UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO

Uso de *Machine Learning* para identificação de solicitação de teste de confirmação em projeto de teste de software

---

Trabalho de Conclusão de Curso
de Bacharelado em Sistemas de Informação na modalidade Empresa

---

Aluno
Victor Leuthier dos Santos

Orientador
Cleviton Vinicius Fonsêca Monteiro
Departamento de Estatística e Informática

Coorientador
Gabriel Alves de Albuquerque Junior
Departamento de Estatística e Informática

Recife
Junho de 2022

Victor Leuthier dos Santos

Uso de Machine Learning para identificação de solicitação de teste de confirmação em projeto de teste de software

Relatório Técnico apresentado ao Curso de Bacharelado em Sistemas de Informação da Universidade Federal Rural de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

Universidade Federal Rural de Pernambuco – UFRPE

Departamento de Estatística e Informática

Curso de Bacharelado em Sistemas de Informação

Orientador: Cleviton Vinicius Fonsêca Monteiro
Coorientador: Gabriel Alves de Albuquerque Junior

Recife

Junho de 2022

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

S237u Santos, Victor Leuthier
Uso de Machine Learning para identificação de solicitação de teste de confirmação em projeto de teste de software / Victor Leuthier Santos. - 2022.
31 f. : il.

Orientador: Cleviton Vinicius Fonseca Monteiro.
Coorientador: Gabriel Alves de Albuquerque Junior.
Inclui referências e apêndice(s).

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal Rural de Pernambuco, Bacharelado em Sistemas da Informação, Recife, 2022.

1. Teste de confirmação. 2. Aprendizado de máquina. 3. Qualidade de software. 4. Machine learning. 5. Confirmation test. I. Monteiro, Cleviton Vinicius Fonseca, orient. II. Junior, Gabriel Alves de Albuquerque, coorient. III. Título

CDD 004

VICTOR LEUTHIER DOS SANTOS

Uso de Machine Learning para identificação de solicitação de teste de confirmação em projeto de teste de software

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

Aprovada em: 06 de Junho de 2022.

BANCA EXAMINADORA

Cleviton Vinicius Fonsêca Monteiro
Departamento de Estatística e Informática
Universidade Federal Rural de Pernambuco

Gabriel Alves de Albuquerque Junior
Departamento de Estatística e Informática
Universidade Federal Rural de Pernambuco

Cícero Garrozi
Departamento de Estatística e Informática
Universidade Federal Rural de Pernambuco

Resumo

Um estudo recente publicado pela Cambridge Judge Business School descobriu que os desenvolvedores perdem 620 milhões de horas por ano depurando falhas de software, o que acaba custando às empresas aproximadamente US$ 61 bilhões por ano. Este processo de depuração se torna ainda mais complexo e custoso às organizações quando o desenvolvedor não possui acesso aos recursos necessários. Utilizando a biblioteca TPOT como ferramenta de Auto *Machine Learning* para encontrar a melhor *pipeline* de um modelo foram analisados comentários do Jira afim de identificar pedidos de reteste por parte de desenvolvedores para os testadores de uma empresa. Foi construído um modelo para criação da ferramenta chamada “Preste atenção ao reteste ou Pay attention to retest” - PATRE, que utiliza Aprendizado de Máquina (Machine Learning) para identificação automática de pedido de teste de confirmação, otimizando assim a rotina dos profissionais envolvidos no desenvolvimento do software. O classificador gerado após 5 gerações foi o *GradientBoostingClassifier* e obteve uma precisão de 0.562, e um *recall* de 0.529, enquanto o *f1-score* encontrado foi de cerca de 0.545. Enquanto que o classificador escolhido pelo TPOT após 20 gerações foi o *StackingEstimator* obteve os seguintes resultados: precisão de 0.48, *recall* de 0.735 e *f1-score* de 0.581. Mostrando a influência direta do número de gerações na qualidade do modelo e do classificador final. Nenhuma informação ou dado confidencial foi utilizado para a realização deste trabalho.

Sumário

1 Introdução

Definitivamente, a construção de software não é uma tarefa simples. Pelo contrário, pode se tornar bastante complexa, dependendo das características e dimensões do sistema a ser criado. Por isso, está sujeita a diversos tipos de problemas que acabam resultando na obtenção de um produto diferente daquele que se esperava. [1]

Frente a essa problemática inicial se evidencia a importância da qualidade durante todo o processo de desenvolvimento de software. Segundo Pressman [2], qualidade é algo difícil de definir, entretanto perceptível ao vê-la. Para andamento deste artigo adotaremos a definição de qualidade de Garvin [3], da *Harvard Business School*, destacando a complexidade e as múltiplas faces deste processo. Destacaremos dentre os cinco pontos de vista do autor sobre o que é qualidade a perspectiva baseada em valor, que mede a qualidade tomando como base o quanto um cliente estaria disposto a pagar por um produto. Corroborando com David Garvin, Pressman destaca qualidade de software como sendo aquilo que agrega valor efetivo mensurável para aqueles que produzem e aqueles que utilizam.

Como a maioria dos softwares evoluem e são modificados constantemente durante o desenvolvimento e manutenção, é necessário testar as partes novas e as já existentes que podem ter sido afetadas pelas mudanças. Esta atividade é chamada de teste de regressão e pode ser responsável por uma grande porcentagem do custo geral de desenvolvimento de software. [4]

Um estudo recente publicado pela Cambridge Judge Business School descobriu que os desenvolvedores perdem 620 milhões de horas por ano depurando falhas de software, o que acaba custando às empresas aproximadamente US$ 61 bilhões por ano.[5] Este processo de depuração se torna ainda mais complexo quando o desenvolvedor não possui acesso aos recursos necessários, como por exemplo um modelo específico de celular, uma placa em que o erro foi reportado, ou um sistema operacional com versão diferente, fazendo com que além de mais custoso necessite de um esforço em conjunto de diversas áreas da organização para a resolução do defeito encontrado.

O *International Software Testing Qualifications Board - ISTQB*, destaca que quando são feitas alterações em um sistema, seja para corrigir um defeito ou por causa de uma funcionalidade nova, deve-se testar para confirmar se as alterações corrigiram o defeito ou implementaram a funcionalidade corretamente e não causaram consequências adversas imprevistas. Entre os tipos de testes relacionados à mudança temos o **teste de regressão** e o **teste de confirmação**.

No teste de confirmação, depois que um defeito é corrigido, o software pode ser testado com todos os casos de teste que falharam devido ao defeito, que devem ser executados novamente na nova versão do software. O software também pode ser testado com novos testes se, por exemplo, para um defeito estiver faltando uma funcionalidade. [6]. Assim, pode-se dizer que o objetivo de um teste de confirmação é garantir que o defeito original foi corrigido com sucesso, e é recomendável que sejam reproduzidas na nova versão do software as mesmas etapas que causaram a falha.

Já no teste de regressão, é possível que uma alteração feita em uma parte do código, seja uma correção ou outro tipo de alteração, possa afetar acidentalmente o comportamento de outras partes do mesmo código, seja dentro do mesmo componente, em outros componentes do código, no mesmo sistema, ou em outros sistemas. [6] Versionamento do sistema operacional e do banco de dados são

exemplos de alterações que podem ocorrer. Esses comportamentos não intencionais são chamados de regressões e o teste de regressão envolve a execução de testes para detectá-los.

O teste de confirmação ou *confirmation test*, como é comumente visto na literatura, é um tipo de teste relacionado com alterações, realizado após a correção de um defeito para confirmar que uma falha causada por esse defeito não ocorre novamente. Esse tipo de teste é importante pois assegura que o software seja confiável e garante a satisfação do cliente; Garante a qualidade do produto, o que acaba fidelizando o cliente; Reduz os custos de manutenção; Reduz os custos de corrigir *bugs* em fases mais avançadas do desenvolvimento de software. Ou seja, é uma parte do processo de qualidade de software durante o ciclo de vida de um produto em que o teste é executado novamente para certificar-se de que a falha foi corrigida ou não, sem a necessidade de criar novos casos de testes para estes testes, podendo ser realizado antes do teste de regressão dado o seu escopo reduzido. No contexto deste trabalho, um pedido de reteste corresponde sempre a um teste de confirmação.

1.1 Contexto

O processo de desenvolvimento de software presente na organização de origem deste trabalho segue os moldes do modelo incremental, combinando elementos do modelo cascata (aplicado repetidamente) com a filosofia iterativa da prototipação. Como visto na Figura 1, os testes variam de acordo com a fase de desenvolvimento do software.

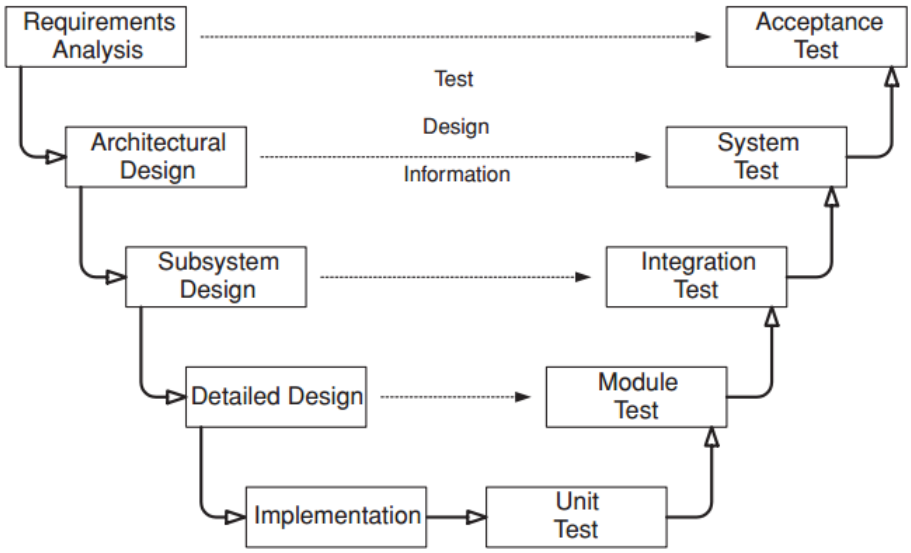

Figura 1: Níveis de teste com base na fase de desenvolvimento do software. (Sommerville, 2011) [7]

Além deste modelo a empresa faz uso da metodologia ágil, especialmente as ferramentas Kanban e Scrum, com *sprints* semanais, permitindo que os diversos times respondam rapidamente às mudanças de prioridade que ocorrem enquanto conseguem administrar facilmente o seu fluxo de trabalho. Sendo capazes de acompanhar suas tarefas, comunicar-se e realizar os trabalhos com rapidez e eficiência utilizando o Jira como plataforma central de controle de atividades de desenvolvimento e testes.

O Jira é uma ferramenta que permite planejar, gerenciar e monitorar as atividades de um projeto de acordo com as peculiaridades da empresa. Com o uso desta plataforma há a possibilidade de: criar relatórios de falhas para o time de desenvolvimento, o que é chamado de *Change Request* (CR); alocar atividades para os diferentes usuários além de monitorar e estimar qual o tempo necessário para a realização de uma atividade. Neste trabalho, a ferramenta Jira foi utilizada para a coleta de todos os comentários contidos nas CRs recuperadas.

1.1.1 *Change Request* (CR)

O procedimento de criação de uma CR começa quando o testador encontra um *bug* que faz com o produto se comporte de uma maneira não esperada. Então, para corrigir este problema, o testador cria uma CR para que o time de desenvolvimento possa encontrar uma solução para o *bug*. Uma CR tem apenas um *reporter*, que é a pessoa responsável pela criação deste defeito no Jira, no contexto deste trabalho é a própria pessoa que testa e encontra defeitos. As pessoas que tem algum interesse

no erro criado e que devem receber e-mails a cada atualização da CR são adicionadas pelo *reporter* como *watcher* da CR, nas circunstâncias deste artigo são adicionados todos os outros funcionários e líderes da equipe. As *labels* ou rótulos, são utilizadas com o intuito de rastrear as CRs, indicando quais produtos e ciclos de testes estão sendo afetatos por esta falha, além de informar a criticidade e se este defeito já passou pela triagem e foi movido para algum time. A equipe de integração, desenvolvedores ou testadores podem a qualquer momento do ciclo de vida de uma CR adicionar ou remover *labels*.

Na Figura 2 é possível ter uma visualização geral das informações, ferramentas e pessoas envolvidas no processo ao reportar um defeito que foi descrito no parágrafo anterior.

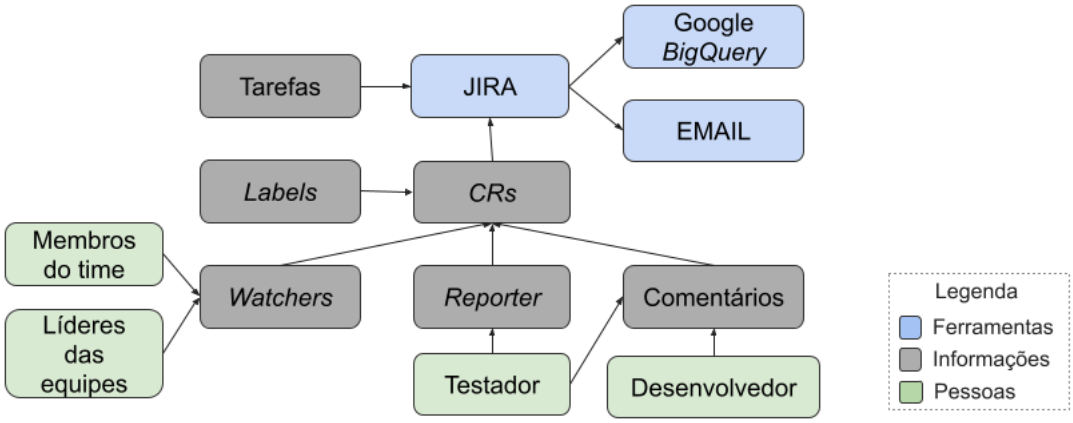

Figura 2: Visão geral acerca das informações contidas ao realizar a criar uma nova CR no Jira.
Fonte: Elaborada pelo autor

Após uma CR ter sido criada, ela sofre um processo de triagem manual por uma equipe responsável por fazer essa análise conforme o produto em questão e que define de acordo com as informações descritas nos campos do defeito qual componente esta inconsistência está relacionada. Todas as CRs tem um *status* e que pode ser classificado em vários tipos diferentes. Neste trabalho, citaremos alguns tipos de *status* que consideraremos importantes para análise futura.

- *New*: estado inicial após ter sido criada, aguardando o processo de triagem;
- *Unresolved*: significa que nenhuma solução para esse *bug* foi encontrada, apesar de estar com o time que tenha a maior chance de ter afinidade com a inconsistência relatada;
- *Working*: após a análise inicial da equipe de desenvolvimento é iniciado o processo de depuração do defeito e então o *status* é alterado para *working*, indicando que há algum desenvolvedor focado em solucionar o defeito;
- *Ready*: quando a investigação para o defeito é concluída e a solução encontrada é validada de fato, o desenvolvedor fica apenas aguardando aprovação para implantação da correção no código;
- *Closed*: quando a correção foi implantada e uma nova versão do software foi finalizada, o status muda para *closed*. Indicando que todas as medidas necessárias para a correção entrar em vigor foram feitas, incluindo testes de confirmação, seja ele tendo sido executado pelo desenvolvedor ou testador.

Os *status* de uma CR descritos anteriormente podem ser observados na Figura 3, destacando a interação entre desenvolvedores e testadores ao realizar a depuração de uma CR em busca de uma solução.

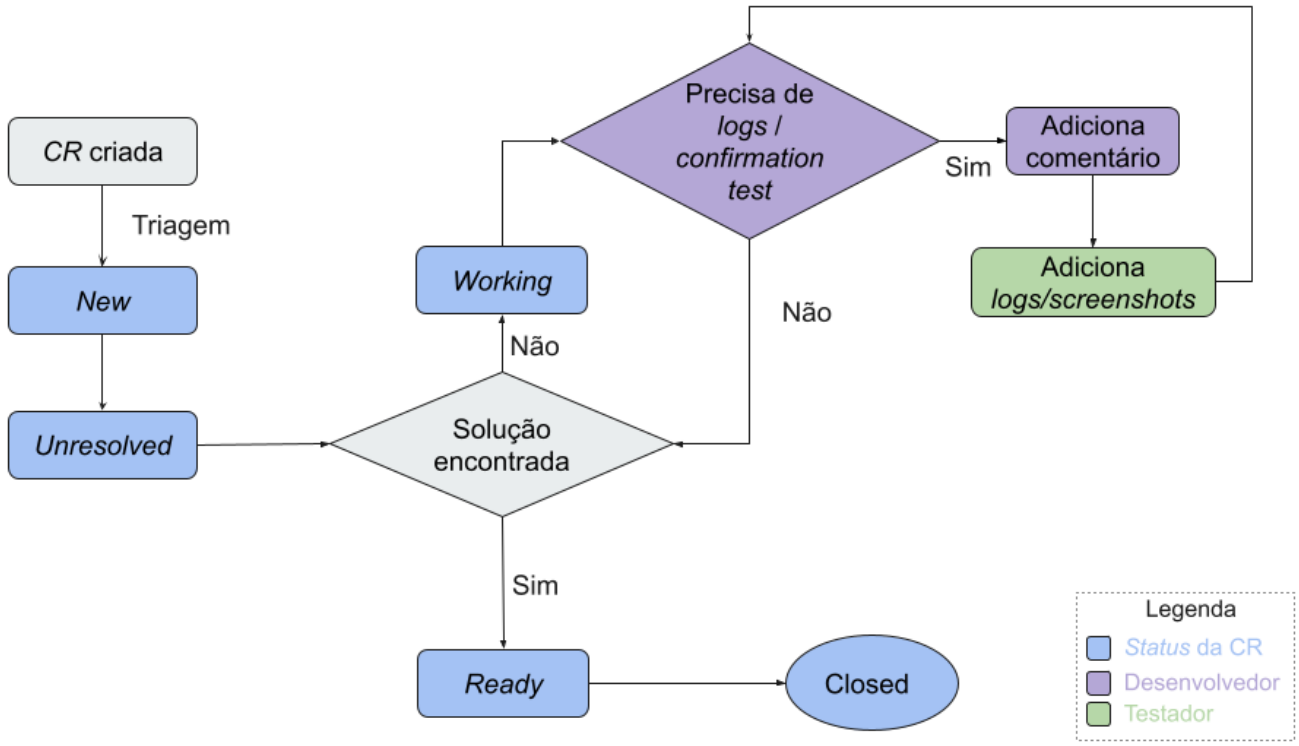

Figura 3: Ciclo de vida de uma *CR*.
Fonte: Elaborada pelo autor

Na eventualidade do desenvolvedor designado para o *debugging* da inconsistência não conseguir identificar de maneira satisfatória a causa para o problema, na maioria das vezes é necessário a coleta de novas evidências (*logs* e *screenshots*) e como o desenvolvedor, na maioria das vezes, não tem disponível os recursos (produtos) necessários para esta coleta, frequentemente entra em contato com os testadores para o auxílio nesta atividade.

1.2 Problema e solução proposta

Os maiores desafios encontrados diariamente por líderes de equipes, engenheiros de testes, residentes e estagiários acerca das atividades relacionadas a testes de software dentro da empresa objeto de estudo deste trabalho são:

- Grande quantidade de erros reportados por diferentes pessoas;
- Inconsistências semelhantes descritas de maneira não uniforme;
- Frequência elevada de recebimento de e-mails, sendo eles de relatórios (*reports*) de execução de testes, informações importantes como mudança de planejamento, atualizações sobre todas as CRs que a pessoa está na lista de *watchers*, marcação de reuniões, entre outros tipos de e-mails;

- Quem reporta um erro em que é solicitado reteste, nem sempre visualiza o pedido através do e-mail, sendo relativamente frequente os desenvolvedores fazerem essa solicitação via chat quando não obtém retorno esperado por intermédio do campo de comentários da CR cadastrada no Jira;
- Os líderes das equipes têm dificuldade para monitorar erros reportados por membros do seu time que precisam de atenção, normalmente para verificar se o erro persiste, ou para capturar novos *logs* conforme solicitado pelo desenvolvedor que está analisando este defeito;

1.2.1 Objetivo

Desta forma, o presente estudo propõe a criação de uma ferramenta chamada de "Preste atenção ao reteste" ou *Pay attention to retest* - PATRE, que utiliza Aprendizado de Máquina (*Machine Learning*) para identificação automática de pedido de teste de confirmação, neste contexto chamado de reteste através da análise dos comentários do Jira quando há solicitação por parte do desenvolvedor para que o testador execute este tipo de teste, uma vez que muitas vezes o produto em questão não está ao alcance dos desenvolvedores.

1.3 Justificativa

Os desenvolvedores envolvidos na correção do defeito em questão atualmente tem a necessidade de entrar em contato via chat com os testadores para solicitar uma nova verificação, muitas vezes isso ocorre pois o volume de e-mails recebidos diariamente pelos membros da área de qualidade é consideravelmente elevado, uma vez que toda e qualquer atualização nas CR's reportadas ou em que a pessoa está como *watcher* sofre uma alteração, um novo e-mail é disparado e o testador acaba por não visualizar esse e-mail referente ao pedido de reteste solicitado pelo desenvolvedor através do campo do comentário na CR aberta no JIRA.

Este problema se torna ainda mais grave quando a pessoa é responsável pelo time e faz parte do processo colocá-la como *watcher* em todas as CR's criadas pela equipe. Fica praticamente impossível acompanhar todas essas atualizações diariamente em busca de pedidos para aquele erro ser novamente verificado.

2 A empresa e sua atuação

O Projeto CIn/Motorola tem como objetivo principal incentivar a formação de recursos humanos com alto grau de especialização em testes de *software* embarcado e aplicações em computação móvel, com os incentivos e benefícios previstos na Lei de Informática. O foco de atuação é o planejamento, projeto, automação e execução de diversos tipos de testes, realizados em aplicações para celulares. A atuação do autor deste estudo consiste na atividade de execução de testes, enquanto este trabalho figura na área de pesquisa desenvolvida dentro da organização.

Este projeto é gerido pela Fundação de Apoio ao Desenvolvimento da Universidade Federal de

Pernambuco (Fade-UFPE), que foi criada em 10 de agosto de 1981 e é constituída na forma de fundação de direito privado, sem fins lucrativos. Está regulamentada pela Lei n.º 8.958/94 e Decreto n.º 7.423/2010, sendo condicionada ao prévio registro e credenciamento nos Ministérios da Educação (ME) e da Ciência, Tecnologia, Inovações e Comunicações (MCTIC), com renovação a cada cinco anos. É fiscalizada pelo Ministério Público de Pernambuco (Centro de Apoio Operacional às Promotorias de Justiça de Fundações e Entidades de Interesse Social). Pelo apoio prestado à Universidade Federal de Pernambuco (UFPE), também é fiscalizada pela Controladoria Geral da União (CGU) e Tribunal de Contas da União (TCU), além dos órgãos financiadores dos projetos.

A Fade-UFPE tem como finalidade dar apoio a projetos de pesquisa, ensino, extensão e de desenvolvimento institucional, científico e tecnológico, de interesse das instituições federais de ensino superior (IFES) e também das instituições de pesquisa (ICT), o que a torna parte fundamental na implementação da Política de Ciência, Tecnologia e Inovação do País.

Os recursos administrados pela Fundação provêm dos contratos, convênios e acordos ou outros instrumentos jurídicos firmados com instituições privadas e/ou públicas para execução de atividades na área da pesquisa, do ensino e da extensão. Por meio de parcerias firmadas com a UFPE e com outras entidades, a fundação estabelece o elo entre essas instituições e os órgãos financiadores, promovendo benefícios para a sociedade em geral.

Parceira da UFPE desde 2002, a Motorola Mobility possui um laboratório de ponta no Centro de Informática dedicado a pesquisa, simulações e testes de redes de dados de 4G. Além disso, o Projeto CIn/Motorola oferece o Curso Sequencial de Formação Complementar, que tem como objetivo principal incentivar a formação de recursos humanos com alto grau de especialização em testes de *software* embarcado e aplicações em computação móvel, com os incentivos e benefícios previstos na Lei de Informática. A empresa conta com um quadro de cerca de 300 funcionários.

3 Desenvolvimento realizado na empresa

Nesta seção será apresentada a ferramenta desenvolvida como solução ao problema de solicitação de reteste no Projeto CIn/Motorola, abordando as técnicas, tecnologias utilizadas e contribuições para a empresa.

3.1 A problemática e a solução proposta

A criação de uma ferramenta capaz de identificar pedidos de reteste de maneira automática surge como alternativa promissora e eficiente para que estas CRs possam facilmente ser identificadas pelos *reporters* ou *watchers* das CRs, reduzindo o esforço diário com a leitura destes e-mails e tornando possível o aproveitamento destas horas em outras atividades relevantes no dia-a-dia do trabalho.

Assim que foi definida a temática do trabalho, o primeiro passo foi solicitar autorização à gerência e setores para realizá-lo, deixando claro todos os cuidados que seriam tomados com as informações e também os ganhos para o negócio. Só após este passo, foi inicializada a captura dos dados.

3.2 Obtenção de dados

Google BigQuery é o banco de dados feito para grandes volumes de dados que usa os recursos computacionais disponibilizado pela Google para realizar as operações em quantidades gigantes e obter um resultado em poucos segundos, sem a necessidade de grandes investimentos em hardware. Através do *BigQuery* foi possível recuperar todos os comentários das CRs utilizados para pré-processamento, treinamento e teste da ferramenta em poucos segundos, sem demandar uma máquina extremamente potente que fosse capaz de realizar esta tarefa de maneira satisfatória.

Desta forma, os comentários de algumas CR's abertas durante os ciclos de desenvolvimento de três versões diferentes do Android resultaram em mais de 9 milhões de linhas de texto, totalizando cerca de 583mb em volume de dados referente a mais de 410 mil comentários.

Ao gerar um nuvem de palavras (*wordcloud*) foi possível descobrir as palavras mais utilizadas nos comentários das CRs, como é possível visualizar na Figura 4. As palavras que aparecem em maior destaque, são os termos que aparecem com maior frequência nos comentários. Entre as sentenças que são observadas repetidamente se destacam as palavras: *issue*, *log*, *please*, *code* e *thanks*.

Figura 4: Nuvem de palavras com os termos mais utilizados nos comentários resgatados do Jira.
Fonte: Elaborada pelo autor.

Ao analisar os dados obtidos, foi identificado que cerca de 95% dos comentários não continha em seu conteúdo a palavra "retest", comumente utilizada pelos desenvolvedores ao solicitar um teste de confirmação por parte da equipe de qualidade.

3.3 Pré-processamento

De acordo com o contexto e as políticas da empresa, foi tomada a decisão de remover da base de dados utilizada para treinamento do modelo os comentários que continham as seguintes características:

- Informações confidenciais, como links e outros arquivos;
- Comentários automáticos de ferramentas internas;
- Análises de *logs*;

No conjunto do *corpus* (vocabulário do texto) algumas palavras podem se apresentar com muita frequência, porém apresentam baixa relevância para expressar o significado de um sentença. Essas palavras são chamadas de *stopwords* [8].

Comentários com menos de duas palavras, ou que ficaram vazios após a remoção de números, caracteres especiais, em chinês ou outro idioma diferente do inglês e *stopwords* foram descartados da base como forma de reduzir o esforço computacional e por não possuírem valor para a aplicação de *Machine Learning*. Entretanto, o caractere "~" (til) não foi removido quando seguido de alguma palavra, por exemplo: ~leuthier, pois trata-se da notação do Jira para os casos onde o escritor do comentário marca um outro usuário. Essa informação foi mantida com o intuito de utilizá-la como uma *feature* para treinamento do modelo.

3.4 Classificação manual

Respeitando a proporção identificada na etapa de Obtenção de dados (95% de comentários que não contém a palavra "retest" e apenas 5% que contém, o que indica um desbalanceamento das classes), foi extraída uma amostra aleatória de 1000 comentários para trabalhar como base de treinamento e teste, dado o prazo para desenvolvimento deste trabalho, sendo 950 comentários que não contém o termo"retest" e os outro 50 que possui essa palavra. Dentre esses mil comentários, 750 foram utilizados para treinamento e 250 para testes.

Nesta seção foram realizados experimentos utilizando algoritmos de aprendizado supervisionado. No aprendizado supervisionado, todo exemplo possui um atributo especial, o rótulo ou classe, que descreve o fenômeno de interesse, isto é, a meta que se deseja aprender e poder fazer previsões a respeito [9]. Desta forma, foi necessário classificar manualmente a base de dados.

Não é possível afirmar que todos os comentários que possuem a palavra "*retest*" são de fato pedidos de reteste. Então, para identificar se os comentários tinham ou não um pedido de reteste por parte do desenvolvedor, foram escolhidos 950 comentários aleatórios que não contém a palavra "*retest*" e outros 50 que contém e estes foram classificados manualmente, como pode ser observado na Figura 5.

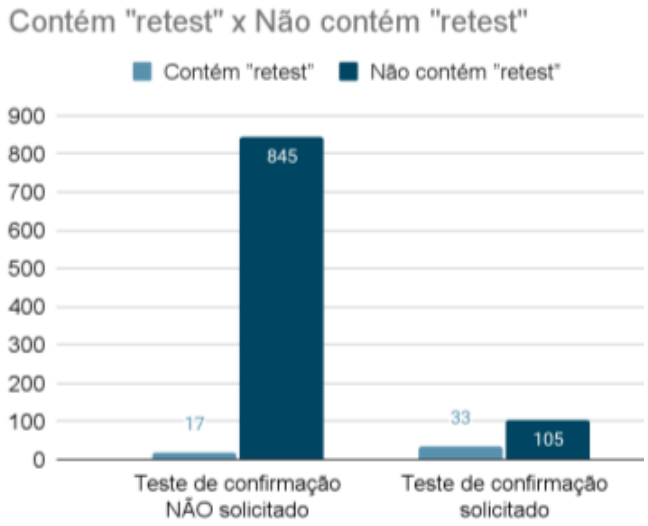

Figura 5: Proporção de comentários que contém a palavra “restest” e de comentários que não contém.

Fonte: Elaborada pelo autor

Dos 50 comentários que contém a palavra “*retest*”, 33 solicitavam pedidos de teste de confirmação e 17 não. Em contrapartida, dos 950 comentários que não contém a palavra “*retest*”, 105 solicitavam pedidos de teste de confirmação e 845 não. Ou seja, a base de dados pode ser classificada como não balanceada, conforme mostra a Tabela 1 pois possui 862 comentários que **NÃO** pedem reteste e 138 que pedem reteste.

	Contém ”retest”	**Não contém ”retest”**
Teste de confirmação solicitado	33	105
Teste de confirmação NÃO solicitado	17	845

Tabela 1: Análise descritiva sobre os dados utilizados para a classificação manual.

Fonte: Elaborada pelo autor

Também foi classificado nesta etapa se um comentário tinha marcação de algum usuário Jira e se era destinado a um colaborador da organização. Foi fornecida pela empresa uma lista com os nomes de usuários Jira de todos os colaboradores e através dela, realizada a verificação por meio do uso expressão regular (*regex*) para agilizar este processo.

3.5 Tokenização, Lematização e Vetorização

Utilizando a biblioteca NLTK foi realizada uma tokenização [10], onde cada comentário (frase) foi separada por palavras e posteriormente, cada palavra dos comentários foi transformada em uma coluna. O processo de lematização consiste no uso de um vocabulário e na análise morfológica das palavras, com o objetivo de remover apenas terminações que flexionam e assim devolver a forma base de uma palavra, que é conhecida como lema. Entretanto essa técnica envolve um alto custo computacional, na Figura 6 é possível visualizar um exemplo da aplicação da lematização em um comentário fictício.

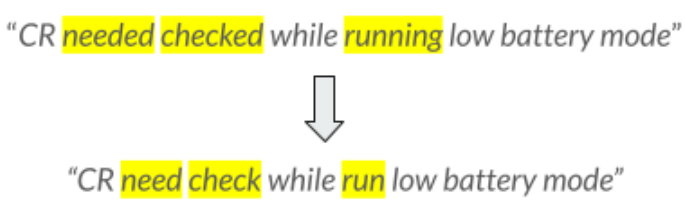

Figura 6: Exemplo de um comentário ao passar pelo processo de lematização.
Fonte: Elaborada pelo autor

Para aplicar técnicas de aprendizado de máquina em busca da solução ao problema proposto, é preciso determinar variáveis. Por se tratarem de comentários, as variáveis do nosso problema são chamadas variáveis categóricas ou nominais, pois contém valores em formato de texto ao invés de valores numéricos.

Existem algumas técnicas de pré-processamento para atribuir valores numéricos à essas palavras, a citar a *Label Enconding* e a *One-Hot Encoding*. Na técnica de *Label Encoding*, a classificação ocorre de acordo com a ordem alfabética da palavra. Caso essa técnica fosse aplicada ao nosso caso, haveria uma probabilidade muito alta de que o modelo capturasse a relação ordinal entre essas palavras. Isso é algo que não condiz com a realidade e poderia resultar em uma performance ruim para o modelo, portanto foi descartado o seu uso.

Foi utilizada então a técnica de *One-Hot Encoding*. Desta forma, as colunas foram classificadas binariamente, onde o número 1 representa a ocorrência daquela palavra no comentário analisado e o número 0 a não ocorrência. Para cada comentário na base, uma nova rodada de classificação era feita, ou seja, já que estamos trabalhando com 1000 comentários, foram necessárias 1000 rodadas classificatórias.

comentário1: "please retest this error ~victor"
comentário2: "hi ~ronaldo check this"

	Total de 3394 palavras únicas									Classificação manual		
	hi	~ronaldo	this	please	error	could	~victor	retest	check	RETEST_ REQUESTED	PESSOA_ MARCADA	PESSOA_ CIN
1000 comentários	0	0	1	1	1	0	1	1	0	1	1	1
	1	1	1	0	0	0	0	0	1	0	1	0
	...	...	...	...	...	...	...	...	...	...	...	...

Figura 7: Exemplo da base de dados antes de iniciar o processo de *machine learning*
Fonte: Elaborada pelo autor

Após concluída essas etapas para a criação do modelo, foi obtido um *dataframe* com cerca de 1001 linhas e 3397 colunas, uma vez que temos 3394 palavras únicas e mais três colunas referentes à classificação manual, como se pode observar na Figura 7 e entender o funcionamento desta estratégia com os dois comentários criados para demonstrar a prática. As linhas são os 1000 comentários selecionados na etapa de Obtenção de dados e mais uma linha com todas as colunas.

3.6 *Auto Machine Learning*

Segundo Balaji [11], o *Auto Machine Learning* (AutoML) serve como a ponte entre vários níveis de conhecimento ao projetar sistemas de aprendizado de máquina e agilizar o processo de ciência de dados. De acordo com os criadores do TPOT [12], esta biblioteca de código aberto (*open-source*), bem documentada e que constantes atualizações são feitas, criada em 2016 por pesquisadores da Universidade da Pensilvânia, automatiza a parte mais tediosa do aprendizado de máquina, explorando de maneira inteligente milhares de *pipelines* possíveis para encontrar a melhor *pipeline* de acordo com os dados inseridos utilizando programação genética para tal finalidade [12].TPOT é a abreviação para *Tree-based Pipeline Optimization Tool* ou em tradução literal algo como: "Ferramenta para otimização de *pipeline* baseada em árvores".

Neste trabalho foi utilizado o TPOT como ferramenta para identificar o modelo mais adequado para os tipos de dados apresentados e os parâmetros ideais para se obter o melhor modelo possível, além de seus hiperparâmetros para gerar um modelo capaz de predizer se um comentário é pedido de reteste ou não. É válido destacar que pode ser realizada como trabalho futuro uma comparação entre os algoritmos sugeridos através do TPOT, e outros *frameworks* de AutoML, como auto-sklearn[1] e Hyperopt[2] e suas métricas, com o intuito de avaliar se o resultado obtido durante trabalho de Balaji [11] é reproduzido também com a base de dados utilizada neste artigo.

Na Figura 8 é possível ter uma melhor visualização do que ocorre ao utilizar esta poderosa ferramenta. Devido a toda sua gama de tentativas e comparações é importante destacar que o tempo gasto até a definição de um modelo tido como bom ou satisfatório e seus parâmetros.

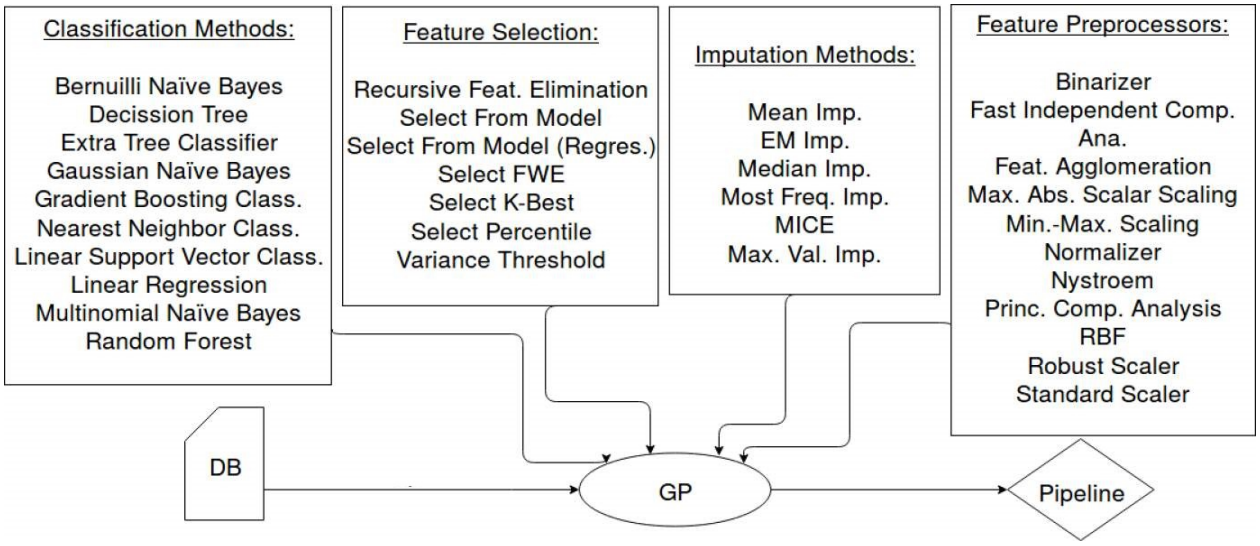

Figura 8: Fluxo de trabalho do TPOT
Fonte: Adaptado de Garcia, 2017[13]

O processo de criação de um modelo após os dados estarem prontos para serem trabalhados envolve um fluxo entre as etapas que é dito como uma "tubulação" onde cada estágio do modelo utiliza as informações que foram processadas em cada fase anterior, se assemelhando a um encanamento de uma casa onde normalmente tudo está interligado. Definir quais os melhores parâmetros para

[1] https://automl.github.io/auto-sklearn/master/
[2] https://github.com/hyperopt/hyperopt-sklearn/

um modelo não é uma tarefa simples ou fácil e dada a sua importância para a generalização do modelo seria uma tarefa que levaria muito tempo, entretanto isso ocorre de maneira automatizada ao se utilizar de ferramentas AutoML como o TPOT. Para o entendimento deste trabalho é importante definir o que é uma *pipeline* e que são os hiper-parâmetros.

- ***Pipeline***: é o agrupamento das etapas de pré-processamento e modelagem para que se possa fazer uso de todo o pacote configurável como se fosse uma única etapa. Dentre alguns dos benefícios é válido destacar: menos bugs, uma vez que menos parâmetros precisarão ser reescritos; torna mais fácil a aplicação de modelos tidos como protótipos ou *baselines* para a algo que pode ser implantado em escala.
- **Hiper-parâmetros** (*hyperparameters*): de acordo com Terra [14], são parâmetros que não são aprendidos diretamente pelos modelos, portanto precisam ser imputados antes do treinamento. No *Scikit-Learn*, eles são passados como argumentos para o construtor das classes dos modelos (chamados no contexto do scikit-learn de estimadores). Os valores escolhidos para os hiperparâmetros influenciam na qualidade dos modelos, na sua capacidade de aprender com o treinamento e de generalizar para dados não vistos anteriormente.

3.6.1 *Stratify*

Este parâmetro de estratificação faz uma divisão para que a proporção de valores na amostra produzida seja igual à proporção de valores fornecidos para estratificar o parâmetro, ou seja, preserva a proporção do alvo de acordo com o conjunto de dados original, nos conjuntos de dados de treinamento e teste também.

Por exemplo, se a variável y for uma variável categórica binária com valores 0 e 1, como é o caso neste trabalho, e houver 86% de zeros e 14% de uns, o parâmetro stratify=y do scikit-learn[3] garantirá que em sua divisão aleatória tenha 86% de 0's e 14% de 1's. Nesse contexto, estratificação significa que o método train_test_split retorna subconjuntos de treinamento e teste que têm as mesmas proporções de rótulos de classe que o conjunto de dados de entrada. Isso pode ser observado com clareza ao analisar a Figura 9.

Este parâmetro utilizado para eliminar o enviesamento de amostras num conjunto de testes, além de possibilitar a criação de um conjunto de teste com uma população que melhor representa toda a base de dados. A amostragem aleatória estratificada é diferente da amostragem aleatória simples, que envolve a seleção aleatória de dados de toda a população, para que cada amostra possível tenha a mesma probabilidade de ocorrer. Se ocorrer um enviesamento da amostragem ao construir um conjunto de testes, então ao testar um modelo de aprendizagem de máquina mostra que o modelo está obtendo um mau desempenho, uma vez que o conjunto de testes não representa toda a população. É importante destacar que o *stratify* não faz o balanceamento da base, embora mantenha a proporção nos subconjuntos. Para balancear a base pode-se utilizar em trabalhos futuros a técnica de *Synthetic Minority Over-sampling Technique* - SMOTE.

[3] https://scikit-learn.org/stable/

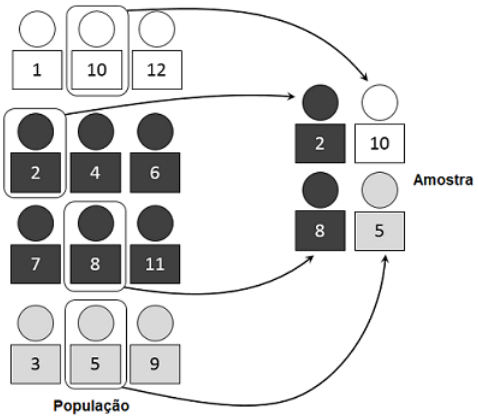

Figura 9: Exemplo do funcionamento do *stratify* ao utilizá-lo
Fonte: Traduzido de D. Kernler, 2014 [15]

A implementação do conceito de amostragem estratificada na validação cruzada garante que os conjuntos de treinamento e teste tenham a mesma proporção do recurso de interesse que no conjunto de dados original. Fazer isso com a variável de destino garante que o resultado da validação cruzada seja uma aproximação do erro de generalização.

3.6.2 Validação cruzada (*cross-validation*)

É comum o uso desta técnica em aprendizado de máquina para comparar e selecionar um modelo para um determinado problema de modelagem preditiva devido sua facilidade de implementação e compreensão. Uma das últimas etapas antes da criação de um modelo de *machine learning* é o processo de separação de parte dos dados para testes, e a outra parte restante utilizar para validação, uma vez que estes dados não foram usados para a criação do modelo e portanto são desconhecidos.

Para enfrentar o problema de *overfitting*, que ocorre quando o modelo está sobreajustado aos seus dados de treinamento, mas não se sai bem com dados desconhecidos é necessário aplicar a validação cruzada. Um desafio muito grande com o *overfitting*, e com a aprendizagem de máquina em geral, é que não podemos saber até que ponto o nosso modelo irá funcionar bem com dados desconhecidos até que o testemos de fato. Para resolver este problema, podemos dividir o nosso conjunto de dados iniciais em formações separadas e subconjuntos de testes. Existem diferentes tipos de técnicas de *cross-validation*, mas o conceito global permanece o mesmo: dividir os dados em vários subconjuntos, utilizar um conjunto de cada vez e treinar o modelo no conjunto restante, enquanto há um conjunto de teste em espera e assim repetir esse processo para cada subconjunto do conjunto de dados.

Dentre os métodos de validação cruzada existentes, vamos focar neste trabalho nos métodos K-fold.

- ***k-Fold***: o conjunto de treinamento é dividido em k conjuntos menores e então o procedimento da divisão entre treinamento e teste é seguido para cada uma das *k* “dobras”:
  - O modelo é treinado usando as dobras como dados de treinamento;

– O modelo resultante é validado na parte restante dos dados (um conjunto de teste é utilizado para calcular uma medida de desempenho, como precisão).

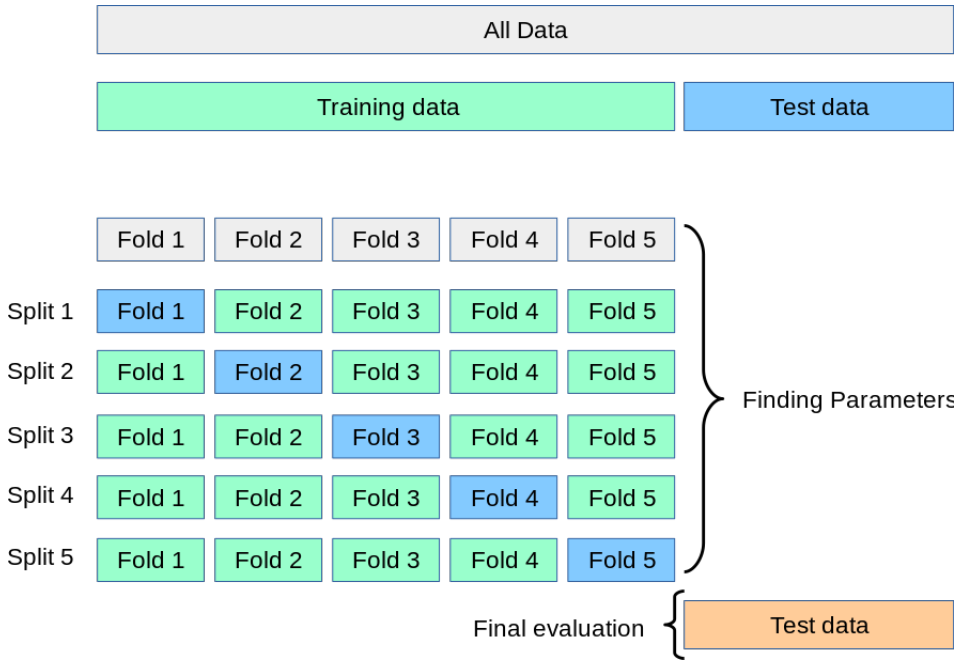

Figura 10: Funcionamento do k-fold
Fonte: Scikit-learn.org[16]

Se o k=10, o conjunto de dados será divido em 10 partes iguais, e conforme descrito na Figura 10 rodará 10 vezes, cada vez com uma parte do conjunto de dados diferente. Pode-se encontrar na literatura a orientação para a utilização de k=5 ou k=10, deve-se ter cuidado ao mudar este valor. A medida de desempenho relatada pela validação cruzada k-fold é então a média dos valores calculados no *loop*, ou seja cada *fold* gera uma espécie de mini modelo, que então é submetido a avaliação e realização de previsões utilizando este mini modelo gerado a partir dos folds remanescentes. Essa abordagem pode ser computacionalmente custosa.

De acordo com H. He e Y. Ma [17], uma validação cruzada de 10 vezes, em particular, o método de estimativa de erro mais comumente usado em aprendizado de máquina (k-fold), pode falhar facilmente no caso de desequilíbrios de classe, mesmo que o viés seja menos extremo do que o considerado anteriormente. Isto acontece pelo fato de que os dados são divididos com uma distribuição de probabilidade uniforme.

- ***StratifiedKFold***: é uma versão de validação cruzada *k-fold* que preserva a distribuição de classe desequilibrada em cada dobra para corresponder à distribuição no conjunto de dados de treinamento completo. Como se pode visualizar na Figura 11 o funcionamento para um k=5 com a classe M com uma quantidade consideravelmente maior de elementos que a classe F, mantém as mesmas proporções em cada dobra.

Figura 11: Funcionamento do *Stratified* k-fold
Fonte: Adaptado de M. Peng, 2015[18]

3.7 Experimentos

3.7.1 Métricas de avaliação do modelo

Baseando-se na separação dos dados entre treinamento e teste, é possível analisar algumas importantes informações acerca dos resultados obtidos, como por exemplo:

- **Verdadeiro positivo** (*True Positive* - TP): previsões acertadamente tidas como verdadeiras, no contexto trabalho é quando o modelo foi capaz de identificar um pedido de teste de confirmação;
- **Verdadeiro negativo** (*True Negative* - TN): previsões classificadas de maneira correta como não sendo uma solicitação de reteste;
- **Falso positivo** (*False Positive* - FP): previsões ditas como um pedido de teste de confirmação, embora, de fato, não tivesse havido essa solicitação;
- **Falso negativo** (*False Negative* - FN): previsões que foram classificadas como não sendo um pedido de reteste, apesar de ter ocorrido uma solicitação para que o reteste fosse feito.

Para problemas de classificação temos algumas das métricas mais comuns com a finalidade de garantir a qualidade do modelo criado e testado:

1. **Acurácia** (*accuracy*): é a proporção da contagem dos resultados verdadeiros em relação ao número total de casos. Sendo assim, uma métrica essencial e de fácil compreensão, adequada tanto para problemas de classificação binários como de classificação multi-classe, segundo Taylor [19]. É dada matematicamente por:

$$acuracia = \frac{numero\ de\ previsoes\ corretas}{numero\ total\ de\ previsoes} = \frac{TP + TN}{TP + TN + FP + FN}$$

No numerador, temos todos os exemplos que acertamos com nosso algoritmo. No denominador temos toda a nossa amostra.

Podemos utilizar a precisão como métrica quando o conjunto de dados está bem equilibrado e não é muito enviesado. Felizmente, existem outras métricas que podemos usar que são mais adequadas para outros tipos de amostras. A partir da acurácia não podemos afirmar quão boas são as previsões do modelo, uma vez que apenas dirá a probabilidade de previsões corretas do modelo acontecerem;

2. **Precisão** (*precision*): é a taxa de verdadeiro positivos em relação ao total dos positivos (verdadeiros e falsos) que foram preditos corretamente, segundo Powers [20]. É dada pela equação:

$$precisao = \frac{TP}{TP + FP}$$

3. **Sensibilidade** (*recall*): esta métrica é de grande relevância quando o objetivo do modelo é identificar um evento que ocorre com uma frequência menor que outro evento, ou seja, quando a base é desbalanceada. Através dela é mostrada a proporção em que o modelo está classificando com precisão os verdadeiros positivos [20]. Matematicamente demonstrada por:

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

4. **F1-score** (*F-measure*): é a média harmônica entre precisão e *recall*, obtendo valores entre 0 e 1. Quanto mais o valor se aproxima de 1, melhor é o modelo de aprendizado de máquina. Proporciona uma métrica de credibilidade mais elevada

$$f1 = \frac{2 * Precision * Recall}{Precision + Recall} = \frac{2 * TP}{2 * TP + FP + FN}$$

Esta métrica leva em consideração não apenas o número de erros de previsão que seu modelo comete, mas também os tipos de erros que foram cometidos.

5. **Matriz de confusão**: sintetiza os resultados de um problema de classificação através de uma imagem sucinta que nos permite avaliar o desempenho do modelo em questão, buscando compreender o vínculo entre acertos e erros apresenta.

		Valor Predito	
		Sim	**Não**
Real	**Sim**	Verdadeiro Positivo (TP)	Falso Negativo (FN)
	Não	Falso Positivo (FP)	Verdadeiro Negativo (TN)

Figura 12: Exemplo de matriz de confusão para um algoritmo de classificação binária
Fonte: D. Nogare, 2020 [21]

Apesar da utilização dessas cinco medidas de desempenho, a F1-score foi escolhida para ser utilizada como critério de comparação entre as *pipelines* geradas. Essa medida é a mais confiável para avaliar o desempenho real do modelo, uma vez que leva em consideração a média harmônica entre precisão e sensibilidade.

3.7.2 Utilizando *stratified k-fold* como técnica de validação cruzada

Após realizar todas as etapas descritas anteriormente neste trabalho, é necessário escrever o código em Python responsável por fazer uso do TPOT e assim testar as diversas *pipelines* e seus parâmetros, além de utilizar as técnicas descritas na seção de validação cruzada para obter uma performance melhor do classificador.

Listing 1: Parâmetros utilizados no TPOT, com validação cruzada *Stratifield k-fold*

```
1
2  X = comments.drop(columns=['RETEST_REQUESTED'], axis=1)
3  y = comments.RETEST_REQUESTED
4
5  X_train, X_test, y_train, y_test = train_test_split(X.astype(int), y.astype(int),
       train_size=0.75, test_size=0.25, random_state=42, stratify=y)
6
7  cv = StratifiedKFold(n_splits=10, random_state=1, shuffle=True)
8
9  tpot = TPOTClassifier(generations=5, population_size=50, verbosity=2, random_state=42,
       cv=cv, scoring='f1')
10
11 # Começou a montagem/busca do melhor algoritmo
12 tpot.fit(X_train, y_train)
13 print('Sessão de TPOT de treinamento/montagem concluída.')
14
15 # Obtenha a pontuação do TPOT no conjunto de teste (a métrica padrão é 'precisão', mas
       foi alterada para f1-score na linha 10)
16 print('Score do TPOT nos testes foi de...')
17 print(tpot.score(X_test, y_test))
18
19 # Exporta a melhor pipeline encontrada
20 tpot.export('all_columns_pipeline_StratifiedKFold.py')
21
22 y_pred = tpot.predict(X_test)
23 cf_matrix = confusion_matrix(y_test, y_pred)
24
25 print(r'Matriz de confusão\n', cf_matrix)
26 print(r'Precisão', precision_score(y_test, y_pred))
27 print('Recall', recall_score(y_test, y_pred))
28 print('F1-Score:', f1_score(y_test, y_pred))
29
30 print('\nMelhor pipeline:', end='\n')
31 for idx, (name, transform) in enumerate(tpot.fitted_pipeline_.steps, start=1):
32     print(f'{idx}. {transform}')
```

3.8 Resultados

3.8.1 Após cinco gerações

Após o TPOT finalizar sua execução, com 5 gerações e população de tamanho 50, algo que levou cerca de 1h 58m para ser concluída com os parâmetros acima, foi possível obter o seguinte classificador selecionado pelo TPOT:

```
GradientBoostingClassifier(learning_rate=0.5, max_depth=1,
                           max_features=0.35000000000000003,
                           min_samples_leaf=10, min_samples_split=14,
                           random_state=42, subsample=0.55)
```

Utilizando o *F1-score* como métrica de avaliação para a *pipeline* gerada pelo TPOT temos o seguinte resultado: $f1 = 0.545$

Na Tabela 3 é possível observar e comparar os resultados do modelo ao utilizar dados de treinamento e dados de testes para avaliação.

	Dados de treinamento	**Dados de teste**
Precisão	0.926	0.562
Recall	0.935	0.529
F1-score	0.930	0.545

Tabela 2: Tabela com as métricas de avaliação do modelo gerado pelo TPOT ao utilizar *Stratified k-fold* como técnica de validação cruzada com 5 gerações.

Fonte: Elaborada pelo autor.

Matriz de confusão

Na Figura 13 é possível visualizar a matriz de confusão gerada após a utilização do modelo com a melhor *pipeline* identificada através do uso do TPOT para as 5 gerações.

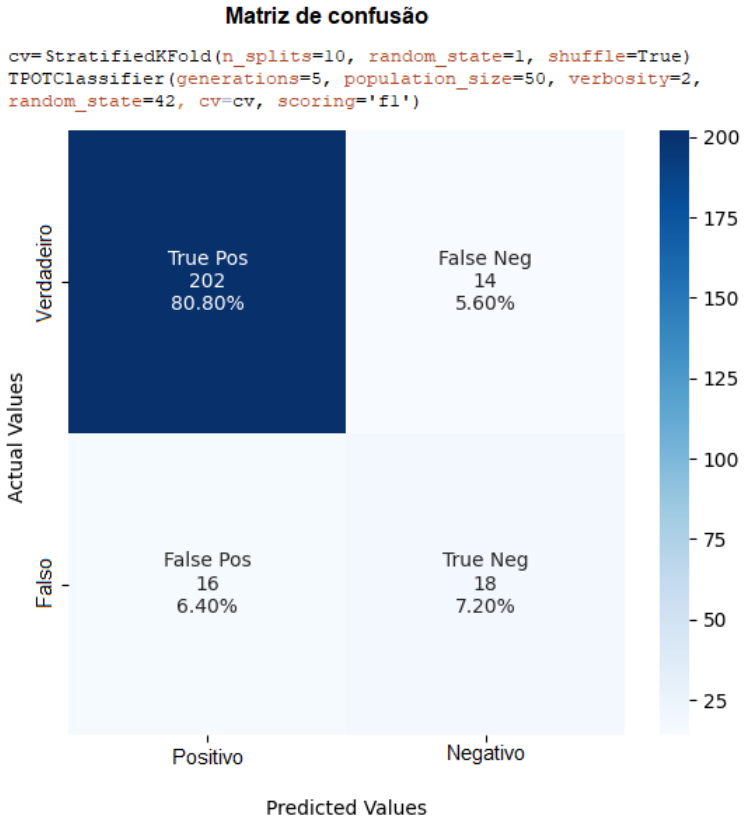

Figura 13: Matriz de confusão.
Fonte: Elaborada pelo autor

A precisão equilibrada (*balanced accuracy*) no TPOT foi descrita por Urbanowicz [22] como: a média de sensibilidade e especificidade é calculada para cada classe e, em seguida, calculada a média sobre o número total de classes. É diferente da precisão balanceada do *scikit-learn*. Na *pipeline* gerada, a *balanced accuracy* foi de 0.732.

3.8.2 Após vinte gerações

Após o TPOT finalizar sua execução, com 20 gerações e população de tamanho 50, algo que levou cerca de 9h 48m para ser concluída, foi possível obter o seguinte classificador selecionado pelo TPOT:

```
StackingEstimator(estimator=XGBClassifier(base_score=0.5, booster='gbtree',
                                          callbacks=None, colsample_bylevel=1,
                                          colsample_bynode=1,
                                          colsample_bytree=1,
                                          early_stopping_rounds=None,
                                          enable_categorical=False,
                                          eval_metric=None, gamma=0, gpu_id=-1,
                                          grow_policy='depthwise',
                                          importance_type=None,
                                          interaction_constraints='',
```

```
learning_rate=1.0, max_bin=256,
max_cat_to_onehot=4, max_delta_step=0,
max_depth=10, max_leaves=0,
min_child_weight=13, missing=nan,
monotone_constraints='()',
n_estimators=100, n_jobs=1,
num_parallel_tree=1, predictor='auto',
random_state=42, reg_alpha=0,
reg_lambda=1, ...))
```

Utilizando o *F1-score* como métrica de avaliação para a *pipeline* gerada pelo TPOT temos o seguinte resultado: $f1 = 0.581$

Na Tabela 3 é possível observar e comparar os resultados do modelo ao utilizar dados de treinamento e dados de testes para avaliação.

	Dados de treinamento	**Dados de teste**
Precisão	0.954	0.480
Recall	0.875	0.735
F1-score	0.913	0.581

Tabela 3: Tabela com as métricas de avaliação do modelo gerado pelo TPOT ao utilizar *Stratified k-fold* como técnica de validação cruzada com 20 gerações.
Fonte: Elaborada pelo autor.

Matriz de confusão

Na Figura 14 é possível visualizar a matriz de confusão gerada após a utilização do modelo com a melhor *pipeline* identificada através do uso do TPOT.

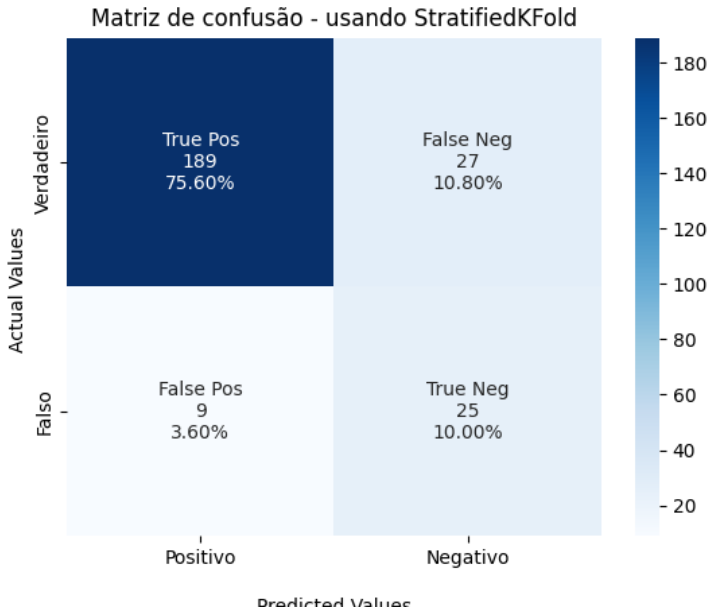

Figura 14: Matriz de confusão.
Fonte: Elaborada pelo autor

Na *pipeline* gerada após as 20 gerações, a *balanced accuracy* foi de 0.805.

Ao utilizar o TPOT, com o parâmetro de *stratify*, para manter a proporção das classes entre pedidos de reteste e não pedido de reteste para o treinamento do modelo, em conjunto com técnicas de validação cruzada, como *stratified k-fold* foi possível criar um modelo de machine learning que fosse capaz de identificar de maneira correta uma solicitação de reteste com uma precisão de cerca de 56.2% para 5 gerações e de 48.0% após 20 gerações, entretanto o f1-score após 20 gerações foi de 0.581 enquanto que com 5 gerações ficou em torno de 0.545, demonstrando que de fato um número maior de gerações fez o classificador obter uma melhor performance, pois a variação dos indivíduos acaba sendo mais elevada. O *f1-score* é uma métrica mais precisa da performance do modelo do que apenas a precisão, uma vez que o *f1-score* leva em consideração a precisão e o *recall* (sensibilidade) pois a ideia do classificador deste trabalho é conseguir identificar o pedido de reteste e este evento ocorre com uma frequência menor. Através da sensibilidade é mostrada a proporção em que o modelo está classifi- cando com precisão os verdadeiros positivos, ou seja, ao utilizar o TPOT com 20 gerações conseguimos aumentar a taxa de acerto para a identificação correta de um pedido de reteste para 73.5% ao invés dos 52.9% com 5 gerações.

3.9 Tecnologias utilizadas

A linguagem de programação *Python* foi escolhida para a elaboração deste trabalho devido a sua grande relevância em áreas como análise de dados, *Machine Learning* através de bibliotecas e por sua curva de aprendizado ser menor em comparação com outras linguagens existentes atualmente no mercado de tecnologia. Entre as principais bibliotecas utilizadas neste projeto destaca-se:

- Pandas[4]: torna o trabalho com dados mais fácil e intuitivo, fazendo uso de sintaxe de alto nível para fazer análises e manipulação dos dados de maneira prática;
- NumPy[5]: está no centro dos ecossistemas científicos para *Python* e adiciona poderosas estruturas de dados a esta linguagem que garantem cálculos eficientes com *arrays* e matrizes e fornece uma enorme biblioteca de funções matemáticas de alto nível. É amplamente utilizada em Pandas, SciPy, Matplotlib, scikit-learn e na maioria dos outros pacotes de ciência de dados;
- MatplotLib[6]: utilizada para criar visualizações estáticas, animadas e interativas. Segundo sua própria descrição, o Matplotlib torna as coisas simples fáceis e as difíceis possíveis;
- Scikit-learn[7]: desenvolvida especificamente para aplicação prática de *machine learning*. Esta biblioteca dispõe de ferramentas simples e eficientes para análise preditiva de dados, principalmente por utilizar outras bibliotecas eficientes (NumPy, SciPy, e matplotlib) nas suas respectivas propostas de uso;
- *Natural Language Toolkit*[8]: uma das principais bibliotecas de Processamento de Linguagem Natural (PLN ou NLP em inglês), foi amplamente pela comunidade graças ao trabalho muito

[4]https://pypi.org/project/pandas/
[5]https://numpy.org/doc/stable/index.html
[6]https://matplotlib.org/
[7]https://scikit-learn.org/stable/
[8]https://www.nltk.org/

bem feito acerca das documentações. O NLTK é adequado para linguistas, engenheiros, estudantes, educadores, pesquisadores e usuários do setor de tecnologia, em geral;

- WordCloud[9]: é utilizada neste trabalho com a finalidade de representar visualmente a frequência de cada palavra em toda a base de dados analisada. Através dessa biblioteca foi criada a Figura 4 para visualizar os termos mais frequentes na base de dados utilizada.

3.10 Contribuição

O auxílio deste estudo para o Projeto CIn/Motorola está na criação de um modelo de machine learning a ser usado para a criação da ferramenta "*Pay Attention To Retest*" - PATRE. Há a expectativa de gerar grande impacto no dia-a-dia dos desenvolvedores, testadores e líderes de equipes ao reduzir a necessidade da comunicação síncrona entre esses pares. Para a gestão, a aplicação efetiva de um mecanismo como esse se faz necessário para ter uma melhor visão do processo necessário para a realização dos pedidos de reteste, do processo necessário para a realização dos pedidos de reteste, fazendo com que se tenha uma estimativa mais precisa das atividades que o time pode assumir naquela *sprint*. Também é possível a criação de *dashboards* no Jira apenas com erros que foram solicitados no reteste e através deles extrair valor ao conseguir visualizar dados que hoje não estão disponíveis.

Uma maneira de identificar pedidos de teste de confirmação através dos comentários nas CRs é a etapa inicial para a automação deste processo utilizando os dispositivos disponíveis nos servidores da companhia.

4 Dificuldades encontradas

Uma das maiores dificuldades encontradas ao longo do desenvolvimento deste estudo foi o fato de trabalhar com informações extremamente sigilosas. É necessário muito mais responsabilidade e cuidado com o tratamento dos dados utilizados, desde a escolha do ambiente de versionamento de código utilizado até a questão dessas informações não poderem ser distribuídas para um computador pessoal.

O uso de *Machine Learning* em uma base de dados extensa demanda um poder computacional relativamente alto, o que consequentemente faz com que haja a necessidade de uma máquina com processador melhor e maior quantidade de memória RAM. Pelos motivos citados, todo o trabalho foi realizado utilizando um computador da organização.

Isto impactou diretamente na quantidade de gerações utilizadas para encontrar o melhor classificador, com 5 gerações e população de 50 demorou cerca de 2 horas, enquanto que para 20 gerações e a mesma população levou cerca de 9 horas e 48 minutos para a execução do TPOT ser concluída. Como o *f1-score* e as demais métricas tem grande chance de melhorar após mais gerações e uma população maior, isso foi um fator determinante para o desenvolvimento deste trabalho, uma vez que poderia fazer uso de computação em nuvem com o uso do Google Colab[10].

[9] https://amueller.github.io/word_cloud/

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

A grande quantidade de comentários em chinês, comentários adicionados automaticamente por ferramentas internas, links diversos e erros gramaticais fizeram com que a fase de pré-processamento demandasse um esforço e tempo considerável para o desenvolvimento deste trabalho.

5 Impactos da formação no seu trabalho

O curso de Bacharelado em Sistemas de Informação tem a expertise de formar alunos com um perfil multidisciplinar e preparado para o mercado de TI. Nesta sessão serão exploradas as disciplinas curriculares do curso que foram de extrema importância na vivência profissional.

A linguagem de programação Python, abordada em várias disciplinas durante o curso, é utilizada principalmente para automatizar tarefas repetitivas através de diversos *scripts* com finalidade de usar o tempo ganho em outras atividades que necessitem de atenção dentro da organização.

A Metodologia ágil também é algo que está presente no meu cotidiano, desde a forma de trabalhar com entregas por *sprints* semanais até as cerimônias de *planning* e *sprint review*.

A Estatística é fundamental para analisar corretamente os dados coletados ao longo de todo o processo de desenvolvimento de um produto. Utilizada, por exemplo, quando se faz necessário saber qual produto está apresentando mais defeitos em um determinado período, ou até mesmo em que área os novos defeitos tem tido mais impacto (câmera, *wi-fi*, etc.). Na construção de relatórios e entregas também é utilizada ao apresentar dados, criar gráficos e filtros, sendo dessa maneira possível fornecer informações com mais propriedade e que agregam mais valor para a gestão.

O processo de entregas e prazos ao desenvolver os projetos das disciplinas e os trabalhos feitos em grupo, foram essenciais para a postura enquanto profissional de buscar sempre cumprir os prazos estabelecidos e ter responsabilidade com os processos e informações confiados ao autor deste trabalho, sempre mantendo uma relação saudável e respeitosa com todas as partes envolvidas nas atividades.

6 Conclusão

Este trabalho teve como principal objetivo o desenvolvimento de uma ferramenta (PATRE, uma sigla para *Pay attention to retest*) capaz de identificar pedidos de reteste, onde o usuário, seja ele líder da equipe de teste ou analista de qualidade, conseguisse realizar essa análise de maneira eficiente.

Através do que foi demonstrado na seção de experimentos, foi possível concluir que o uso de uso de *Machine Learning* integrado ao Jira é de grande relevância, visto que após todo o processo de coleta, pré-processamento e análise dos dados, foi possível criar uma ferramenta de fácil entendimento e interação, facilitando o processo de identificação de pedido de reteste e substituindo métodos com menos eficiência, como a leitura de e-mails, por exemplo, além de ser de grande ajuda no dia-a-dia de uma equipe que lida com uma grande quantidade de defeitos abertos e que será necessário monitorar o andamento desses defeitos e verificação posterior se aquele defeito ainda ocorre ou não.

Uma particularidade deste trabalho é que um número maior de falsos positivos é pior do que falso

negativo, uma vez que caso o classificador identifique um pedido de reteste de maneira errada, ou seja, não deveria ser considerado reteste, um testador irá abrir a CR a fim de realizar o teste de confirmação, embora não seja necessário. Apesar da pessoa conseguir identificar que não é um pedido de reteste, isso vai ter tomado tempo para analisar e entender o contexto da CR. Enquanto que ao obter falsos negativos, o desenvolvedor não obtendo uma resposta por parte do testador, certamente vai entrar em contato solicitando que o reteste seja feito para que a falha encontrada seja solucionada, permanecendo com o problema atual de postergação da correção. É válido destacar que além de necessitar da atenção do testador ao ter como resultado da classificação um falso positivo, isso gera ao modelo um certo grau de desconfiança por parte dos usuários da ferramenta de maneira geral, entre testadores, líderes de equipes e gerência.

Ao utilizar o TPOT, com os parâmetros de *stratify* em conjunto com técnicas de validação cruzada, como *stratified k-fold* foi possível criar um modelo de machine learning que fosse capaz de identificar de maneira correta uma solicitação de reteste com uma precisão de cerca de 56.2% para 5 gerações e de 48.0% após 20 gerações, entretanto o *f1-score* após 20 gerações foi de 0.581 enquanto que com 5 gerações ficou em torno de 0.545, demonstrando que de fato um número maior de gerações faz o classificador obter uma melhor performance, pois a variação dos indivíduos acaba sendo mais elevada.

Apesar dos fatores positivos do uso da ferramenta, notou-se durante a implementação do PATRE, uma necessidade de identificar pedidos de reteste de maneira mais complexa e menos inflexível, visto que os mecanismos criados e utilizados oferece limitações nesse processo de identificação. Aplicação de *Deep learning* talvez consiga criar modelos mais generalistas ao se tratar da identificação destes pedidos de teste de confirmação.

No que se refere às melhorias e trabalhos futuros, há alguns pontos a serem implementados de modo a complementar o uso do PATRE:

1. Implementação de um campo para melhoramento do modelo treinado de predição de pedidos de reteste, onde solicitaria ao usuário um *feedback* sobre a CR listada, a fim de penalizar o algoritmo caso seja um defeito falso positivo para a solicitação de reteste;

2. Implementação de API ligada ao BigQuery a fim de monitorar apenas as CRs em que a pessoa em questão reportou ou está de *watcher*, sendo capaz de mostrar ao usuário ou líder do time uma lista de defeitos que necessitam de atenção naquele dia, sem a necessidade da leitura de e-mails;

3. Ao invés de utilizar o PATRE de maneira individualizada, implementar a integração com uma das ferramentas já existentes na organização, de maneira a tornar possível a identificação de um pedido de reteste sem um esforço maior de configuração de ambiente, além desta ferramenta já ser distribuída via *PyPi packages*, facilitando todo a implantação para os usuários;

4. Implementação de uma nova ferramenta, capaz de iniciar o processo de reteste de maneira automatizada, de acordo com as configurações solicitadas identificadas no defeito cadastrado (produto, versão de software e/ou pré condição específica). A fim de otimizar o dia-a-dia dos testadores ao focar em uma atividade específica em que se faz necessária a análise de um humano. Além de otimizar o uso de aparelhos celulares ociosos durante os intervalos de execução das diversas suítes de testes;

5. Possibilidade de criação de um *plugin* a ser disponibilizado no *marketplace* do Jira capaz de informar ao usuário quando um pedido de reteste fosse solicitado diretamente através da plataforma, sem a necessidade de verificação da caixa de e-mail, ou instalação e manutenção de uma nova ferramenta na organização;

6. Verificar a oportunidade de adicionar uma nova *label* à CR identificando que aquele defeito necessita de atenção por motivos de solicitação de reteste ou até mesmo a criação de um novo campo personalizado informando que esta CR passou pelo processo de teste de confirmação, a fim de melhorar filtros utilizados em *dashboards* facilitando a visualização destas métricas;

7. Testar um algoritmo com uma base de dados balanceada, ou seja, ter a mesma quantidade de comentários que são pedidos de teste de confirmação e que não são, aplicando técnicas como SMOTE. Pode-se ainda criar um classificador super especializado na classe de reteste buscando otimizar os resultados obtidos neste trabalho;

8. Utilizar o Auto-sklearn e comparar os resultados obtidos com o TPOT, para verificar se obtemos o mesmo resultado que o trabalho de Balaji [11], uma vez que TPOT se mostrou melhor para problemas de regressão, enquanto auto-sklearn se mostrou melhor para problemas de classificação.

Referências Bibliográficas

[1] M. Delamaro, M. Jino, and J. Maldonado, *Introdução ao teste de software*. Elsevier Brasil, 2013.

[2] R. Pressman and B. Maxim, *Engenharia de Software - 8ª Edição*. McGraw Hill Brasil, 2016. [Online]. Available: https://books.google.com.br/books?id=wexzCwAAQBAJ

[3] D. A. Garvin and W. D. Quality, "Really mean," *Sloan management review*, vol. 25, pp. 25–43, 1984.

[4] M. J. Harrold and A. Orso, "Retesting software during development and maintenance," in *2008 Frontiers of Software Maintenance*, 2008, pp. 99–108.

[5] S. Fogel, "Why software bugs are like mini outages," Apr 2021. [Online]. Available: https://www.forbes.com/sites/forbestechcouncil/2021/04/26/why-software-bugs-are-like-mini-outages/

[6] B. BSTQB, "Certified tester foundation level syllabus," *Sl: sn*, 2018.

[7] I. Sommerville *et al.*, "Engenharia de software.[sl]," *Pearson Education*, vol. 19, p. 23, 2011.

[8] H. Lane, C. Howard, and H. Hapke, *Natural Language Processing in Action Video Edition*. Manning Publications, 2019.

[9] S. O. Rezende, *Sistemas inteligentes: fundamentos e aplicações*. Editora Manole Ltda, 2003.

[10] J. Perkins, *Python text processing with NLTK 2.0 cookbook*. PACKT publishing, 2010.

[11] A. Balaji and A. Allen, "Benchmarking automatic machine learning frameworks," *arXiv preprint arXiv:1808.06492*, 2018.

[12] R. S. Olson, N. Bartley, R. J. Urbanowicz, and J. H. Moore, “Evaluation of a tree-based pipeline optimization tool for automating data science,” in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, ser. GECCO ’16. New York, NY, USA: ACM, 2016, pp. 485–492. [Online]. Available: http://doi.acm.org/10.1145/2908812.2908918

[13] U. Garciarena, R. Santana, and A. Mendiburu, “Evolving imputation strategies for missing data in classification problems with tpot,” *arXiv preprint arXiv:1706.01120*, 2017.

[14] K. Terra, “Ajuste de hiperparâmetros em modelos de machine learning,” Jun 2021, [Online; Acessado 26-Maio-2022]. [Online]. Available: https://medium.com/programacaodinamica/ajuste-de-hiperpar%C3%A2metros-em-modelos-de-machine-learning-946dec10f90b

[15] D. Kernler, “Stratified sampling,” https://commons.wikimedia.org/wiki/File:Stratified_sampling.PNG, 2014, [Online; Acessado 08-Junho-2022].

[16] “Scikit-learn cross-validation: Evaluating estimator performance.” [Online]. Available: https://scikit-learn.org/stable/modules/cross_validation.html

[17] H. He and Y. Ma, “Imbalanced learning: foundations, algorithms, and applications,” 2013.

[18] M. Peng, “General tips for participating kaggle competitions,” https://www.slideshare.net/markpeng/general-tips-for-participating-kaggle-competitions, 2015, [Online; Acessado 04-Junho-2022].

[19] J. Taylor and S. Taylor, *Introduction To Error Analysis: The Study of Uncertainties in Physical Measurements*, ser. ASMSU/Spartans.4.Spartans Textbook. University Science Books, 1997. [Online]. Available: https://books.google.com.br/books?id=giFQcZub80oC

[20] D. M. Powers, “Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation,” *arXiv preprint arXiv:2010.16061*, 2020.

[21] D. Nogare, “Performance de machine learning - matriz de confusão,” Apr 2020, [Online; Acessado 27-Maio-2022]. [Online]. Available: https://diegonogare.net/2020/04/performance-de-machine-learning-matriz-de-confusao/

[22] R. J. Urbanowicz and J. H. Moore, “Exstracs 2.0: description and evaluation of a scalable learning classifier system,” *Evolutionary intelligence*, vol. 8, no. 2, pp. 89–116, 2015.