistic Regression* é ilustrado pela Figura 5-20. Para a variável número de testes classificados corretamente pelo algoritmo *Naive Bayes* é observado pela Figura 5-21, o gráfico de dispersão para a variável número de testes classificados corretamente pelo algoritmo *C45* é ilustrado pela Figura 5-22, e o gráfico de dispersão para a variável número de testes classificados corretamente é ilustrado pela Figura 5-23.

Figura 5-20. Gráfico de dispersão para a variável número de testes classificados corretamente pelo algoritmo Logistic Regression



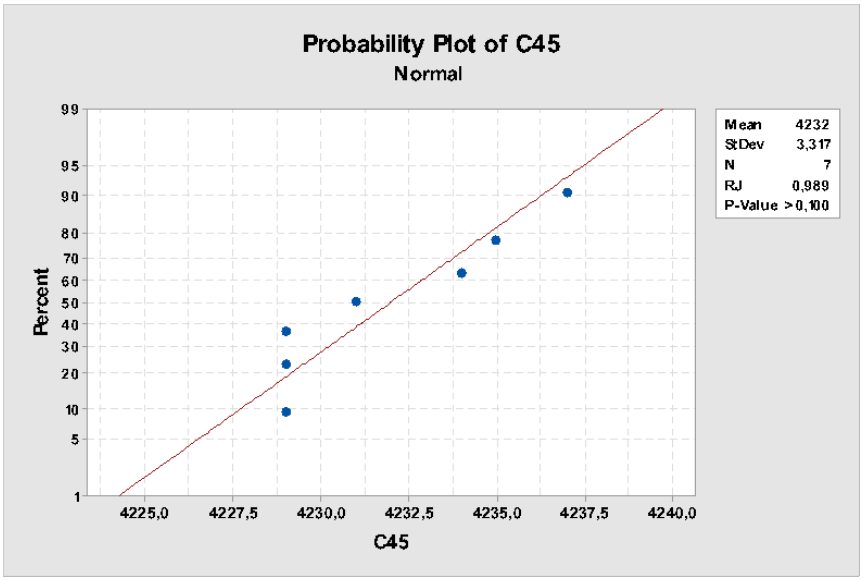
Fonte. Elaborado pelo próprio autor

Figura 5-21. Gráfico de dispersão para a variável número de testes classificados corretamente pelo algoritmo Naive Bayes



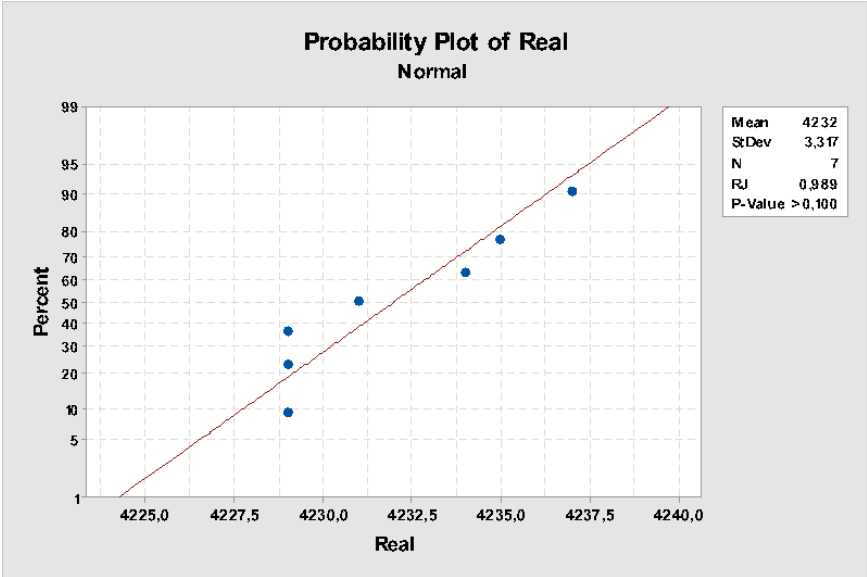
Fonte. Elaborado pelo próprio autor

Figura 5-22. Gráfico de dispersão para a variável número de testes classificados corretamente pelo algoritmo C45



Fonte. Elaborado pelo próprio autor

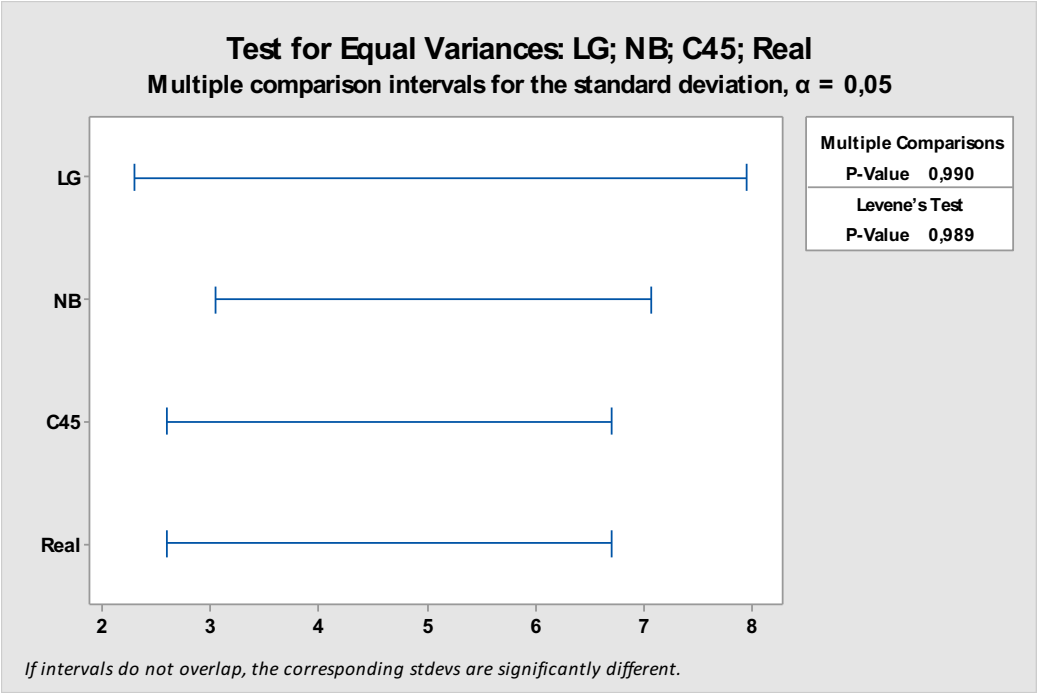
Figura 5-23. Gráfico de dispersão para a variável número de testes classificados corretamente



Fonte. Elaborado pelo próprio autor

Após o teste de normalidade, aplicou-se o método de Levene para verificar a homocedasticidade dos dados. Como observa-se pela Figura 5-24, o p-value = 0,989 é maior que o nível de significância estabelecido de 0,05. Desta forma os dados são homocedásticos.

Figura 5-24. Homocedasticidade dos dados para Logistic Regression



Fonte. Elaborado pelo próprio autor

Como os dados utilizados são normais e homocedásticos, foi utilizado o teste paramétrico ANOVA. Como mostra a Figura 5-25, pode-se observar que o p-value = 0,974 é maior que o nível de significância estabelecido de 0,05. Então, não há indícios para aceitação da hipótese alternativa, aceitando-se a hipótese nula de que as médias são iguais, ou seja, não existe diferença entre as previsões realizadas entres os três algoritmos.

Figura 5-25. Resultado Anova

One-way ANOVA: LG; NB; C45; Real

Method

Null hypothesis All means are equal

Alternative hypothesis At least one mean is different

Significance level $\alpha = 0,05$

Equal variances were assumed for the analysis.

Factor Information

Factor	Levels	Values
Factor	4	LG; NB; C45; Real

Analysis of Variance

Source	DF	Adj SS	Adj MS	F-Value	**P-Value**
Factor	3	2,571	0,8571	0,07	**0,974**
Error	24	283,143	11,7976		
Total	27	285,714			

Fonte. Elaborado pelo próprio autor

Os resultados obtidos evidenciam a viabilidade da infraestrutura proposta com relação às previsões realizadas e os resultados reais. Acredita-se que, no contexto de um cenário de desenvolvimento real, tais previsões ajudariam os desenvolvedores a identificar *builds* críticos propensos às falhas, e ter mais atenção a todos os seus componentes, antes que ocorra o *build,* dessa forma, auxiliando no *feedback*. Contudo, análises adicionais devem ser realizadas, pois apenas um projeto foi utilizado para a validação, o que implica em resultados que não podem ser generalizados ou atingidos por outros projetos.

5.5 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Este capítulo apresentou uma avaliação da infraestrutura proposta. Foram abordadas todas as etapas necessárias em um experimento, segundo WOHLIN *et al.* (2000). Ao final, foram realizados 8 estudos experimentais baseados em dados de um projeto real disponível no GitHub.

Os estudos experimentais mostraram a viabilidade da infraestrutura e a acurácia das previsões realizadas. Por fim, foram apresentadas as ameaças à validade dos estudos experimentais, bem como suas limitações.

6 CONSIDERAÇÕES FINAIS E PESPECTIVAS FUTURAS

A atividade de teste é complexa. São diversos fatores que podem contribuir para a ocorrência de erros. Os testes de regressão devem ser executados a cada modificação realizada no sistema visando assegurar que novos defeitos não foram introduzidos e o conjunto continua funcionando como planejado ou como anteriormente à adição.

Entretanto, com o mercado mais exigente e com maiores necessidades em qualidade, rapidez e resultados, os processos foram aperfeiçoados através engenharia de software contínua. Ela se baseia no uso do *feedback* de execuções para alçar uma melhoria contínua e pela realização das atividades de maneira contínua, sendo assim, um grande número de informações são geradas diariamente.

Manter o histórico dos dados pode auxiliar de diversas maneiras, como por exemplo, evitar problemas recorrentes, diferentes técnicas de priorização/seleção de casos de teste podem ser utilizadas para otimizar a execução dos testes de regressão, buscar por melhores configurações, realizar previsões sobre o software. Existem poucos estudos na área que oferecem recursos para os testes de regressão. Dentre as publicações encontradas observou-se a escassez de propostas que permeiam as previsões e o suporte à gerência de testes.

Este trabalho apresentou uma infraestrutura amparado por ontologia e algoritmos de predição, para prever os resultados dos casos de testes sem que eles sejam executados. Essas previsões são realizadas através da coleta de informações históricas e métricas de um projeto.

Essa infraestrutura pode ser uma ferramenta importante no processo de tomada de decisão, auxiliando no planejamento dos testes de regressão, na execução dos *builds*, na otimização da suíte de testes, bem como, na melhoria de futuras execuções.

6.1 CONTRIBUIÇÕES

As principais contribuições deste trabalho são:

- uma revisão sistemática de literatura com o objetivo de analisar algoritmos e modelos utilizados para a predição de falha dos testes ou predição de falha do software no contexto de engenharia de software;

- um mapeamento sistemático cujo objetivo era analisar ontologias relacionadas ao ambiente de testes de software;
- elaboração de uma infraestrutura capaz de capturar, processar e gerar informações relativas aos testes de regressão, a qual implementa as seguintes funcionalidades:
    - captura de informações contidas em relatórios gerados pelo servidor de integração contínua, Jenkins;
    - definição de uma ontologia, *Regression Test Execution Ontology (RTE-Ontology)*, para o enriquecimento das informações de execução dos testes de regressão;
    - disponibilização de três algoritmos, Logistic Regression, Naive Bayes e C45, para realizar as previsões dos resultados dos testes de unidade;
    - disponibilização de um módulo de rastreabilidade, responsável por encontrar os casos de testes associados e os códigos fonte;
    - implementação de uma técnica de otimização baseada no histórico de falhas dos testes de unidade, disponibilizando os resultados através de arquivos XML para que aplicações consigam priorizar, selecionar o minimizar os casos de testes;
    - disponibilização das informações capturadas para que aplicações externas possam acessar e utilizar os dados através de serviços;
- realização de estudos experimentais utilizando a infraestrutura proposta, avaliando as previsões realizadas com os resultados reais.

6.2 LIMITAÇÕES E AMEAÇAS À VALIDADE

Algumas ameaças e limitações foram identificadas. Umas delas provem da quantidade de projetos utilizados para a validação da infraestrutura proposta. Apenas um projeto foi utilizado para a validação, o que implica em resultados que não podem ser generalizados ou atingidos por outros projetos.

Outra ameaça seria a criação de um modelo de cada algoritmo para cada caso de teste. Tal fato implica em modelos com percentuais diferentes, o que pode gerar alguns modelos melhores do que outros.

Por fim, a necessidade de aplicações externas, Jenkins e SonarQube, para se obter os dados que são utilizados para realizar as previsões. Essa necessidade limita o escopo da aplicação.

6.3 TRABALHOS FUTUROS

A realização deste trabalho de pesquisa levou ao desenvolvimento de uma infraestrutura que captura e gerencia informações sobre as execuções dos testes de regressão visando apoiar futuros pontos críticos nos *builds*. Como dito anteriormente, os resultados encontrados nos estudos experimentais realizados evidenciam que as previsões obtiveram percentuais assertivos acima de 77,8%. A menor acurácia obtida foi 99,95%, entretanto, o mínimo de acerto para as instâncias verdadeiras foi 90,90%. Contudo, apenas um projeto foi utilizado nestes estudos experimentais, o que torna os resultados não generalizáveis. Como trabalhos futuros novos estudos devem ser conduzidos para que se avalie a infraestrutura em outros contextos.

Um dos problemas relatados anteriormente são os conjuntos de dados desbalanceados. De acordo com GALAR *et al.* (2012) o problema de conjuntos de dados desbalanceados surgiu como um dos desafios na comunidade de mineração de dados. Esse problema está presente em outros de classificação do mundo real. Devido a este fato, uma grande quantidade de técnicas foi desenvolvida para resolver o problema. Essas técnicas se categorizam em três grupos: *Data Level*, no qual as amostras podem ser duplicadas ou retiradas, diminuindo o desbalanceamento do conjunto, entretanto, a duplicação de amostras pode causar um *overfitting* e a retirada de amostras pode remover amostras importantes para a classificação; *Algorithms Level*: no qual o algoritmo é modificado para acomodar o desbalanceamento; e *Cost-Sensitive*: combina as abordagens anteriores para melhorar as classificações. Contudo, para o problema em questão, é necessário realizar uma análise de contexto antes de aplicar qualquer uma dessas abordagens, visto que, um caso de teste pode ter falhado há muitas execuções passadas e se mantém estável agora. Duplicar ou calibrar o algoritmo para acomodar esse desbalanceamento pode desconsiderar o comportamento do teste. Futuras análises devem ser feitas para resolver o problema em questão sem que se perca o comportamento histórico do teste.

Com o objetivo de enriquecer a infraestrutura visando que consiga melhorar o processo de teste de maneira geral, pretende-se adicionar mais análises para a extração

de novas métricas relativas aos métodos de testes tendo em vista o enriquecimento dos dados utilizados para realizar as previsões.

Cobrir mais abordagens de teste, permitiria que a infraestrutura pudesse ser utilizada para outros tipos de testes além do de regressão e unidade. Adicionar mais abordagens, como por exemplo, priorizar os testes que possuem maior probabilidade de falha de acordo com as previsões, para otimização da suíte de testes dado que tais abordagens são muito utilizadas no contexto de testes de regressão.

Por fim, realizar um estudo experimental que tenha como objetivo verificar a usabilidade da infraestrutura proposta.

REFERÊNCIAS

ANANDARAJ, A; PADMANABHAN, Kalaivani; RAMESHKUMAR, V. **Development Of Ontology-Based Intelligent System For Software Testing**. v. 2 , 2013.

BAI, Xiaoying *et al.* **Ontology-based test modeling and partition testing of web services.** Proceedings of the IEEE International Conference on Web Services, ICWS 2008 p. 465–472 , 2008.9780769533100.

BALDAUF, Matthias; DUSTDAR, Schahram; ROSENBERG, Florian. **A Survey on Context-Aware Systems.** Int. J. Ad Hoc Ubiquitous Comput. v. 2, n. 4, p. 263–277 , 2007.

BALDONADO, Michelle Q.; WOODRUFF, Allison; KUCHINSKY, Allan. **Guidelines for Using Multiple Views in Information Visualization**. AVI '00, 2000, New York, NY, USA: ACM, 2000. p.110–119. Disponível em: <http://doi.acm.org/10.1145/345513.345271>. 1-58113-252-2. .

BARBOSA, Ellen *et al.* ***Ontology-based Development of Testing Related Tools.*** .20th International Conference on Software Engineering and Knowledge Engineering, SEKE 2008. [S.l: s.n.]. , 2008

BARBOSA, Ellen; NAKAGAWA, Elisa; MALDONADO, José. ***Towards the Establishment of an Ontology of Software Testing***. .18th International Conference on Software Engineering and Knowledge Engineering, SEKE 2006. [S.l: s.n.]. , 2006

BATISTA, Gustavo E. A. P. A.; PRATI, Ronaldo C.; MONARD, Maria Carolina. A **Study of the Behavior of Several Methods for Balancing Machine Learning Training Data.** ACM SIGKDD Explorations Newsletter - Special issue on learning from imbalanced datasets v. 6, n. 1, p. 20–29 , 2004. Disponível em: <http://doi.acm.org/10.1145/1007730.1007735%5Cnhttp://dl.acm.org/ft_gateway.cfm?id=1007735&type=pdf>.1931-0145.

BECK, Kent. **Extreme Programming Explained: Embrace Change**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. .0-201-61641-6.

BELLINGARD, Fabrice. ***About SonarQube.*** Disponível em: <https://www.sonarqube.org/about/>. Acesso em: 30 jan. 2018.

BEZERRA, Daniella; COSTA, Afonso; OKADA, Karla. S**wTOI(Software Test Ontology Integrated) and its application in Linux test.** CEUR Workshop Proceedings v. 460, p. 25–36 , 2009.

BISHNU, P S. **Application of K-Medoids with Kd-Tree for Software Fault Prediction** ACM SIGSOFT Software Engineering Notes. v. 36, n. 2, p. 1–6 , 2011.

BISHNU, P S; BHATTACHERJEE, V. **Application of K-Medoids with Kd-Tree for Software Fault Prediction.** SIGSOFT Softw. Eng. Notes v. 36, n. 2, p. 1–6 , 2011. Disponível em: <http://doi.acm.org/10.1145/1943371.1943381>.

BOETTICHER, Gary D. **Nearest Neighbor Sampling for Better Defect Prediction**. p. 1–6 , 2005.1595931252.

BOSCH, Jan. **Continuous Software Engineering**. [S.l.]: Springer Publishing Company, Incorporated, 2014. .3319112821, 9783319112824.

BOWES, David *et al.* **Mutation-aware Fault Prediction**. ISSTA 2016, 2016, New York, NY, USA: ACM, 2016. p.330–341. Disponível em: <http://doi.acm.org/10.1145/2931037.2931039>. 978-1-4503-4390-9. .

BUNEMAN, Peter; KHANNA, Sanjeev; TAN, Wang-chiew. Why and Where : A **Characterization of Data Provenance ?** , 2001.

CAI, L *et al.* **Test Case Reuse Based on Ontology.** nov. 2009, [S.l: s.n.], nov. 2009. p.103–108.

CANFORA, Gerardo *et al.* **Defect prediction as a multiobjective optimization**

problem. n. March, p. 426–459 , 2015.

CARNEIRO, G F; CONCEIÇÃO, C F R; DAVID, J. A **Multiple View Environment for Collaborative Software Comprehension.**, p. 15–21 , 2012. Disponível em: <http://www.thinkmind.org/index.php?view=article&articleid=icsea_2012_1_30_10246>.9781612082301.

CARVALHO, André B; POZO, Aurora; VERGILIO, Silvia Regina. **A symbolic fault-prediction model based on multiobjective particle swarm optimization.** Journal of Systems and Software v. 83, n. 5, p. 868–882 , 2010. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121209003367>.

CHARNIAK, E.; MCDERMOTT, D. **Introduction to Artificial Intelligence (Addison-Wesley Series in Computer Science)**. [S.l.]: Addison-Wesley; 1St Edition edition (May 1, 1985), 1985. 701 p. .978.

CHEN, Lianping; POWER, Paddy. Continuous Delivery. **Continuous delivery** p. 497 .

CLAPS, Gerry; SVENSSON, Richard; AURUM, Aybüke. **On the journey to continuous deployment: Technical and social challenges along the way.** Information and Software Technology v. 57, n. 1, p. 21–31 , 2015. Disponível em: <http://dx.doi.org/10.1016/j.infsof.2014.07.009>.09505849.

DA SILVA, Fabio Q B *et al.* **A Critical Appraisal of Systematic Reviews in Software Engineering from the Perspective of the Research Questions Asked in the Reviews.** ESEM '10, 2010, New York, NY, USA: ACM, 2010. p.33:1--33:4. Disponível em: <http://doi.acm.org/10.1145/1852786.1852830>. 978-1-4503-0039-1. .

DAPRA, Humberto L. O. *et al.* **Using ontology and data provenance to improve software processes Using Ontology and Data Provenance to Improve Software Processes.** n. June 2016 , 2015.

DELAMARO, Márcio Eduardo; MALDONADO, José Carlos; JINO, Mario. **Introdução ao teste de software**. 13º ed. [S.l: s.n.], 2007. 394 p. .978-85-352-2634-8.

DIEHL, Stephan. **Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007. .3540465049.

DROWN, D J; KHOSHGOFTAAR, T M; SELIYA, N. **Evolutionary Sampling and Software Quality Modeling of High-Assurance Systems.** IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans v. 39, n. 5, p. 1097–1107 , 2009.

DSSOULI, Rachida *et al.* Chapter Three - **Testing the Control-Flow, Data-Flow, and Time Aspects of Communication Systems**: A Survey. In: MEMON, Atif M (Org.). . Advances in Computers. [S.l.]: Elsevier, 2017. 107 v. p. 95–155. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0065245817300281>.

EDITION, Second; ANTONIOU, Grigoris; HARMELEN, Frank Van. **A Semantic Web Primer**. 2º ed. [S.l: s.n.], 2008. 264 p. .9780262012423.

ELENA, P. **Improving fault prediction using Bayesian networks for the development of embedded software applications** ‡. n. September 2005, p. 157–174 , 2006.

FITZGERALD, Brian; STOL, Klaas-Jan. **Continuous software engineering: A roadmap and agenda.** Journal of Systems and Software v. 123, p. 176–189 , 2017. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121215001430>.

G. SAPNA, P; MOHANTY, Hrushikesha. *An Ontology Based Approach for Test Scenario Management* .**Communications in Computer and Information Science**. [S.l: s.n.]. , 2011

GALAR, M. and *et al.* **A Review on Ensembles for the Class Imbalance Problem: Bagging-, Boosting-, and Hybrid-Based Approaches**. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) v. 42, p. 463–484 , 2012.

GAO, Ruizhi *et al.* **Effective Software Fault Localization Using Predicted Execution Results.** Software Quality Journal v. 25, n. 1, p. 131–169 , 2017. Disponível em: <https://doi.org/10.1007/s11219-015-9295-1>.

GARCIA, Salvador; FERNÁNDEZ, Alberto; HERRERA, Francisco. Enhancing the **Effectiveness and Interpretability of Decision Tree and Rule Induction Classifiers with Evolutionary Training Set Selection over Imbalanced Problems**. Appl. Soft Comput. v. 9, n. 4, p. 1304–1314 , 2009. Disponível em: <http://dx.doi.org/10.1016/j.asoc.2009.04.004>.

GHHS, Sudpdq *et al.* **Software defect prediction analysis using machine learning algorithms**. p. 775–781 , 2017.9781509035199.

GROTH, P.; MOREAU, L. ***PROV-Overview. an overview of the PROV Family of Documents*** . [S.l: s.n.]. Disponível em: <https://www.w3.org/TR/prov-overview/>. , 2013

GRUBER, Thomas R. **Toward principles for the design of ontologies used for knowledge sharing?** International Journal of Human-Computer Studies v. 43, n. 5, p. 907–928 , 1995. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1071581985710816>.

GUI, Lin; SI, Yuan Jie; YANG, Xin Yu. **Combining Model Checking and Testing with an Application to Reliability Prediction and Distribution Categories and Subject Descriptors**. p. 101–111 , 2005.9781450321594.

HADI, H M *et al.* **Classification of heart sound based on s-transform and neural network**. 2013, [S.l: s.n.], 2010. p.189–192.

HALL, Mark *et al.* **The WEKA Data Mining Software: An Update**. SIGKDD Explor. Newsl. v. 11, n. 1, p. 10–18 , 2009. Disponível em: <http://doi.acm.org/10.1145/1656274.1656278>.

HAND, David J; SMYTH, Padhraic; MANNILA, Heikki. **Principles of Data Mining**.

Cambridge, MA, USA: MIT Press, 2001. .0-262-08290-X, 9780262082907.

HAWKINS, Douglas M. **The Problem of Overfitting**. Journal of Chemical Information and Computer Sciences PMID: 14741005, v. 44, n. 1, p. 1–12 , 2004. Disponível em: <https://doi.org/10.1021/ci0342472>.

HERBOLD, Steffen. **Training Data Selection for Cross-project Defect Prediction.** PROMISE '13, 2013, New York, NY, USA: ACM, 2013. p.6:1--6:10. Disponível em: <http://doi.acm.org/10.1145/2499393.2499395>. 978-1-4503-2016-0. .

HOSMER, David W; LEMESHOW, Stanley. **Applied logistic regression (Wiley Series in probability and statistics)**. 2. ed. [S.l.]: Wiley-Interscience Publication, 2000. Disponível em: <http://www.amazon.com/Applied-logistic-regression-probability-statistics/dp/0471356328%3FSubscriptionId%3D192BW6DQ43CK9FN0ZGG2%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0471356328>. .0471356328.

JING, Xiao-Yuan *et al.* Dictionary **Learning Based Software Defect Prediction**. p. 414–423 , 2014. Disponível em: <http://doi.acm.org/10.1145/2568225.2568320%5Cnfiles/360/Jing et al. - 2014 - Dictionary Learning Based Software Defect Predicti.pdf%5Cnfiles/1187/Jing et al. - 2014 - Dictionary Learning Based Software Defect Predicti.pdf>.978-1-4503-2756-5.

JONGSAWAT, N; PREMCHAISWADI, W. **Developing a Bayesian Network Model Based on a State and Transition Model for Software Defect Detection**. 2012, [S.l: s.n.], 2012. p.295–300.

KITCHENHAM, Barbara; CHARTERS, Stuart. **Guidelines for performing Systematic Literature reviews in Software Engineering Version 2.3.** Engineering v. 45, n. 4ve, p. 1051 , 2007. Disponível em: <http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Guidelines+for+performing+Systematic+Literature+Reviews+in+Software+Engineering#0%5Cnhttp://www.dur.ac.uk/ebse/resources/Systematic-reviews-5-8.pdf>.1595933751.

KOTSIANTIS, Sotiris B. S**upervised Machine Learning: A Review of Classification Techniques**. Informatica v. 31, p. 249–268 , 2007.1586037803.

LEBO, T. *et al.* Prov-o: **The prov ontology.** 2013, [S.l: s.n.], 2013. Disponível em: <https://www.w3.org/TR/prov-o/>.

LI, Han *et al.* **Using ontology to generate test cases for GUI testing**. International Journal of Computer Applications in Technology v. 42, n. 2/3, p. 213 , 2011. Disponível em: <http://www.inderscience.com/link.php?id=45407>.

LI, Xuexiang; ZHANG, Wenning. **Ontology-based testing platform for reusing.** Proceedings - 6th International Conference on Internet Computing for Science and Engineering, ICICSE 2012 p. 86–89 , 2012.9780769547053.

LIM, Chunhyeok *et al.* **Prospective and Retrospective Provenance Collection in Scientific Workflow**. , 2010.9780769541266.

LIM, TS; LOH, WH; SHIH, YS. **A comparison of prediction accuracy, complexity, and training time of thirty three old and new classification algorithms.** Machine Learning v. 40, n. 3, p. 203–229 , 2000.0885-6125.

MICHLMAYR, Martin *et al.* **Why and How Should Open Source Projects Adopt Time- Based Releases ?** n. April, p. 55–63 , 2015.

MOREAU, L.; MISSIER, P. **PROV-DM: The prov data model**. 2013, [S.l: s.n.], 2013. Disponível em: <https://www.w3.org/TR/prov-dm/>.

MOREAU, Luc *et al.* **The Open Provenance Model: An Overview.** 2008, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p.323–326. 978-3-540-89965-5. .

NAKAGAWA, Elisa Yumi; BARBOSA, Ellen Francine; MALDONADO, José Carlos. **Exploring ontologies to support the establishment of reference architectures: An example on software testing.** 2009 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, WICSA/ECSA 2009

p. 249–252 , 2009.9781424449859.

NAM, Jaechang; PAN, Sinno Jialin; KIM, Sunghun. **Transfer Defect Learning.** ICSE '13, 2013, Piscataway, NJ, USA: IEEE Press, 2013. p.382–391. Disponível em: <http://dl.acm.org/citation.cfm?id=2486788.2486839>. 978-1-4673-3076-3. .

NEELY, Steve; STOLT, Steve. **Continuous delivery? Easy! Just change everything (well, maybe it is not that easy)**. Proceedings - AGILE 2013 p. 121–128 , 2013.9780769550763.

OSTRAND, Thomas J; WEYUKER, Elaine J; BELL, Robert M. **Automating Algorithms for the Identification of Fault-prone Files.** ISSTA '07, 2007, New York, NY, USA: ACM, 2007. p.219–227. Disponível em: <http://doi.acm.org/10.1145/1273463.1273493>. 978-1-59593-734-6. .

OSTRAND, Thomas; PARK, Florham. **Software Fault Prediction Tool.** p. 275–278 , 2010.9781605588230.

PÉREZ-MIÑANA, Elena; GRAS, Jean-Jacques. **Improving Fault Prediction Using Bayesian Networks for the Development of Embedded Software Applications: Research Articles.** Softw. Test. Verif. Reliab. v. 16, n. 3, p. 157–174 , 2006. Disponível em: <http://dx.doi.org/10.1002/stvr.v16:3>.

QUINLAN, J Ross. **C4.5: Programs for Machine Learning**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. .1-55860-238-0.

REFAEILZADEH, Payam; TANG, Lei; LIU, Huan. Cross-Validation. In: LIU, LING; OZSU, M. TAMER (Orgs.). . **Encyclopedia of Database Systems**. [S.l.]: Springer US, 2009. p. 532--538. Disponível em: <https://doi.org/10.1007/978-0-387-39940-9_565>. 978-0-387-39940-9.

ROYCE, W W. **Managing the Development of Large Software Systems: Concepts and Techniques.** ICSE '87, 1987, Los Alamitos, CA, USA: IEEE Computer Society Press, 1987. p.328–338. Disponível em:

<http://dl.acm.org/citation.cfm?id=41765.41801>. 0-89791-216-0. .

RUSSELL, Stuart J.; NORVIG, Peter. **Artificial intelligence A Modern Approach**. [S.l: s.n.], 2017. 1145 p. .9780136042594.

RYU, H; RYU, D K; BAIK, J. **A Strategic Test Process Improvement Approach Using an Ontological Description for MND-TMM.** 2008, [S.l: s.n.], 2008. p.561–566.

SAFF, D.; ERNST, M. D. Reducing wasted development time via continuous testing. **Proceedings - International Symposium on Software Reliability Engineering, ISSRE** v. 2003–Janua, p. 281–292 , 2003.0769520073.

SIMON, P. **Too Big to Ignore: The Business Case for Big Data**. [S.l.]: Wiley, 2013. Disponível em: <https://books.google.com.br/books?id=Dn-Gdoh66sgC>. (Wiley and SAS Business Series). .9781118642108.

SINGH, P Deep; CHUG, **A. Software defect prediction analysis using machine learning algorithms**. jan. 2017, [S.l: s.n.], jan. 2017. p.775–781.

SINGH, Yogesh; KAUR, Arvinder; MALHOTRA, Ruchika. **Empirical Validation of Object-oriented Metrics for Predicting Fault Proneness Models.** Software Quality Journal v. 18, n. 1, p. 3–35 , 2010. Disponível em: <http://dx.doi.org/10.1007/s11219-009-9079-6>.

SMART, John Ferguson. **Jenkins: The Definitive Guide**. [S.l.]: O'Reilly Media, Inc., 2011. .1449305350, 9781449305352.

SOHN, Jeongju. FLUCCS : **Using Code and Change Metrics to Improve Fault Localization**. n. July, p. 273–283 , 2017.9781450350761.

SOMMERVILLE, I. **Engenharia de software**. [S.l.]: PEARSON BRASIL, 2011. Disponível em: <https://books.google.com.br/books?id=H4u5ygAACAAJ>. .9788579361081.

SOUZA, E. F.; FALBO, R. A.; VIJAYKUMAR, N. L. **Using ontology patterns for building a reference sofware testing ontology.** Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC p. 21–30 , 2013.1541-7719.

SPENCE, Robert. **Information Visualization: Design for Interaction (2Nd Edition)**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2007. .0132065509.

SU, C T; HSIAO, Y H. **An Evaluation of the Robustness of MTS for Imbalanced Data.** IEEE Transactions on Knowledge and Data Engineering v. 19, n. 10, p. 1321–1332 , 2007.

WARD, Jonathan Stuart; BARKER, Adam. Undefined By Data: **A Survey of Big Data Definitions.** , 2013. Disponível em: <http://arxiv.org/abs/1309.5821>.9781612083957.

WEYUKER, Elaine J; OSTRAND, Thomas J; BELL, Robert M. **Comparing negative binomial and recursive partitioning models for fault prediction.** 2008, [S.l: s.n.], 2008. p.3. Disponível em: <http://portal.acm.org/citation.cfm?doid=1370788.1370792>. 9781605580364. .

WINSTON, P. H. **Artificial Intelligence**. [S.l.]: Pearson; 3 edition (May 10, 1992), 1992. 737 p. .978-0201533774.

WOHLIN, C. *et al.* **Experimentation in Software Engineering**. 1. ed. [S.l.]: Springer-Verlag Berlin Heidelberg, 2012. XXIV, 236 p. .978-3-642-29044-2.

WOHLIN, Claes *et al.* **Experimentation in Software Engineering: An Introduction**. Norwell, MA, USA: Kluwer Academic Publishers, 2000. .0-7923-8682-5.
YU, L *et al.* A Framework of Testing as a Service. 2009, [S.l: s.n.], 2009. p.1–4.

YU, Liyang. **A developer's guide to the semantic Web**. 2. ed. [S.l.]: Springer-Verlag Berlin Heidelberg, 2014. 829 p. .978-3-662-43796-4.

ZHANG, Y; ZHU, H. **Ontology for Service Oriented Testing of Web Services.** 2008,

[S.l: s.n.], 2008. p.129–134.

ZHANG, Zhi-Wu; JING, Xiao-Yuan; WANG, Tie-Jian. Label **Propagation Based Semi-supervised Learning for Software Defect Prediction**. Automated Software Engg. v. 24, n. 1, p. 47–69 , 2017. Disponível em: <https://doi.org/10.1007/s10515-016-0194-x>.

ZHU, Hong; HUO, Qingning. **Developing A Software Testing Ontology in UML for A Software Growth Environment of Web-Based Applications.** Software Evolution with UML and v. 1060, n. January, p. 263–295 , 2005.9781591404620.