FERNANDA YUKARI KAWASAKI

APLICAÇÃO DE BERT EM *LOGS* PARA CLASSIFICAÇÃO DE CÓDIGOS MALICIOSOS

(*versão pré-defesa, compilada em 11 de dezembro de 2023*)

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: André Ricardo Abed Grégio.

CURITIBA PR

2023

RESUMO

Este trabalho teve como objetivo avaliar o desempenho da aplicação do modelo BERT na classificação de códigos maliciosos e anomalias utilizando arquivos de *log*. Para isso, foram realizados estudos sobre o tema e assuntos correlatos, bem como um experimento prático utilizando um modelo BERT pré-treinado e um *dataset* de *logs* estáticos brasileiros. O resultado foi que o modelo BERT é ineficiente para classificação de códigos malicioso e detecção de anomalias.

Palavras-chave: BERT. Logs. Clasificação de malware.

ABSTRACT

The purpose of this study is to evaluate the performance of a BERT based model to classify malicious codes and detect anomalies using log files. To achieve this goal, a research about the topic and related works was conducted, along with a practical experiment using a pre-trained BERT model and a dataset of Brazilian static logs. The findings are as follows: the chosen BERT model was inefficient at classifying malicious codes and detecting anomalies.

Keywords: BERT. Logs. Mawlware classification.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Os sistemas e dispositivos tecnológicos têm sido aplicados em um número cada vez maior de atividades, tornando-se parte essencial de diversos setores. Sendo assim, é imprescindível que esses sejam seguros, isto é, capazes de garantir características como confidencialidade, disponibilidade e integridade.

Para isso, é possível utilizar *logs*, arquivos de texto que, ao registrar a atividade dos sistemas e aplicações, permitem entender e monitorar acontecimentos importantes. De forma geral, a análise de *logs* e detecção de anomalias vinham sendo feitas manualmente por especialistas ou estaticamente com regras pré-definidas. No entanto, esses tipos de análise possuem custo alto e progressivo em relação ao aumento crescente de dados, tornando-se inviáveis economicamente.

Ao mesmo tempo, a presença de mais dados pode viabilizar o uso de modelos de *machine learning*. Assim, diversos estudos têm sido conduzidos para avaliar sua aplicação nas áreas de detecção de anomalias e classificação de códigos maliciosos.

1.1 OBJETIVOS

Este trabalho tem como objetivo avaliar a viabilidade da aplicação do modelo pré-treinado BERT para a classificação de códigos maliciosos utilizando *logs* estáticos.

1.1.1 Objetivos Específicos

- Analisar os principais métodos de detecção de anomalias e classificação de *malware*;
- Compreender o funcionamento do modelo BERT e como ele tem sido aplicado ao processamento de linguagem e à segurança computacional;
- Realizar análise exploratória de dados em um *dataset* de *logs*;
- Verificar o desempenho do modelo BERT para detecção de anomalias e classificação de códigos maliciosos no *dataset* escolhido;
- Comparar os resultados obtidos com trabalhos similares.

1.2 ORGANIZAÇÃO DO DOCUMENTO

Esta monografia é composta por 5 capítulos. Este capítulo contém uma breve introdução e contextualização do tema, bem como apresentação dos objetivos que se pretendem alcançar com a realização do projeto.

O capítulo 2 descreve os conceitos essenciais para este trabalho, como a detecção de anomalias, análise de *logs* e o funcionamentos de modelos de *machine learning* e BERT. Por fim,

é feita a revisão da literatura, com análise dos trabalhos mais relevantes que empregam modelos de *machine learning* para detecção de anomalias e classificação de *malware*.

O capítulo 3 aborda a metodologia seguida durante a execução dos experimentos, que são descritos no capítulo subsequente.

O capítulo 4 apresenta os experimentos realizados, os resultados obtidos e limitações atuais.

Finalmente, o capítulo 5 traz um panorama geral do que foi realizado e descreve as conclusões obtidas, além de fazer um levantamento de trabalhos futuros a serem desenvolvidos.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 ANOMALIAS

Anomalia é uma observação que diverge tanto das outras observações a ponto de levantar suspeitas de que tenha sido gerado por um mecanismo diferente (Hawkins, 1980, tradução nossa). Contudo, determinar se uma observação é um outlier é um julgamento subjetivo, pois não há uma definição matemática geral. Existem métodos de detecção que possuem uma definição de outlier. Um exemplo visual é o boxplot, em que os pontos que não estão no intervalo [Q1-IQ*1,5, Q3+IQ*1,5] são considerados outliers. Por outro lado, também existem modelos em que analistas definem um limiar específico.

Aggarwal (2016) destaca que os outliers são gerados quando os sistemas e processos se comportam de forma atípica. Assim, estes dados podem ser analisados para extrair informações úteis sobre a aplicação. Este é o objeto de estudo da área de reconhecimento de padrões, que engloba algoritmos para identificar irregularidades automaticamente. Essas informações permitem que ações sejam tomadas para mitigar o problema.

2.1.1 Tipos de Anomalias

Para realizar a detecção de anomalias, é importante entender o tipo de anomalia que se quer detectar. Existem três tipos de anomalias (Chandola et al., 2009): anomalias pontuais, anomalias contextuais/condicionais e anomalias coletivas. As anomalias pontuais são o tipo mais comum, e ocorre se um dado individual difere consideravelmente de todos os outros dados. Já as anomalias contextuais ou condicionais são aquelas consideradas anômalas apenas quando estão inseridas em um contexto específico. O comportamento normal é descrito por meio de um atributo. Por exemplo, um *timestamp* pode ser utilizado como atributo de contexto ao analisar séries temporais. Por fim, as anomalias coletivas ocorrem quando um conjunto de dados é considerado anômalo ao se comparar com o dataset completo.

2.1.2 Detecção de Anomalias

Antes da difusão dos algoritmos de *machine learning* na área, a detecção de anomalias era feita principalmente de forma manual ou estática.

A detecção manual, como o nome indica, é feita manualmente por especialistas. As principais desvantagens incluem subjetividade da análise e a necessidade de muito conhecimento da área e do negócio. Ademais, o investimento de tempo para fazer as análises é considerável. Com uma quantidade cada vez maior de dados gerados, torna-se cada vez mais difícil manter esse tipo de atividade de forma escalável.

Já a detecção estática é baseada em regras que sinalizam a presença de anomalias e devem ser definidas por especialistas. O principal problema é que somente as anomalias compreendidas pelas regras serão identificadas. Logo, se uma anomalia aparecer, ela não será identificada inicialmente e será necessário criar uma nova regra. Vale pontuar a importância das regras serem capazes de separar corretamente as classes. Se forem muito abrangentes, elas podem gerar muitos falsos positivos. Por outro lado, se forem muito restritivas, elas podem não identificar vários verdadeiros positivos. Para cada caso, é necessário levar em conta essas contrapartidas e identificar a combinação que melhor se adapta ao contexto.

Por fim, a detecção de anomalias com algoritmos tem se tornado cada vez mais comum, pois é mais adaptável e necessita de pouca intervenção humana. O fluxo geral consiste em treinar modelos de machine learning apenas com dados normais. Assim, se o modelo não conseguir identificar o comportamento normal, considera-se que é uma anomalia. Nas fases de teste e de uso em produção, tanto anomalias quanto dados normais são utilizados.

O modelo LogBERT (Guo et al., 2021), por exemplo, recebe uma sequência de *logs* que tiveram alguns de seus *tokens* substituídos por máscaras. Em seguida, ele identifica cada uma delas e calcula os *tokens* que têm maior chance de ser o valor mascarado. Se o *token* original não aparece nos G tokens mais prováveis, então a sequência de *logs* é considerada anomalia.

2.2 LOGS

Logs são arquivos de texto semi-estruturados que registram a atividade de sistemas operacionais e *softwares*. Eles permitem entender e monitorar acontecimentos dentro do sistema. Ao utilizar técnicas de detecção de anomalias em *logs*, é possível identificar possíveis ataques e intrusões que ocorreram ou que estão prestes a ocorrer.

Porém, problemas tais como diferentes padrões de *logging* e a quantidade crescente de dados gerados tornam a análise manual e estática deles bastante dispendiosa. Ainda, os ataques vêm se tornado cada vez mais sofisticados, sendo capazes de contornar métodos que visam identificá-los. Além disso, o aumento da complexidade dos sistemas e aplicações abre espaço para mais *bugs* e vulnerabilidades que podem ser exploradas (Du et al., 2017).

Os sistemas de detecção de intrusão (IDS) são um exemplo disso. São analisados vários dados do sistema, como de tráfego de rede e de chamadas dentro do sistema operacional. Neste caso, um *outlier* pode ser um bom indicativo de atividades maliciosas e de intrusão, o que é bastante útil no contexto de segurança de sistemas.

Por fim, também é de suma importância considerar a possibilidade de que os *logs* tenham sido comprometidos (Arp et al., 2020). A fim de camuflar atividades suspeitas, alguns ataques envolvem a alteração dos *logs* para remover evidências.

2.2.1 Logs estáticos e dinâmicos

É possível categorizar *logs* com base no momento em que as informações foram registradas. Os *logs* estáticos são gerados antes da compilação ou execução do código. Um exemplo é o *Brazilian Malware Dataset* (Ceschin et al., 2018), em que as informações foram extraídas do cabeçalho de arquivos *Portable Executable*, que são executáveis no sistema operacional *Windows*.

Já os *logs* dinâmicos são gerados durante a execução do código ou do sistema. Eles podem conter vários tipos diferentes de informação, como erros, tráfego de rede e chamadas de sistema. Na literatura, existem diversos *datasets* públicos obtidos dinamicamente, como o *BGL*, que contém dados de *logs* coletados no *BlueGene/L supercomputer system*.

2.3 MACHINE LEARNING

Aprendizado de máquina, ou, em inglês, *machine learning* (ML) é uma área de ciência de computação baseada no entendimento que sistemas são capazes de aprender com base em dados e utilizar as informações obtidas para identificar padrões, gerar estimativas e auxiliar a tomada de decisões (Shalev-Shwartz e Ben-David, 2014). Esta área, que vem se tornando cada vez mais disseminada, engloba o estudo dos algoritmos, modelos e técnicas que permitem que as máquinas consigam aprender e se adaptar, melhorando sua performance com o mínimo de intervenção humana.

2.3.1 Tipos de Aprendizado

Uma das formas de se classificar de modelos de *machine learning* é utilizando as características do *dataset* de treinamento e a maneira como o modelo aprende com eles. Os três tipos de aprendizado mais conhecidos são supervisionado, não supervisionado e auto-supervisionado.

O **aprendizado supervisionado** necessita de entradas rotuladas durante a fase de treinamento, ou seja, elas devem ser acompanhadas pelas respectivas saídas esperadas. Com isso, o modelo consegue identificar as relações e padrões existentes entre entrada e saída.

Já o **aprendizado não supervisionado** caracteriza-se pela ausência de dados rotulados, inclusive na etapa de treinamento. Assim, o modelo precisa calcular os resultados de outras formas. A clusterização, por exemplo, é um método que separa os dados em grupos com características semelhantes, chamados *clusters*. Apesar de ser possível treinar o modelo sem os rótulos, eles são necessários para a avaliação do desempenho, já que baseia-se na comparação entre o resultado esperado e o obtido.

Também é possível utilizar tarefas auxiliares que, ao serem executadas, ajudam o modelo a capturar relações entre os dados fornecidos. Esta forma de aprendizado chama-se aprendizado auto-supervisionado. O modelo descrito em *LogBERT* é um exemplo disso, já que o treinamento consiste nas tarefas de estimativa de *log key* mascarada e minimzação de volume da hiperesfera.

2.3.2 Aprendizado por transferência (*Transfer Learning*)

O aprendizado por transferência é um método em que o modelo é pré-treinado com antecedência utilizando grandes quantidades de dados. As configurações e pesos obtidos são armazenados e, depois, este conhecimento é utilizado como base para outras tarefas. Então, os modelos pré-treinados são apenas ajustados com o *dataset* escolhido.

Com a criação modelos mais complexos que precisam de quantidades cada vez maiores de dados e de poder de processamento, esta abordagem tornou-se muito relevante e inclusiva ao permitir que eles sejam utilizados em computadores normais.

2.4 REDES NEURAIS

As redes neurais são modelos que consistem no uso de neurônios interconectados, que propagam informações pelas várias camadas da rede. Cada uma delas é composta por múltiplos neurônios, que processam entradas e propagam os resultados obtidos para as camadas seguintes.

É válido pontuar que os erros também se propagam pela rede e são acumulados com os erros das camadas anteriores, gerando um efeito "bola de neve". Então, a cada iteração, são realizados os cálculos de gradiente, que quantificam a taxa de erro em relação aos pesos. Depois, esta informação é utilizada para o ajuste dos pesos das iterações seguintes, o que permite amenizar o efeito dos erros na rede.

Os modelos mais conhecidos são:

2.4.1 *Recurrent Neural Network*

Recurrent Neural Network (RNN) é um tipo de rede neural que utiliza os valores que precedem a entrada atual para produzir o *output* correspondente. Sendo assim, modelos deste tipo são capazes de reconhecer padrões de natureza sequencial. Assim, são bastante relevantes no contexto de processamento de linguagem natural, em que os dados são sequenciais.

Entretanto, esse modelo sofre com o *Vanishing Gradient Problem*, em que o gradiente, utilizado para ajustar os pesos da rede, diminui de forma exponencial. Dessa forma, a rede perde informações sobre as iterações anteriores. Para evitar esse problema, duas soluções foram propostas: GRUs e LSTMs.

2.4.2 *Gated Recurrent Unit*

Gated Recurrent Unit (GRU) é um tipo de rede neural que introduz o uso de *gates*, mecanismos que controlam o fluxo de informação. Para definir a importância das informações da célula anterior e quais informações devem ser mantidas, são utilizados, respectivamente, o *reset gate* e o *update gate*. Assim, é possível definir o que deve ser lembrado e o que deve ser esquecido em cada etapa.

2.4.3 *Long Short-Term Memory*

Long Short-Term Memory (LSTM) é um tipo de rede neural bastante similar ao GRU, pois também utiliza *gates* para lidar com o *Vanishing Gradient Problem*. Apesar de serem utilizados de forma semelhante, as LSTMs utilizam 3 *gates* para controlar o fluxo de informações dentro da rede (*input gate* , *forget gate* e o *output gate*), o que faz com que elas sejam mais eficazes do que as GRUs. Outra adição é o *cell state*, que funciona como uma memória de longo prazo.

Porém, a desvantagem é que tanto GRU quanto LSTM precisam de poder de processamento maior do que as RNNs simples. Por fim, é importante notar que os modelos discutidos anteriormente são todos sequenciais. Ou seja, é preciso ter o resultado das iterações anteriores para a calcular os valores da iteração atual.

2.5 BERT

Bidirectional Encoder Representations from Transformers (BERT) (Devlin et al., 2019) é um *framework* que se baseia no uso dos *transformers* e de seu mecanismo de atenção para processar textos (Vaswani et al., 2017).

2.5.1 Transformer

O *transformer* (Vaswani et al., 2017) é uma arquitetura de rede neural que soluciona os problemas de memória e processamento das GRUs e LSTMs. Para isso, utiliza o mecanismo de atenção, em que a rede analisa todas as palavras ao mesmo tempo e define pesos para indicar a sua importância em relação à palavra atual. Esta mudança viabiliza a detecção de dependências e relações entre palavras em posições distantes. Ainda, as camadas de atenção podem ser utilizadas em conjunto, o que permite que várias características da entrada sejam observadas simultaneamente.

Por não haver dependência entre os mecanismos de atenção de cada palavra e entre as camadas, é possível realizar os cálculos paralelamente, o que reduz significativamente seu tempo de execução.

2.5.2 Avanços de BERT

Um dos principais diferenciais de BERT é sua capacidade de compreender o contexto de palavras em ambas as direções. Isso permite ao modelo compreender significados e relações semânticas de maneira mais precisa, em contraste com modelos anteriores que eram predominantemente unidirecionais. Dessa forma, o modelo BERT oferece diversas vantagens, incluindo a capacidade de entender particularidades semânticas e adaptar-se a tarefas específicas.

Por outro lado, suas principais desvantagens são a necessidade de grandes volumes de dados de treinamento e a alta demanda por recursos computacionais. Por isso, vários modelos BERT pré-treinados podem ser encontrados na Internet.

2.5.3 Treinamento

O treinamento de BERT envolve o uso de grandes conjuntos de dados de texto e *masking tasks*. Durante o pré-treinamento, partes do texto são mascaradas, e o modelo é treinado para prever as palavras mascaradas com base no contexto . Isso leva à criação de representações de palavras ricas em informações contextuais.

Após o pré-treinamento, os modelos de BERT podem ser ajustados com dados específicos, processo denominado *fine tuning*. Assim, estes modelos podem ser aplicados em tarefas específicas de PLN, como classificação de texto, resolução de questões, análise de sentimento, entre outras.

2.6 PROCESSAMENTO DE LINGUAGEM NATURAL

O processamento de linguagem natural (NLP) é uma área da computação que estuda problemas como geração e compreensão de linguagem natural humana. Abordaremos duas tarefas, que são utilizadas no início do processo de NLP.

2.6.1 Tokenização

É a ação de dividir sentenças em unidades menores chamadas *tokens*. Dessa forma, os algoritmos de *machine learning* podem processar e interpretar cada parte do texto separadamente, o que se aproxima do funcionamento da linguagem humana. Por isso, a escolha de um tipo adequado de tokenização afeta diretamente o desempenho de um modelo de ML.

Existem diversas formas de tokenizar uma sentença, como, por exemplo, baseada em espaços, caracteres ou *WordPieces*.

2.6.1.1 Tokenização baseada em caracteres

Uma das formas mais simples para tokenizar sequências é utilizando caracteres. Ou seja, cada letra do alfabeto é considerada como um *token*. Porém, a separação baseada em caracteres perde informações semânticas das palavras, o que, na maioria das vezes, leva a um desempenho pior dos modelos.

2.6.1.2 Tokenização baseada em espaços

Realiza a separação de *tokens* utilizando em espaços no texto. Por exemplo, `The quick brown fox jumps over the lazy dog.` torna-se `[The, quick, brown, fox, jumps, over, the, lazy, dog, .]`.

No entanto, esta forma de tokenização tem dois problemas:

- O tamanho do vocabulário de uma linguagem é muito extenso, o que dificulta o processamento computacional
- As variações de uma única palavra são consideradas como *tokens* diferentes.
- A separação baseada em espaços nem sempre é precisa. Por exemplo, `Campo Grande` é dividido em `['Campo', 'Grande']`. As duas palavras são utilizadas para expressar o nome de uma única cidade. Mas, ao separá-las em *tokens* diferentes, cada uma apresenta um valor semântico diferente.

2.6.1.3 Tokenização baseada em WordPieces (BERT)

Uma outra maneira de tokenizar textos é utilizando *WordPieces*, que são partes das palavras do texto. Cada palavra é representada por um número variável de *tokens*.

A principal vantagem deste método é a melhor adaptação a palavras mais raras, já que o modelo consegue utilizar combinações de subpalavras já vistas para reconhecer seu significado. Ademais, este tipo de tokenização também é capaz de processar vocabulários extensos de maneira eficiente.

Neste trabalho, foi utilizado o modelo de tokenização baseada em *WordPieces* disponível na biblioteca `transformers`. Como esta implementação foi criada específicamente para BERT, os parâmetros e configurações já estão definidos corretamente, como por exemplo o número máximo de *tokens*, que é 512 para o BERT.

2.6.2 *Encoding* e *Embedding*

Depois que os dados estiverem tokenizados, é necessário transformá-los em um formato adequado para que possam ser utilizados como entrada para os modelos. Para textos, isso geralmente significa fazer o *encoding* de informações, isto é, codificando os dados já tokenizados em vetores numéricos. Esta representação das palavras no espaço vetorial chama-se *Word Embedding*.

É relevante mencionar que o uso de vetores numéricos para representação tende a ser mais eficiente. O processamento é realizado em menos tempo, e, dependendo do método utilizado, o espaço ocupado em memória também é menor. Existem variados métodos para realizar esta conversão, como, por exemplo, o *Word2Vec*.

2.6.2.1 Word2Vec

Ao contrário de modelos tradicionais de *encoding*, o *Word2Vec* é capaz de representar os dados textuais de uma forma que captura o significado de cada palavra e suas relações, preservando-os dentro do espaço vetorial. Assim, palavras com significados parecidos, ou que aparecem em contextos parecidos, possuem menores distâncias entre si dentro do espaço vetorial.

O treinamento do *Word2Vec* envolve a utilização de grandes volumes de texto, como um corpus de documentos. O modelo ajusta seus parâmetros com base na tarefa de previsão de palavras vizinhas. Durante o treinamento, o *Word2Vec* atribui vetores a palavras, de modo que palavras semelhantes no contexto tenham vetores próximos no espaço vetorial. Hiperparâmetros como o tamanho da janela, a dimensão do vetor e a taxa de aprendizado podem ser ajustados para otimizar o desempenho.

No entanto, vale mencionar que este modelo requer expressivas quantidades de dados para treinamento eficaz. Ainda, pode ter dificuldade em representar palavras raras.

2.6.3 BERT *embeddings*

A representação de palavras de BERT é um pouco diferente. Ao invés de atribuir vetores representativos para cada palavra, BERT utiliza as informações do contexto para representar a palavra.

Além disso, outras características também são levadas em consideração, como o posicionamento de cada palavra na sentença. Essas informações adicionais são concatenadas aos dados de entrada e depois propagadas pelas camadas, que refinam as representações a cada passo.

2.7 MÉTRICAS DE AVALIAÇÃO

Depois de treinar e testar os modelos, são utilizadas métricas de avaliação que comparam os resultados obtidos com os resultados esperados. Dessa forma, é possível analisar o desempenho do modelo sob diversas perspectivas.

Para o cálculo das métricas, são utilizadas as quantidades de erros e acertos para cada classe. Estas informações podem ser observadas na matriz de confusão.

2.7.1 Matriz de Confusão

A matriz de confusão é uma maneira de visualizar os acertos e erros do modelo, separados por classe. Os valores "Verdadeiro Negativo"e "Verdadeiro Positivo"indicam, respectivamente, o número de previsões corretas para *label* 0 (*goodware*) e para *label* 1 (*malware*. Já os valores "Falso Negativo"e "Falso Positivo"indicam, respectivamente, o número de previsões incorretas para *label* 0 (*goodware*) e para *label* 1 (*malware*).

True Label	Verdadeiro Negativo (TN)	Falso Positivo (FP)
	Falso Negativo (FN)	Verdadeiro Positivo (TP)
	Predicted Label	

Tabela 2.1: Configuração da matriz de confusão

2.7.2 Métricas

Com os valores obtidos da matriz de confusão, é possível calcular diversas métricas. As mais conhecidas são acurácia, precisão, recall e F1-Score.
A acurácia calcula a porcentagem de acertos, independentemente da classe.

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

A precisão calcula a porcentagem que o de acertos do modelo, considerando apenas os casos considerados como anomalias.

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

A revocação ou recall calcula a porcentagem de acertos do modelo, considerando apenas as anomalias de fato.

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

O F1-Score calcula a média harmônica entre precisão e recall

$$F1 = \frac{2 * precision * recall}{precision + recall}$$

Outra métrica relevante é a curva ROC (*Receiver Operating Characteristic*), que relaciona a taxa de falsos positivos (eixo x) e verdadeiros positivos (eixo y). Assim, é possível verificar se o modelo está prevendo corretamente as *labels* positivas. A AUC (*area under curve*) é uma métrica obtida ao calcular a área sob a curva ROC.

Quanto maior for a AUC, melhor a eficácia do modelo. Graficamente, é possível notar que a curva se aproxima de $y = 1$ à medida que a taxa de verdadeiros positivos é aumenta e a de falsos positivos diminui. Um exemplo é a figura 4.7, obtida durante os experimentos.

2.7.3 *Overfitting*

Overfitting é um termo utilizado quando um modelo se ajusta muito bem aos dados de treino, mas não é genérico o suficiente para ser utilizado com outros dados. Assim, durante o treinamento é possível observar um desempenho muito bom, porque o modelo foi ajustado de forma excessiva aos dados fornecidos. No entanto, ao utilizar o modelo com outros dados novos, percebe-se que as previsões são imprecisas.

É comum observar este fenômeno quando o modelo é treinado demasiadamente, como, por exemplo, utilizando muitas iterações (*epochs*). Até certo ponto, treinar mais o modelo melhora seu desempenho. No entanto, se o limite for ultrapassado, ocorrerá o efeito contrário.

2.7.4 Métricas são o suficiente?

Apesar de serem bons indicadores de performance, convém ressaltar que uma métrica sozinha não é capaz de capturar todos os detalhes sobre o desempenho do modelo. Por exemplo, ao observar uma acurácia de 90%, pode ser instintivo concluir que o modelo é efetivo. Todavia, existem casos em que isso não se confirma. Um exemplo é um modelo que classifica se uma compra é fraude ou não. O desequilíbrio entre classes é inerente ao contexto, já que a quantidade de compras legítimas é consideravelmente maior. Logo, mesmo quando o modelo prevê que todas as compras são legítimas, a acurácia obtida é reletivamente alta.

Para ilustrar este caso, consideremos que 10% do dataset corresponde à fraudes. Então, a matriz de confusão seria parecida com:

True Label	90 (TN)	0 (FP)
	10 (FN)	0 (TP)
	Predicted Label	

Tabela 2.2: Exemplo de matriz de confusão

Calculando a acurácia, temos:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} = \frac{90 + 0}{90 + 0 + 10 + 0} = 90\%$$

Portanto, é fundamental que a análise do resultado seja feita de forma detalhada, além de prestar atenção na forma como os dados estão distribuídos entre as classes.

2.8 TRABALHOS RELACIONADOS

O modelo BERT é bastante versátil e, na literatura, é possível encontrar diversas aplicações com alta eficácia, sendo que, em alguns casos, foi alcançado o estado da arte. As principais atividades em que o modelo é utilizado envolvem Processamento de Linguagem Natural (NLP), o que engloba tarefas como tradução de textos, análise de sentimentos e geração de respostas para perguntas.

O framework CCBERT (Zhang et al., 2022), por exemplo, permite melhorar os títulos de perguntas do *Stack Overflow*, que é um fórum utilizado por desenvolvedores. Um problema bastante comum nesta plataforma é a criação de perguntas e títulos mal formulados, o que dificulta a resolução da questão. O modelo é capaz de captar informações semânticas presentes no corpo da questão, gerando títulos concisos, mas que envolvem as informações relevantes.

Existem também modelos que envolvem linguagens de programação, tal como o CodeBERT (Feng et al., 2020). Este *framework* tem a capacidade de captar as relações entre linguagem de programação e linguagem natural, o que permite a realização de tarefas como

pesquisa de trechos de código utilizando linguagem natural e geração de documentação de códigos, uma etapa que geralmente é ignorada ou feita às pressas.

Outra tarefa cada vez mais comum é a detecção de *malware* a partir de arquivos de log. Será feita uma breve análise de alguns *frameworks* e modelos.

2.8.1 Modelos de detecção de anomalias

Nota-se que a maioria dos trabalhos desta área seguem um fluxo similar, composto pelas etapas iniciais de *parsing*, agrupamento e representação de *logs*. Por fim, são utilizados modelos para detectar anomalias, que indicam possíveis ataques e invasões (Le e Zhang, 2022).

2.8.1.1 LogBERT

O framework proposto por Guo et al. (2021) consiste em aplicar BERT para detecção de anomalias em sequências de logs. Para isso, o modelo BERT é treinado apenas com logs considerados normais. Para o aprendizado auto-supervisionado, são utilizadas as tarefas **(i)** *Masked Log Key Prediction (MLKP)* e **(ii)** *Volume of Hypersphere Minimization (VHM)*. A primeira tarefa tem o foco em prever *log keys* mascaradas, enquanto a segunda é uma função matemática que visa melhorar os resultados da tarefa anterior.

A ideia central do trabalho é agrupar *keys* de sequências normais no espaço vetorial, utilizando como base a ideia de que sequências normais de logs possuem padrões similares entre si. Para isso, cada sequência de *log* tem uma *key* substituída pelo *token* especial `[MASK]`. O modelo recebe as sequências mascaradas e, se a *key* substituída estiver entre os G valores com maior probabilidade, então ela não é considerada anomalia. Caso contrário, a *key* é considerada anômala. Se a sequência de *logs* escolhida possuir mais do que R *keys* consideradas anômalas, então a sequência também é considerada anômala.

Os testes foram realizados em três *datasets* públicos: HDFS, BGL e ThunderBird. Além disso, a implementação original está disponível publicamente. Assim, foi possível reproduzir os testes e resultados.

2.8.1.2 DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning

DeepLog (Du et al., 2017) é um *framework* que aplica um modelo de rede neural LSTM para detecção e diagnóstico de anomalias.

Os dados foram extraídos de *logs* de sistema, pois registram eventos importantes em tempo real. A ideia é aplicar o modelo para o monitoramento de eventos que acabaram de ocorrer, o que é chamado de monitoramento *online*.

As entradas de *log* são vistas como elementos de uma sequência que segue certos padrões e regras gramaticais, que são originados pela lógica e fluxos de controle da implementação. E, apesar de ser mais restrita, é possível encontrar semelhanças com a linguagem natural. Assim como outros trabalhos, o modelo é treinado apenas para reconhecer *logs* normais.

Muitos modelos de detecção de anomalias da literatura funcionam como uma caixa preta. Isto é, não é possível identificar as características utilizadas para a previsão do modelo, o que dificulta a análise das anomalias. Por isso, o DeepLog também tem a proposta de auxiliar o diagnóstico, utilizando autômatos finitos para construir o *workflow* das tarefas.

Os testes foram realizados com dois *datasets* públicos: HDFS e OpenStack. O código para execução do *framework* não é fornecido pelos autores. No entanto, os autores de outro trabalho relacionado (van Ede et al., 2022) disponibilizaram a implementação do *DeepLog* que fizeram como parte de seu artigo com o intuito de comparar os resultados dos modelos.

2.8.1.3 Log-based Anomaly Detection with Deep Learning: How Far Are We?

Este artigo (Le e Zhang, 2022) foi elaborado considerando o contexto de surgimento de vários modelos propostos para identificar anomalias de sistema utilizando logs. Os resultados obtidos indicam alto poder preditivo, com performance igual ou mesmo melhor do que o estado da arte até então. Contudo, percebe-se que esta questão ainda não foi totalmente solucionada fora do ambiente acadêmico. Por isso, 5 modelos são analisados a fim de apresentar uma perspectiva geral do cenário presente de detecção de anomalias baseada em logs. Para os experimentos, foram utilizados 4 datasets públicos.

Um dos principais problemas é a baixa capacidade de generalização dos modelos, o que explica a baixa performance na maioria dos datasets. Além disso, identificou-se o uso de métricas insuficientes para avaliar o desempenho e *data leakage*, nos casos em que os dados de treinamento forem selecionados aleatoriamente.

Assim, os resultados aparentemente satisfatórios não são refletidos no cenário real de detecção de anomalias em logs. Por isso, é imprescindível que metodologias mais rigorosas sejam aplicadas ao realizar os estudos e que os autores façam análises críticas sobre seus resultados.

2.8.2 Modelos de classificação

Durante a fase de treinamento, os modelos de classificação aprendem a reconhecer cada uma das classes por meio de regras e padrões. Portanto, eles são treinados com dados de ambas as classes.

2.8.2.1 The Need for Speed: An Analysis of Brazilian Malware Classifiers

Este trabalho descreve o processo de análise de um *dataset* de *malware* brasileiro. Os dados são extraídos com um *crawler*, e, depois, normalizados.

As colunas textuais são processadas, e têm valores atípicos removidos utilizando o cálculo TF-IDF, que define a importância de cada palavra dentro do texto. Os valores com frequência menor do que 10% e maior do que %45 são removidos. Estes dados são concatenados às colunas numéricas, e, depois, essa combinação obtidas é utilizada para treinar os modelos.

Os testes utilizaram quatro tipos diferentes de modelo de classificação: *multilayer perceptron* (MLP), *support vector machine* (SVM), *K-nearest-neighbors* (KNN) e *Random Forest*. Também vale pontuar que os testes são feitos de maneira incremental, com o intuito de simular o uso real dos modelos.

De forma geral, todos os modelos apresentaram bons resultados. Apenas em alguns meses específicos houve piora do desempenho.

3 METODOLOGIA

A metodologia seguida para a execução do experimento deste trabalho consiste em:

- Realizar a análise exploratória dos dados. O intuito é entender mais sobre o *dataset* e explorar a forma como as classes estão distribuídas. Por fim, verificar a presença de valores atípicos e os possíveis efeitos no desempenho do modelo;
- Fazer o pré-processamento do *dataset*, garantindo que esteja no formato adequado para ser utilizado pelos modelos;
- Tokenizar os dados e fazer o *encoding*;
- Realizar o *fine tuning* dos modelos BERT pré-treinados de classificação e de *masked language prediction*;
- Verificar o desempenho dos modelos com as métricas estudadas;
- Executar testes com diferentes combinações do número máximo de *tokens* e o tamanho das *batches* para verificar o impacto nos resultados.

4 EXPERIMENTOS E RESULTADOS

Com o objetivo de avaliar o desempenho do modelo BERT para detecção de anomalias e a classificação de códigos maliciosos a partir de arquivos de *logs*, foram criados dois códigos experimentais, que utilizam como base modelos pré-treinados BERT. Os detalhes técnicos de implementação e resultados obtidos serão analisados a seguir.

4.1 AMBIENTE

O ambiente utilizado para executar os experimentos foi o *Google Colab Pro*. Há uma pequena variação das especificações de cada sessão. Porém, as GPUs disponibilizadas sempre possuem entre 12 e 16 GB de memória.

4.2 *DATASET*

O *Brazilian Malware Dataset* (Ceschin et al., 2018) foi selecionado para a realização dos experimentos. O *dataset* é composto por *logs* estáticos obtidos através da análise do cabeçalho do código executável *Portable Executable*. Os dados foram extraídos a partir de amostras de *goodware* e *malware* coletadas por meio de um *crawler* no cyberespaço brasileiro no período de 2012 a 2018. O arquivo possui 50181 registros, divididos em 29.065 *malwares* e 21.116 *goodwares*. Devido à uma queda no servidor de armazenamento, os dados de *malware* no período de Janeiro a Julho de 2016 foram perdidos. A figura 4.8 ilustra a distribuição dos dados ao longo do tempo e o impacto deste fato será discutido na seção 4.4.

4.3 EXPERIMENTOS

Foram realizados dois experimentos. O primeiro utiliza o modelo BERT para classificar entre *malware* e *goodware*. Já o segundo utiliza BERT para prever os *tokens* mascarados.

4.3.1 Preparação do *dataset*

O processo de preparação do *dataset* foi semelhante para ambos os experimentos. O *dataset* foi divido entre treinamento e teste, representando 80% e 20% dos dados, respectivamente.

O escopo deste trabalho está limitado a duas colunas textuais, que contém dados das bibliotecas dinâmicas e dos símbolos importados. É importante notar que estes dados foram previamente processados e normalizados (Ceschin et al., 2018). Por isso, os textos são compostos apenas por letras minúsculas e não contém acentos ou caracteres especiais.

Vale ressaltar que, como estes dados são influenciados pelo tempo, o *dataset* foi ordenado pela data de criação antes de ser separado. Dessa forma, evita-se um *data leakage*, no qual o modelo tem acesso a dados futuros, que não estariam disponíveis em um cenário real.

4.3.2 Pré-processamento

As duas colunas textuais utilizadas contém os nomes das DLLs (Dynamic Link Libraries) e os símbolos importados. No entanto, o modelo BERT deve receber como entrada uma única sequência. Por isso, foi criada uma nova coluna `Corpus`, que é a concatenação dessas colunas.

Como os dados de entrada estão em formato textual (*strings*), é necessário realizar dois métodos de PLN: *tokenization* e *encoding*. Apesar de os valores contidos nas colunas não serem linguagem humana natural, ainda é possível separá-los em *tokens* que permitem que o modelo identifique relações semânticas e padrões. Além disso, observa-se que, muitas vezes, os termos utilizados para nomear bibliotecas dinâmicas e símbolos possuem alguma semelhança com a linguagem humana.

Neste experimento, foi utilizado o *BertTokenizer*, que, além de tokenizar, também realiza o *encoding* das sequências. Também é possível configurar a adição de *tokens* especiais, que são indicados entre colchetes. Dessa forma, é possível sinalizar ao modelo o momento em que uma sentença se inicia, por exemplo.

```
[CLS] oleaut32dll advapi32dll user32dll kernel32dll kernel32dll user32dll
msim [SEP]
```

Sentença 4.1: Sentença original

```
['ole', '##au', '##t', '##32', '##dl', '##l', 'ad', '##va', '##pi', '##32',
'##dl', '##l', 'user', '##32', '##dl', '##l', 'kernel', '##32', '##dl',
'##l', 'kernel', '##32', '##dl', '##l', 'user', '##32', '##dl', '##l', 'ms',
'##im']
```

Sentença 4.2: Sentença tokenizada

Vale notar que existem diversas repetições nos textos, como, por exemplo na sentença 4.1, onde `kernel32dll` aparece duas vezes. Ao analisar o *dataset* completo, percebe-se que algumas bibliotecas se repetem inúmeras vezes. Este fator pode afetar negativamente o desempenho do modelo, pois dificulta a diferenciação entre as classes.

DLLs	Nº de linhas	% de linhas	Nº de ocorrências
kernel32.dll	40089	79.8%	89119
user32.dll	34181	68.1%	54312
advapi32.dll	33402	66.5%	49982
oleaut32.dll	29243	58.2%	51728

Tabela 4.1: Frequência de DLLs

Além disso, a repetição de termos gera o problema de truncamento de sentenças. Isto é, se um texto ultrapassa o limite de *tokens*, ele será reduzido até ter o tamanho adequado, possivelmente descartando informações mais relevantes. Em função da sua arquitetura, BERT consegue receber até 512 *tokens* por registro. Porém, é possível escolher um valor específico, desde que esteja dentro do limite. Foram realizados testes com os valores 16, 32, 64, 128 e 256.

A figura 4.1 é um histograma que representa a quantidade de *tokens* por sentença. Mesmo para o limite de 256 *tokens*, observa-se que a maioria delas foi truncada.

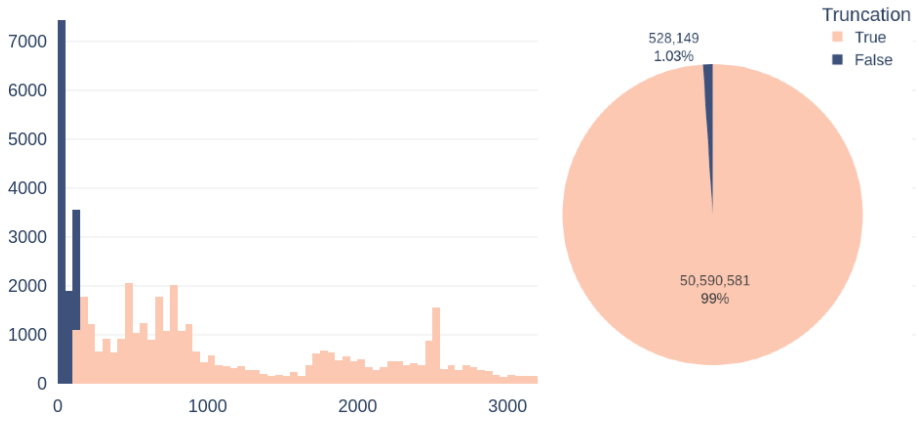

Figura 4.1: Distribuição do número de tokens e proporção de truncation no dataset original.

Para tentar contornar esta situação, também foram feitos testes com os termos únicos de cada sentença. A nova distribuição pode ser observada na figura 4.2. Comparativamente, a quantidade de truncamentos é menor, mas continua sendo a grande maioria.

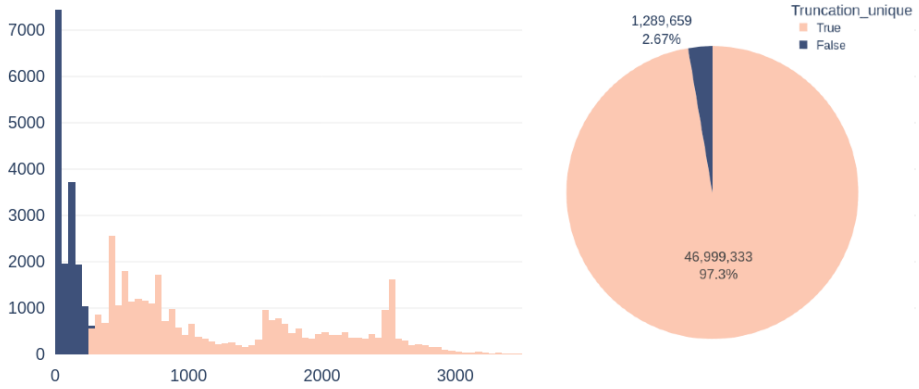

Figura 4.2: Distribuição do número de tokens e proporção de truncation no dataset com remoção de duplicidade.

Por um lado, a presença de termos repetidos e seu posicionamento no texto pode conter informações importantes. Por outro, a remoção de repetições torna possível que uma parte maior de símbolos e DLLs sejam recebidas pelo modelo.

4.3.3 Modelo

Como visto na seção de fundamentação teórica, BERT tem implementações de modelos pré-treinados. Então, o foco será realizar o *fine tuning* para as tarefas de classificação e *masked language model*, utilizando os dados de DLLs e símbolos importados.

O modelo escolhido para este experimento foi o `bert-base-uncased` com a implementação da biblioteca *PyTorch*.

4.4 RESULTADOS

4.4.1 Aprendizado Supervisionado

O modelo *BertForSequenceClassification* é treinado com ambas as classes (*malware* e *goodware*). Apesar de o dataset ser equilibrado, todos os dados de teste foram considerados maliciosos. Verificou-se que todos os *scores* eram valores entre 0.9 e 1.0, o que indica provável *overfitting*. Por padrão, os valores maiores que 0.5 são considerados como *label* 1 (*malware*) e os menores, *label* 0 (*goodware*). Assim, com o *threshold* padrão do BERT, foram obtidos: 0.46 de acurácia, 0.46 de precisão e 0.63 de F1-Score.

Mesmo com valores diferentes para tamanho de *batch* e máximo de *tokens*, os resultados foram iguais. A remoção de termos duplicados também não apresentou diferenças significativas.

4.4.1.1 Novo threshold

Tendo isso em mente, a mediana foi utilizada como novo *threshold*. Dessa forma, a definição das *labels* fica menos sensível a valores altos. Isso quer dizer que o modelo deve ter bastante certeza de que a *label* é 1 para, de fato, ser considerada 1. Os resultados são apresentados nas figuras 4.3 e 4.4. A partir da *epoch* 14, nota-se uma gradativa piora da acurácia, o que é um indicativo de *overfitting*.

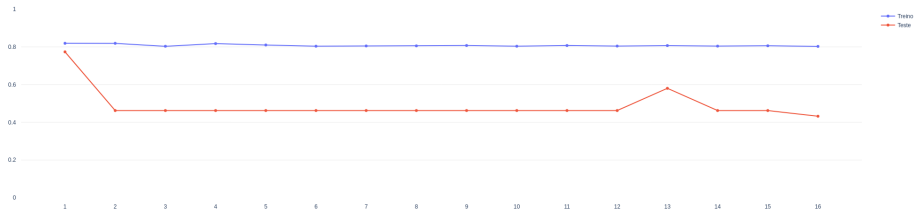

Figura 4.3: Acurácia por epochs no *dataset* original

Com o novo *threshold*, também é possível comparar o desempenho da *dataset* com remoção de duplicidade. Os *batches* de tamanho 4 resultaram em bastante *overfitting*, pois até mesmo os *scores* calculados pelo modelo eram iguais.

A maior acurácia é vista em 4.5 utilizando *dataset* original, com 32 *tokens* e *batch* de tamanho 8. Contudo, os outros valores neste mesmo gráfico são consideravelmente menores.

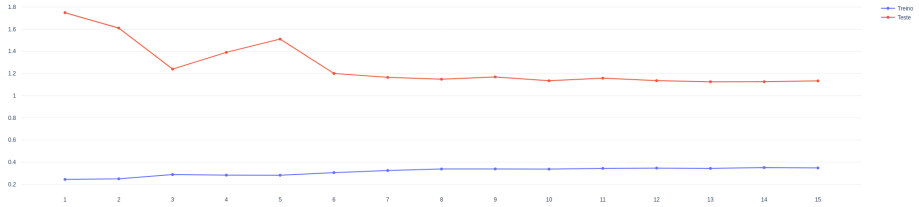

Figura 4.4: Loss por epochs no *dataset* original

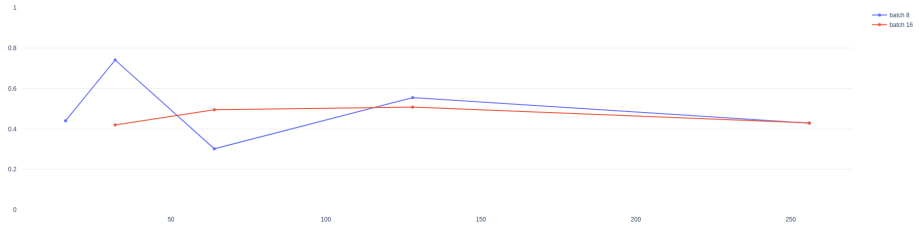

Figura 4.5: Acurácia no *dataset* original

Já na figura 4.6, observa-se o desempenho do *dataset* sem duplicidade. Nele, a maior acurácia é vista com 128 *tokens* e *batch* de tamanho 16. Entretanto, outros pontos também possuem resultados semelhantes. Assim, em uma análise inicial, este *dataset* aparentemente favorece a consistência dos resultados.

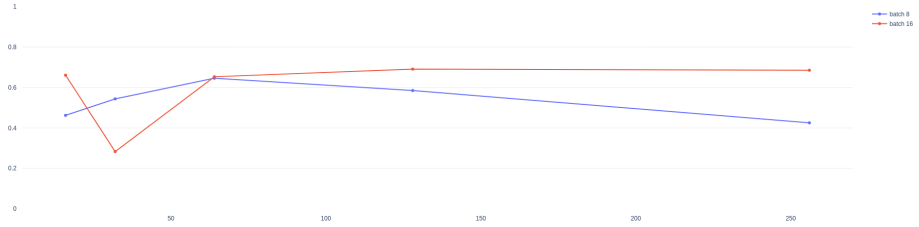

Figura 4.6: Acurácia no *dataset* com remoção de termos duplicados

Por fim, a curva ROC-AUC apresentou algumas variações dependendo do número máximo de *tokens* e do tamanho das *batches*. No entanto, a maior parte das AUCs foram próximas a 0.50, e continham curvas no formato de "S", como podeser observado em 4.7. O melhor resultado foi $AUC = 0.99$, obtido no teste com 128 *tokens* e *batches* de tamanho 4), enquanto o pior resultado foi $AUC = 0.29$, observado no teste com 256 *tokens* e *batches* de tamanho 16), que são os maiores valores utilizados nos testes.

4.4.2 Aprendizado Supervisionado Incremental

Uma outra forma de testar o desempenho do modelo é de forma incremental. Isto é, para o teste de cada mês, utilizar todos os dados anteriores a ele para realizar o *fine tuning*. Assim, simulamos a forma como o modelo seria utilizado em ambiente de produção. Por exemplo, para

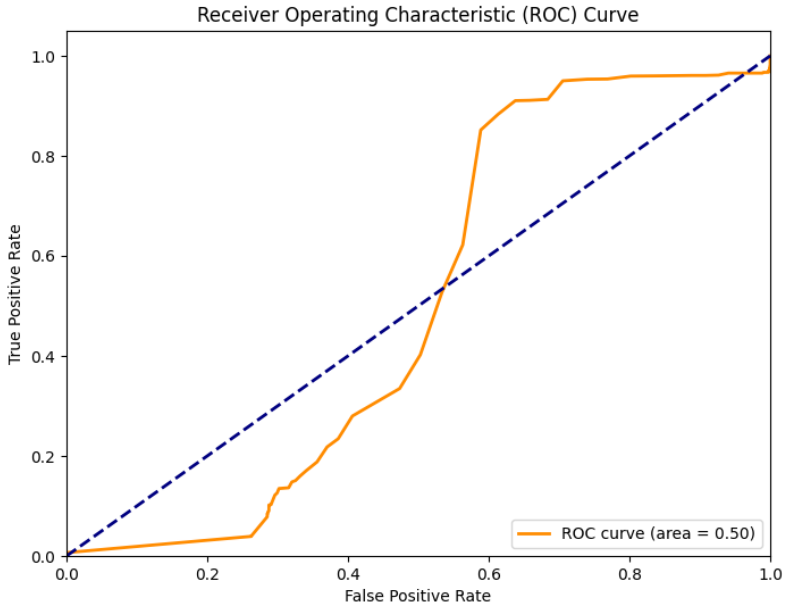

Figura 4.7: Exemplo de curva ROC-AUC (64 *tokens* e *batches* de tamanho 16)

os dados de teste do mês Julho/2013, treinamos o modelo com todos os dados vistos nos meses anteriores, ou seja, os dados de Janeiro/2013 a Junho/2013.

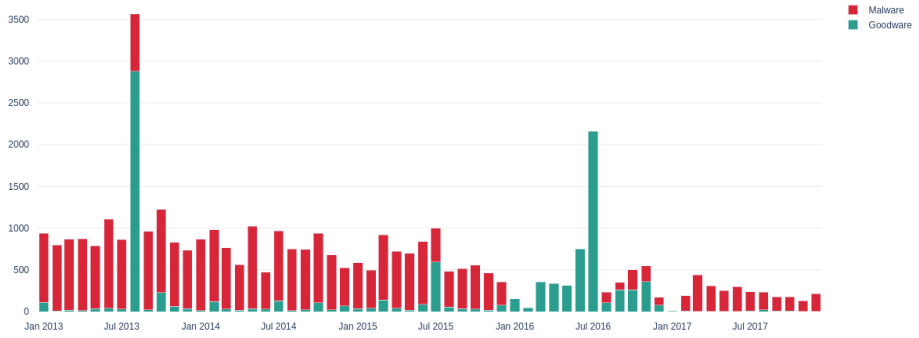

Figura 4.8: Distribuição dos dados ao longo do tempo.

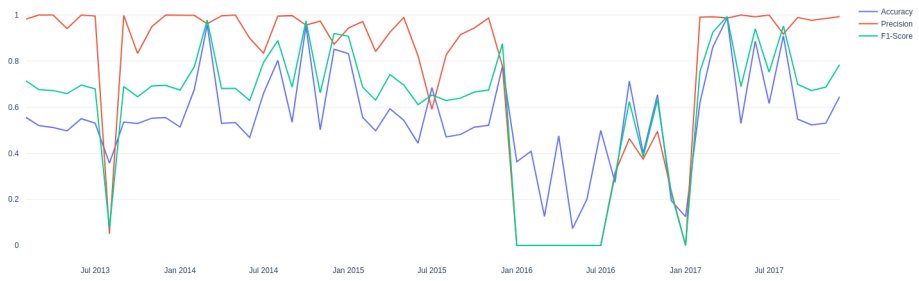

Figura 4.9: Métricas ao longo do tempo.

Há uma queda de desempenho abrupta no desempenho em Janeiro de 2016. Como discutido anteriormente, uma queda do servidor de armazenamento resultou na ausência de dados de *malware* até Julho do mesmo ano.

Nota-se que o modelo apresenta uma performance muito melhor nos meses em que existe predominância de *malwares*, como no período de Setembro de 2013 até Junho de 2015. Por outro lado, nos meses em que a quantidade de *goodwares* é maior, as métricas pioram significativamente.

4.4.3 Aprendizado Não-Supervisionado

Para treinar o modelo de aprendizado não-supervisionado, foi aplicada uma lógica similar ao do *framework LogBERT* (Guo et al., 2021). Isto é, um *masked language model* foi treinado para prever os *tokens* mascarados. Se o *token* real estiver nos K *tokens* com maior probabilidade, então a sentença é considerada normal. Caso contrário, a sentença é considerada uma anomalia.

O corpo do texto foi mascarado aleatoriamente, sendo que cada uma das sentenças teve 15% dos seus *tokens* substituídos pela máscara [MASK]. Neste caso, o *dataset* de treinamento contém apenas dados de *goodware*. A ideia é que o modelo consiga prever o valor mascarado apenas para os casos de teste de *goodware*, ao mesmo tempo em que não é capaz de completar os textos de *malware*, pois não foram vistos na etapa de treinamento.

A tabela 4.2 ilustra os resultados do modelo para a sentença 4.1. Neste caso, quatro *tokens* foram substituídos por máscaras, e, para cada um, o modelo calcula os K *tokens* mais prováveis e os retorna em ordem. Para o exemplo, foi utilizado K=12. Observa-se que o modelo conseguiu prever corretamente os *tokens*, e foi capaz, inclusive, de acertar no primeiro *token*.

Masked tokens	Predicted tokens
ole	ole mall user win versionex wi net http setup close isle
##32	##32trust version compromm80basecr86ex64
##dl	##dlddct3dudexmdddbhddmmm
##32	##32 versiontrust compro close6480mmbasefi86

Tabela 4.2: Sentença 1 - *Tokens* mascarados e previstos

Dataset	Critério	Acurácia
Original	Pelo menos um	0.936
Original	Todos	0.791
Sem duplicidade	Pelo menos um	0.939
Sem duplicidade	Todos	0.806

Tabela 4.3: Resultados para K=5

Os resultados obtidos mostram que o modelo consegue prever, com bom desempenho, todas as sentenças, e não há diferenças significativas ao remover a duplicidade. Em cerca de 80% dos casos, o modelo foi capaz de prever corretamente todos os *tokens* mascarados.

Assim, a ideia central, que se baseava na premissa de que o modelo não conseguiria prever os *tokens* mascarados de registros maliciosos, não funciona neste caso.

É possível que os dados das sentenças não tenham diferenças suficientes entre as classes. Como visto, algumas bibliotecas aparecem em mais de 60% das sentenças, o que evidencia que existem pelo menos algumas bibliotecas em comum. Outra possibilidade é o tamanho deste dataset de treinamento, que, por ser composta apenas por *goodwares*, contém apenas cerca de 16 mil sentenças.

4.4.4 Comparativo com trabalhos relacionados

A principal diferença com os trabalhos da literatura encontra-se nos dados. A maioria utiliza *datasets* bem conhecidos, como o *HDFS*. Uma das vantagens é a possibilidade de comparar os resultados de um *dataset* específico entre artigos diferentes. Todavia, o viés inerente ao *dataset* escolhido pode gerar resultados incompatíveis com a realidade.

Além disso, o contexto de segurança computacional no cyberespaço brasileiro conta com várias peculiaridades. Os ataques, por exemplo, são bastante motivados por dinheiro.

4.4.5 Limitações e trabalhos futuros

Um fator que influencia diretamente nos resultados é o número de *tokens* e o tamanho dos *batches*. Por limites de memória do ambiente, os testes utilizam, no máximo, 256 tokens e batches com 16 elementos. Assim, seria interessante utilizar uma variedade maior destes valores.

Como visto, um dos testes realizados envolve a remoção de todos os valores duplicados, garantindo que cada sentença só possui elementos únicos. Porém, neste processo, algumas informações são perdidas, como a posição de cada termo, a ordem em que aparecem e o número de vezes. Então, seria importante analisar formas mais sutis para reduzir a repetição nos textos. Por exemplo, utilizar o cálculo do TF-IDF como *threshold* para remover valores atípicos (Ceschin et al., 2018).

Além disso, o escopo deste trabalho ficou limitado aos dados de DLLs e símbolos importados. Então, a realização experimentos que incluam os outros dados deste *dataset* e o uso de outros *datasets* pertinentes seria um avanço interessante e tornaria os resultados mais robustos.

Por fim, o uso de outros modelos pré-treinados BERT também pode ser explorado, a fim de avaliar sua eficácia e compará-los entre si.

5 CONCLUSÃO

Em conclusão, a aplicação de um modelo pré-treinado BERT para a classificação de código malicioso não se mostrou eficaz. Em boa parte dos testes, foi possível observar *overfitting*, e foi necessário alterar o *threshold* dos *scores* para diminuir a sensibilidade relacionada aos *malwares*.

Por outro lado, ao utilizar o mesmo modelo como um *masked language model* treinado apenas com *goodwares*, nota-se a capacidade avançada para o reconhecimento de padrões e relações entre os *tokens*, independentemente da classe. Por este motivo, não é possível distinguir entre elas utilizando a detecção de anomalias como base.

REFERÊNCIAS

Aggarwal, C. C. (2016). *Outlier Analysis*. Springer.

Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L. e Rieck, K. (2020). Dos and don'ts of machine learning in computer security. *CoRR*, abs/2010.09470.

Ceschin, F., Pinage, F., Castilho, M., Menotti, D., Oliveira, L. S. e Gregio, A. (2018). The need for speed: An analysis of brazilian malware classifiers. *IEEE Security Privacy*, 16(6):31–41.

Chandola, V., Banerjee, A. e Kumar, V. (2009). Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3).

Devlin, J., Chang, M.-W., Lee, K. e Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding.

Du, M., Li, F., Zheng, G. e Srikumar, V. (2017). Deeplog: Anomaly detection and diagnosis from system logs through deep learning. Em *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, página 1285–1298, New York, NY, USA. Association for Computing Machinery.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D. e Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages.

Guo, H., Yuan, S. e Wu, X. (2021). Logbert: Log anomaly detection via bert. Em *2021 International Joint Conference on Neural Networks (IJCNN)*, páginas 1–8.

Hawkins, D. M. (1980). *Identification of Outliers*. Chapman and Hall.

Le, V.-H. e Zhang, H. (2022). Log-based anomaly detection with deep learning: How far are we? Em *Proceedings of the 44th International Conference on Software Engineering*. ACM.

Shalev-Shwartz, S. e Ben-David, S. (2014). *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.

van Ede, T., Aghakhani, H., Spahn, N., Bortolameotti, R., Cova, M., Continella, A., van Steen, M., Peter, A., Kruegel, C. e Vigna, G. (2022). DeepCASE: Semi-Supervised Contextual Analysis of Security Events. Em *Proceedings of the IEEE Symposium on Security and Privacy (SP)*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. e Polosukhin, I. (2017). Attention is all you need. *CoRR*, abs/1706.03762.

Zhang, F., Yu, X., Keung, J., Li, F., Xie, Z., Yang, Z., Ma, C. e Zhang, Z. (2022). Improving stack overflow question title generation with copying enhanced codebert model and bi-modal information.