Mineração de Dados Utilizando o EMBER

Arthur Labaki[1], **Pedro Leale**[2]

[1]Faculdade de Computação – Universidade Federal de Uberlândia (UFU)

{arthur.labaki, pedro.leale}@ufu.br

Resumo. *Este trabalho descreve a implementação de uma árvore de decisão para classificação de malwares, utilizando como base de dados o EMBER. O devido projeto foi dividido em três etapas. A primeira etapa de pré-processamento dos dados envolveu a remoção de campos menos relevantes e dos malwares classificados como desconhecidos. A segunda etapa se resume em criar e executar o algoritmo de árvore simples da biblioteca Scikit-learn para classificar os elementos como malignos ou benignos. Por fim, na etapa de pós-processamento foi calculado a acurácia do modelo, que varia entre 78% e 87%, o que chega a ser relevante tendo como ponto de vista utilizar uma tecnica simples de mineração de dados.*

1. Introdução

Nesse trabalho utilizamos um *dataset* de *malwares*, para realizar a classificação de amostras entre malignas (que contem malwares) ou benignas. Entendendo melhor, o *EMBER Dataset* [Anderson and Roth 2018] é um conjunto de dados amplamente utilizado no campo da segurança cibernética para pesquisa e desenvolvimento de técnicas de detecção de malware. Ele foi desenvolvido pela equipe da *Endgame*, uma empresa de segurança cibernética, e é composto por uma grande quantidade de arquivos executáveis em formato PE (*Portable Executable*), incluindo tanto amostras de malware quanto arquivos benignos.

Para essa pesquisa, é divulgado um conjunto de dados com cerca de 1,1 milhões de amostras de malwares, porém esse *dataset* contém apenas as características extraídas desses arquivos executáveis, como informações estáticas e dinâmicas, que podem ser usadas para treinar algoritmos de aprendizado de máquina e desenvolver modelos de detecção de malware. Essas características incluem:

- *Unique ID*: As funções de *hash SHA256* e *MD5*;
- *Date*: Tempo estimado em mês/ano de quando a amostra foi criada e espalhada;
- *Label*: Um rotulo da amostra, sendo 0 benigno, 1 maligno e -1 não rotulado;
- *AVClass*: *Tags* indicando o tipo de malware, como classe e família.

Nesse trabalho será apresentado uma implementação de uma árvore de decisão para classificação de arquivos benignos ou malignos. Para realizar essa tarefa da melhor maneira possível, esse projeto foi dividido em três etapas diferentes:

1. Pré-processamento: Prepara os dados para a analise. Nela realizamos a amostragem dividindo nossa base em pedaços, a filtragem para remover amostras não rotuladas, a extração dos rótulos para criar um conjunto de dados somente de *labels*, e por fim a seleção de um subconjunto de *features*, removendo características irrelevantes ou inapropriadas para nossa analise;
2. Mineração: Utilizamos a arvore de decisão simples para criar um classificador simples e facilmente interpretável;
3. Pós-processamento: Calculamos a acurácia dos resultados obtidos e montamos a imagem da árvore gerada, para melhor visualização de sua dimensionalidade.

Vale ressaltar que todos os códigos desenvolvidos, links para o *dataset*, imagens e slides estão disponíveis no repositório [Labaki 2023].

2. Desenvolvimento

Esse projeto foi desenvolvido e implementado utilizando a linguagem de programação *Python*, utilizando a biblioteca *Sciki-learn* [Pedregosa et al. 2011], que é uma poderosa ferramenta para realizar tarefas de aprendizado de máquina, fornecendo algoritmos e funcionalidades para treinamento, avaliação e previsão de modelos em uma ampla variedade de problemas de análise de dados. No projeto, ela foi utilizada para criar, treinar e visualizar a árvore de decisão.

2.1. Estrutura dos dados

O conjunto de dados *EMBER* consiste em uma coleção de arquivos de linhas *JSON*, onde cada linha contém um único objeto *JSON*. Um objeto *JSON*, ou *JavaScript Object Notation*, é uma estrutura de dados usada para representar informações de forma organizada e legível por máquinas. Ele é uma coleção de pares chave-valor, onde as chaves são strings e os valores podem ser de diferentes tipos. Os pares chave-valor são separados por vírgulas e o objeto é delimitado por chaves.

Os dados fora extraídos de amostras de malwares e softwares reais, utilizando a ferramenta Lief [Thomas and other contributors 2023], que é uma biblioteca de código aberto que permite analisar, modificar e manipular arquivos executáveis em diferentes formatos, como *PE* (*Portable Executable*), *ELF* (*Executable and Linkable Format*) e *Mach-O*.

Como o projeto *EMBER* tem o intuito de criar um *dataset* para pesquisadores, então o conjunto de dados contem a maior quantidade de dados brutos das amostras possíveis, para que cada pesquisador que for utilizar o *dataset*, manipule esses dados de acordo com o for necessário para eles, criando um grande conjunto de dados adequado para praticamente todos os pesquisadores. A imagem 1 contem um exemplo dos dados de uma única amostra.

```
"sha256": "000185977be72c8b007ac347b73ceb1ba3e5e4dae4fe98d4f2ea92250f7f580e",
"appeared": "2017-01",
"label": -1,
"general": {
  "file_size": 33334,
  "vsize": 45056,
  "has_debug": 0,
  "exports": 0,
  "imports": 41,
  "has_relocations": 1,
  "has_resources": 0,
  "has_signature": 0,
  "has_tls": 0,
  "symbols": 0
},
"header": {
  "coff": {
    "timestamp": 1365446976,
    "machine": "I386",
    "characteristics": [ "LARGE_ADDRESS_AWARE", ..., "EXECUTABLE_IMAGE" ]
  },
  "optional": {
    "subsystem": "WINDOWS_CUI",
    "dll_characteristics": [ "DYNAMIC_BASE", ..., "TERMINAL_SERVER_AWARE" ],
    "magic": "PE32",
    "major_image_version": 1,
    "minor_image_version": 2,
    "major_linker_version": 11,
    "minor_linker_version": 0,
    "major_operating_system_version": 6,
    "minor_operating_system_version": 0,
    "major_subsystem_version": 6,
    "minor_subsystem_version": 0,
    "sizeof_code": 3584,
    "sizeof_headers": 1024,
    "sizeof_heap_commit": 4096
  }
},
"imports": {
  "KERNEL32.dll": [ "GetTickCount" ],
  ...
},
"exports": []
"section": {
  "entry": ".text",
  "sections": [
    {
      "name": ".text",
      "size": 3584,
      "entropy": 6.368472139761825,
      "vsize": 3270,
      "props": [ "CNT_CODE", "MEM_EXECUTE", "MEM_READ"]
    },
    ...
  ]
},
"histogram": [ 3818, 155, ..., 377 ],
"byteentropy": [0, 0, ... 2943 ],
"strings": {
  "numstrings": 170,
  "avlength": 8.170588235294117,
  "printabledist": [ 15, ... 6 ],
  "printables": 1389,
  "entropy": 6.259255409240723,
  "paths": 0,
  "urls": 0,
  "registry": 0,
  "MZ": 1
},
}
```


Figure 1. Estrutura dos dados do EMBER

2.2. Pré-processamento dos dados

A etapa de pré-processamento de dados consiste em um conjunto de técnicas e procedimentos aplicados aos dados antes de sua utilização em um modelo de aprendizado de máquina ou análise de dados. Essa etapa tem por objetivo a adequada preparação dos dados, assegurando sua qualidade e potencializando sua utilidade e eficiência para as tarefas subsequentes.

Em nosso projeto foi primeiramente realizado uma amostragem dos dados. Amostragem é a criação de amostras representativas dos dados originais. Como o *dataset* contém 1,1 milhão de amostras, é muito custoso e inviável utilizá-lo completo. Então esse conjunto de dados foi separado em 5 *datasets* de treino, contendo 180 mil cada e um único de teste com 200 mil. Cada conjunto foi treinado separadamente.

Após isso, foi feito uma filtragem nos dados. A filtragem de dados refere-se ao processo de selecionar e restringir os dados com base em critérios específicos ou condições estabelecidas. No *dataset* original existem amostras rotuladas como -1, podendo ser tanto benignas quanto malignas. Em nossa análise, esse tipo de amostra pode comprometer a sua qualidade, então foi necessário remover esses objetos.

Também foi realizado a extração dos rótulos do *dataset* original para um novo. Para realizar a mineração corretamente, é necessário extrair o *label* (0 ou 1) da base de dados, criando um novo conjunto (normalmente chamado de *y-train* para treino e *y-test* para o teste). Essa etapa foi realizada em todos os conjuntos de dados, tanto nos de treino quanto no de teste.

Por fim, realizamos uma seleção de um subconjunto de *features* para nosso *dataset*. A seleção de um subconjunto de *features* refere-se ao processo de escolher um conjunto reduzido de variáveis ou atributos das quais se pretende extrair informações relevantes para uma determinada tarefa de análise de dados ou modelagem. No *dataset* original, existem algumas *features* que não são importantes para nossa mineração escolhida, como o *sha256*, *md5*, *appeared*, *label*, *avclass*.

Além dessas, existem características que nossa análise não é capaz de suportar, como strings. Então foram removidos também as *features header*, *section*, *imports*, *exports*, *datadirectories*. Essas remoções provavelmente vai resultar em uma análise menos precisa, mas é necessária para poder realiza-la de forma mais simplificada, o que é proposto no projeto.

Com isso, a estrutura do *dataset* ficou da forma:

- Histogram: 256 classes;
- Byteentropy: 256 classes;
- Strings: 104 classes;
- General: 10 classes.

2.3. Mineração dos dados

Como o trabalho se resume em um problema de classificação, foi utilizado a Árvore de Decisão. Árvore de Decisão é um bom um algoritmo de aprendizado de máquina supervisionado, além de ser muito utilizado devido à sua simplicidade e interpretabilidade. Ela é uma representação visual de um conjunto de regras de decisão hierárquicas, onde cada nó interno representa um atributo ou característica, cada ramo representa uma condição possível e cada folha representa uma decisão ou resultado.

Uma árvore de decisão é construída através da divisão recursiva dos dados com base nos atributos relevantes. Começando com um nó raiz contendo todos os dados, o algoritmo seleciona o atributo mais informativo para dividir os dados em subconjuntos puros. A divisão continua até que os subconjuntos sejam homogêneos em relação à classe ou atinjam um critério de parada. As folhas representam as classes previstas (em classificação) ou contêm valores numéricos de predições (em regressão).

Para a criação dessa árvore, foi utilizado a biblioteca *Scikit-learn* no *Python*. Ela fornece uma ampla variedade de ferramentas e algoritmos para todas as etapas de uma mineração de dados. Ela possui uma API consistente que facilita a criação, treinamento e aplicação de modelos de aprendizado de máquina. Ela inclui uma variedade de algoritmos populares, como árvores de decisão, regressão linear, regressão logística, máquinas de vetor de suporte (SVM), k-vizinhos mais próximos (KNN), entre outros.

Em nosso projeto, utilizamos a *DecisionTreeClassifier*, que é uma implementação da árvore de decisão para problemas de classificação. Ela faz parte do módulo *tree* da biblioteca e é utilizada para construir modelos de classificação baseados em árvores de decisão. Ao criar um objeto *DecisionTreeClassifier*, é possível especificar os hiper parâmetros que controlam a construção e o comportamento da árvore de decisão, como o critério de divisão (como o ganho de informação ou índice de *Gini*), a estratégia de divisão, a profundidade máxima da árvore, entre outros. No nosso caso, utilizamos a árvore mais pura possível, simplificando o uso ao máximo.

Uma vez criado o objeto *DecisionTreeClassifier*, ele pode ser treinado com um conjunto de dados de treinamento utilizando o método *fit(X, y)*, onde X é a matriz de características e y é o vetor de rótulos ou classes correspondentes. Durante o treinamento, a árvore de decisão é construída de acordo com as regras especificadas pelos hiper parâmetros, dividindo os dados com base nas características para obter as melhores decisões de classificação.

Após o treinamento, o modelo de árvore de decisão pode ser usado para fazer previsões em novos dados usando o método *predict(X)*, onde X é um conjunto de dados de teste ou de entrada. O método retorna as previsões de classe correspondentes para os exemplos fornecidos. Abaixo está um pedaço do código para demonstrar as funções utilizadas.

```
# Organiza os dados como um array estilo numpy
X_train = np.array(X_train)
y_train = np.array(y_train)
X_test = np.array(X_test)
y_test = np.array(y_test)

# Cria a arvore de decizão vazia
clf = DecisionTreeClassifier()
num_features = len(X_train[0])

# Treina o classificador usando o conjunto de treinamento
clf.fit(X_train, y_train)

# Faz as previsões usando o conjunto de teste
y_pred = clf.predict(X_test)
```

Figure 2. Código para a árvore de decisão

2.4. Pós-processamento dos dados

Após a análise dos dados, é necessário obter os resultados para verificar se o que foi produzido foi relevante. O objetivo do pós-processamento de dados é melhorar a utilidade ou compreensão dos resultados, tornando-os mais adequados para a análise ou tomada de decisões. Ele pode incluir diversas atividades, dependendo do contexto e dos requisitos específicos do problema.

Dentre as mais diversas técnicas possíveis, em nosso projeto foi realizado as mesmas técnica utilizadas pela analise do *EMBER*, a fim de comparação dos resultados. Elas são o calculo da acurácia dos dados e a visualização da árvore criada.

A acurácia é uma métrica utilizada para avaliar o desempenho de modelos de classificação. Ela mede a proporção de exemplos classificados corretamente em relação ao total de exemplos. Matematicamente, a acurácia é calculada dividindo o número de amostras corretamente classificados pelo número total de amostras observadas:

Acurácia = (Verdadeiros Positivos + Verdadeiros Negativos) / (Total de Observações)

Das árvores treinadas e testadas, obtivemos os resultados:

Dataset	1	2	3	4
Acurácia	0.7876	0.8709	0.7993	0.8173

Além dessa métrica, também foram obtidos os *plots* das árvores de decisão geradas com o intuito de visualizar e entender a estrutura da árvore construída pelo algoritmo de aprendizado de máquina. Essa representação gráfica permite a visualização de forma intuitiva as decisões tomadas em cada nó da árvore e como os atributos são utilizados para classificar ou prever os exemplos.

No nosso projeto, como existem muitas classes, é inviável analisar qual classe foi escolhida para cada nó, então seu *plot* funciona apenas para obter uma noção de sua dimensionalidade. Com isso, os *plots* das árvores obtidas foram:

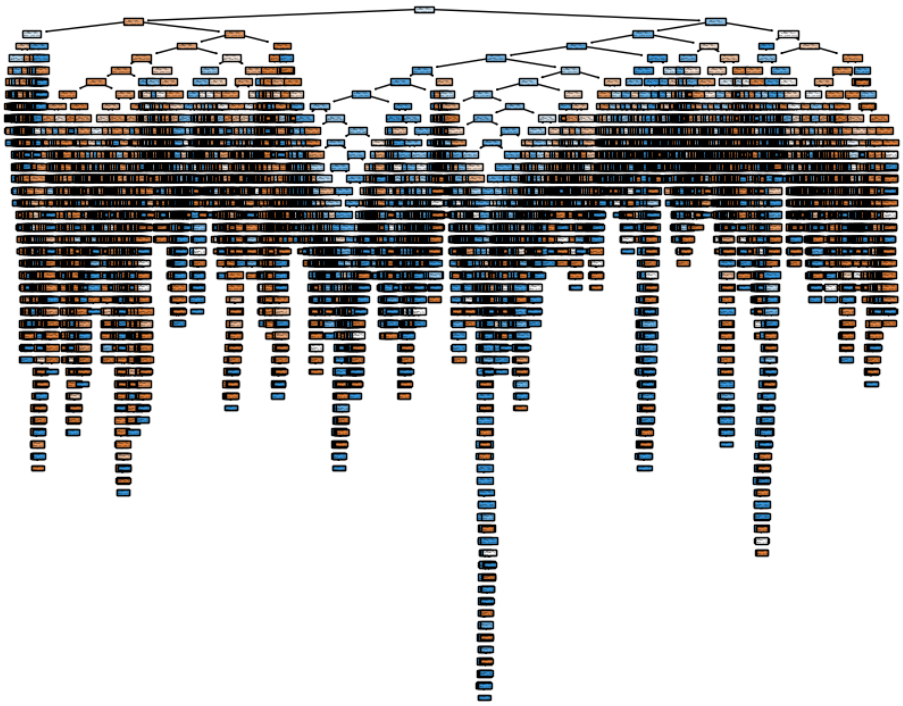

Figure 3. Árvore de decisão do treino 1 - Acurácia: 0.7876

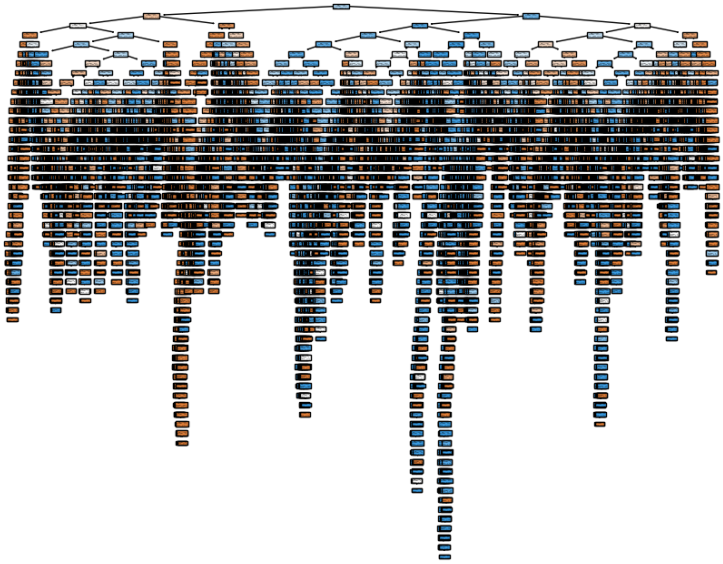

Figure 4. Árvore de decisão do treino 2 - Acurácia: 0.8709

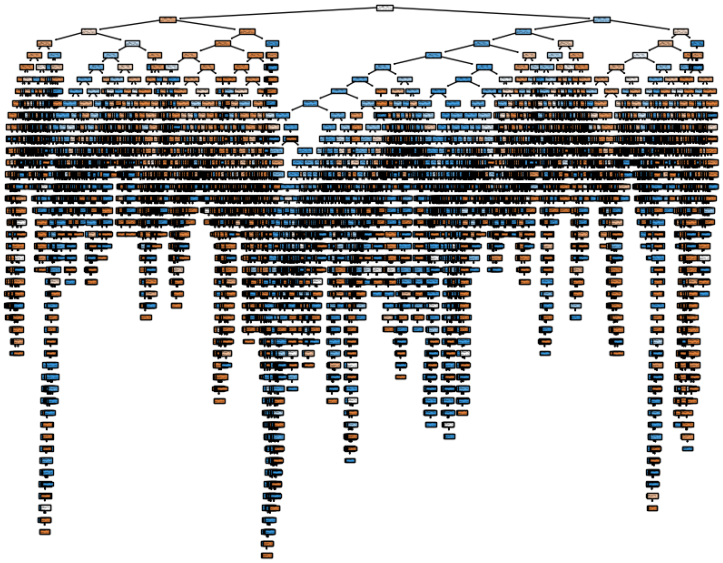

Figure 5. Árvore de decisão do treino 3 - Acurácia: 0.7993

Observando os dados obtidos, é possível perceber que mesmo retirando boa quantidade de *features* relevantes, o resultado ainda é satisfatório. Certamente que comparando com outros trabalhos grandes ou com a própria análise do *EMBER*, nosso

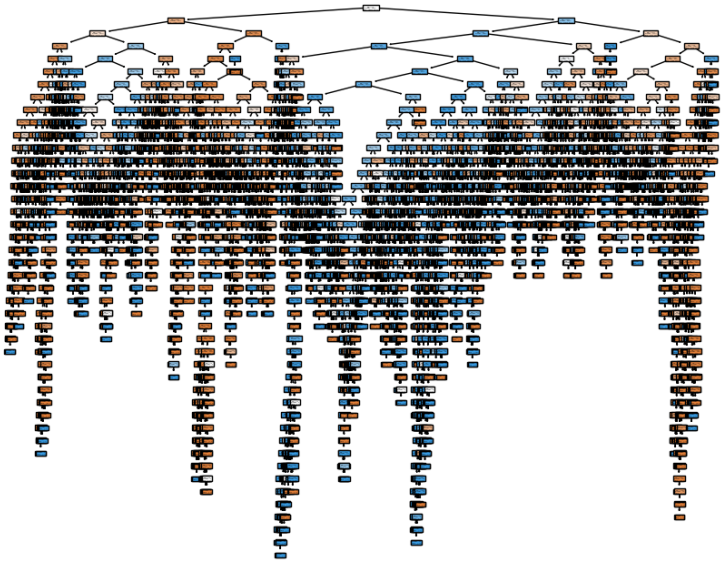

Figure 6. Árvore de decisão do treino 4 - Acurácia: 0.8173

resultado foi muito inferior (0.99911 de acurácia do *EMBER*), mas utilizamos apenas uma árvore de decisão simples.

É importante ressaltar que, mesmo com o intuito de fazer algo semelhante ao que foi realizado no projeto *EMBER*, teria sido útil e interessante realizar outras avaliações de desempenho, como precisão, *recall*, *F1-score* ou curva ROC, fornecendo uma visão mais completa do desempenho da árvore de decisão em diferentes classes ou contextos.

3. Conclusão

Neste trabalho foi apresentado a implementação de uma árvore de decisão simples para classificação de malwares, seguindo com base o projeto *EMBER*. Como resultados, foram obtidos uma acurácia que varia de *78%* a *87%*. Esta abordagem gerou um resultado com baixa acurácia, visto que neste contexto é necessário que seja o mais próximo de t100% possível, porém é possível realizar mudanças nesta árvore para melhorar seus resultados.

Um dos maiores problemas para a execução do trabalho foi o pouco espaço de tempo disponível, visto que foi dado poucas semanas para concluir o projeto e que cada execução de treino da árvore com uma base de dados demorou cerca de 7 horas para concluir. Um segundo problema foi que, o *dataset EMBER* possui um tamanho de aproximadamente 10 gigabytes, sendo muito espaçoso, o que dificultou muito seu manuseio.

Também é importante informar que o objetivo primário do trabalho era criar uma árvore única, sem dividir os dados, para obter a melhor acurácia possível. Porem essa execução excedeu 28 horas e foi cancelada devido à necessidade de otimizar o tempo de processamento. Diante dessa situação, decidimos revisar a abordagem e buscar alternativas para alcançar o objetivo estabelecido de maneira mais eficiente.

Para trabalhos futuros, uma possível continuação para esse trabalho seria desenvolver algum método para utilizar as strings que foram removidas, como utilizar *Bag-of-Words* ou *TF-IDF* (*Term Frequency-Inverse Document Frequency*). Também ainda seria possível utilizar os dados não rotulado (-1) para avaliar e testar a robustez dos modelos de detecção de malware, como outros projetos utilizaram.

Uma outra ideia para uma continuação do projeto seria alterar o tipo de árvore usada, utilizando como, por exemplo, *Random Forest* ou *Gradient Boosting*. Também seria possível trocar o estilo do classificador, utilizando por exemplo o *Naive Bayes*. Porem será importante manter a ideia de simplicidade do projeto, buscando obter o máximo possível com o mais simples.

References

Anderson, H. S. and Roth, P. (2018). EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints*.

Labaki, A. (2023). Repositório do github contendo os códigos do projeto: https://github.com/arthurlabaki/materias_ufu/tree/main/md.

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

Thomas, R. and other contributors (acessado em 2023). Lief: Library to instrument executable formats.