UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS DOIS VIZINHOS
CURSO DE ESPECIALIZAÇÃO EM CIÊNCIA DE DADOS

BRENDA SABRINA COPATTI

PRÉ-PROCESSAMENTO DE DADOS DE TEXTO DE REQUISITOS DE SOFTWARE UTILIZANDO TÉCNICAS DE APRENDIZADO DE MÁQUINA

TRABALHO DE CONCLUSÃO DE CURSO DE ESPECIALIZAÇÃO

DOIS VIZINHOS
2022

BRENDA SABRINA COPATTI

PRÉ-PROCESSAMENTO DE DADOS DE TEXTO DE REQUISITOS DE SOFTWARE UTILIZANDO TÉCNICAS DE APRENDIZADO DE MÁQUINA

Trabalho de Conclusão de Curso de Especialização apresentado ao Curso de Especialização em Ciência de Dados da Universidade Tecnológica Federal do Paraná, como requisito para a obtenção do título de Especialista em Ciência de Dados.

Orientador: Prof. Dr. Dalcimar Casanova
Coorientadora: Prof.ª Eliane de Bortoli Fávero

DOIS VIZINHOS
2022

BRENDA SABRINA COPATTI

PRÉ-PROCESSAMENTO DE DADOS DE TEXTO DE REQUISITOS DE SOFTWARE UTILIZANDO TÉCNICAS DE APRENDIZADO DE MÁQUINA

Trabalho de Conclusão de Curso de Especialização apresentado ao Curso de Especialização em Ciência de Dados da Universidade Tecnológica Federal do Paraná, como requisito para a obtenção do título de Especialista em Ciência de Dados.

Data de aprovação: 11/novembro/2022

Dalcimar Casanova
Doutorado
Universidade Tecnológica Federal do Paraná - Câmpus Pato Branco

Eliane Maria De Bortoli Fávero
Doutorado
Universidade Tecnológica Federal do Paraná - Câmpus Pato Branco

Jefferson Tales Oliva
Doutorado
Universidade Tecnológica Federal do Paraná - Câmpus Pato Branco

Ives Renê Venturini Pola
Doutorado
Universidade Tecnológica Federal do Paraná - Câmpus Pato Branco

DOIS VIZINHOS
2022

RESUMO

A execução de algoritmos de aprendizado de máquina aplicando processamento de linguagem natural vem auxiliando os seres humanos e automatizando muitas tarefas do dia a dia. Contudo, uma das etapas que mais consome tempo no processo de implementação é o tratamento do *dataset* para que fique ideal para o treinamento do algoritmo, o que acontece por conta da impossibilidade de existir um padrão de digitação ou de estrura dos dados, fazendo com que os textos disponíveis para as pesquisa contenham muitos dados ruidosos. Diante disso, o presente estudo se dedica à execução de técnicas de pré-processamento de dados de texto utilizando expressões regulares, para tratamento de ruídos que possuem um padrão de escrita e técnicas de aprendizado de máquina para buscar soluções para substituição de ruído em textos que não possuem um padrão de escrita, com intuito de agilizar o pré-processamento dos dados utilizados para a implementação de algoritmos de processamento de linguagem natutal.

Palavras-chave: Aprendizado de Máquina; Pré-processamento de Textos; Processamento de Linguagem Natural; PLN.

ABSTRACT

The execution of machine learning algorithms applying natural language processing has been helping humans and automating many everyday tasks. However, one of the most time-consuming steps in the implementation process is the treatment of dataset so that it is ideal for training the algorithm, which happens due to the impossibility of having a typing pattern or data structure, causing the texts available for searches to contain a lot of noisy data. Therefore, the present study is dedicated to the execution of techniques for pre-processing text data using regular expressions, to treat noises that have a writing pattern and machine learning techniques to seek solutions to replace noise in texts that they do not have a writing standard, in order to speed up the pre-processing of the data used for the implementation of natural language processing algorithms.

Keywords: Machine Learning; Text pre-processing; Natural Language Processing; NLP.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

PLN	Processamento de linguagem natural
URL	*Uniform Resource Locator*
IDE	*Integrated Development Environment*
BERT	*Bidirectional Encoder Representations from Transformers*
IA	Inteligência Artificial
CSV	*Comma-separated values*
IP	*Internet Protocol*
HTML	*HyperText Markup Language*

SUMÁRIO

1 INTRODUÇÃO

O uso de métodos de inteligência artificial (IA) para automatizar processos que antes só podiam ser realizados por seres humanos vem crescendo a cada dia. Atualmente, uma das áreas bastante exploradas em aplicações de IA, tem sido a possibilidade da máquina interpretar textos com a linguagem humana, técnica conhecida como Processamento de Linguagem Natural (PLN).

Um dos principais objetivos, e que tem sido um desafio para a área de PLN, é a capacidade da máquina de analizar o contexto no qual as palavras do texto estão inseridas, e não apenas o sentido literal delas, simulando da maneira mais aproximada possível, uma interpretação realizada por um ser humano. Tarefas como *chatbots* (AHMAD et al., 2018), tradução (KHAN; ABID; ABID, 2020), detecção de *spam* (CHOWDHURY et al., 2020), análise de sentimentos (HASAN; MALIHA; ARIFUZZAMAN, 2019), (RAMACHANDRAN; PARVATHI, 2019), são exemplos de soluções que usam de técnicas de PLN para a interpretação de dados textuais recebidos como entrada.

Entretanto, uma etapa muito importante do PLN e que pode impactar positiva ou negativamente nos resultados obtidos na implementação de soluções é o pré-processamento dos dados textuais. Por se tratarem de um tipo de dados não estruturado, os textos estão sujeitos a apresentar ruídos, o que prejudica a interpretação realizada pela máquina. É comum serem encontrados muitos ruídos em dados textuais disponibilizados para treinamentos de métodos de aprendizado em PLN, tais como: símbolos, caracteres especiais, endereços da web, endereços de IP, trechos de códigos de programação, entre outras palavras que não se encaixam no contexto.

Esses ruídos nos dados, muitas vezes não apresentam um padrão de escrita, tornando necessário o uso de técnicas diferentes para resolver cada situação, e então, realizar a identificação dos trechos inválidos com mais eficiência.

Alguns estudos foram realizados objetivando desenvolver técnicas específicas para o pré-processamento de dados textuais de fontes específicas, em busca de melhorias nos resultados de métodos de aprendizado com base nesses dados .

Ao analisar as características de dados textuais de requisitos de *software* (ex. histórias de usuário), coletados de diferentes projetos de código aberto, a fim de formar um *dataset* voltado à realização de estudos e experimentos em engenharia de *software*, é possível encontrar diversos dos ruídos já citados, o que prejudica a identificação correta da semântica das palavras, compromentendo também a identificação do contexto dos requisitos. Sendo assim, buscando mitigar/resolver esse problema, esse trabalho busca implementar técnicas para o pré-processamento de dados textuais, visando remover ruídos que contenham ou não uma estrutura padrão de escrita, a fim de contribuir para o desenvolvimento de soluções de PLN para a área de engenharia de *software*.

1.1 Problema de Pesquisa

Ao analisar as características de dados textuais de requisitos de *software* (ex. histórias de usuário), coletados de diferentes projetos de código aberto (CHOETKIERTIKUL et al., 2018), a fim de formar um *dataset* voltado à realização de estudos e experimentos em engenharia de *software*, é possível encontrar diversos dos ruídos já citados (ex. trechos de código, endereços IP, endereços da web, entre outros), o que prejudica a identificação correta da semântica das palavras, compromentendo também a identificação do contexto dos requisitos.

Sendo assim, buscando mitigar/resolver esse problema, esse trabalho busca implementar técnicas para o pré-processamento de dados textuais, visando remover ruídos que contenham ou não uma estrutura padrão de escrita, a fim de contribuir para o desenvolvimento de soluções de PLN para a área de engenharia de *software*, como por exemplo, em métodos inteligentes para estimativa de esforço de desenvolvimento de *software* (FÁVERO; CASANOVA; PIMENTEL, 2022).

1.2 Objetivos

1.2.1 Objetivo Geral

Realizar o pré-processamento inteligente de textos de requisitos de *software* fazendo uso de métodos de PLN.

1.2.2 Objetivos Específicos

- Realizar um estudo de métodos e técnicas que podem ser utilizadas para realizar a substituição de palavras inválidas.
- Implementar métodos para substituição de elementos indesejados no texto (ex. *tags* HTML, URL's, entre outros).
- Estudar e implementar métodos de PLN para auxiliar no pré-processamento de textos de forma inteligente;
- Realizar a validação do modelo proposto.

1.3 Justificativa

É notável o número de soluções que podem ser desenvolvidas para auxiliar ou até mesmo substituir o trabalho humano utilizando PLN (ex. tradução, perguntas e respostas, identificação de entidade nomeada, entre outros). Todas essas aplicações são possíveis por conta da implementação de conceitos de PLN que tornam o dia a dia muito mais prático, por realizarem processos de análise de dados que seriam massantes ou até impossíveis de serem realizados por seres humanos.

Um dos maiores desafios encontrados por pesquisadores ao propor soluções fazendo uso de PLN, é a disponibilidade de um *dataset* de dados textuais volumoso o suficiente, e

dentro de uma área específica (ex. saúde, engenharia de *software*). Quando encontrados, ou construídos, esses *datasets* apresentam problemas e consequentemente baixo desempenho quando aplicados a algoritmos de aprendizado. Esses problemas estão relacionados ao formato desses textos, mas mais fortemente ao número de ruídos encontrados. Esses ruídos dificultam a interpretação pela máquina. Um exemplo, é o uso de frases informais em textos que são geralmente encontrados em mídias sociais (RAMACHANDRAN; PARVATHI, 2019), as quais geralmente apresentam caracteres especiais, endereços da web, códigos, entre outros.

Essa mesma problemática é encontrada em textos de requisitos de *software*, como os utilizados no trabalho de Choetkiertikul et al. (2018) e Fávero, Casanova e Pimentel (2022).

Visando contribuir com a limpeza e pré-processamento dos dados de texto de requisito de *software*, utilizando uma base altamente populada com textos específicos da área. surge o interesse de buscar métodos adequados de pré-processamento que auxiliem na remoção de palavras ou partes do texto que não contribuam positivamente com métodos de PLN.

1.4 Organização do Trabalho

Este texto está organizado em capítulos. O Capítulo 2 apresenta conceitos sobre PLN. No Capítulo 3 estão os materiais e o método para a realização do trabalho. Os resultados obtidos são apresentados no Capítulo 4. No Capítulo 5 está a conclusão obtida após a análise dos resultados.

2 PROCESSAMENTO DE LINGUAGEM NATURAL E TÉCNICAS DE PRÉ-PROCESSAMENTO

Esse capítulo objetiva apresentar alguns conceitos acerca de Processamento de Linguagem Natural (PLN), com foco especial na etapa de pré-processamento. Também serão apresentadas técnicas de pré-processamento e o modelo de linguagem pré-treinado utilizado na implementação de uma técnica de pré-processamento inteligente.

2.1 Processamento de Linguagem Natural

De fato a comunicação é um dos processos mais importantes do cotidiano dos seres humanos. A ascensão da tecnologia tornou mais comum o fato de muitas dessas informações estarem registradas no formato textual, e disponíveis para uso a qualquer momento. Tendo em vista a ampla quantidade de conteúdo textual disponível em redes sociais, aplicativos de bate papo, sites de busca, ou em qualquer outro meio de armazenamento de dados, notou-se a necessidade de desenvolvimento de uma forma de interpretação desses dados por parte dos computadores, a fim de automatizar muitas tarefas.

O PLN, é uma subárea da IA que busca desenvolver a habilidade dos computadores de interpretar a linguagem humana. Diferente da linguagem de programação, o PLN busca extrair características e entender o contexto do texto sendo analisado. Com isso, o principal objetivo é o de obter conclusões simulando o pensamento humano. Alguns exemplos de atividades são: detecção de spam, tradutor de textos, elaborar respostas de um diálogo (ex. em *chatboots* ou assistentes virtuais disponíveis em *smartphones* mais modernos), entre outros.

O intuito dos sistemas desenvolvidos utilizando PLN é ser capaz de compreender não apenas o sentido literal das palavras, e sim analisar o contexto no qual está inserida, interpretando de fato o texto, tal qual realizando a análise sintática e semântica de toda a sentença e não apenas de uma só palavra. Como aponta Kao A. e Poteet (2007), o PLN busca extrair a representação de um significado em um texto. Complementam os autores, que para realizar essa representação é feito uso de várias formas de conhecimento, como o linguístico, gramatical, compreensão de anáforás e ambiguidades tanto de palavras quanto de estruturas gramaticais.

Segundo Kao e Poteet (2007), as técnicas de PLN são usadas para análise de texto no nível sintático, usando informações de uma gramática formal e um léxico. A informação resultante é então interpretada semanticamente e usada para extrair informações sobre o que foi dito. Os autores complementam dizendo que o PLN inclui técnicas como a remoção de palavras indesejadas (*stopwords*), remoção de sufixos (reduzindo uma palavra ao seu formato raiz), lematização (substituindo uma palavra flexionada por sua forma base), agrupamento de palavras múltiplas, normalização de sinônimo, etiquetamento de partes da fala de acordo com as classes gramaticais (do inglês - *Part-of-speech*), entre outros.

Todas essas técnicas são usadas para realizar uma análise inteligente do texto escrito, extraindo características, com as quais é possível automatizar ou auxiliar outros processos (ex. classificação de textos, resumo automático, tradução de textos, entre outros) (GHAREHCHOPOGH; KHALIFELU, 2011). Sendo assim, o PLN é o processo de análise de um texto para extrair informações que são úteis para fins específicos e, para isso, envolve um conjunto de técnicas para organizar, classificar e extrair informações relevantes de coleções de textos, as quais podem ser chamadas de corpus. Um corpus é uma coletânea de textos em certo idioma, disponibilizados em formato eletrônico (WYNNE, 2005).

2.2 Pré-processamento de Textos

Essa etapa consiste na preparação do texto para possibilitar a extração adequada de suas características, necessárias a um dado contexto de aplicação. O pré-processamento visa a transformação de um texto desestruturado em um formato formalizado, permitindo a extração de padrões. De forma geral, o pré-processamento de texto normalmente envolve (CONRADO et al., 2014): padronização do texto para letras minúsculas, remoção de acentuações e caracteres especiais, tokenização, remoção das *stopwords* e normalização de palavras.

A tokenização consiste em particionar o texto de um documento em uma sequência de *tokens*. Um *token* correponde a uma palavra ou termo, ou seja, o termo mais geral usado para incluir pontuação e outros símbolos (STRAKA; HAJIC; STRAKOVÁ, 2016).

As *stopwords* são palavras que não agregam valor para o contexto da aplicação, como por exemplo: as preposições, os artigos e as conjunções. Seu principal uso é evitar que o processamento a seguir seja excessivamente influenciado por palavras muito frequentes (CESKA; FOX, 2011). A remoção das *stopwords* possibilita gerar características mais reais com relação aos objetivos da aplicação, fazendo com que sejam gerados melhores resultados a partir dos termos utilizados. Podem ainda serem incluídos novos termos à lista de *stopwords*, de acordo com o contexto aplicado. A Tabela 1 apresenta um exemplo de remoção de *stopwords*, acentuações e caracteres especiais.

Tabela 1 – Exemplo de aplicação das etapas de pré-processamento de texto.

Original (sem acentos e caracteres especiais):	O empregador sera responsavel por manter informaçoes dos empregados.
Sem stopwords:	Empregador responsavel manter informaçoes empregados.

Fonte: Autoria própria.

A normalização de palavras consiste em unificá-las, reduzindo suas variações. Esse

processo pode ser realizado comumente fazendo uso de técnicas conforme descritas a seguir:

(a) lematização: consiste na redução de cada palavra de um texto para a sua forma canônica, ou seja, sem flexões (MANNING; RAGHAVAN; SCHÜTZE, 2008). Nessa etapa, os verbos são reduzidos ao infinitivo (ex. estudamos → estudar) e os substantivos e adjetivos reduzidos para o masculino singular (ex. alunas → aluno);

(b) *stemming*: também chamada de radicalização, objetiva a redução das palavras de um texto para o seu radical (ex. estudamos → estud).

Ambos os processos de normalização são específicos do idioma, o que exige que o idioma do texto seja conhecido, pois a maioria das linguagens possui diferentes padrões de escrita. Embora seja um tipo de processo bastante utilizado em PLN, tais passos do pré-processamento não são obrigatórios. Inclusive destaca-se que ao utilizar vetores de *embeddings* como representação textual, não é aconselhável a aplicação de normalização aos textos, pois pode ocorrer uma mistura de contextos e algumas palavras podem ficar sem sentido.

2.2.1 Técnicas de Pré-processamento

A primeira etapa do pré-processamento de texto é a remoção de ruídos, que se trata da remoção de pontuação, caracteres especiais, *tags* HTML, letras maiúsculas, entre outros. Para isso uma ferramenta que se faz bastante útil quando a escrita desses ruídos possuem um determinado padrão são as expressões regulares (do inglês, *RegEx*).

As expressões regulares são uma sequência de caracteres capazes de formar um padrão de pesquisa em um texto, o que permite implementar operações como: busca, substituição, validação de formatos ou filtragem de informações. Para realizar a busca em um texto, se faz necessário escrever um padrão de pesquisa, esse padrão pode contar com o texto a ser buscado e também com o uso de alguns caracteres que representam operações, como por exemplo (SERVIAN, 2020):

- Flags: Incluem comportamentos adicionais às regras, como:
  - "g": indicar achar todas as ocorrências da *regex*;
  - "i": ignora case sensitive;
  - "m": multilinha, lida com caracteres de inicio e fim ao operar em múltiplas linhas.
- Operador *pipe* "|": Utilizado quando precisa-se realizar a busca de mais de um termo, basicamente possui a mesma funcionalidade do operador lógico *OR*.
- Conjuntos "[ ]": Utilizado quando precisa-se realizar a busca de qualquer um dos valores informados. Um exemplo bastante utilizado é a busca de um intervalo de caracteres "[a-z]"
- Metacaracteres: Diferente dos caracteres literais, cujo significado é o valor literal do caractere, os metacaracteres representam funções distintas, como por exemplo:
  - "-": para indicar o uso do range em uma regra de conjunto;
  - ".": usado como um "curinga" para busca de qualquer caractere.
- Qualificadores: Utilizados para dizer quantas vezes uma mesma regra pode aparecer em

sequência.
  - "?": representar zero ou um ocorrência;
  - "*": representar zero ou mais ocorrências;
  - "+": representar uma ou mais ocorrências;
  - "n, m" - representar de n até m.
- Grupos "()": Possibilita a criação de regras isoladas, criação de referências (retrovisores) para o reuso da mesma regra em outro local dentro de um mesmo regex e ainda cria a possibilidade de validações dentro da regex.

No Quadro 1 é exemplificado o uso de *regex* para busca de *tags* HTML, sendo destacado em amarelo os caracteres encontrados de acordo com a expressão regular:

Quadro 1 – Texto com palavras inválidas sem padrão.

Expressão regular: <.*?> Texto com trechos encontrados grifados: Texto utilizando tags html para exemplificar o uso de Regex .

Fonte: Autoria própria.

Outra técnica de pré-processamento, geralmente executada após a remoção de ruído é a tokenização. Trata-se de uma forma de estruturar o texto em formato de lista, removendo as chamadas *stop words*, que são palavras não transmitem significado quando a frase termina por exemplo: "para", "mas", "o". Essas palavras são importantes para leitores humanos, mas não transmitem significado para a máquina.

Na sequência, utiliza-se uma técnica chamada normalização, que possui duas formas (KENNEDY, 2021):

- *Stemming*: Trata-se da remoção de prefixos e sufixos de palavras. Isso se torna útil quando a parte da fala (POS) não é conhecida e é necessária uma varredura rápida.
- Lematização: Trata-se da normalização de cada palavra em uma lista de *tokens* para sua raiz base. A lematização pode devolver verbos complexos como "é" para "ser", mas precisa que a parte do discurso seja realmente eficaz.

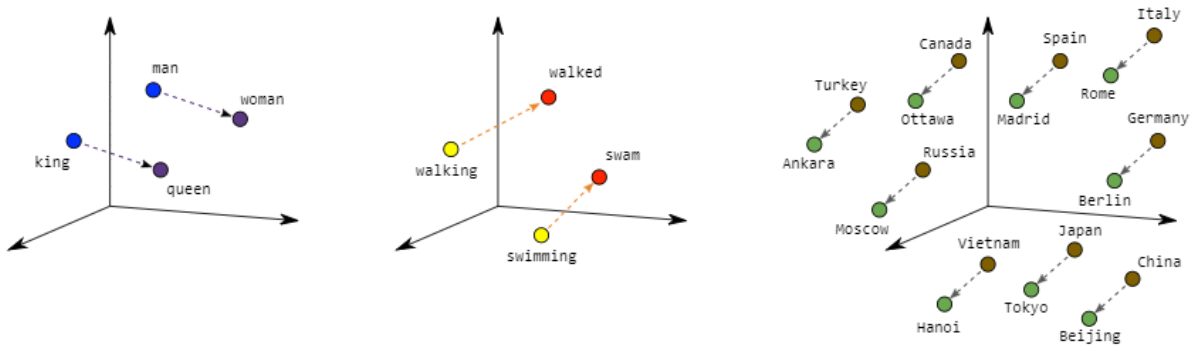
2.2.2 *Word Embeddings* e Representação Textual

Atualmente, os modelos de *word embeddings* são uma forte tendência em PLN. O primeiro modelo de *word embeddings* utilizando redes neurais artificiais foi publicado em 2013 (MIKOLOV et al., 2013) por pesquisadores da Google, e desde então, esse conceito tem feito parte da maioria das pesquisas em PLN dos últimos anos.

Modelos de incorporação de palavras, do inglês *word embeddings*, são algoritmos que surgiram para representar textos, superando algumas limitações do modelo PLN, ou seja, considerando o contexto das palavras, diferente da representação por espaço vetorial (MANNING; RAGHAVAN; SCHÜTZE, 2008).

Além disso, vetores de *word embeddings* permitem reduzir a dimensionalidade e a esparsidade dos vetores de características textuais. A Figura 1 apresenta um exemplo de representação vetorial de *embeddings* para formas comparativas e superlativas de adjetivos, tempos verbais e relações entre países e capitais. A partir da representação geométrica entre os *embeddings*, é possível capturar relações semânticas (ex. entre um país e sua capital). Assim, torna-se possível modelar a importância semântica de uma palavra no formato numérico (ZHANG et al., 2015).

Figura 1 – Representação vetorial de *word embeddings*.



Fonte: (adaptado de (MIKOLOV et al., 2013)).

Observa-se que cada palavra é representada como um vetor com valores numéricos. Na Figura 1, são apresentadas as palavras ilustradas em formato vetorial. Assim, é possível calcular a distância entre duas palavras, realizando cálculos de distância entre os pontos.

Assim, um modelo de *word embedding*, é constituído de vetores de representação de palavras, por meio dos quais é possível identificar o relacionamento semântico entre as palavras de um determinado domínio. Isso é possível a partir das propriedades observadas em um *corpus* de treinamento, realizando a geração de *word embedding* de forma automática (MIKOLOV et al., 2013).

Os modelos de *word embedding* usam redes neurais profundas para representar cada palavra como um vetor denso, com baixa dimensionalidade e com foco na relação entre as palavras (MIKOLOV et al., 2013). Tais vetores são usados tanto de maneira independente, para calcular semelhanças entre termos, quanto como base de representação para tarefas de PLN (ex. classificação de texto, reconhecimento de entidades, análise de sentimento).

Dentre os métodos para geração de *embeddings* a partir de textos, destacam-se os preditivos, os quais podem ser classificados em (HAJ-YAHIA; SIEG; DELERIS, 2019):

- sem-contexto (também chamados de estáticos), e
- contextualizados (também chamados de sensíveis ao contexto ou dinâmicos)

A abordagem de representação por meio da geração de *embeddings* contextualizados é facilmente aplicável a muitas tarefas da PLN, onde as entradas geralmente são sentenças e, portanto, as informações de contexto estão disponíveis (ex.requisitos textuais de software). Dentro desse novo paradigma, o BERT (DEVLIN et al., 2019) tem sido o algoritmo que

apresentou os melhores resultados em tarefas de PLN e, portanto, vem sendo largamente utilizado em diversas aplicações. Sua aplicação tem ocorrido por meio de modelos pré-treinados e disponibilizados por seus autores (ex. BERT_base e BERT_large (DEVLIN et al., 2019)), os quais serão melhor apresentados na sequência.

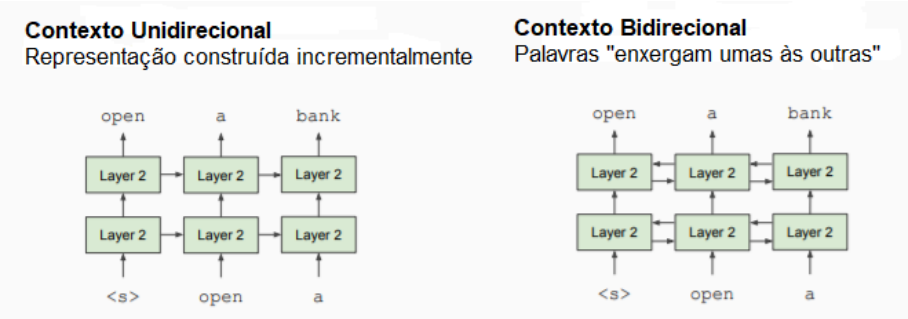
2.2.3 *Bidirectional Encoder Representations from Transformers* (BERT)

O BERT vem sendo considerado o estado da arte na representação de linguagem pré-treinada (DEVLIN et al., 2019). O motivo desse destaque é que modelos de *embeddings* BERT são considerados modelos contextualizados (também chamados de dinâmicos), e têm apresentado ótimos resultados em diversas tarefas de PLN (DAI; LE, 2015; PETERS et al., 2018; RADFORD et al., 2018; HOWARD; RUDER, 2018), como classificação de sentimentos, cálculo de tarefas semânticas de similaridade textual e reconhecimento de tarefas de vinculação textual.

Esse método contextualizado de representação de linguagem, teve origem a partir de diversas ideias e iniciativas voltadas para essa finalidade, que surgiram na área de PLN nos últimos anos, tais como: coVe (MCCANN et al., 2017), ELMo (PETERS et al., 2018), ULMFiT (HOWARD; RUDER, 2018), CVT (CLARK et al., 2018), context2Vec (MELAMUD et al., 2016), o *transformer* OpenAI (GPT, GPT-2, GPT-3) (RADFORD et al., 2018) e o *Transformer* (VASWANI et al., 2017).

O algoritmo do BERT foi projetado para pré-treinar representações textuais bidirecionais profundas a partir de texto não rotulado. Assim, durante o treinamento, são considerados, ao mesmo tempo, os contextos esquerdo e direito de uma palavra. A representação de uma palavra é dada pela concatenação desses dois contextos. A Figura 2 mostra a diferença entre os modelos unidirecionais (ex. Word2Vec) e os bidirecionais (ex. BERT).

Figura 2 – Contexto unidirecional *versus* contexto bidirecional.



Fonte: (adaptado de Devlin et al. (2019).

Essa interpretação do contexto de forma bidirecional, é indispensável para permitir uma compreensão completa do significado de uma sentença. O exemplo a seguir demonstra essa situação. Na Tabela 2 é apresentado um exemplo em que a palavra "banco" ocorre na frase 1 e na frase 2.

Tabela 2 – Exemplo de palavras ambíguas, em que se faz necessária a análise bidirecional de contexto.

Frase 1:	Combinamos de nos encontrar na praça e irmos ao banco.
Frase 2:	Combinamos de nos encontrar na praça, irmos ao banco e fazermos o pagamento.

Fonte: Autoria própria.

Ao tentar prever o significado da palavra banco tomando como base o contexto da esquerda ou da direita, ocorrerá um erro para uma das duas frases, pois será omitido o contexto da direita, necessário para que seja identificado o contexto completo da palavra "banco".

A característica dinâmica que o BERT apresenta, se deve ao mecanismo de atenção, mais especificamente de auto-atenção (do inglês, *self-attention*) - o que será melhor explicado na seção 3.6 - que possui, o qual denomina-se *Transformer* (DEVLIN et al., 2019). Esse mecanismo permite analisar o contexto de cada palavra em um texto de forma individual, o que permite verificar se cada palavra já foi utilizada anteriormente em um mesmo contexto. Com isso, o método permite aprender relações contextuais entre palavras (ou sub-palavras) em um texto. Desta forma, o BERT é um modelo de linguagem em larga escala que consiste em várias camadas de blocos do tipo *Transformer* (VASWANI et al., 2017). O conceito de *Transformer* será apresentado em mais detalhes a seguir.

Além de ser bidirecional e dinâmico, os modelos BERT pré-treinados podem ser ajustados com apenas algumas camadas de saída adicionais (DEVLIN et al., 2019). Esses modelos pré-treinados e ajustados, podem ser usados para criar modelos de alta eficiência que podem ser aplicados em diversas tarefas de PLN.

Portanto, a abordagem BERT consiste em duas etapas principais (DEVLIN et al., 2019):

1. Treinamento de um modelo de linguagem em um grande corpus de texto não rotulado (não supervisionado) - incluindo toda a Wikipedia (2.500 milhões de palavras) e *Book Corpus* (800 milhões de palavras) (ZHU et al., 2015). - e,
2. Ajuste desse modelo de linguagem pré-treinado para aplicação em tarefas (supervisionadas) específicas de PLN.

2.3 Trabalhos Relacionados

Na Tabela 3 são apresentados alguns trabalhos relacionados à essa pesquisa. Na pesquisa realizada por Pogiatzis (2019), é detalhado o uso do BERT para a geração de *embeddings*, tanto de palavras como de sentenças. Esse trabalho foi utilizado como base para implementar a geração dos *embeddings* que são usados como base de treinamento para o modelo classificador.

Tabela 3 – Trabalhos relacionados.

Titulo	Autor(es)
NLP: Contextualized word embeddings from BERT	Pogiatzis (2019)
How to Clean Data in Natural Language Processing (NLP)	Kennedy (2021)
Analysis of twitter specific preprocessing technique for tweets	Ramachandran e Parvathi (2019)

Fonte: Autoria própria.

Na pesquisa realizada por Kennedy (2021), são citadas e exemplificadas algumas formas de limpeza de dados para o processamento de linguagem natural, tais como: o uso de expressões regulares, tokenização, que seria resumidamente uma forma de adicionar um tipo de estrutura ao texto e normalização.

No trabalho de Ramachandran e Parvathi (2019) é realizado um experimento de classificação de *tweets* para avaliar o desempenho das técnicas de pré-processamento aplicadas. Neste caso foram aplicadas técnicas de pré processamento como tokenização, remoções de *stopword*, além de explorar recursos de uma técnica chamada *Part-Of-Speech* que identifica a classe de palavras com base na posição da palavra na frase e é capaz de identificar URL's, menções de outros usuários e até *Emoticons*.

Nenhum trabalho aplicando técnicas de pré-processamento para a preparação de bases de dados de requisitos de *software*, aplicando *embeddings* foi encontrado.

3 MATERIAIS E MÉTODOS

A seguir é apresentada o *dataset* sobre o qual foi aplicada a abordagem proposta. Na sequência são descritas as etapas do pipeline desenvolvido, incluindo os métodos estudados e selecionados. Por fim, são descritos os materiais utilizados para a implementação da abordagem de pré-processamento de textos, objetivando a limpeza e pré-processamento adequados de textos de requisitos de *software* (ex. histórias de usuário) sem prejudicar a identificação do contexto.

3.1 Dataset Aplicado

O *dataset* utilizado como objeto deste estudo possui textos de requisitos de software. Esses textos apresentam linguagem técnica utilizada por equipes de desenvolvimento de *softwar*e, a fim de realizar o detalhamento de regras de negócio que devem ser interpretadas pelos desenvolvedores. O objetivo de um requisito textual é realizar o desenvolvimento de novas funcionalidades ou alterações em fluxos já existentes em aplicações.

Os requisitos textuais que compõe o *dataset* é composto de histórias de usuário (CHOETKIERTIKUL et al., 2018). Destaca-se que, apesar de serem referenciadas como histórias de usuários, o texto dos requisitos não apresenta uma estrutura padronizada. Esse dataset é considerado por Choetkiertikul et al. (2018), como sendo o primeiro conjunto de dados em que o foco está no nível de requisitos (ex. histórias de usuário) e não somente no nível de projeto, como ocorre na maioria dos conjuntos de dados disponíveis para pesquisas na área de engenharia de *software*.

Os textos de requisitos foram obtidos de grandes fontes abertas, a partir de sistemas de gestão de projetos (ex. Jira). Tratam-se de 16 grandes projetos de código aberto obtidos de 9 repositórios (*Apache, Appcelerator, DuraSpace, Atlassian, Moodle, Lsstcorp, Mulesoft, Spring e Talendforge*) totalizando 23.313 requisitos (Tabela 4). Esses requisitos foram disponibilizados inicialmente por (PORRU et al., 2016). Na sequência, (CHOETKIERTIKUL et al., 2018) utilizou o mesmo *dataset* para a realização da sua pesquisa visando estimativa de esforço de *software* por analogia.

Tabela 4 – Número de requisitos textuais (histórias de usuário) e descrição de cada um dos 16 projetos utilizados nos experimentos (CHOETKIERTIKUL et al., 2018).

Número do projeto	**Descrição**	**Requisitos/projeto**
0	Mesos	1680
1	Usergrid	482
2	Appcelerator Studio	2919
3	Aptana Studio	829
4	Titanium SDK/CLI	2251
5	DuraCloud	666
6	Bamboo	521
7	Clover	384
8	JIRA Software	352
9	Moodle	1166
10	Data Management	4667
11	Mule	889
12	Mule Studio	732
13	Spring XD	3526
14	Talend Data Quality	1381
15	Talend ESB	868
Total		**23.313**

Fonte: (CHOETKIERTIKUL et al., 2018) .

Tendo em vista essas informações, foi realizada uma breve análise textual, considerando um pequeno número de dados do *dataset*. Ficou evidente que em inúmeras partes dos textos são utilizadas *tags*, códigos de programação ou até mesmo URL's para descrever ou exemplificar alguns fluxos, como é exemplificado no Quadro 2.

Quadro 2 – Texto com trechos de palavras inválidas.

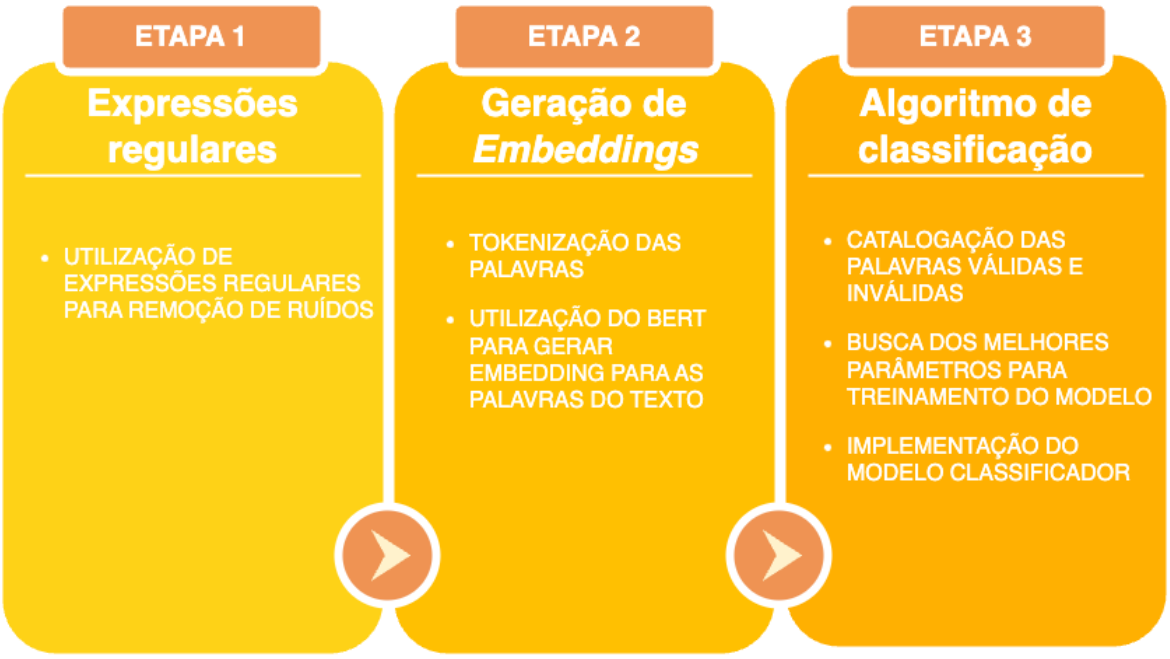
Developers would like to have the ability to add custom launch parameters to the android emulator such as setting a HTTP proxy. Here is one example of the parameters they would like to be able to include: {code} # start the emulator emulator_cmd = [self.sdk.get_emulator(), '-avd', avd_name, '-port', '5560', '-sdcard', self.sdcard, '-logcat', '"*:d *"', '-no-boot-anim', # '-http-proxy', # 'http://127.0.0.1:8888', '-partition-size', '128' # in between nexusone and droid] {code} Reference HD Tickets: http://appc.me/c/APP-364571 http://appc.me/c/EPH-39417-198

Fonte: *Dataset* (CHOETKIERTIKUL et al., 2018) .

3.2 Abordagem Proposta

A seguir, na Figura 3 serão apresentadas as etapas do *pipeline* proposto para limpeza e pré-processamento do *dataset* de requisitos de *software*, a qual foi utilizada nesse trabalho.

Figura 3 – *Pipeline* proposto.



Fonte: Autoria própria.

Na etapa 1 foi realizado um estudo para tratar o maior número possível dessas palavras indesejadas, utilizando expressões regulares. Com a análise realizada nos textos do dataset, foi possível encontrar alguns padrões de apresentação dos textos inválidos, conforme segue:

- Tags que começam com o caractere $<$, conhecido pelo nome "menor que" e são finalizadas com o caractere $>$, conhecido pelo nome "maior que".
- Códigos de programação que ficam entre a tag "$\{code\}$" ou "$<code>$".
- Tags que começam com o caractere "{" e são finalizadas com o caractere "}" conhecidos pelo nome "chave".
- URL's que se iniciam com a sequência de caracteres "http://" e "https://".

A segunda etapa foi o uso de um algoritmo de geração de *embeddings* para entender a possibilidade da verificação de agrupamento de dados inválidos, visto que esse algoritmo retorna a representação das palavras em forma de vetor de 768 dimensões. Utilizando o t-SNE como redutor de dimensionalidade, para reduzir para um vetor de duas dimensões, se tornou possível realizar uma análise dos agrupamentos das palavras em um gráfico para chegar à conclusão da possibilidade de implementação de um algoritmo de classificação.

Por fim, na etapa 3, foi implementado o algoritmo de classificação, iniciando pela catalogação de uma amostra de palavras entre válidas e inválidas para utilizar como *dataset* de treinamento, seguindo pela implementação da técnica de *GridSearch* para realizar a busca do melhor cenário de parâmetros para o treinamento do modelo classificador, e finalizando com o treinamento do modelo utilizando o classificador *KNeighborsClassifier* da biblioteca *Scikit-learn*.

3.3 Materiais

A Tabela 5 apresenta a lista de ferramentas e tecnologias utilizadas para desenvolvimento do algoritmo que é resultado dessa pesquisa.

Tabela 5 – Lista de ferramentas e tecnologias.

Ferramenta / Tecnologia	Versão	Finalidade
Colaboratory	-	IDE de desenvolvimento
Python	3.7.13	Linguagem de Programação
Tensorflow	2.11.0	Biblioteca para aprendizado de máquina
Scikit-learn	1.1.3	Biblioteca para aprendizado de máquina

Fonte: Autoria própria.

O *dataset* utilizado para pesquisa traz informações textuais de descrições de requisitos de software, o arquivo CSV disponibilizado para o estudo possui 23.313 linhas, cada uma delas contendo o título e o detalhamento dos requisitos. Além de explicações para que seja realizado o desenvolvimento das funcionalidades, esses textos são compostos por informações de URL's, *tags*, IP's e até mesmo trechos de código de programação. Todos os textos do *dataset* estão na língua inglesa.

4 RESULTADOS

Nessa sessão, serão apresentados os resultados obtidos com a realização desse trabalho. Inicialmente, será apresentada uma análise dos textos quando executada a etapa de pré-processamento dos dados utilizando expressões regulares. Logo após, será demonstrada a análise dos resultados obtidos ao aplicar o algoritmo para geração de *embeddings* com intuito de realizar o agrupamento das palavras em grupos de válidas e inválidas. Por fim, serão apresentados os resultados do algoritmo que foi implementado para a realização da classificação das palavras.

4.0.1 Método para Substituição Textual Utilizando Expressões Regulares

Considerando alguns padrões para textos inválidos, tornou-se possível desenvolver um algoritmo de substituição desses dados utilizando expressões regulares. Sendo assim, foram criados métodos isolados para tratamento de cada um dos padrões, visando a qualidade da escrita e do entendimento do código.

Na Quadro 3, é apresentado o método escrito na linguagem Python implementado para realizar a remoção do texto entre *tags*. Na primeira linha da função, é definido o padrão da expressão regular, que define o padrão de texto que será buscado, nesse caso, é buscado todo o texto que encontra dentro de caracteres "maior que" e "menor que", incluindo também esses caracteres. Tendo encontrado essas *tags*, alimenta uma lista de *strings*, a qual será percorrida para que cada palavra seja substituída por um texto padrão que pode ser facilmente alterado conforme os resultados esperados no modelo que está se treinando.

Quadro 3 – Remoção de tags HTML.

```
    def remove_html_tags(dirty_text):
        pattern = re.compile(r'<.*?>')
        html_tags = re.findall(pattern, dirty_text)
        for tag in html_tags:
            dirty_text = dirty_text.replace(tag, TAG_REPLACEMENT)
        return dirty_text
```

Fonte: Autoria própria.

No Quadro 4 é mostrado o método escrito para realizar a substituição de todo o texto que se encontrar entre a *tag* "{*code*}". Inicialmente é removido o espaçamento dentro da *tag*, pois durante a análise dos dados foi encontradas algumas *tags* que possuiam espaçamento antes ou depois da palavra "*code*". Logo após, é realizada a mesma lógica do código anterior para substituir todo o texto encontrado por um texto pré-definido em uma constante (CODE_TAG_REPLACEMENT).

Quadro 4 – Remoção de textos entre *tag* {*code*}.

```
    def remove_text_between_code_brace_tag(dirty_text):
        dirty_text = re.sub(r"{\s+", "{", dirty_text)
        dirty_text = re.sub(r"\s+}", "}", dirty_text)
        pattern = re.compile(r'{code}(.*?){code}')
        code_text = re.findall(pattern, dirty_text)
        for code in code_text:
            dirty_text =
                dirty_text.replace(code, CODE_REPLACEMENT)
        return dirty_text
```

Fonte: Autoria própria.

No Quadro 5, foi implementada uma lógica praticamente idêntica ao bloco de código anterior, da Listagem 3.2, onde foi apenas trocada a *tag* {*code*} por <*code*>.

Quadro 5 – Remoção de textos entre *tag* <*code*>.

```
    def remove_text_between_code_tag(dirty_text):
        dirty_text = re.sub(r"<\s+", "<", dirty_text)
        dirty_text = re.sub(r"\s+>", ">", dirty_text)
        pattern = re.compile(r'<code>(.*?)<code>')
        code_text = re.findall(pattern, dirty_text)
        for code in code_text:
            dirty_text =
                dirty_text.replace(code, CODE_REPLACEMENT)
        return dirty_text
```

Fonte: Autoria própria.

Já no Quadro 6, foi implementada uma lógica para remover todas os textos que se encontram entre os caracteres { e }. Inicialmente é definido o padrão da expressão regular, que estabelece que serão buscadas todas as palavras dentro dos caracteres especificados. Após isso, é alimentada uma lista de strings que será percorrida para realizar a substituição de cada uma das palavras selecionadas.

Quadro 6 – Remoção de *tags* HTML.

```
    def remove_text_between_brace_tag(dirty_text):
        pattern = re.compile(r'{\w*\S*}')
        html_tags = re.findall(pattern, dirty_text)
        for tag in html_tags:
            dirty_text =
                dirty_text.replace(tag, BRACE_TAG_REPLACEMENT)
        return dirty_text
```

Fonte: Autoria própria.

No Quadro 7 é realizada uma substituição das URL's que utilizam o padrão de escrita iniciado por "http://" ou "https://".

Quadro 7 – Remoção de URL's.

```
    def remove_url(sample):
        dirty_text = re.sub(r"http:\S+", URL_REPLACEMENT, sample)
        return re.sub(r"https:\S+", URL_REPLACEMENT, dirty_text)
```

Fonte: Autoria própria.

Além dos métodos apresentados acima, foi escrito um trecho de código para realizar a remoção de espaços em branco duplicados encontrados no texto, conforme mostra o Quadro 8.

Quadro 8 – Remoção de espaçoes em branco duplicados.

```
    def remove_repeated_spaces(dirty_text):
        dirty_text = re.sub(r"\s\s+", "␣", dirty_text)
        return dirty_text
```

Fonte: Autoria própria.

Para unir a chamada desses vários métodos em apenas um, foi desenvolvido o código apresentado no Quadro 9. Onde a entrada é o texto que precisa ser pré-processado, e a saída seria o texto livre dos caracteres e textos explicados a cima.

Quadro 9 – Método para limpeza do texto.

```
    def get_clean_text(dirty_text):
        dirty_text =
            remove_text_between_code_brace_tag(dirty_text)
        dirty_text = remove_text_between_brace_tag(dirty_text)
        dirty_text = remove_text_between_code_tag(dirty_text)
        dirty_text = remove_html_tags(dirty_text)
        dirty_text = remove_url(dirty_text)
        clean_text = remove_repeated_spaces(dirty_text)
        return clean_text
```

Fonte: Autoria própria.

Ao realizar a chamada do método de limpeza do texto, após todos os tratamentos é possivel visualizar no Quadro 10 o texto livre das *tags*, códigos de programação e URL's.

Quadro 10 – Texto pré-processado.

Developers would like to have the ability to add custom launch parameters to the android emulator such as setting a HTTP proxy. Here is one example of the parameters they would like to be able to include: Reference HD Tickets:

Fonte: *Dataset* (CHOETKIERTIKUL et al., 2018).

No entanto, mesmo após a substituição aplicando os métodos que utilizam expressões regulares, ainda há textos que possuem palavras inválidas, cuja escrita não possui um padrão para uma possível representação utilizando as expressões regulares. Dessa forma, optou-se por explorar formas de classificação de palavras utilizando processamento de linguagem natural.

4.0.2 Método de Classificação de *Embeddings*

Observa-se que, ainda que seja realizado o tratamento dos textos usando expressões regulares, essa opção está limitada a trechos de palavras inválidas que não possuem um padrão de escrita. No exemplo do Quadro 11 é possível notar que ainda existem palavras que devem ser removidas por se tratarem de caminhos de *endpoints*, nomes de classes da linguagem de programação, entre outros.

Quadro 11 – Texto com palavras inválidas sem padrão.

Error and possible race condition in the assets/uuid/data endpoints. It looks like we might have a race condition in the assets/uuid/data endpoints. perhaps the assets in general. I noticed my calls were successful about 5% of the time. I put in a 1s wait, and now itś over 95%. When it fails, I get this error: {"error":"web_application","timestamp":1391230754494,"duration":0,"exception":"javax.ws.rs.WebApplicationException","error_description":"com.sun.jersey.api.MessageException: A message body reader for Java class com.sun.jersey.multipart.FormDataMultiPart, and Java type class com.sun.jersey.multipart.FormDataMultiPart, and MIME media type application/octet-stream was not found."}

Fonte: *Dataset* (CHOETKIERTIKUL et al., 2018).

Nessa etapa foi realizado um estudo para a aplicação de *embeddings*, que são a representação de cada palavra do texto no formato de vetor de dimensões fixas. Assim buscou-se compreender a existência de agrupamentos dessas representações que tornem facilmente identificáveis os grupos de palavras consideradas indesejáveis, ou seja, que se encontram fora de contexto. Dessa forma torna-se possível a implementação de um método de aprendizado de máquina, o qual realize a interpretação desses *embeddings* e a classificação de forma automática das palavras válidas e inválidas.

Inicialmente foi realizada a geração de *embeddings* das palavras utilizando o BERT. Os módulos desse codificador que foram importados foram: *modeling*, que inclui a implementação do modelo do BERT e *tokenization*, que inclui a parte da tokenização das sequências. Existem diversos modelos pré-treinados do BERT disponíveis no repositório oficial do GitHub, nesse estudo será usado o modelo chamado de uncased_L-12_H-768_A-12, em que cada parte representa um parâmetro (Pogiatzis (2019)):

- uncased representa que serão geradas sequências com letras minúsculas,
- L-12 os blocos transformadores,
- H-768 o tamanho da camada oculta e
- A-12 as *attention heads*.

Também foram definidas classes para a entrada e saída do processamento, conforme mostra o Quadro 12, a classe *InputText* espera receber um identificador único e um texto no qual será realizado o *embedding*, além disso, possui a propriedade "*text_b*" que por padrão foi definida como "*None*".

Quadro 12 – Classe InputText.

```
class InputText(object):

  def __init__(self, unique_id, text_a, text_b=None):
    self.unique_id = unique_id
    self.text_a = text_a
    self.text_b = text_b
```

Fonte: Pogiatzis (2019).

Para realizar a entrada dos dados no BERT, foi criada a classe *InputFeatures*, como atributos, essa classe espera receber um identificador único, um vetor de *tokens*, *input_ids* são os identificadores que correspondem ao token do vocabulário, *input_mask* anota a sequência de *token* real do preenchimento e, por último, *input_type_ids* separa o segmento A do segmento B. (Pogiatzis (2019)).

Quadro 13 – Classe InputFeatures.

```
class InputFeatures(object):

  def __init__(self, unique_id, tokens, input_ids, \\
  input_mask, input_type_ids):
    self.unique_id = unique_id
    self.tokens = tokens
    self.input_ids = input_ids
    self.input_mask = input_mask
    self.input_type_ids = input_type_ids
```

Fonte: Pogiatzis (2019).

O início do fluxo espera receber uma lista contendo um ou mais textos, que será convertida para outra lista de objetos do tipo *InputText*. Essa será usada para obter como resultado uma terceira lista já com os dados *tokenizados* (*InputFeatures*), a qual servirá com base para os próximos métodos que serão executados.

Por último temos o método principal da geração dos *embeddings*, que tem como objetivo receber a lista de textos e realizar o pré-processamento para obter a lista de dados *tokenizados*. Também espera receber o número de dimensões que cada *embedding* deve possuir, que pode ser no máximo de 768. Esse método utiliza o BERT para realizar as previsões com base nos dados fornecidos e, como saída, retorna um dicionário com o *token* como chave e o vetor de *embedding* como valor. A saída obtida foi exportada para um arquivo CSV, que posteriormente será utilizado como *dataset* para realizar o treinamento de um modelo classificador.

4.0.3 Método Automatizado para Classificação de Palavras Indesejadas

Nessa etapa, foi implementado um algoritmo de classificação que tem como objetivo identificar se a palavra é válida ou inválida. Como se trata de um modelo de aprendizado de máquina supervisionado, espera que sejam inseridos dados de treino e teste rotulados. Para isso foi utilizada realizado um pré-processamento no *dataset* que foi obtida no processo anterior, rotulando manualmente os dados como 0 para palavras válidas e 1 para palavras inválidas.

Outra etapa de pré-processamento que foi realizada nesse *dataset*, foi a redução da dimensionalidade, de 768 dimensões para 2, utilizando *t-SNE*, que se trata uma técnica para redução de dimensionalidade que é particularmente adequada para a visualização de conjuntos de dados de alta dimensão (Maaten (2022)). Isso possibilita realizar a visualização e análise dos *embeddings* em gráficos de 2 dimensões, o que permite tomar decisões importantes ao realizar a criação do algoritmo.

Para automatizar a busca pelo melhor cenário de parâmetros do algoritmo, foi utilizada uma técnica de *GridSearch* chamada de classificação de árvore de decisão que, conforme Madan (2019), são uma excelente maneira de classificar classes, se tratam de um classificador transparente, o que significa que podemos realmente encontrar a lógica por trás da classificação da árvore de decisão. No Quadro 14 foi exemplificada o uso do *GridSearch*.

Quadro 14 – *GridSearch*.

```
param_grid = {
    "criterion" : ['gini', 'entropy'],
    "splitter": ['best', 'random'],
    'min_samples_split': [2, 5, 10],
    "min_samples_leaf": [1, 5, 10]
}

grid_search = GridSearchCV(
    estimator=DecisionTreeClassifier(),
    param_grid=param_grid
)
grid_search.fit(X_train, y_train)
```

Fonte: Autoria própria.

Ainda no Quadro 14 nota-se que o método *fit* espera uma entrada de dados *X_train* e *y_train*, que são os dados e os rótulos do *dataset* de treino, os mesmos utililzados para o método *fit* do classificador, para isso foi dividido o *dataset* com os *embeddings* de 2 dimensões, que ao total conta com 1000 dados rotulados, entre 70% para o treino e 30% para os testes, conforme mostra o código no Quadro 15.

Quadro 15 – *Dividindo dados de treino e teste.*

```
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.3,
    shuffle=True,
    random_state=1,
    stratify=y
)
```

Fonte: Autoria própria.

Dessa forma, foi realizado o treinamento do modelo de classificação utilizando o melhor cenário de parâmetros encontrado pelo método de *GridSearch* e os dados de treino e testes, como pode ser observado no Quadro 16

Quadro 16 – *Utilização do melhor classificador treinado pelo* GridSearch.

```
new_classifier=grid_search.best_estimator_
y_test_pred=new_classifier.predict(X_test)
```

Fonte: Autoria própria.

4.1 Análise dos Resultados

4.1.1 Aplicação de Métodos de Pré-Processamento Baseados em Expressões Regulares

O uso de expressões regulares para realizar o pré-processamento dos textos se mostrou muito satisfatório para o tratamento de trechos onde existiam palavras inválidas dentro de um determinado padrão. O Quadro 17 apresenta um exemplo, contendo a comparação entre o texto enviado como entrada para o método e o texto recebido como resultado da sua execução.

Quadro 17 – Análise da entrada e resultado do método.

Entrada	**Resultado**
iOS: Font property doesn't work for Picker and PickerColumn Steps to reproduce: 1. Copy Paste the code in app.js and run the app on iOS device. Expected Result: 1. Must observe the difference in font size than normal. Actual Result: 1. No differences appear. Font Size appears the same. Working fine for Android. See the issue TIMOB-14007 {code} var win ... {code} Test case for Date Picker: {code} Ti.UI.backgroundColor = 'white';	iOS: Font property doesn't work for Picker and PickerColumn Steps to reproduce: 1. Copy Paste the code in app.js and run the app on iOS device. Expected Result: 1. Must observe the difference in font size than normal. Actual Result: 1. No differences appear. Font Size appears the same. Working fine for Android. See the issue TIMOB-14007 HERE WAS A PROGRAMMING CODE code Test case for Date Picker: . HERE WAS A PROGRAMMING CODE

Fonte: *Dataset* (CHOETKIERTIKUL et al., 2018).

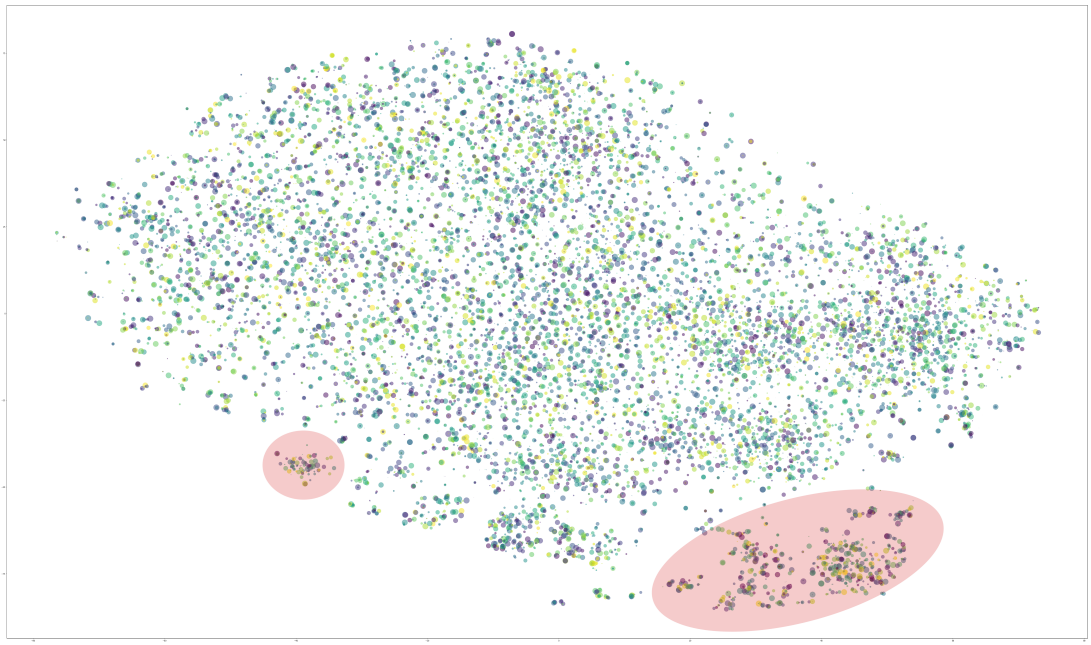
Nota-se que, nesse caso, quando as palavras que fazem parte de um grupo de código de programação, possuem no início e no fim do trecho a sequência de caracteres $\{code\}$, isso tornou possível implementar o método que substitui todo o texto que se encontra nesse trecho pela frase "*HERE WAS A PROGRAMMING CODE*", que pode facilmente ser substituída por outra frase de preferência modificando a constante implementada no código.

4.1.2 Aplicação de Métodos de Pré-Processamento Baseados em *Embeddings*

A geração de *embeddings* utilizando o modelo de linguagem pré-treinado BERT (DEVLIN et al., 2019), tornou possível visualizar de forma clara os agrupamentos de palavras que não se encaixam no contexto geral das frases, ou seja, que se referem a palavras consideradas estranhas ao contexto.

Na Figura 4 estão sendo representados por pontos no gráfico, os *embeddings* de uma parcela das palavras que compõem os textos da base de dados utilizada para o estudo. Nota-se, nos destaques em vermelho, grupos de palavras que fogem do agrupamento central, quando feita uma análise mais aprofundada desses grupos, percebeu-se que se tratavam, de maneira predominante, de grupos de palavras inválidas.

Figura 4 – Gráfico representando os *embeddings* de 2 dimensões.



Fonte: Autoria própria.

Feita essa análise compreende-se que é possível treinar um algoritmo de aprendizado de máquina para interpretar esses *embeddings* e classificar entre palavras válidas e inválidas. Para isso, foi necessário catalogar manualmente uma amostra de dados composta por palavras válidas e inválidas que fosse o suficiente para treinar o algoritmo de classificação, como exemplificado na Tabela 6.

Tabela 6 – Exemplos de palavras catalogadas.

Palavra	Classificação
found	Válida
shows	Válida
saved	Válida
##coll	Inválida
##ection	Inválida
export	Inválida
}	Inválida

Fonte: Autoria própria.

4.1.3 Análise dos resultados do algoritmo de classificação

Ao implementar o algoritmo de agrupamento sobre os embeddings das palavras, obteu-se um resultado satisfatório em testes com uma amostra de 1000 palavras catalogadas, sendo esta dividida em 70% de palavras válidas e 30% de palavras inválidas.

Ao realizar a busca pelos parâmetros ideais para realizar o treinamento, utilizando *GridSearch*, foram recebidos como resultados os seguintes parâmetros:

Quadro 18 – Melhor cenário de parâmetros encontrado pelo *GridSearch*.

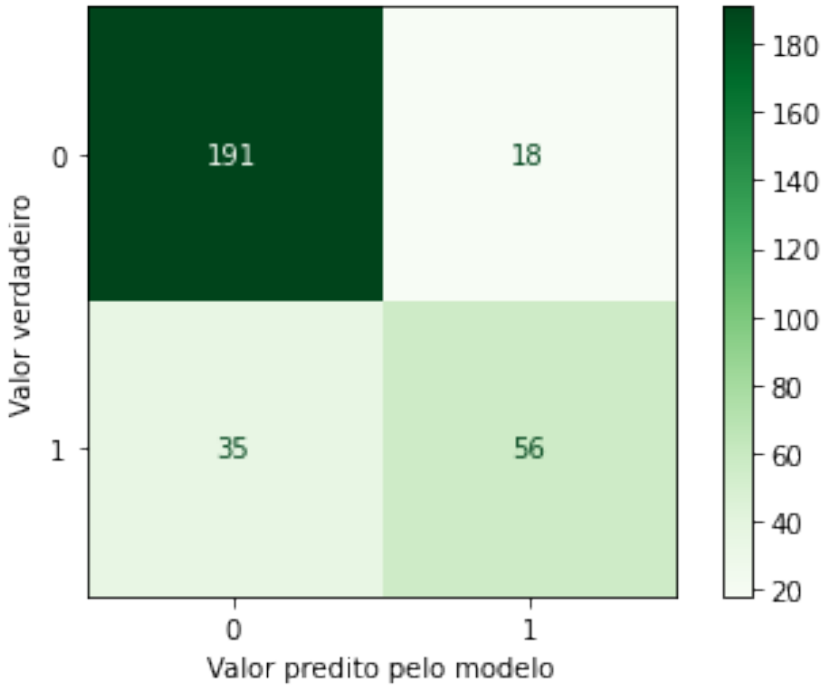
{‘criterio‘: ‘entropy‘, ‘min_samples_leaf‘: 5, ‘min_samples_split‘: 2, ‘splitter‘: ‘random‘}

Fonte: Autoria própria.

Foi realizada e implementação do treinamento de um modelo de classificação utilizando o melhor cenário de parâmetros encontrado pelo *GridSearch*, sendo que, a amostra de 1000 palavras catalogadas foi divida entre 70% para treinamento e 30% para testes do classificador. Sendo assim, obteu-se como resultado final uma acurácia de 82,33%.

Na Figura 5 podem ser observados na matriz de confusão os totalizadores dos resultados obtidos, sendo que, para palavras válidas foram classificadas corretamente 191 palavras, de um total de 226. Já com relação às palavras inválidas, foram classificadas corretamente 56 palavras de um total de 74.

Figura 5 – Matriz de confusão representando resultados do classificador.



Fonte: Autoria própria.

5 CONCLUSÃO

A evolução no PLN torna possível automatizar tarefas de grande importância que antes precisavam ser executadas por seres humanos e devido ao grande volume de dados a se analisar, se tornava um processo exaustivo ou até mesmo impossível. Contudo, por se tratarem de dados não estruturados, um dos maiores problemas enfrentados ao implementar esses algoritmos é a alta quantidade de ruído nos dados disponíveis. Esses ruídos podem ou não possuir um padrão de escrita, o que torna necessário que sejam pensadas formas diferentes de se tratar essas duas ocasiões.

Diante da necessidade da realização de limpeza e transformação dos dados de texto para implementar algoritmos de PLN, esse trabalho propôs desenvolver métodos de realizar o pré-processamento de dados de texto, utilizando-se de técnicas de expressões regulares (*RegEx*) para a remoção de ruídos para aqueles textos que apresentam algum padrão de escrita. Aliado à essas técnicas, também foi proposto o uso de técnicas de aprendizado de máquina para busca de ruídos cujo texto não possui um padrão de escrita. Sendo assim, objetivou-se a aplicação de um método de pré-processamento inteligente, com intuito de contribuir na etapa de pré-processamento de textos ao implementar algoritmos de PLN.

Como objeto de estudo, foram utilizados textos de especificações de requisitos de software (CHOETKIERTIKUL et al., 2018) e (FÁVERO; CASANOVA; PIMENTEL, 2022), os quais continham trechos de códigos de programação, *tags* HTML, enderços da web, entre outros tipos de ruído. Para o tratamento de ruídos com formato de escrita padrão, como por exemplo *tags* HTML e códigos de programação que se encontravam entre as *tags* "$\{code\}$" ou "$<code>$", foi aplicado *RegEx*.

Já para ruídos considerados mais complexos, ou seja, que não apresentavam um padrão na escrita, foi preciso estudar o uso de técnicas de inteligência artificial. Neste caso, definiu-se por aplicar técnicas de representação textual das palavras por *embeddings*. Para isso, foi utilizado o modelo pré-treinado BERT (DEVLIN et al., 2019), um dos modelos mais atuais e robustos que se tem acesso. Ao aplicar a representação das palavras por meio de *embeddings*, foi possível notar que as palavras inválidas mantinham-se agrupadas, o que tornou possível catalogá-las como "válidas" e "inválidas" e, assim, realizar a aplicação de um algorítmo de *machine learning* objetivando classificá-las. O modelo classificador de ruídos obteve um resultado bastante satisfatório de predição, o que torna possível a implementação de um algoritmo para automatização de classificação de ruídos em textos que contém especificações técnicas de *software* ou códigos de programação.

Dessa forma, a implementação dessas técnicas de classificação de textos traz uma ideia de implementação futura, que seria um método que deve receber como entrada um texto ruidoso e devolver um texto já pré-processado, visando facilitar ainda mais a etapa de pré-processamento dos dados textuais e diversas tarefas de PLN.

Referências

AHMAD, N. A. et al. Review of chatbots design techniques. **International Journal of Computer Applications**, v. 181, n. 8, p. 7–10, 2018. Citado na página 10.

CESKA, Z.; FOX, C. The influence of text pre-processing on plagiarism detection. In: ASSOCIATION FOR COMPUTATIONAL LINGUISTICS. [S.l.], 2011. Citado na página 14.

CHOETKIERTIKUL, M. et al. A deep learning model for estimating story points. **IEEE Transactions on Software Engineering**, IEEE, 2018. Citado 9 vezes nas páginas 7, 11, 12, 21, 22, 28, 29, 33 e 36.

CHOWDHURY, R. et al. A method based on nlp for twitter spam detection. Preprints, 2020. Citado na página 10.

CLARK, K. et al. Semi-supervised sequence modeling with cross-view training. **arXiv preprint arXiv:1809.08370**, 2018. Citado na página 18.

CONRADO, M. da S. et al. A survey of automatic term extraction for brazilian portuguese. **Journal of the Brazilian Computer Society**, Springer, v. 20, n. 1, p. 12, 2014. Citado na página 14.

DAI, A. M.; LE, Q. V. Semi-supervised sequence learning. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2015. p. 3079–3087. Citado na página 18.

DEVLIN, J. et al. Bert: Pre-training of deep bidirectional transformers for language understanding. **North American Association for Computational Linguistics (NAACL)**, 2019. Citado 5 vezes nas páginas 17, 18, 19, 33 e 36.

FÁVERO, E. M. D. B.; CASANOVA, D.; PIMENTEL, A. R. Se3m: A model for software effort estimation using pre-trained embedding models. **Information and Software Technology**, Elsevier, v. 147, p. 106886, 2022. Citado 3 vezes nas páginas 11, 12 e 36.

GHAREHCHOPOGH, F. S.; KHALIFELU, Z. A. Analysis and evaluation of unstructured data: text mining versus natural language processing. In: IEEE. **2011 5th International Conference on Application of Information and Communication Technologies (AICT)**. [S.l.], 2011. p. 1–4. Citado na página 14.

HAJ-YAHIA, Z.; SIEG, A.; DELERIS, L. A. Towards unsupervised text classification leveraging experts and word embeddings. In: **Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics**. [S.l.: s.n.], 2019. p. 371–379. Citado na página 17.

HASAN, M. R.; MALIHA, M.; ARIFUZZAMAN, M. Sentiment analysis with nlp on twitter data. In: IEEE. **2019 International Conference on Computer, Communication, Chemical, Materials and Electronic Engineering (IC4ME2)**. [S.l.], 2019. p. 1–4. Citado na página 10.

HOWARD, J.; RUDER, S. Universal language model fine-tuning for text classification. **arXiv preprint arXiv:1801.06146**, 2018. Citado na página 18.

KAO, A.; POTEET, S. R. **Natural language processing and text mining**. [S.l.]: Springer Science & Business Media, 2007. Citado na página 13.

KAO A. E POTEET, S. R. **Natural Language Processing and Text Mining**. [S.l.], 2007. 272 p. Disponível em: <https://learnersdesk.weebly.com/uploads/7/4/1/9/7419971/natural_language_processing_and_text_mining.pdf>. Acesso em: 13 de setembro de 2022. Citado na página 13.

KENNEDY, K. **How to Clean Data in Natural Language Processing (NLP)**. [S.l.], 2021. Disponível em: <https://python.plainenglish.io/nlp-data-cleansing-steps-6b4150cf87cf>. Acesso em: 19 de outubro de 2022. Citado 2 vezes nas páginas 16 e 20.

KHAN, N. S.; ABID, A.; ABID, K. A novel natural language processing (nlp)–based machine translation model for english to pakistan sign language translation. **Cognitive Computation**, Springer, v. 12, n. 4, p. 748–765, 2020. Citado na página 10.

MAATEN, L. v. d. **t-SNE**. [S.l.], 2022. Disponível em: <https://lvdmaaten.github.io/tsne/>. Acesso em: 16 de outubro de 2022. Citado na página 31.

MADAN, R. **NLP: Contextualized word embeddings from BERT**. [S.l.], 2019. Disponível em: <https://medium.com/analytics-vidhya/decisiontree-classifier-working-on-moons-dataset-using-gridsearchcv-to-find-best-hyperparameters-ede24a06
Acesso em: 16 de outubro de 2022. Citado na página 31.

MANNING, C. D.; RAGHAVAN, P.; SCHÜTZE, H. **Introduction to information retrieval**. [S.l.]: Cambridge university press, 2008. Citado 2 vezes nas páginas 15 e 16.

MCCANN, B. et al. Learned in translation: Contextualized word vectors. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2017. p. 6294–6305. Citado na página 18.

MELAMUD, O. et al. The role of context types and dimensionality in learning word embeddings. **arXiv preprint arXiv:1601.00893**, 2016. Citado na página 18.

MIKOLOV, T. et al. Efficient estimation of word representations in vector space. **arXiv preprint arXiv:1301.3781**, 2013. Citado 2 vezes nas páginas 16 e 17.

PETERS, M. E. et al. Deep contextualized word representations. **arXiv preprint arXiv:1802.05365**, 2018. Citado na página 18.

POGIATZIS, A. **NLP: Contextualized word embeddings from BERT**. [S.l.], 2019. Disponível em: <https://towardsdatascience.com/nlp-extract-contextualized-word-embeddings-from-bert-keras-tf-67ef29f60a7b>. Acesso em: 8 de outubro de 2022. Citado 4 vezes nas páginas 19, 20, 29 e 30.

PORRU, S. et al. Estimating story points from issue reports. In: **Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering**. [S.l.: s.n.], 2016. p. 1–10. Citado na página 21.

RADFORD, A. et al. Improving language understanding by generative pre-training. **URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf**, 2018. Citado na página 18.

RAMACHANDRAN, D.; PARVATHI, R. Analysis of twitter specific preprocessing technique for tweets. **Procedia Computer Science**, Elsevier, v. 165, p. 245–251, 2019. Citado 3 vezes nas páginas 10, 12 e 20.

SERVIAN, A. **Regex: Um guia prático para expressões regulares**. [S.l.], 2020. 272 p. Disponível em: <https://medium.com/xp-inc/regex-um-guia-pratico-para-expressÃµes-regulares-1ac5fa4dd39f>. Acesso em: 13 de setembro de 2022. Citado na página 15.

STRAKA, M.; HAJIC, J.; STRAKOVÁ, J. Udpipe: trainable pipeline for processing conll-u files performing tokenization, morphological analysis, pos tagging and parsing. In: **Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)**. [S.l.: s.n.], 2016. p. 4290–4297. Citado na página 14.

VASWANI, A. et al. Attention is all you need. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2017. p. 5998–6008. Citado 2 vezes nas páginas 18 e 19.

WYNNE, M. **Developing linguistic corpora: A guide to good practice**. [S.l.]: Oxbow Books Limited, 2005. Citado na página 14.

ZHANG, D. et al. Chinese comments sentiment classification based on word2vec and svmperf. **Expert Systems with Applications**, Elsevier, v. 42, n. 4, p. 1857–1863, 2015. Citado na página 17.

ZHU, Y. et al. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In: **Proceedings of the IEEE international conference on computer vision**. [S.l.: s.n.], 2015. p. 19–27. Citado na página 19.