Malware Prediction - Midsem Project Report

Aditya Gupta
IIITD
Delhi, India
aditya22031@iiitd.ac.in

Avi Sharma
IIITD
Delhi, India
avi22119@iiitd.ac.in

Rishabh Jay
IIITD
Delhi, India
rishabh22401@iiitd.ac.in

Sahil Gupta
IIITD
Delhi, India
sahil22430@iiitd.ac.in

1. Abstract

We initiated this project to address the growing sophistication of malware, which poses significant threats to both individuals and organizations. Traditional security measures often fall short against evolving threats, so we sought to leverage machine learning to predict and prevent malware infections before they occur. By analyzing large datasets, we aim to enhance early detection capabilities and provide a more proactive solution to safeguarding digital systems.

2. Introduction

The malware industry is a sophisticated and well-resourced sector , continuously devising new strategies to evade conventional security mechanisms. Malware infections on computers can have severe consequences , leading to data theft , financial damage , and significant disruptions for both individual and organizations.
The goal of this project is to develop a predictive model capable of determining the likelihood of a computer being compromised by malware in the near future. Leveraging an extensive dataset provided by Microsoft , which contains detailed information on previous malware incidents , the project aims to create an innovative solution that can proactively identify and mitigate malware risks. This initiative will contribute to the broader effort of advancing open-source methodologies for malware prediction and enhancing the overall security landscape.

3. Literature Survey

The existing body of literature that explores this area of machine learning includes:

3.1. Malware Analysis and Detection Using Machine Learning Algorithms

This paper addresses polymorphic malware detection, evaluating Decision Trees (DT), Convolutional Neural Networks (CNN), and Support Vector Machines (SVM). DT achieved the highest accuracy (99%), followed by CNN (98.76%) and SVM (96.41%), with low false positive rates: DT (2.01%), CNN (3.97%), and SVM (4.63%). Link to paper

3.2. Evaluation of Machine Learning Algorithms for Malware Detection

The study focuses on dynamic malware detection using classifiers in a simulated environment. Random Forest (RF) and Gaussian Naive Bayes (NB) achieved 100% accuracy, precision, recall, and F1-score, demonstrating effective behavior-based detection. Link to paper

3.3. Microsoft Malware Prediction Using LightGBM Model

This research applies the LightGBM model for Windows malware detection, emphasizing feature engineering and evaluating performance using AUC-ROC. LightGBM achieved the highest AUC-ROC score of 0.684, outperforming Catboost and XGBoost. Link to paper

4. Dataset

4.1. Dataset details

The dataset used in this project is obtained from Kaggle platform , where the goal is to predict the probability of Windows machines becoming infected by various families of malware, based on machine-specific properties . The telemetry data was generated by integrating heartbeat and threat reports collected by Microsoft's endpoint protection solution, Windows Defender.

Each record in the dataset represents an individual machine, uniquely identified by a **MachineIdentifier** . The target variable, **HasDetections**, indicates whether malware was detected on the machine (1 for detected, 0 for not detected). The training data (**train.csv**) contains labels for HasDetections, while the test data (**test.csv**) requires predictions for this target variable.

The dataset comprises **8,921,483** unique machines in the training set and **7,853,253** unique machines in the test set. There are **83** features (including HasDetections in the training set), capturing a wide range of machine properties and configurations relevant to malware infection prediction.

4.2. EDA and preprocessing on the Dataset

We conducted an Elementary Data Analysis (EDA) on the dataset to extract key features and insights from the complex dataset with numerous variables.

Data types were optimized for memory efficiency by setting categorical variables as categories and converting numerical values to smaller types like `int8` and `float32`. Using a custom `reduce_memory_usage` function, memory usage was reduced by 17.1%, improving data handling and analysis speed.

The analysis identified several columns with significant data quality issues, including `PuaMode` and `Census_ProcessorClass`, which have over 99% missing values, and the `DefaultBrowsersIdentifier`, where 95% of entries belong to a single category.

Additionally, 26 other columns displayed similar imbalances, with over 90% of their values concentrated in one category, suggesting they should be removed from the dataset to enhance its quality and relevance.

Further investigation is warranted for *Census_IsFlightingInternal*,

which exhibited unusual patterns. We observe that the target classes are well-balanced, with **4,462,591** instances belonging to class 0 and **4,458,892** instances belonging to class 1 in the training dataset.

Thus, no additional balancing techniques were necessary. We conducted Exploratory Data Analysis (EDA) on the feature `Census_IsTouchEnabled` as follows:

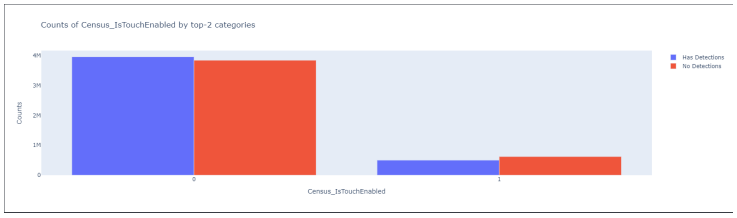

Figure 1. Touch_Based

We first examine variables with a high number of categories, followed by an analysis of those with a limited number of categories . An example of this is the feature *EngineVersion*, for which the plots are presented below:

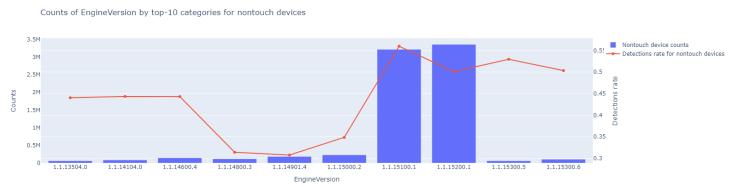

Figure 2. EngineVersion_NoTouch

Figure 3. EngineVersion_Touch

We observe that two categories account for 84% of all values, indicating a significant disparity in detection rates. The remaining categories exhibit varying detection rates, primarily due to the low number of samples available in those categories. Notably, the patterns for touch and non-touch devices are quite similar . Similarly, we identified comparable observations for other variables, such as *AppVersion*, *AvSigVersion*, *AVProductStatesIdentifier*, *AVProductsInstalled*, and others. The EDA also enabled us to identify instances where a variable was assigned an incorrect data type, such as `float16`/`float32`, despite its distribution following a categorical pattern . This way , we changed the datatype of such variables to **category** . One such example is of the variable *Wdft_RegionIdentifier*

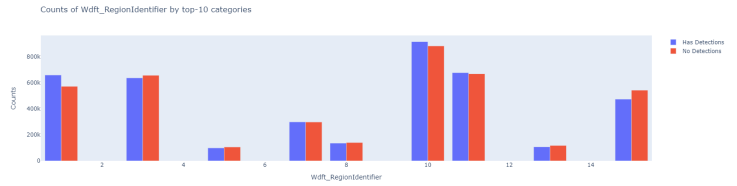

Figure 4. Categorical_Feature_Misclassified

Based on the insights from the EDA, we implemented the necessary adjustments and saved the modified training and testing datasets using pickle . This included the preprocessing using EDA on the dataset . Additional plots and details are available at the provided Github Link.

4.3. Feature Engineering

We start with handling the *'OsBuildLab'* column in which we prepare the column for training by converting it into categorical data type , adding a new category , and filling missing values which were present.

We first check if the *OsBuildLab* column in both the train and test datasets is of categorical data type or not. If it is not then we convert it into a categorical type.

We do this as categorical data types are more memory-efficient and it improves the performance of the learning model when dealing with non-numeric data.

We then define a new category which is *0.0.0.0.0-0*. Now , this category is used to replace missing values. This helps in maintaining data consistency and ensures learning can be done without any issues.

Now , after this we start our feature extraction which aims to extract relevant information from raw features enabling better analysis and model performance.

We create new categorical features by extracting specific components from version numbers of software and operating system details like *EngineVersion* , *AppVersion* , *AvSigVersion* , *OsBuildLab* and *Census_OSVersion*.

We set some disk and display calculations which are:

1) Primary drive ratio and size: this calculates the ratio of system volume capacity to total disk capacity and the size of the non primary drive.

2) Aspect ratio and display information: aspect ratio and dimensions of the display are calculated and stored as new features. DPI (dots per inch) and screen area are also derived, which give details about the screen quality and size.

3) Monitor Dimensions: this creates a combined representation of monitor's dimensions as categorical feature.

Then we have performance based features which are:

1) RAM per processor: this feature indicates the amount of RAM available per processor core.

2) Other display features: this creates additional features that combine diagonal screen size with number of processor cores to give interaction effects.

At the end we add missing values for some columns like *Census_IsFlightingInternal* , *Census_ThresholdOptIn* , *Census_IsWIMBootEnabled* and *Wdft_IsGamer* with default values.

Additional information about feature engineering are available at the provided GitHub Link.

5. Methodology and Model Details

We are utilizing the first 4,000,000 data points, which represent half of the total dataset, for training. This approach is intended to expedite the training process, as using the complete dataset would significantly increase the time required for model training.

This problem is a classification task, and we applied various methodologies to address it effectively. A range of models was explored, each chosen based on its appropriateness for the dataset and the nature of the problem. The models employed include:

1) Light Gradient Boosting Machine(LightGBM): Light Gradient Boosting Machine (LightGBM) is a gradient boosting framework that uses tree-based learning algorithms, designed for high performance and efficiency. We employ a robust training strategy

with k-fold cross validation(k=5) , enabling multiple models to be evaluated and selected based on their AUC performance . It handles large datasets efficiently by making chunked predictions and thus managing memory constraints during training and inference . The approach also allows for detailed insights into model performance through AUC scores and feature importance, making it suitable for complex classification problems with large datasets . The showcase of the feature importance is as follows :

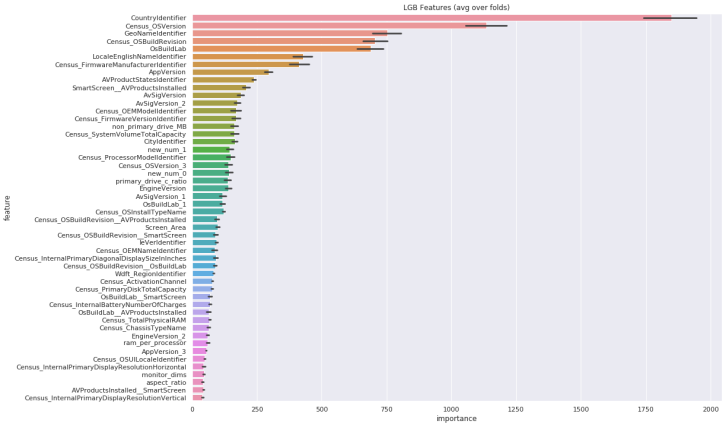

Figure 5. Feature_Selection

2) Extreme Gradient Boosting(XGBoost) : Extreme Gradient Boosting (XGBoost) is a highly efficient machine learning algorithm that enhances speed and performance through its parallelized tree-building and regularization techniques. This makes it a popular choice for predictive modeling in structured data, effectively minimizing overfitting. The implementation of XGBoost focuses on leveraging k-fold cross validation(stratified cross validation) to assess the model's performance robustly , while efficiently doing memory handling and AUC scoring provide insights into the model's predictive capabilities . The key hyperparameters are num_boost_round (set to 400) which determines the maximum number of boosting iterations early_stopping_rounds (set to 200), allowing early termination if validation performance does not improve.

3) Random forest (RF) model : This is an ensemble-based model that uses many multiple weak learners (decision trees) to classify its final predictions. It has three main hyperparameters, the number of trees, the maximum depth of each tree, and the number of features to be sampled. We employed a stratified k-fold which is a variation of the standard K-fold cross-validation technique. This variation ensures that each fold has the same class composition as the original dataset.The model was run with the following parameters:

$$\text{Number of Trees} = 100$$
$$\text{Depth} = 10$$

and it was run for 5 folds and the final prediction was the average prediction from the 5 models trained.

4) Decision Tree (DT) Model :

Preparation of the dataset for decision tree

While training the model using decision tree , we first prepare our dataset by dividing the columns based on the **continuous valued columns** and **version valued columns**. We exclude the columns *HasDetections* and *MachineIdentifier* and other than that for each column we calculate the unique values and the proportion of most common values. If a column's most frequent value accounts for more than 90% of the data, the column is removed from the dataset. We also check the columns which have more than 500 unique values, and we delete those columns as they don't have any significance in model training. Then for each categorical column except *HasDetections* , *MachineIdentifier* and those having datatype *int8* we calculate the detection ratio for each unique value, representing the mean value of the target variable *HasDetections* when unique value is present. Then we sort the unique values by their detection ratios, and mapping is created that assigns each unique value an integer based on its position in the sorted list.This mapping is added to conversion dictionary, where each column is mapped to its corresponding value to index mapping. Finally , conversion dictionary is stored and is used to convert categorical data into numerical representations based on the relationship to the target variable. Then version-related columns are broken down into their constituent parts for finer analysis. By systematically converting these non numeric columns into meaningful numerical values , we ensure that dataset is suitable for decision tree algorithm. This also improves the model's capacity to interpret categorical and version based information accurately. After forming the train and test dataset from this, we see the missing values within the dataset and replace any missing values with their corresponding column mean. This imputation ensures that the dataset remains complete without introducing bias, and it retains overall distribution of the data. Also, by filling *NaN* values systematically, the quality and consistency of the data are enhanced , improving model's ability to make accurate predictions. Similarly , we remove the single unique value columns from the dataset, as these features are not meaningful.

Training of decision tree model

Now, after forming the training dataset from the preprocessing steps , we perform hyperparameter tuning on Decision Tree classifier using k-fold cross-validation to identify the optimal *min_sample_leaf* value. By evaluating the model's performance across different values of *min_sample_leaf* , we aim to find the parameter that maximizes the area under the ROC curve (AUC) ,ensuring the best balance between bias and variance. The model's performance is assessed using cross-validation to ensure robustness and generalizability. The results are visualized using error bars to show the variability in performance, helping to select the most suitable configuration for decision tree. This process is crucial for improving model accuracy and stability. The maximum AUC score we get from this is 70.23%. Then after training the decision tree on the pre-processed training data , we apply the trained model to predict the probabilities for the test dataset. First, missing values in the test data are filled using pre-computation statistics to maintain consistency with the training data and then we apply the trained model. The accuracy we get from this model on the test dataset is 63.833%. More can be found at this link to Github.

6. Results and analysis

6.1. Results

In this section, we present the results of our classification models applied to the dataset, focusing on their performance metrics and comparative analysis. The primary metric used for the evaluation is the accuracy in the testing set and the training in the AUC score, which provides insight into the predictive capabilities of the models. We also discuss the implications of the importance of the features and any patterns observed during the model evaluation.

Model	Accuracy	AUC Score
LightGBM	0.675	0.717
XGBoost	0.657	0.708
Random Forest	0.658	0.659
Decision Tree	0.63833	0.7023

Table 1. Model Accuracy and AUC Summary

6.2. Observations

From the results obtained from different models , we can see that LightGBM model has the highest accuracy of 0.675 as well as the highest AUC score. XGBoost and Random Forest follows closely with an accuracy of 0.657 and 0.658 respectively. However Random Forest had the lowest AUC score of 0.659. Decision Tree on the other have the lowest accuracy of 0.63833 and a higher AUC score of 0.7023 than Random Forest.

7. Conclusion

7.1. Learning from the project

We can see that boosting methods like LightGBM and XGBoost outperform the traditional models like Random Forest and Decision Tree. Boosting techniques like LightGBM and XGBoost incrementally improving on their errors, making them able to generalize better.They can capture the complex relationship between the features.On the other hand , models like Random Forest and Decision Tree are prone to overfitting, a lower AUC score for Random Forest and a higher accuracy signifies that the model is able to correctly predict the right class , however the model struggles with ranking predictions, which means that it can fail to assign higher probablities to postivie class consistently in case of an imbalanced dataset.

7.2. Work Left

We plan to do exhaustive hyperparameter tuning for optimizing the performance of the models, and also plan to explore other techniques in feature engineering.

7.3. Contribution

- **Aditya Gupta**: Conducted Exploratory Data Analysis (EDA) and initial data preprocessing, as well as testing and implementation of the LightGBM model.
- **Avi Sharma**: Carried out data preprocessing, along with testing and implementation of the XGBoost model.
- **Rishabh Jay**: Performed Exploratory Data Analysis (EDA) and tested and implemented the Random Forests model.
- **Sahil Gupta**: Executed feature engineering to enhance model performance and tested and implemented the Decision Tree model.

8. References

- Malware Analysis and Detection Using Machine Learning Algorithms(Paper-1)
- Evaluation of Machine Learning Algorithms for Malware Detection(Paper-2)
- Microsoft Malware Prediction Using LightGBM Model(Paper-3)
- Eda and other references
- Feature engineering reference
- LGBM reference
- Fast ROC-computation
- Hyperparameter Testing
- Model and training reference
- Model and training reference