Malware Prediction for Windows System

Group 6: Xiaoliang Xu and Changxu Ren
CS 6140 Fall 2023
Instructor: Dr. Anurag Bhardwaj

Abstract

This project investigates the challenge of malware detection in cybersecurity, emphasizing the limitations of traditional signature-based methods and the potential of machine learning approaches. We conducted a study using a large dataset from the Microsoft Malware Prediction Kaggle competition, involving exploratory data analysis, feature engineering, and the application of various machine learning models including decision trees, LightGBM, and neural networks.

Our key findings include the identification of significant predictors for malware detection such as AvSigVersion, AVProductStatesIdentifier, and CountryIdentifier. We observed the limitations of certain engineered features, which did not significantly contribute to the predictive power of the models. The decision tree model faced challenges of overfitting, which were mitigated by constraining the tree depth. The LightGBM model showed a slight improvement in accuracy, benefiting from its gradient-boosting capabilities. However, the neural network model, despite a sophisticated architecture, achieved modest results, indicating the complexity of the malware detection task and the need for further refinement in model design and feature selection.

In conclusion, the study underscores the complexity of malware detection and the necessity of advanced machine learning techniques to tackle this evolving cybersecurity threat. Future work should focus on exploring different encoding techniques for categorical features and expanding the use of gradient boosting models like CatBoost, which could provide further insights into the nuanced nature of malware detection.

Introduction

Problem Statement

Malware, a harmful software targeting computer systems, presents a major challenge in cybersecurity. The evolving nature of malware renders traditional signature-based detection methods inadequate, highlighting the need for more flexible, machine learning-driven approaches.

Background

Malware's impact is significant and far-reaching, affecting security, economics, technology, and international relations. It endangers personal and corporate data, causing privacy violations and financial harm. Malware hinders technological progress by attacking intellectual property and new technologies. Its continuous evolution complicates prevention efforts, leading to reputational and regulatory challenges for organizations. Additionally, its role in cyber warfare and international espionage underscores its status as a complex, enduring threat to the global digital ecosystem, demanding a collaborative and multifaceted response.

Exploratory Data Analysis

The exploratory data analysis (EDA) for the malware prediction study was a meticulous process aimed at streamlining a massive dataset for more effective analysis.

Dataset Overview

The training and testing datasets were sourced from the Microsoft Malware Prediction Kaggle competition. The training set comprises 83 columns, including labels and machine identifiers, encompassing over 8.9 million rows of observations. The team addressed the challenges posed by the dataset's size by implementing a data truncation strategy. This involved loading the data in chunks from the disk, then randomly selecting a quarter of each chunk's data to append to the final training dataset. This approach enabled us to reduce the dataset to a more manageable size of approximately 2.2 million records.

```python
# ---------------------- WE DO NOT HAVE TO RUN THIS BLOCK EVERYTIME ----------------------------
# truncate the whole data set to 1/4 of its original size randomly

def process_and_save_by_chunk(file_path, output_file, chunk_size=10000, fraction=0.25):
    """
    Reads a large file in chunks, randomly selects a fraction of rows from each chunk,
    and saves all selected rows to a separate file.

    :param file_path: Path to the input file.
    :param output_file: Path to the output file.
    :param chunk_size: Number of rows per chunk.
    :param fraction: Fraction of rows to select from each chunk.
    """
    # Initialize an empty DataFrame for storing selected rows
    selected_rows = pd.DataFrame()

    # Read the file in chunks
    for chunk in pd.read_csv(file_path, chunksize=chunk_size, low_memory=False):
        # Randomly sample a fraction of rows from each chunk
        sampled_chunk = chunk.sample(frac=fraction)
        selected_rows = pd.concat([selected_rows, sampled_chunk])

    # Save the selected rows to a new file
    selected_rows.to_csv(output_file, index=False)


process_and_save_by_chunk(training_set_file_path, training_set_truncated_path)
```

Figure 1. Implementation of data truncation strategy.

Data Cleaning, Data Type Classification, and Canonicalization

The EDA then progressed to the crucial task of data cleansing, a step essential for improving data quality. This process involved identifying and removing features with a high percentage of missing values. In Figure 2, we inspected the correlation between the features and the number of missing values. It was observed that 9 features had over 10% of their values missing, and 3 features had more than 90% of their values missing. Given that features with a majority of missing values contribute minimally to the final prediction, we followed Prof. Anurag's guidelines and decided to remove 7 features with over 40% missing values.

	Missing Values	% of Total Values
PuaMode	2229795	99.974175
Census_ProcessorClass	2221157	99.586885
DefaultBrowsersIdentifier	2121553	95.121081
Census_IsFlightingInternal	1852293	83.048650
Census_InternalBatteryType	1585032	71.065845
Census_ThresholdOptIn	1416990	63.531583
Census_IsWIMBootEnabled	1415108	63.447202
SmartScreen	793784	35.589774
OrganizationIdentifier	687726	30.834601
SMode	134839	6.045586
CityIdentifier	81613	3.659167
Wdft_IsGamer	75574	3.388405
Wdft_RegionIdentifier	75574	3.388405
Census_InternalBatteryNumberOfCharges	66894	2.999232
Census_FirmwareManufacturerIdentifier	45720	2.049883
Census_FirmwareVersionIdentifier	39905	1.789164
Census_IsFlightsDisabled	39674	1.778807
Census_OEMModelIdentifier	25436	1.140438
Census_OEMNameIdentifier	23791	1.066684
Firewall	22655	1.015750

Figure 2. Ranking of features based on missing values.

Post-cleansing, the features were categorized as numerical, categorical, or binary to establish a clear structure for analysis. Particularly for categorical data, a canonicalization process was carried out. This process was essential in distilling numerous variations of categorical entries. For instance, in the 'Census_OSEdition' feature, we encountered values like 'ServerStandard', 'ServerStandardEval', and 'ServerSolution'. These values potentially indicate similar meanings. Therefore, we consolidated all values with similar meanings into the 'Server' category. Such standardization is pivotal for reducing complexity and enhancing the performance of subsequent machine learning models.

Data Distribution

As we learned in the class, inspecting the data distribution is a fundamental step in ensuring the robustness and accuracy of the model's predictions. Diverse data distribution can greatly influence how the model learns and generalizes. For instance, skewed or unbalanced distributions might lead the model to develop biases, overly favoring certain outcomes or characteristics. This is particularly critical in classification tasks where the prevalence of one class over another can result in poor performance on the underrepresented class. Similarly, understanding the distribution helps in identifying outliers and anomalies, which, if not addressed, can skew the model's learning process, leading to overfitting or underfitting. To address all mentioned concerns, we inspected the distribution for label, binary and numerical features.

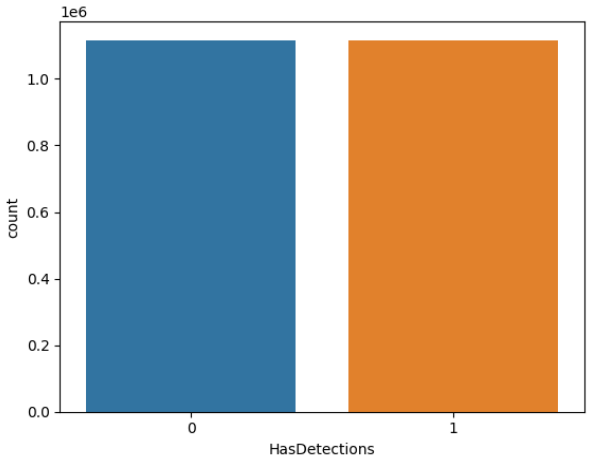

Figure 3. Distribution of data for different labels.

As shown in Figure 3, we observed that the whole training set contains almost equal amounts of positive and negative labeled data. This indicates we have a balanced dataset, which ensures that the model is exposed to an equitable representation of each class during training. Next, we analyzed the distribution of binary features, as illustrated in Figure 4. We observed a significant imbalance in most features, with some displaying a predominant preference for a single value. While at a glance, an imbalanced distribution in feature values might seem less than ideal, in the context of binary features, this can be advantageous. Such disparities can be indicative of strong predictors within the data. For example, a feature where one value is vastly more common than the other may serve as a key differentiator when classifying observations between classes. This characteristic becomes particularly valuable in scenarios where one class is inherently more common or when a specific outcome is more critical to

identify accurately. However, it is important to approach these imbalances with caution, as they can also lead to overfitting if the model overly relies on these features for prediction. Hence, the significant variance we see across the binary features in our dataset necessitates a nuanced approach to model training, ensuring that our model can leverage these disparities to enhance predictive performance without compromising on its ability to generalize to new data.

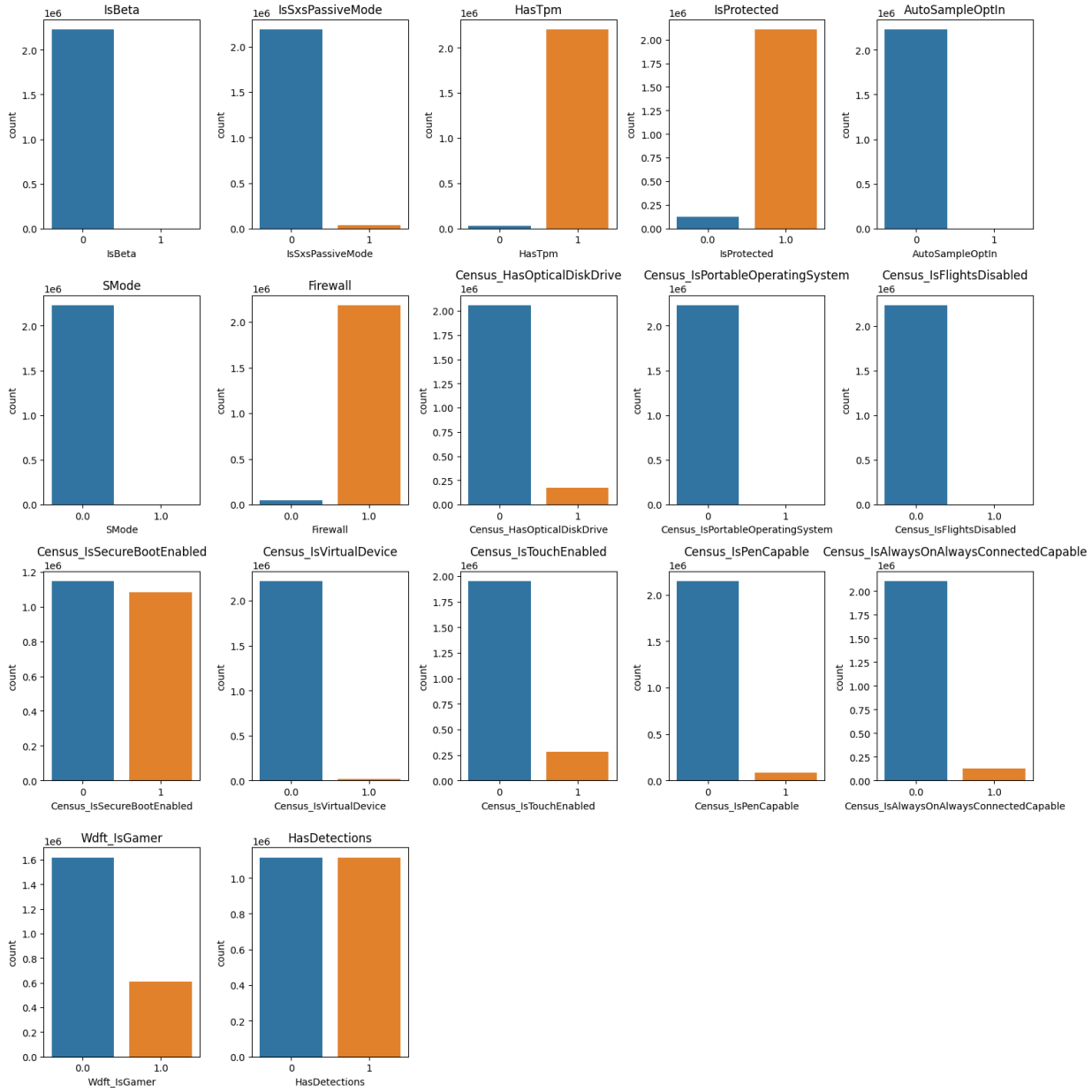

Figure 4. Distribution of binary features.

Lastly, we examined the distribution of numerical features. Figure 5 presents the distribution of these features along with their kernel density estimation plots. We observed that the distribution of some numerical features is strongly positively skewed. Such skewness can lead to several concerns when modeling data. A positively skewed distribution suggests that there are outliers or a long tail to the right, which can distort the mean and pull it toward the tail, leading to a misrepresentation of the typical value in the data. This can adversely affect

the performance of many machine learning algorithms, which assume data is normally distributed. It may also impact the model's ability to generalize by giving undue influence to these outlier values. After researching common data transformation practices, we found that taking the logarithm of right-skewed data is a typical approach to handling this issue, as it can help in making the data conform more closely to a normal distribution. Hence, we applied a logarithmic transformation to the non-normally distributed features. Figure 6 compares the data distribution before and after the transformation, clearly illustrating that the log transformation indeed assists in correcting the skewness of the data distribution, resulting in a more symmetrical dataset that is likely to improve model performance.

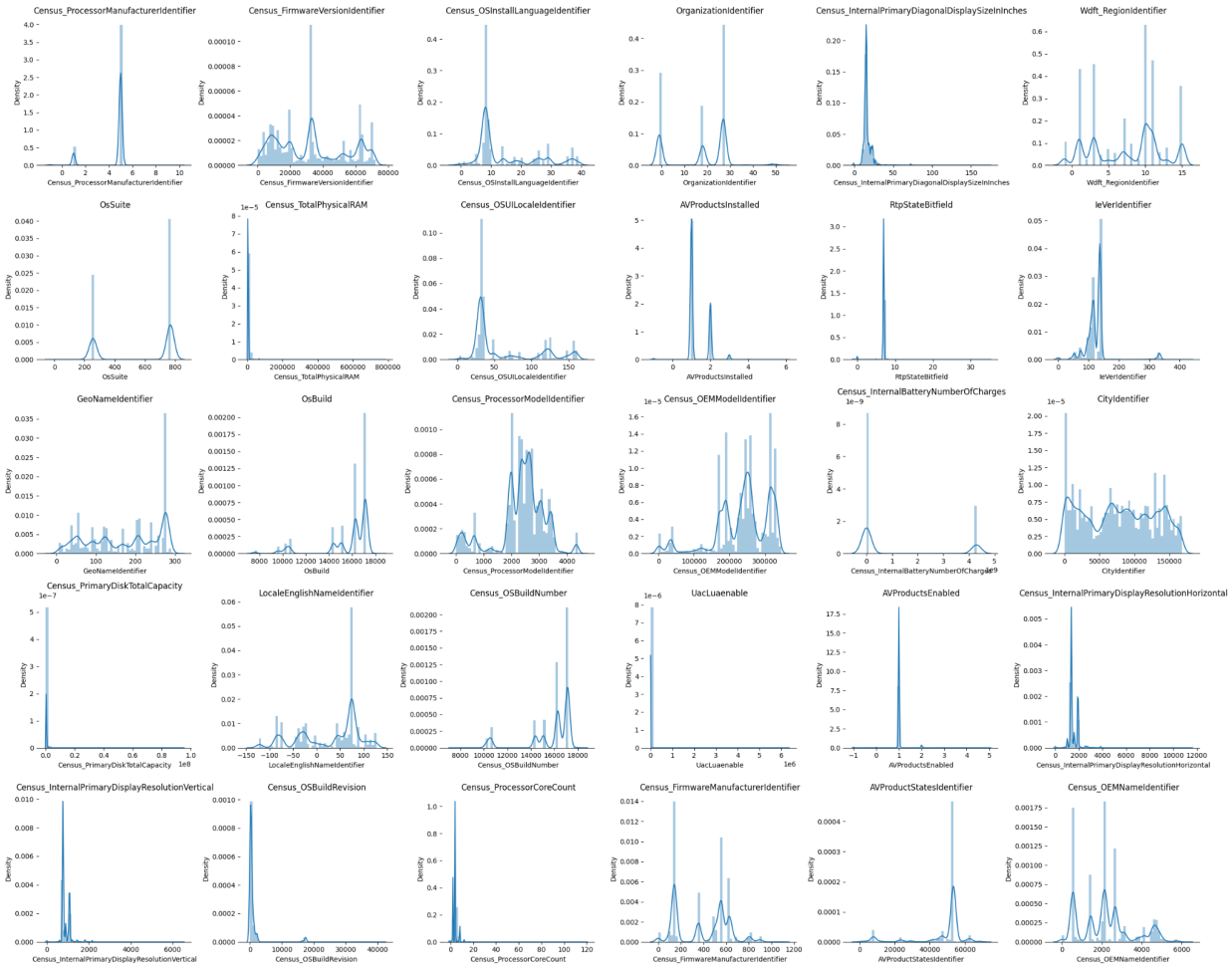

Figure 5. Distribution of numerical features.

Feature Engineering

From what we learned throughout the course, feature engineering is a vital step in the machine learning pipeline because it involves the creation of new features from existing data to enhance the predictive power of a model. By integrating domain knowledge, common sense, and personal experience, we can form hypotheses about the relationships within the data that may not be immediately apparent. This process enables us to transform raw data into informative features that better represent the underlying problem, leading to improved model accuracy and generalizability.

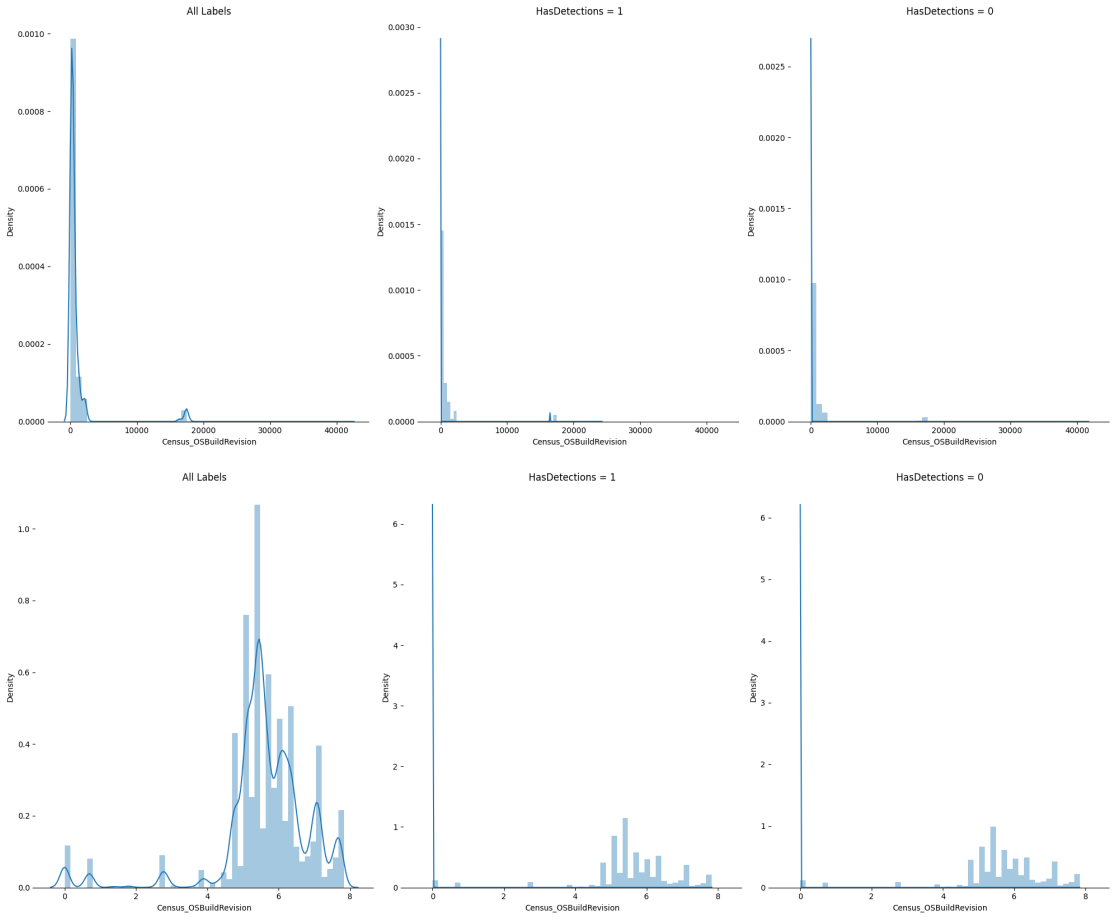

Figure 6. Comparison between before (above) and after (below) log transformation.

In our approach to feature engineering, we drew upon our understanding of the dataset and the context in which the data exists to create new variables. For example, drawing on cybersecurity insights reported by sources like PCMag.com, we crafted the Gamer_Firewall feature based on the notion that gaming devices without adequate firewall protection may be more susceptible to malware attacks. This feature was designed by combining gaming device indicators with firewall status, and then applying a modulo operation to capture the exclusive condition where gaming devices are unprotected.

```
def feature_engineering(data):
    ''' Creating new features based on current features '''
    # Hypothesis: devices that are used for gaming may have higher chance for malware attack if no firewall enabled
    #data['Gamer_NoFirewall'] = ((data['Wdft_IsGamer'] == 1) & (data['Firewall'] == 0)).astype(int)
    data['Gamer_Firewall'] = (data['Wdft_IsGamer'] + data['Firewall']) % 2
    data['Gamer_Firewall'] = data['Gamer_Firewall'].fillna(data['Gamer_Firewall'].mode()[0])
    # Hypothesis: devices that are used for gaming may have higher chance for malware attack if not protected
    data['Gamer_Protected'] = (data['Wdft_IsGamer'] + data['IsProtected']) % 2
    data['Gamer_Protected'] = data['Gamer_Protected'].fillna(data['Gamer_Protected'].mode()[0])
    # Hypothesis: highEndDevice are less vulnerable for malware attack
    data['HighEndDevice'] = (data['Census_ProcessorCoreCount'] >  data['Census_ProcessorCoreCount'].quantile(0.5))
    & (data['Census_TotalPhysicalRAM'] > data['Census_TotalPhysicalRAM'].quantile(0.5)).astype(int)
    data['HighEndDevice'] = data['HighEndDevice'].fillna(data['HighEndDevice'].mode()[0])
    # Hypothesis: devices which have optical dirve are tpically used for years. If they are not protected, there are chanced being affected by the malware
    data['Optical_Drive_Protected'] = (data['Census_HasOpticalDiskDrive'] + data['IsProtected']) % 2
    data['Optical_Drive_Protected'] = data['Optical_Drive_Protected'].fillna(data['Optical_Drive_Protected'].mode()[0])
    # Hypothesis: devices with higher security score are less vulnerable to malware attack
    data['Security_Score'] = (data['HasTpm'] + data['Census_IsSecureBootEnabled'] + (data['Census_IsVirtualDevice'] == 0).astype(int)).astype(int)
    data['Security_Score'] = data['Security_Score'].fillna(data['Security_Score'].mode()[0])
    # Hypothesis: if the system has more capacity, it is vulnerable to be attacked
    data['System_Volume_to_Disk_Capacity'] = data['Census_SystemVolumeTotalCapacity'] / data['Census_PrimaryDiskTotalCapacity']
    data['System_Volume_to_Disk_Capacity'] = data['System_Volume_to_Disk_Capacity'].fillna(-1)
    # Hypothesis: The ratio of total physical RAM to processor core count (RAM_to_Core_Ratio) is an indicator of a device's performance efficiency and
    # can influence its vulnerability to malware attacks. Devices with a higher RAM-to-core ratio might be better at handling multiple processes and
    # thus less prone to performance degradation from malware.
    data['RAM_to_Core_Ratio'] = data['Census_TotalPhysicalRAM'] / data['Census_ProcessorCoreCount']
    data['RAM_to_Core_Ratio'] = data['RAM_to_Core_Ratio'].fillna(-1)
```

Figure 7. Full list of new features created

Similarly, we hypothesized that high-end devices, defined by above-median processor core count and physical RAM, would be less vulnerable to attacks. This led to the creation of the HighEndDevice feature, which could help the model differentiate between varying levels of device security. Our feature engineering strategy aimed to encapsulate such nuanced insights, transforming them into tangible attributes that a machine learning model can leverage, thereby increasing its interpretability and predictive accuracy. A full list of new features is shown in Figure 7 along with the hypothesis.

Model Selection and Experiment Approach

For our malware prediction task, we decided to focus primarily on tree-based machine learning models, including decision tree and lightGBM models, recognizing their particular strengths in handling complex classification challenges inherent in cybersecurity. These models are adept at processing high-dimensional datasets, which is characteristic of the intricate feature space in malware detection problems. Concurrently, to enrich our understanding and apply the theoretical knowledge acquired in class, we ventured into the realm of neural networks. This foray into deep learning allowed us to gain practical experience with these advanced models, even though they are not our primary focus due to their less transparent nature.

Tree-based models distinguish themselves with an architecture that is naturally robust to feature scaling, adept at managing missing data, and capable of ignoring irrelevant features—qualities that simplify the pre-processing steps without compromising on predictive accuracy. Their ability to provide clear insight into the decision-making process is particularly advantageous. Unlike deep learning models, which can act as 'black boxes', tree-based models such as decision trees, random forests, and gradient boosting machines offer interpretable and justifiable decision criteria. In the specific context of malware detection, the explanatory power provided by these models is indispensable. It not only facilitates a deeper understanding of the predictions but also empowers cybersecurity professionals to make informed decisions and implement effective protective measures.

In our experimental approach, we initially considered a range of tree-based algorithms, including random forest and XGBoost. However, to deepen our comprehension of the inner workings of these algorithms, we chose to construct a decision tree model from the ground up. By comparing our homemade decision tree with one derived from an established library, we aim to gain intimate knowledge of the mechanics and nuances of decision tree algorithms. This hands-on comparison is designed to demystify the operational aspects of the model and enhance our grasp of its functionality, thereby solidifying our theoretical foundations with practical insights.

Results and Discussion

Decision Tree

We initiated our analysis with the DecisionTreeClassifier from the sklearn.tree package without setting any constraints on the maximum depth of the tree or the minimum number of

samples per leaf. This unrestricted model yielded a perfect 100% accuracy on the training dataset and a modest 57% on the testing dataset. Such a disparity suggested potential overfitting, where the model excessively learned the idiosyncrasies, including noise, within the training data, rather than discovering the genuine/true underlying patterns that correlate features with labels. To mitigate this overfitting, we introduced a limitation on the maximum depth of the tree. Restricting the depth of the tree helps to prevent overfitting by reducing the complexity of the model—it stops the model from creating excessively deep trees that can lead to learning the training data too precisely, including its noise and outliers.

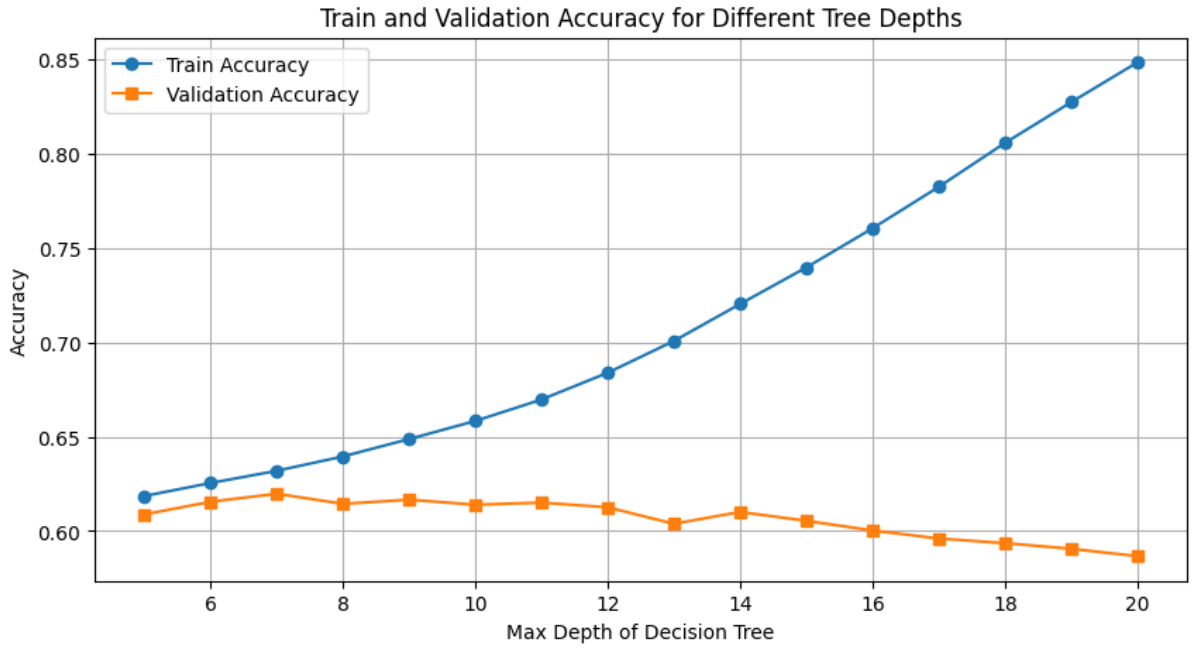

Figure 8. Train and validation accuracy for different tree depths.

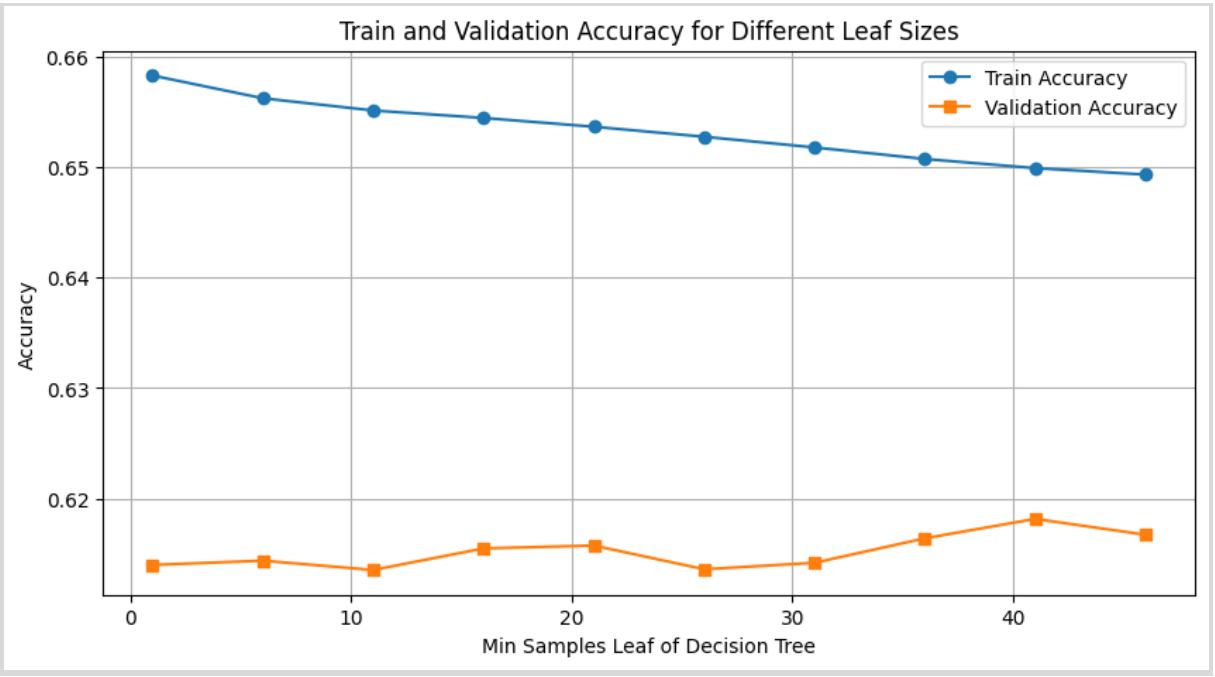

Figure 9. Train and validation accuracy for various min_smaple_leaf.

The accompanying figure (Figure 8) illustrates the train and validation accuracies across varying tree depths. Notably, as the depth increased, the training accuracy improved, while the validation accuracy began to decline, an indication of overfitting. Consequently, a maximum depth of 10 was selected to balance model complexity with generalization ability. Further, we explored the impact of min_samples_leaf on model performance. As anticipated, the training accuracy experienced a slight decrease with larger min_samples_leaf values, indicative of a simpler model less prone to overfitting. Meanwhile, validation accuracy remained relatively stable, showing only minor fluctuations. Thus, for subsequent experiments, we decided to retain the default min_samples_leaf value of 1, maintaining the model's sensitivity to the training data while observing its generalization on unseen data.

Next, we developed a decision tree from scratch, adhering to guidelines outlined in online tutorials (detailed references are provided in the appendix). Consistent with our earlier findings, we capped the tree's maximum depth at 10. Our self-constructed model achieved a comparable testing accuracy of 63%. However, a notable discrepancy was observed in the training duration; the homemade model required approximately 8 hours to complete its training on the same dataset, whereas the sklearn.tree model completed training in under 10 minutes. This substantial difference in computational efficiency could be attributed to several factors. Firstly, the sklearn library is optimized with highly efficient C++ code under the hood, whereas our implementation, likely in a high-level language like Python without such optimizations, would naturally run slower. Secondly, sklearn employs a variety of algorithmic optimizations for tree construction, such as the use of entropy or gini impurity calculations that are computationally optimized, and efficient data structures to manage the training data. In contrast, our homemade version may not have incorporated these sophisticated optimizations and data handling techniques. Lastly, the vectorization of operations and parallel processing are optimizations commonly absent in basic implementations but are present in professional-grade libraries like sklearn, which also contributes to the discrepancy in training times.

LightGBM

In advancing our exploration, we implemented the LightGBM model, a component of the gradient boosting family that constructs models in a stage-wise fashion. LightGBM has several advantages over a vanilla decision tree, notably its use of gradient-based learning. It builds trees one at a time, where each new tree helps to correct errors made by previously trained trees. This iterative correction often results in improved prediction accuracy, as the model dynamically adjusts to the complex patterns in the data.

The optimal testing accuracy reached with LightGBM was 67%, a slight improvement over the decision tree model. This increment can be attributed to LightGBM's ability to focus on the errors of the previous trees, effectively refining the model's predictions at each stage. It is also worth noting that gradient boosting models, due to their additive nature, can model complex relationships in the data more effectively than a single decision tree.

The significance of features in our model, as depicted in Figure 10, reveals that AvSigVersion, AVProductStatesIdentifier, and CountryIdentifier stand out as the top influential features. The AvSigVersion could be pivotal because it likely correlates with the evolution of malware signatures, thus being critical in detecting new malware variants. AVProductStatesIdentifier suggests the presence and status of antivirus products, which is inherently tied to the system's security state. CountryIdentifier might indicate geographic patterns in malware distribution or in the adoption of security measures. SystemVolumeToDiskCapacity also emerges as a key feature, perhaps due to its relation to system usage patterns, which could affect exposure to malware. Notably, some of the engineered features appear to have a minor impact on the model's prediction capability. This could

imply that while these features made logical sense during the feature engineering phase, they may not hold as much discriminative power in the actual dynamics captured by the model, or they could be conveying information that is already captured by other more dominant features.

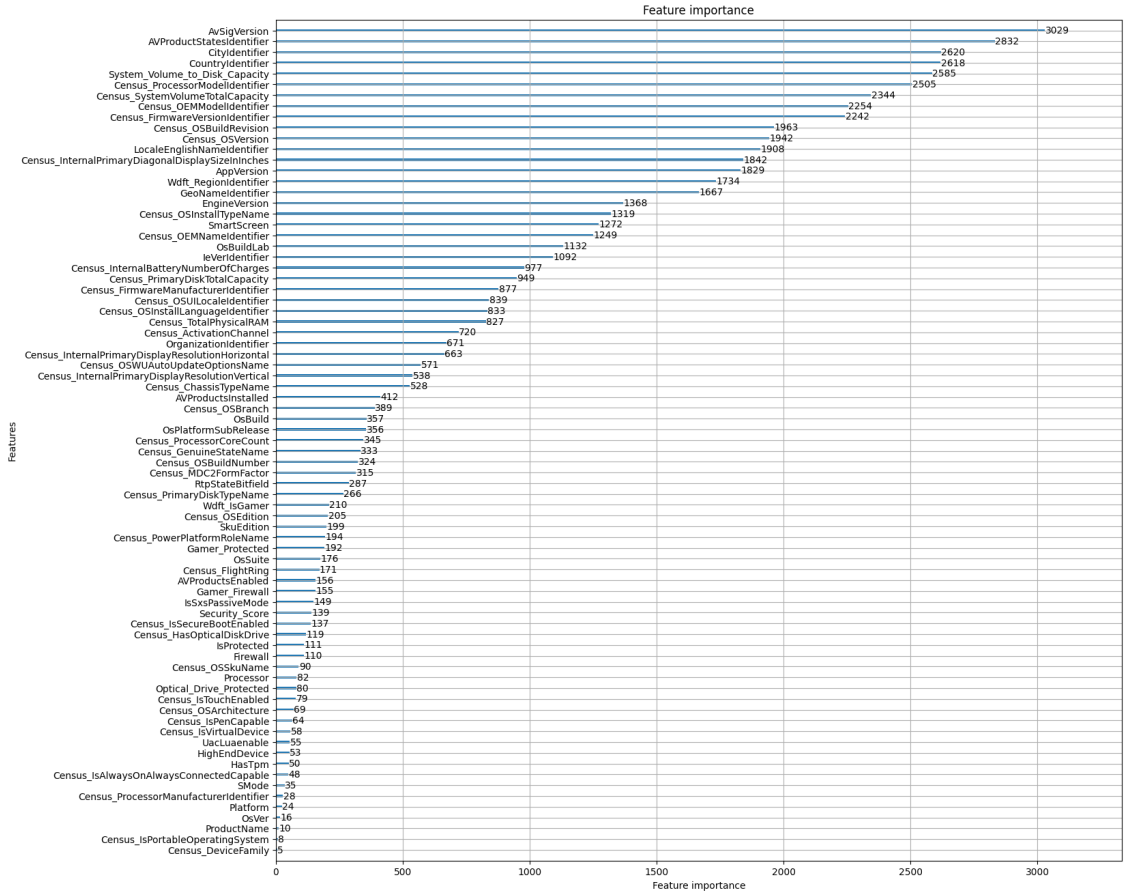

Figure 10. Feature importance.

Neural Network

In our neural network model development, we have designed an architecture that facilitates a clear and linear progression of data from input to prediction. The model architecture consists of a sequential arrangement starting with an input layer, followed by two hidden layers, and ending with an output layer designated for binary classification. The first hidden layer is composed of 100 neurons and uses the ReLU activation function to enable the modeling of complex patterns in the dataset. To prevent overfitting, a 40% dropout rate is applied following each hidden layer to intermittently deactivate neurons during the training phase. Additionally, batch normalization is utilized post-activation within the hidden layers to normalize the outputs, thus enhancing the training efficiency and model stability. The architecture concludes with a sigmoid-activated output neuron, optimized for binary outcome classification.

After training the model for 20 epochs, we review the training output which indicates the loss and accuracy at each epoch for both training and validation datasets.

We see that the accuracy is not improving significantly with more epochs, hovering around 59.19% on the validation set, which suggests the model is performing at chance level. This may imply that the model is not learning effectively from the data provided, and could be due

to several reasons such as an inadequate model architecture for the complexity of the task, insufficient or uninformative features, or a need for more advanced preprocessing.

Conclusions and Future Works

In conclusion, our analysis including various machine learning models has provided insightful observations into the nature of malware detection. Starting with the Decision Tree Classifier from the sklearn.tree package, we confronted the challenge of overfitting, as evidenced by a perfect training accuracy with significantly lower testing accuracy. By constraining the maximum depth and retaining the default minimum samples per leaf, we achieved a balanced model that effectively generalized to unseen data.

Then we constructed a decision tree from scratch which allows us to gain a deeper understanding of the algorithm's intricacies, albeit at the cost of computational efficiency. The disparity in training duration between our homemade model and sklearn's implementation highlighted the sophistication and optimization ingrained in professional libraries.

Advancing with LightGBM, we leveraged its gradient-boosting capabilities, resulting in a slight improvement in testing accuracy over the decision tree model. The interpretation of feature importance provided valuable insights, revealing significant predictors such as AvSigVersion, AVProductStatesIdentifier, and CountryIdentifier, while also identifying the relatively minor role of some engineered features.

Lastly, our neural network model, despite a thoughtfully designed architecture incorporating dropout and batch normalization to enhance performance, exhibited modest results. The plateau in accuracy suggests a need for re-evaluation of the network's complexity, the informativeness of the features, and potentially more sophisticated data preprocessing to better capture the subtleties of the task.

For future work, several potential directions can be considered. Firstly, experimenting with different encoding techniques for categorical features could yield significant improvements. The current encoding method impacts how these features are interpreted by the model, and exploring alternatives may reveal more about the underlying structure of the data. Secondly, expanding our suite of gradient boosting models to include alternatives like CatBoost could offer benefits. CatBoost is renowned for its handling of categorical features and its robustness to overfitting, particularly on datasets with a large number of categorical variables.

Github Repository

https://github.com/sherryxlxu/malware-prediction

Project Member Work Distribution

Based on the breakdown of tasks assigned to each member in Group 6, here is a summary of the contributions made by Xiaoliang and Changxu:

Xiaoliang:

Proposal drafting: Contributed equally with Changxu.

Report drafting: Contributed equally with Changxu.

Exploratory Data Analysis:
Dataset overview: Solely responsible for providing an overview of the dataset.
Missing data removal: Handled the task of identifying and removing missing data.
Data type classification: Took charge of classifying the data into various types.
Neural Network Exploration: Solely responsible for exploring and implementing the neural network model.

Changxu:

Proposal drafting: Contributed equally with Xiaoliang.
Report drafting: Contributed equally with Xiaoliang.
Exploratory Data Analysis:
Categorical feature normalization: Solely responsible for normalizing categorical features.
Data distribution: Analyzed and reported on the distribution of the data.
Feature Engineering: Entirely responsible for the creation and implementation of new features.
Decision Tree Model Exploration: Led the exploration and implementation of the decision tree model.
LightGBM Model Exploration: Conducted the exploration and application of the LightGBM model.

Both members contributed significantly, with Xiaoliang focusing more on the initial stages of the project, including data preparation and analysis, as well as the complex task of neural network exploration. Changxu, on the other hand, contributes to the feature engineering aspect and the exploration of advanced machine learning models like the decision tree and LightGBM. Their combined efforts facilitated a comprehensive approach to the project.

References

[1] An Extensive Step by Step Guide to Exploratory Data Analysis
[2] Step by step process of Feature Engineering for Machine Learning Algorithms in Data Science
[3] How To Defend Your PC From Cyber attacks and Malware when Gaming Online
[4] Transform Data to Normal Distribution in R
[5] How to Implement the Decision Tree Algorithm from Scratch in Python
[6] Python Implement Decision Tree from Scratch
[7] Program a Decision Tree in Python from 0