DSA4266: MALWARE CLASSIFICATION

Group 5: Github Repository

Eric Neo Zi Zheng
National University of Singapore
ericneo@u.nus.edu

Hans Neddyanto Tandjung
National University of Singapore
hans.neddyanto.tandjung@u.nus.edu

Hastuti Hera Hardiyanto
National University of Singapore
hastuti.hera@u.nus.edu

Matthew Antonio Wijaya
National University of Singapore
e0866231@u.nus.edu

Veronica Angelin Setiyo
National University of Singapore
setiyo.veronica@u.nus.edu

Vincent Aurellio Budianto
National University of Singapore
vincent.aurellio@u.nus.edu

***Abstract*—Malware poses a growing threat to cybersecurity, and effective detection methods are crucial for protecting organizations. This report examines the use of machine learning to classify malware, using the Microsoft Malware Classification Challenge dataset from Kaggle. We tested several models, including XGBoost, Multi-Layer Perceptrons (MLP), Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), and a Combined Neural Network Model to determine their ability to accurately identify different types of malwares. The models were evaluated using Multiclass Log Loss as the main performance metric. The best performing input-model combination is ASM with bytes input fed on Combined Neural Network Model.**

***Keywords*—machine learning, deep learning, malware classification**

I. Problem Statement

Malware, a contraction of malicious software, refers to any software designed with the intent of causing harm to data, devices, or users [1]. Traditional detection techniques are being challenged by sophisticated malware variants, making it imperative to develop robust classification systems. Given the growth in the malware industry, quick and accurate classification is essential for cybersecurity solutions to keep pace with rapidly evolving threats. The goal of this project is to develop a multi-class classification model to predict the malware family of a given malware file.

II. Dataset

The dataset for this project comes from the Microsoft Malware Classification Challenge (BIG 2015) on Kaggle [2]. The dataset includes nearly half a terabyte of compressed malware files. It contains a mix of malware samples across 9 families: Ramnit, Lollipop, Kelihos_ver3, Vundo, Simda, Tracur, Kelihos_ver1, Obfuscator.ACY, and Gatak. Each malware sample is uniquely identified by a 20-character hash ID and a class label representing the malware family. Each ID has two corresponding files:

A. *Bytes files*

These contain the hexadecimal representation of its binary content, specifically designed to exclude the Portable Executable (PE) header for safety.

.bytes file

```
00401000 00 00 80 40 40 28 00 1C 02 42 00 C4 00 20 04 20
00401010 00 00 20 09 2A 02 00 00 00 00 8E 10 41 0A 21 01
00401020 40 00 02 01 00 90 21 00 32 40 00 1C 01 40 C8 18
00401030 40 82 02 63 20 00 00 09 10 01 02 21 00 82 00 04
00401040 82 20 08 83 00 08 00 00 00 00 02 00 60 80 10 80
00401050 18 00 00 20 A9 00 00 00 00 04 04 78 01 02 70 90
00401060 00 02 00 08 20 12 00 00 00 40 10 00 80 00 40 19
```

Fig. 1. Bytes file snapshot

B. *ASM files*

These contain disassembled machine code, along with metadata such as function calls, strings, and other low-level code features.

.asm file

```
.text:00401000                          assume es:nothing, ss:nothing, ds:_data, fs:
nothing, gs:nothing
.text:00401000 56                       push    esi
.text:00401001 8D 44 24 08                  lea     eax, [esp+8]
.text:00401005 50                       push    eax
.text:00401006 8B F1                        mov     esi, ecx
.text:00401008 E8 1C 1B 00 00                   call    ??0exception@std@@QAE@ABQBD@
Z ; std::exception::exception(char const * const &)
.text:0040100D C7 06 08 BB 42 00                mov     dword ptr [esi], offset off_
42BB08
```

Fig. 2. ASM file snapshot

The dataset contains the following files:

1. train.7z: Raw data for the training set.
2. trainLabels.csv: IDs and labels for the training set (9 malware families).
3. test.7z: Raw data for the test set.
4. sampleSubmission.csv: Example submission format for Kaggle.
5. dataSample.csv: A sample preview of the dataset.

Only raw data is provided and no pre-extracted features are available. Total train dataset consists of 200GB data, out of which 50Gb of data is bytes files and 150GB of data is ASM files. There are in total 10,868 bytes files and 10,868 ASM files, totalling to 21,736 files.

III. Prior Works

Previous works on the Kaggle Malware Classification Challenge have primarily employed tree-based models, with XGBoost emerging as the most popular and high-performing approach [3] [4][5]. While these models achieved impressive results, there was a noticeable lack of submissions exploring neural network (NN) architectures, which could offer additional insights and performance improvements through their ability to learn complex patterns in data

Outside of the competition, some studies have experimented with CNNs for malware classification, specifically using bytes images as input [6][7]. However, these works did not extend CNN applications to ASM images, leaving a gap in exploring CNN's potential on this feature type.

The top-performing submissions [4] placed significant emphasis on sophisticated feature engineering, demonstrating the value of well-crafted features in enhancing model accuracy. However, these efforts were largely focused on minimizing log loss as the primary evaluation metric, with limited exploration into the interpretability of model

predictions. Moreover, most approaches applied single models rather than leveraging model combinations or ensemble techniques, which might further enhance prediction accuracy and robustness. These gaps in research highlight the potential to not only integrate neural networks but also to explore model interpretability and hybrid models, providing a more comprehensive understanding of the dataset and the models' performance.

IV. METHODOLOGY

A. Exploratory Data Analysis (EDA)

During our initial analysis, we encountered several challenges and notable characteristics in the dataset. Firstly, opening and processing the files proved challenging due to their large sizes and the compressed format. These challenges made initial exploration more complex. There is also substantial variability in file sizes, with some files being much larger than others.

Moreover, the dataset from the train set shows a significant class imbalance. For example, Class 5 only contains 42 instances, while other classes have considerably more data points, potentially impacting model training and requiring targeted handling.

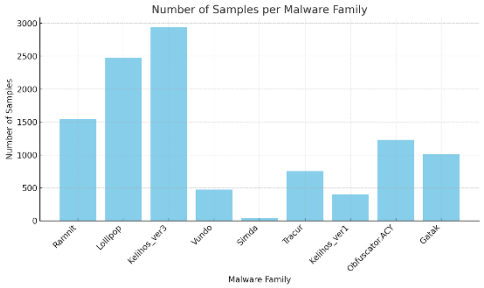


Fig. 3. Number of Samples per Malware Family

Lastly, some sections of the bytes files contain unknown or missing data marked as '??'. In certain files, these unknown sections form a significant portion of the file, which could impact feature extraction and would require special handling.

B. Data Preparation and Preprocessing

Given the large size of the malware dataset and the need to balance classes, we implemented a few preparation process to help with later steps:

1) Undersampling for Class Balance

To handle the imbalance dataset, we undersampled each of the majority classes in the train set down to 200 samples. One class (class 5) with an especially low sample count was excluded entirely to avoid introducing noise and imbalance. The resulting dataset contained eight malware classes, each with 200 samples.

2) Selective Extraction of Required Files

The dataset, stored in a compressed archive, was too large to extract in its entirety. To address this, we only extracted the files that have been sampled from part 1. The aim was to reduce the dataset size while retaining enough samples for robust model training. This selective extraction process began with identifying files for each malware class, which facilitated the following undersampling step.

C. Feature Engineering

Building on insights from prior research, extensive feature engineering has proven essential for achieving strong model performance. Consequently, we have incorporated several established techniques from our research. In addition, we generated the embeddings from the sequences of bytes and ASM opcodes. All of our features are shown in Fig 4.

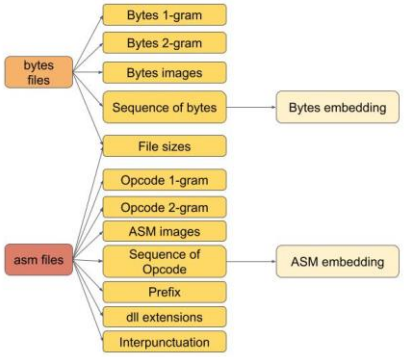


Fig. 4. Feature Engineering Diagram

1) File sizes

File size metrics were calculated to understand the scale of each file, including the ASM file size, bytes file size, and the ratio between these two metrics. This ratio provides insights into the relative density of the ASM representation versus its binary counterpart.

2) Bytes & ASM images

As the addresses in bytes files do not provide any information, the bytes files are first processed by removing the addresses. This is followed by the replacement of the unknown '??' values to '00'. There is an alternative of removing all unknown values altogether. However, after our preliminary experimentation, the former method of handling unknown values tends to perform better in CNN. The byte values are then reshaped to a 2D array and converted to square grayscale image (Fig 5).

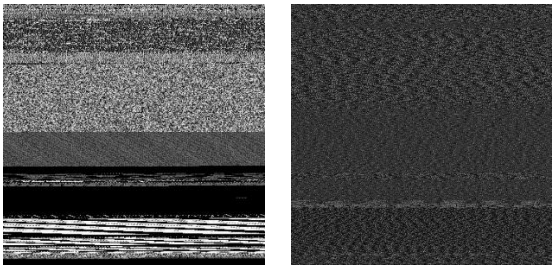


Fig. 5. Bytes (left) and ASM (right) file image

The ASM files are converted directly into grayscale images by first transforming the assembly code into binary data, which is then reshaped into a 2D array. This array is processed as-is, with no additional preprocessing steps, and then converted to a square grayscale image format (refer to Fig 5). This process contrasts with the handling of bytes files, which involved preprocessing (such as replacing '??' into '00'). As illustrated in the figures, the grayscale images generated from ASM files display finer detail compared to those derived from bytes files, likely due to the raw binary-to-pixel mapping that preserves more intricate structural patterns in the assembly code.

The image sizes of byes and ASM are variable, and this will require handling before being fed into the CNN model.

3) dll Extensions Count

The occurrences of file extensions inside the ASM files are counted and stored in a csv file. The extensions are '.dll', '.drv', '.ocx', '.cpl', '.scr', and '.sys'.

4) Interpunction Characters Count

The occurrences of each interpunction character inside the ASM file are counted and stored inside a csv file. The interpunction characters are as listed here:

[":", ",", ";", ".", "[", "]", "(", ")", "*", "+", "-", "=", "#", "$", "@", "?", "&", "!", "|", "^", "<", ">", "\\", "_"]

5) Segment Prefixes

We extracted segment prefixes from the ASM files, counting occurrences for each prefix as distinct features. These prefixes—HEADER:, .text:, .Pav:, .idata:, .data:, .bss:, .rdata:, .edata:, .rsrc:, .tls:, .reloc:, .BSS:, and .CODE—mark specific code or data sections within the ASM files.

6) Number of occurrences of N-grams

N-grams are contiguous sequences of n items from a given sample of text. For both bytes and ASM files, we've extracted 1-gram and 2-gram features using the hexadecimals in bytes files and opcodes in ASM files. For instance, a unigram (1-gram) considers individual words, while a bigram (2-gram) looks at pairs of consecutive words. Subsequently, we will generate the number of occurrences for each of this n-gram being extracted from the files. Additionally, we dropped the features with more than 800 zero counts which is more than half the sample size of 1600.

7) Embeddings

The sequences from bytes and ASM files are extracted as texts. For bytes, the addresses are removed. For ASM, only the opcodes are extracted. The sequences of bytes and opcodes are converted into embeddings of fixed size 384 to capture the sequential information. The embedding model used is a pre-trained model from HuggingFace, SentenceTransformers('all-MiniLM-L6-v2').

D. Train-Test Split

To evaluate the performance of our malware classification model, we split the dataset into training and testing subsets. 80-20 ratio is used, where the larger portion is for training, and the remainder is reserved for testing. In this train-test split, we also keep the proportion of each malware class equal. Ensuring this balance helps prevent model bias towards more prevalent malware types.

E. Primary Objectives

There are 2 primary objectives:

1. Explore which individual model yields the best performance
2. Explore whether using data features from bytes file only, or from ASM files only, or a combination of both yields the best classification results

F. Models

1) XGBoost

a) Data Preparation and Rationale

To prepare the data, all inputs are converted into a tabular format. For images derived from bytes and ASM files, only the first 400 pixels are selected and transformed into individual tabular columns (e.g., bytes_pixel_1, bytes_pixel_2, ..., asm_pixel_1, asm_pixel_2, ...). The rationale behind selecting only the initial pixels is supported by prior research [4], which indicates that for XGBoost classification, the 'texture' of the image is more critical than the entire image. This suggests that the distinctive patterns found in the early pixels are sufficient for effective model training and classification. As XGBoost is extensively used by most kaggle winners, we have chosen it as our baseline model for comparison with the neural network models.

b) Feature Importance

Unlike other models, feature selection through feature importance analysis is applied to determine the top 400 most significant features for each input set (e.g., bytes files only, ASM files only, and combined data from both files). This limit on feature count mitigates overfitting risks, based on insights from prior work [4] that also used XGBoost for a similar task. Although the referenced study had a significantly larger dataset, we maintained a similar ratio of data points to selected features to align with their methodology.

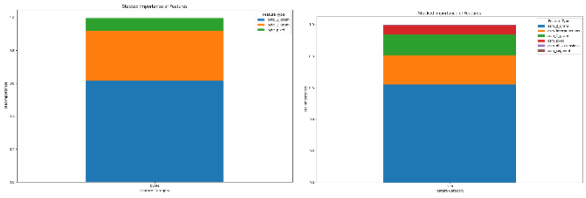


Fig. 6. XGBoost Feature Importance Bytes(Left) and ASM(Right)

Starting with the bytes files, the most important features were byte_2_gram (0.60), byte_1_gram (0.30), and byte_pixel (0.10). It's worth noting that features with more columns, like byte_2_gram, can have a higher cumulative importance, so the number of columns is a factor in interpreting these scores. For ASM files, the most influential features are led by asm_2_gram (0.60), followed by asm_interpunctions (0.20) and asm_1_gram (0.13).

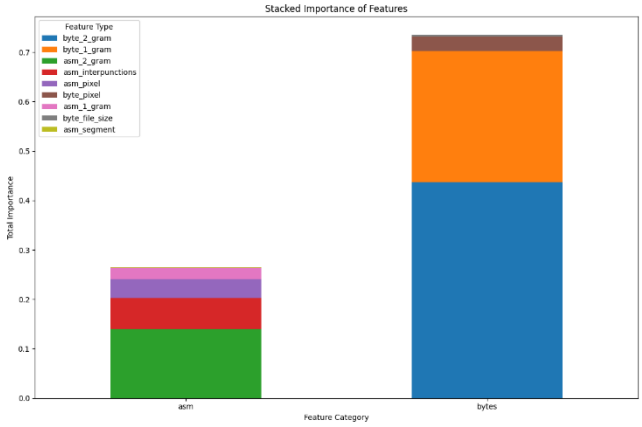


Fig. 7. XGBoost Feature Importance Bytes + ASM

When both ASM and bytes files are combined, bytes features dominate, contributing about 73% of the total importance. The top features are byte_2_gram (0.45),

byte_1_gram (0.27), and then asm_2_gram (0.14). There's a slight shift in ranking within ASM features here; asm_1_gram becomes more significant, while asm_pixel is less so, highlighting how ASM feature importance can vary when combined with bytes data.

2) MLP

a) Data Preparation

To prepare the data, all tabular non-image, non-sequence features are joined into a data frame. Normalisation is then performed on all the data using tf.keras.math.l2_normalize for each feature in the data frame. Train test split is then performed on the data frame based on the previously sampled train and test labels. Class labels are then one hot encoded such that it is compatible with the MLP model.

b) Model and Rationale

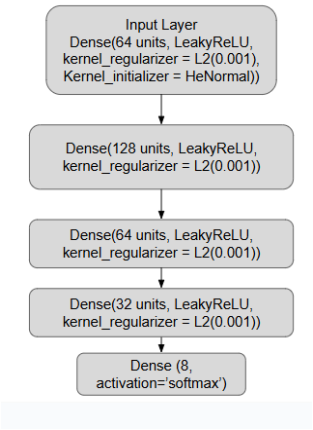


Fig. 8. MLP Model Architecture

We utilized MLP in our pipeline to process tabular non-image, non-sequential input effectively. This will then be combined with other neural networks to complement each other in the combined neural network model. The model is composed of four dense layers with the first layer being the input layer followed by an output layer with softmax activation. All dense layers are regularised with L2 regularisation to prevent gradient issues. The weights are initialised with HeNormal initialisation because of its compatibility with ReLU activation functions and to ensure fast convergence.

3) CNN

a) Input Images

Image size (in pixels) for asm and bytes input

ASM Image Size Statistics		Bytes Image Size Statistics	
count	1280.000	count	1280.000
mean	10209007.238	mean	941943.028
std	18692624.462	std	858387.399
min	19600.000	min	30975.000
25%	850084.500	25%	200704.000
50%	3757802.500	50%	581406.000
75%	9168796.000	75%	1461680.000
max	144180056.000	max	3906552.000
dtype: float64		dtype: float64	

Fig. 9. ASM and Bytes Image Analysis

For our CNN model, we adjusted and tuned input image sizes for ASM and bytes files to enhance model performance. We found that 64x64 worked effectively for ASM, while 512x512 was more suitable for byte files, albeit with increased computational demands. By selecting image sizes that are powers of two, we aim to maximise computational efficiency.

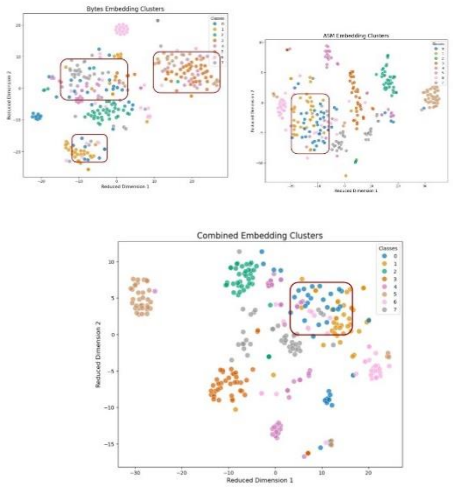
Our research suggests that ASM files contain critical information in their heading, allowing us to use only the first few thousand pixels as input, despite these files having a larger overall image size compared to bytes files. In contrast, bytes files lack this structured information at the beginning, so we opted for a larger input image size for the bytes model. This choice addresses challenges in achieving clear class differentiation, especially given the limited training data available for bytes files.

4) RNN/LSTM

a) Input Embeddings

The rationale of choosing to evaluate RNN is that it is able to process sequential data and capture dependencies between elements in the sequences of bytes and opcodes. As RNN/LSTM only take in continuous values, our initial approach was to tokenize the sequence of bytes and opcodes. However, this led to each file having varying input length due to the different length of sequences. Hence, to resolve this issue, we utilised the embeddings from sequence of bytes and opcodes to have a fixed input length of 384.

Furthermore, we performed an analysis of the embeddings by clustering them using t-SNE dimensionality reduction.



Class 0-7 here refers to class 1-4, 6-9 of the dataset

Fig. 10. RNN/LSTM Input Embedding Visualization

When analysing the bytes embedding clusters, we observed significant overlap among the points from different classes. In the case of the ASM embedding clusters, we found a considerable overlap between the points of class 1 and class 2. On the other hand, when we examined the embedding clusters using features from both bytes and asm files, the overlap between the two classes was reduced. As such, we wish to explore whether the RNN layers will be able to separate these points before parsing into the final Dense layer for classification.

b) Model and Rationale

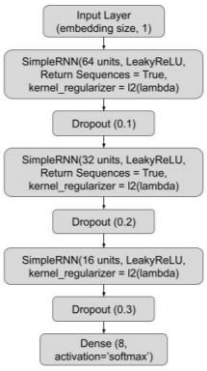


Fig. 11. RNN/LSTM Model Architecture

The model architecture consists of three SimpleRNN layers with decreasing units. Each SimpleRNN layer is followed by a dropout layer to help reduce overfitting. Since the dropout process randomly selects neurons to be ignored, we begin with a lower dropout rate of 0.1 to ensure the model does not underlearn. As the model learns to capture more complex patterns in the data, we gradually increase the dropout rate to prevent overfitting. Finally, the model includes a dense layer that outputs the probabilities for the 8 classes using the softmax activation function.

LSTM shares the similar model architecture as RNN except that it uses tanh as the activation function and sigmoid as the recurrent activation function. The gating mechanisms of LSTM utilise sigmoid activation function, hence they work well with bounded values (https://arxiv.org/pdf/2109.14545). If the input data to the LSTM layers contains large or unbounded values, the network may experience instability during training due to issues like exploding gradients. Hence, leakyReLU's output range of $(-\infty, \infty)$ does not align well with the gating mechanism. Whereas the tanh function's symmetric range of [-1, 1] aligns with LSTM's gating mechanisms, facilitating smoother gradient flow and stable learning, making it the default choice.

In addition, we tried training the LSTM model using three different types of embeddings: byte embeddings, ASM embeddings, and a combination of both. However, the models trained with byte embeddings and combined embeddings encountered an issue where they predicted class 2 for all samples, possibly due to exploding or vanishing gradient. While adjusting the hyperparameters could potentially fix this problem, we chose to keep the hyperparameters consistent in order to facilitate a fair comparison with the RNN model.

5) Combined Neural Network Model

a) Data Preparation

For our ensemble approach, we begin with embeddings from the already-trained neural network models. Table 1 lists down the models used in each combined NN model.

TABLE I. COMBINED MODEL INPUT

Input	List of Models Used
ASM	• ASM CNN model • ASM MLP model • ASM RNN model
bytes	• Bytes CNN model • Bytes MLP model • Bytes RNN model
ASM and bytes	• ASM CNN model • Bytes CNN model • Bytes + ASM MLP model • Bytes + ASM RNN model

To obtain these embeddings, we remove the softmax layer from each model, allowing us to retain the feature representations without any classification probabilities. This approach ensures that we're using the raw, learned features, which are essential for a combined model. After extracting these embeddings, we apply L2 normalization to each one, which is crucial as it brings the embeddings onto a common scale, ensuring that each model contributes equally to the final concatenated feature vector. Finally, we concatenate these normalized embeddings to create a comprehensive feature vector for each input sample.

b) Model Design

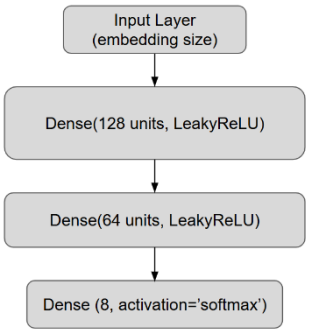


Fig. 12. Combined Model Architecture

After obtaining the concatenated embeddings, we pass them through a three-layer dense architecture where each layer uses the LeakyReLU activation function. We chose LeakyReLU to avoid issues with "dead neurons" that can arise from standard ReLU. The final layer is an 8-unit softmax layer. The model is trained using Adam optimizer for 10 epochs. Due to the naturally separable nature of the input data, a more complex model led to overfitting and resulted in unstable training loss. Training for 10 epochs proved sufficient for this setup, as it allowed the model to learn meaningful patterns without overfitting, as confirmed by stable training and validation loss curves.

6) XGBoost

a) Input

To prepare the data, all inputs are converted into a tabular format. For images derived from bytes and ASM files, only the first 400 pixels are selected and transformed into individual tabular columns (e.g., bytes_pixel_1, bytes_pixel_2, ..., ASM_pixel_1, ASM_pixel_2, ...). The rationale behind selecting only the initial pixels is supported by prior research, which indicates that for XGBoost classification, the 'texture' of the image is more critical than the entire image. This suggests that the distinctive patterns found in the early pixels are sufficient for effective model training and classification.

Unlike other models, feature selection through feature importance analysis is applied to determine the top 400 most significant features for each input set (e.g., bytes files only, ASM files only, and combined data from both files). This limit on feature count mitigates overfitting risks, based on insights from prior work that also used XGBoost for a similar task. Although the referenced study had a significantly larger dataset, we maintained a similar ratio of data points to selected features to align with their methodology. The model uses default parameter values to simplify design and focus on feature engineering and feature importance.

b) XGBoost Feature Importance

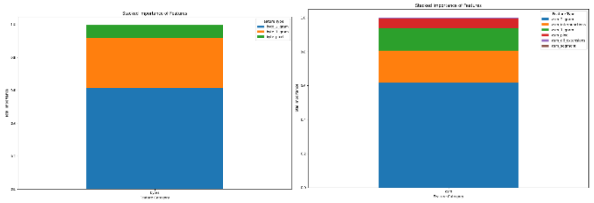


Fig. 13. XGBoost Feature Importance Bytes(Left) and ASM(Right)

Starting with the bytes files, the most important features were byte_2_gram (0.60), byte_1_gram (0.30), and byte_pixel (0.10). It's worth noting that features generating more columns, like byte_2_gram, can have a higher cumulative importance, so the number of columns is a factor in interpreting these scores. For ASM files, the most influential features are led by ASM_2_gram (0.60), followed by ASM_interpunctions (0.20) and ASM_1_gram (0.13).

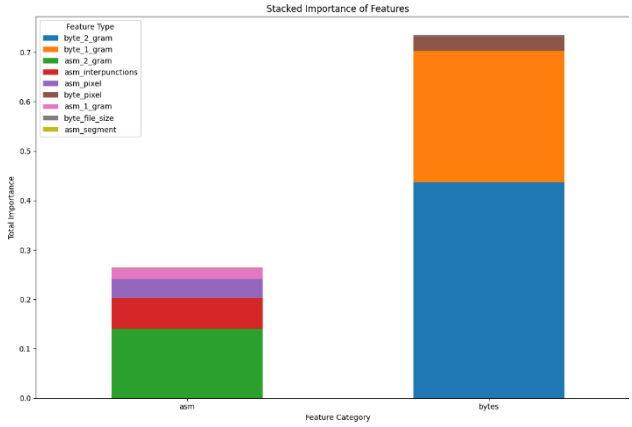


Fig. 14. XGBoost Feature Importance ASM + Bytes

When both ASM and bytes files are combined, bytes features dominate, contributing about 73% of the total importance. The top features are byte_2_gram (0.45), byte_1_gram (0.27), and then ASM_2_gram (0.14). There's a slight shift in ranking within ASM features here; ASM_1_gram becomes more significant, while ASM_pixel is less so, highlighting how ASM feature importance can vary when combined with bytes data.

G. Hyperparameter Choices

1) Activation Function

For the output layer, softmax activation function is used because our problem is a multiclass classification problem. All other layers use Leaky ReLU activation function for its simplicity and its ability to handle negative values better compared to ReLU.

2) Optimizer

Adam Optimizer is used for its broad compatibility with various model types. Adam Optimizer helps models to reach fast and stable convergence.

3) Regularisation

L2 Regularisation is used to reduce overfitting and bound our inputs since our activation function is unbounded to prevent gradient issues in the neural network.

4) Normalisation

Normalisation is performed on all numerical features by using tf.keras.math.l2_normalization to ensure all features are weighed evenly in the model.

H. Choice of Evaluation Metric

MLogLoss measures the uncertainty of the models' predictions based on their probabilities. It takes into account not only the predictions' correctness, but also the models' confidence. A confident wrong prediction will mean that the model assigned a very high probability to the wrong class which results in a larger penalty, while a correct prediction with high confidence is rewarded.

Furthermore, we also calculated accuracy, and utilised confusion matrices (Appendix A) to evaluate the models' performance in predicting each individual malware class. This allows us to analyse where the misclassifications occur and deep dive into finding the root cause. For better visualisations of the misclassifications, we also generated misclassification matrices (Appendix x) by truncating the diagonal values of confusion matrices to 0.

V. Results and Evaluation

The following is the overview of our model results.

TABLE II. Test multiclass log loss of Malware Classification Models

Model	Bytes only	ASM only	ASM + bytes
RNN	0.9687	0.6851	0.5842
LSTM	-	0.9487	-
CNN	5.0120	0.5314	-
MLP	0.7538	0.4560	0.2406
Combined (MLP + CNN + RNN)	0.3216	0.0642	0.0321
XGBoost	0.0909	0.0407	0.0547

TABLE III. Test accuracy of Malware Classification Models

Model	Bytes only	ASM only	ASM + bytes
RNN	0.68	0.78	0.80
LSTM	-	0.69	-
CNN	0.77	0.94	-
MLP	0.77	0.94	0.97
Combined (MLP +	0.92	0.98	0.99

Model	Bytes only	ASM only	ASM + bytes
CNN + RNN)			
XGBoost	0.97	0.99	0.98

A. *XGBoost*

Among all the models tested, XGBoost stands out as the only tree-based model and consistently performs exceptionally well for this malware classification task. Regardless of the input type—whether bytes files only, ASM files only, or a combination of both—XGBoost outperforms most neural network-based models, with mlogloss consistently below 0.10 and accuracy above 0.96.

Interestingly, despite bytes files showing greater feature importance than ASM files during feature analysis when combined (as shown from Fig. 14 feature importance xgboost combined), XGBoost's performance with ASM files alone surpasses its performance with bytes files alone. Specifically, the mlogloss for XGBoost using bytes files only is 0.091—more than double that for XGBoost using ASM files only, which achieves an mlogloss of 0.041.

Surprisingly, XGBoost's performance with ASM files alone also surpasses its performance when both ASM and bytes files are combined, which results in an mlogloss of 0.055. This finding may indicate that, in the context of XGBoost, the bytes files introduce significant noise when combined with ASM files, thereby reducing the model's overall performance with the combined features.

B. *CNN*

a) *Training and Testing Loss*

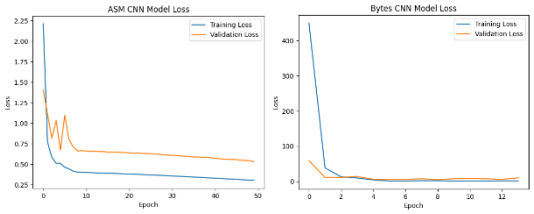


Fig. 15. CNN Loss Plot

Both the CNN model using ASM input and the one using bytes input exhibit a decreasing trend in both training and testing loss over time, indicating that neither model is currently overfitting. However, due to the implementation of early stopping, the bytes CNN model reaches convergence in fewer epochs, suggesting it might start to overfit if training were to continue. In contrast, the ASM CNN model continues to show improvements in learning, even up to the maximum allowed epochs, indicating it has not yet fully converged.

b) *Output Embedding Visualization*

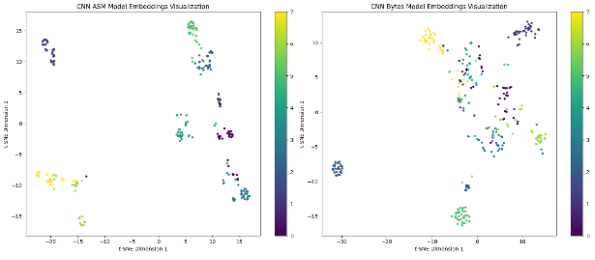


Fig. 16. CNN Embedding Visualisations

Distinct clusters are visible in the visualizations of both models. For the ASM CNN model, the clustering is more pronounced, with classes grouped closely together, although the clusters are not fully separated and remain somewhat close to each other. In contrast, the bytes CNN model shows only two clearly distinguishable clusters, while the remaining classes appear intermingled near the center, indicating less effective separation of features.

C. *MLP*

a) *Training and Testing Loss*

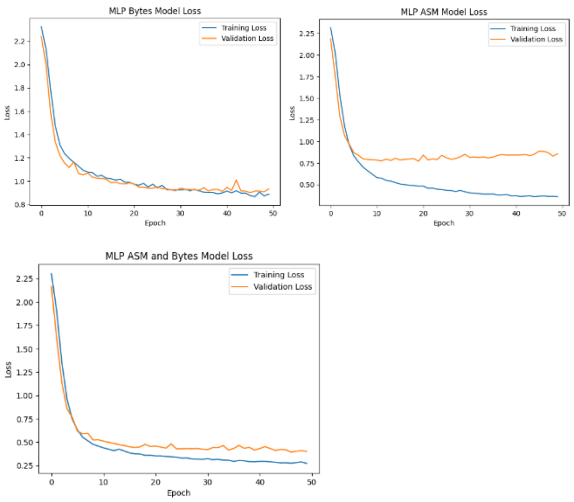
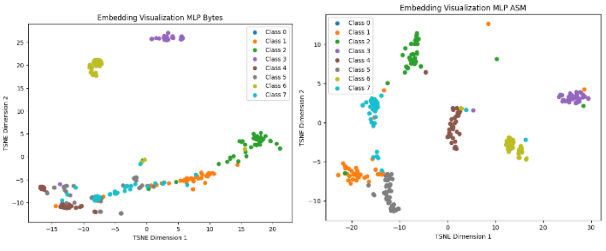


Fig. 17. MLP Loss Plot

From the above plots, the models have converged and are able to successfully capture the patterns in the data generally without overfitting, as evident by both training and testing loss decreasing. However, the graph for ASM shows the possibility of overfitting because the training and testing loss are not converging to the same value. This could be caused by MLP not being able to fully capture the patterns in the training data well enough for generalization because of the simplicity of the model.

b) *Output Embedding Visualization*

To increase the model results' interpretability and investigate the cause of the misclassifications, we visualised the clusters of embeddings that we retrieved from the second last model layer. This is applicable for the other neural network models.



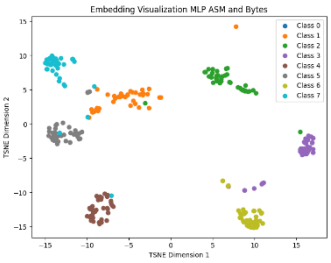


Fig. 18. MLP Embedding Visualizations

When analysing the embedding visualizations in the figure above, we observe that embeddings for ASM and ASM + bytes are able to clearly separate between classes, with some overlap between class 2 and 7 which indicates slightly worse performance compared to ASM + bytes. In the case of bytes embedding, it is unable to separate between the classes clearly and are only able to separate class 3, 4 and 8 while other classes are close to each other with significant overlap. This is also reflected in the performance of our MLP bytes model in other metrics used.

D. Combined Neural Network Model

1) Training and Testing Loss

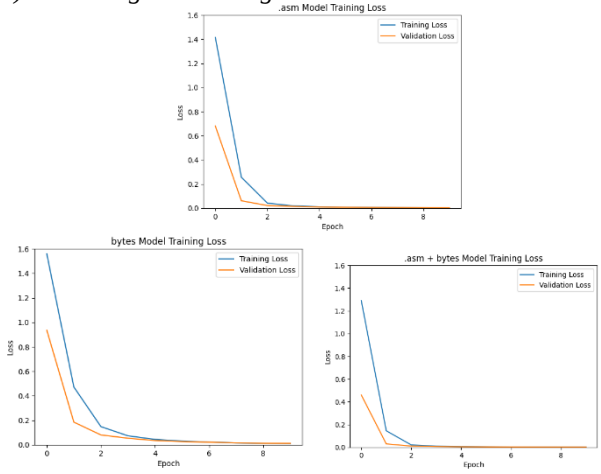


Fig. 19. Combined Model Loss Plot

Each plot shows a rapid drop in both training and validation loss within the first few epochs, indicating that the models are learning effectively from the input data. By around the second or third epoch, all models have nearly reached a point of minimal loss, with validation loss closely matching training loss, suggesting a low risk of overfitting. The similarity in loss trends across all models supports the robustness of our design and justifies our choice of 10 epochs, as further training does not significantly impact the loss.

VI. Discussion

Based on our results in Table 2 and Table 3, most of the models perform best when the input is a combination of bytes and ASM features. This may be attributed to the variable length of the commands in the ASM files.

```
.text:00401000
assume es:nothing, ss:nothing, ds:_data,    fs:nothing, gs:nothing
.text:00401000 56                                          push
esi
.text:00401001 8D 44 24     08
lea     eax, [esp+8]
.text:00401005 50                                          push
eax
.text:00401006 8B F1                                       mov
esi, ecx
.text:00401008 E8 1C 1B     00 00
call    ??0exception@std@@QAE@ABQBD@Z ;
```

Fig. 20. ASM file snapshot

```
00401000 56 8D 44 24 08 50 8B F1 E8 1C 1B 00 00 C7 06 08
```

Fig. 21. Bytes file snapshot

The ASM files retain information about the variable length of instructions, whereas this detail is absent in the bytes files. For instance, in the ASM file, the command push esi is represented by the single byte '56', while 'lea eax, [esp+8]' is represented by the sequence '8D 44 24 08'. In contrast, the bytes files lack this contextual differentiation; the byte '56' may represent 'push esi' when used alone, but could mean something entirely different in combination with other bytes (e.g., '33 56 4F'). This absence of contextual information in the bytes files leads to a loss of important instruction-specific details.

VII. Conclusion

In conclusion, the Combined Neural Network Model using both ASM and bytes features achieved the best performance, with the XGBoost model trained on ASM features followed closely, showing high performance as well. Across all our neural network models, we consistently observed that models with ASM + bytes performed best, followed by ASM only, with both significantly outperforming models using bytes features alone. This trend was also reflected in the XGBoost results, where ASM and ASM + bytes performed better than bytes alone. These findings underscore the value of ASM features in malware classification and support the advantage of combining ASM and bytes for optimal performance. Additionally, they demonstrate the value of integrating individual models into a combined model, as it enables a comprehensive feature representation that captures insights from each model architecture.

VIII. Future Works

For future improvements, we can explore several enhancements to our approach. First, adding 3, 4, or more-gram features may provide more sequential information and capture relationships between each consecutive hexadecimal or opcode, potentially enriching the feature set. We could also include CNN with CLIP [8], which may improve the model's ability to predict unseen classes by leveraging a more generalized representation. Moreover, using different embeddings for RNN and LSTM that can better differentiate the classes could also help improve classification accuracy, as the current embeddings generated by the HuggingFace transformer produces clusters that are not distinctly separated. Finally, training the model with more data would likely increase robustness and enhance its ability to generalize across diverse malware samples.

References

[1] AVG, "What is Malware? How Malware Works & How to Remove it," Avg.com, 2015. https://www.avg.com/en/signal/what-is-malware

[2] "Microsoft Malware Classification Challenge (BIG 2015)," @kaggle, 2015. https://www.kaggle.com/competitions/malware-classification (accessed Sep. 03, 2024).

[3] paulrohan2020, "Microsoft Malware Detection - log loss of 0.0070," Kaggle.com, Oct. 18, 2021. https://www.kaggle.com/code/paulrohan2020/microsoft-malware-detection-log-loss-of-0-0070 (accessed Oct. 01, 2024).

[4] xiaozhouwang, "GitHub - xiaozhouwang/kaggle_Microsoft_Malware: code for kaggle competition Microsoft malware classification," GitHub, 2015.

https://github.com/xiaozhouwang/kaggle_Microsoft_Malware (accessed Oct. 01, 2024).

[5] jiwei liu, “First place approach in Microsoft Malware Classification Challenge (BIG 2015),” YouTube, May 18, 2015. https://www.youtube.com/watch?v=VLQTRlLGz5Y (accessed Sep. 10, 2024).

[6] M. Kalash, M. Rochan, N. Mohammed, N. D. B. Bruce, Y. Wang and F. Iqbal, "Malware Classification with Deep Convolutional Neural Networks," 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), Paris, France, 2018, pp. 1-5, doi: 10.1109/NTMS.2018.8328749. keywords: {Malware;Computer architecture;Machine learning;Gray-scale;Learning systems;Convolution;Support vector machines;Malware classification;convolutional neural networks;deep learning},

[7] D. Pant and R. Bista, “Image-based Malware Classification using Deep Convolutional Neural Network and Transfer Learning,” 2021 3rd International Conference on Advanced Information Science and System (AISS 2021), Nov. 2021, doi: https://doi.org/10.1145/3503047.3503081.

[8] Lee, J. S., Tay, K. K., & Chua, Z. F. (2022). BinImg2Vec: Augmenting Malware Binary Image Classification with Data2Vec. 2022 1st International Conference on AI in Cybersecurity, ICAIC 2022. https://doi.org/10.1109/ICAIC53980.2022.9897062

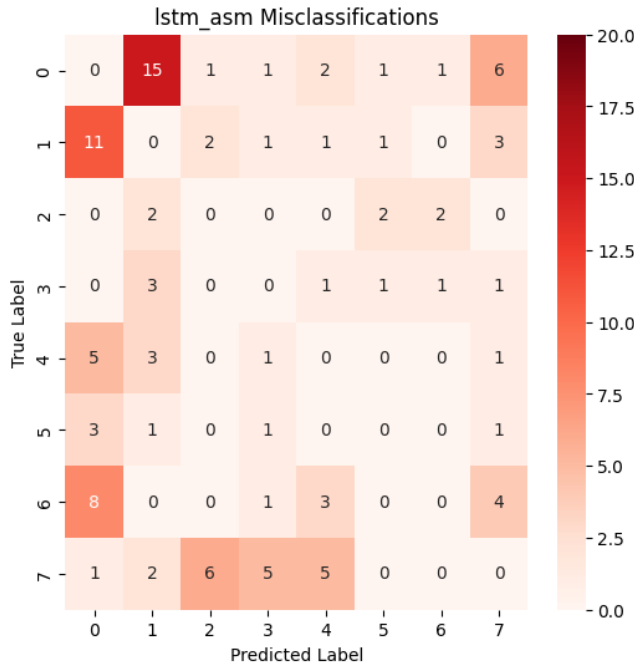
[9] “Microsoft Malware Classification Challenge (BIG 2015),” @kaggle, 2015. https://www.kaggle.com/competitions/malware-classification/discussion/13863 (accessed Sep. 10, 2024).

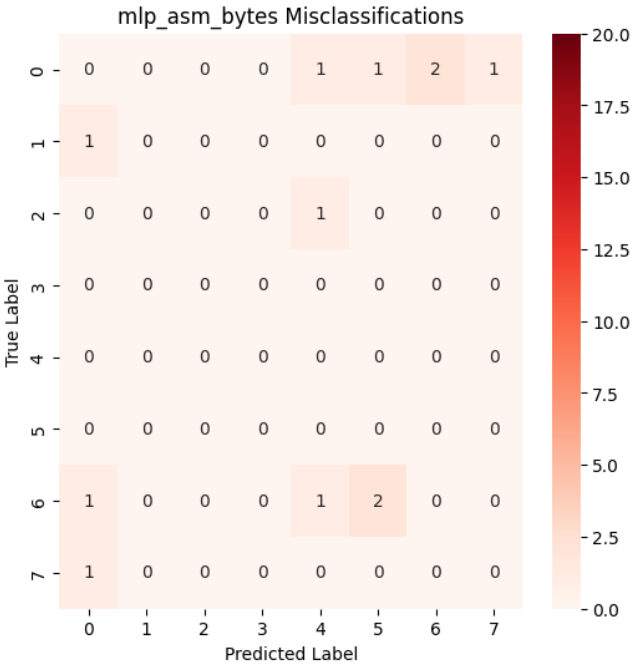
[10] elemento, “Microsoft_MalwareClassification,” Kaggle.com, Oct. 18, 2021. https://www.kaggle.com/code/elemento/microsoft-malwareclassification#4.4.-Machine-Learning-models-on-features-of-ASM-files (accessed Sep. 18, 2024).

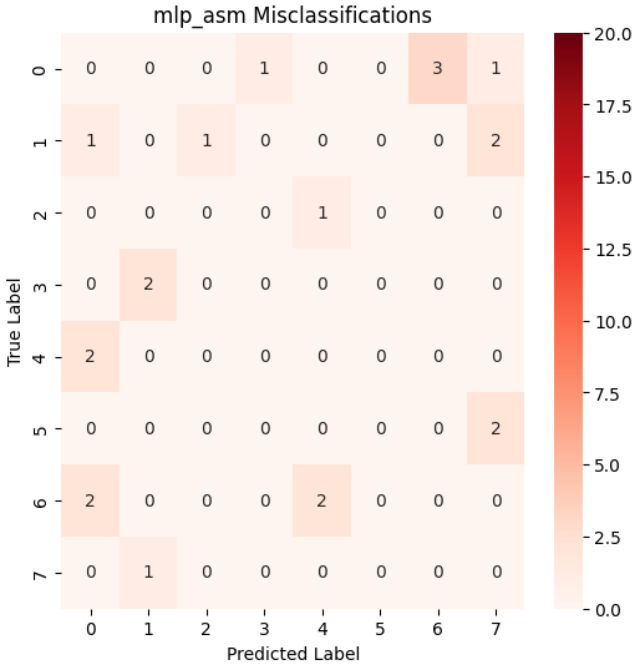
[11] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, “Microsoft Malware Classification Challenge,” arXiv:1802.10135 [cs], Feb. 2018, Available: http://arxiv.org/abs/1802.10135

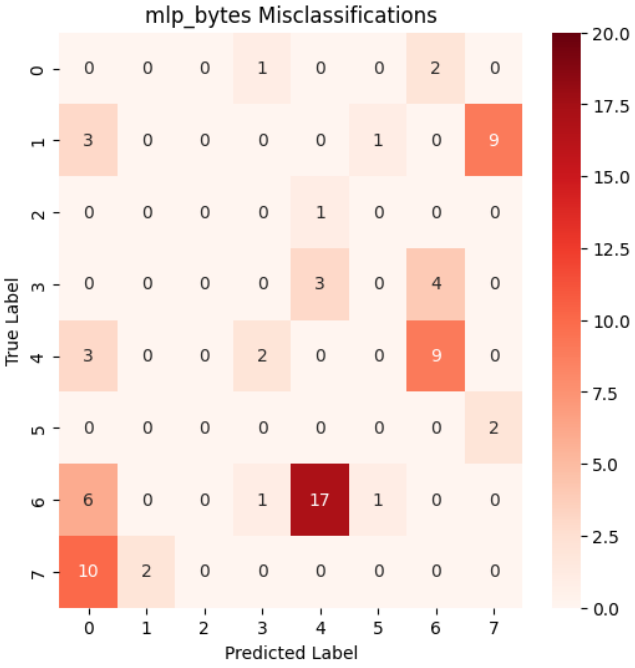
IX. Appendix

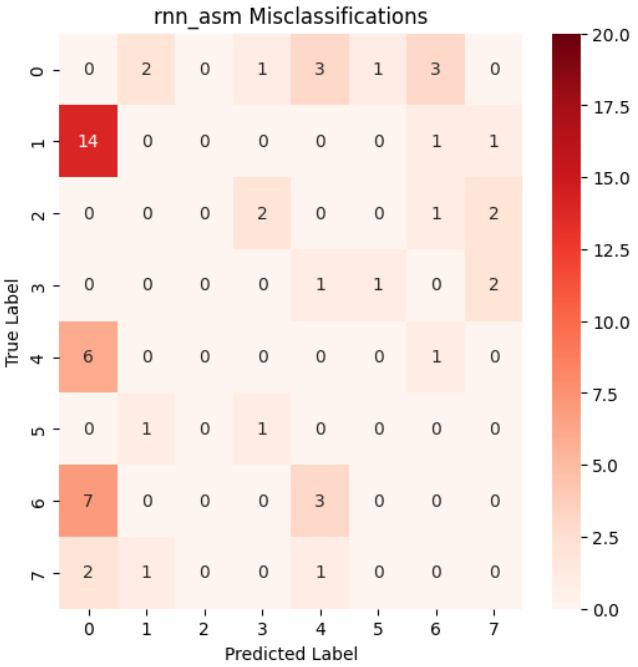
A. *Misclassification*

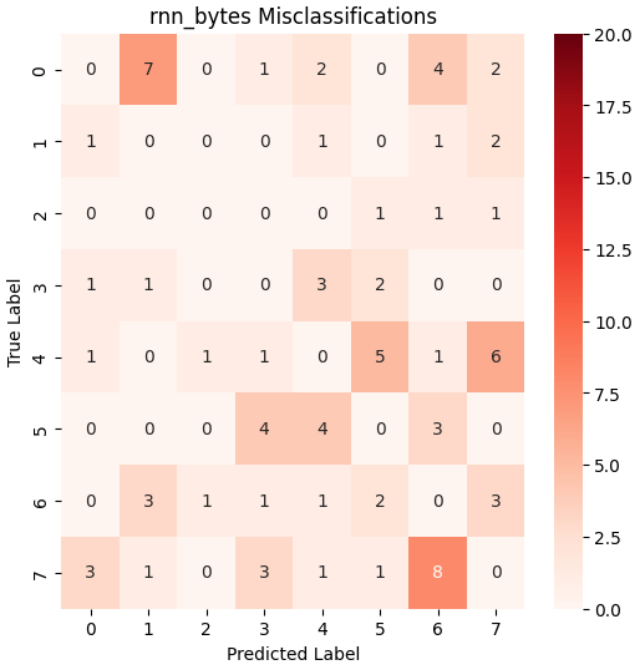


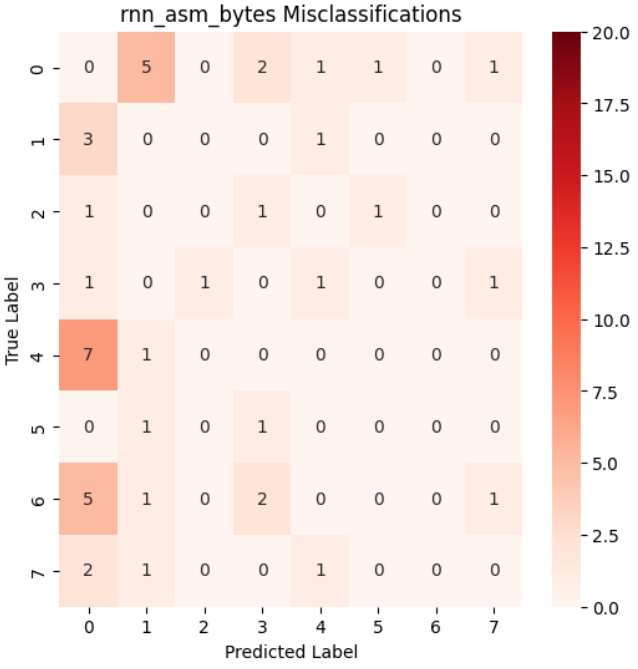


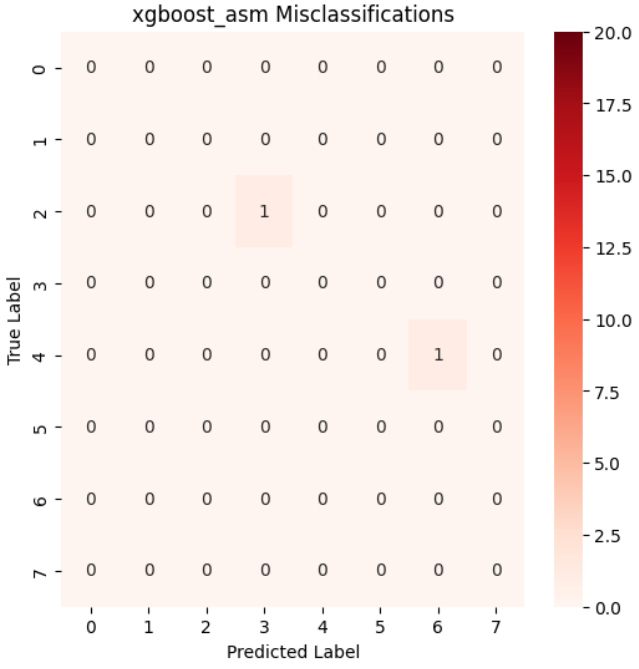


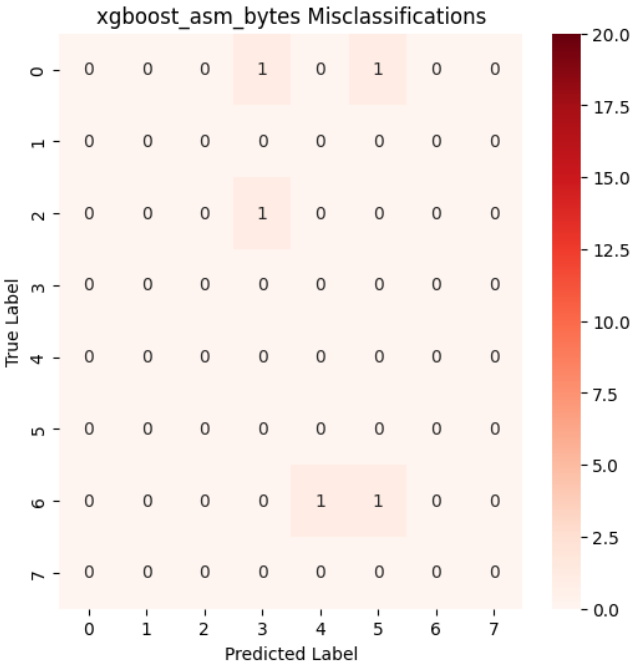


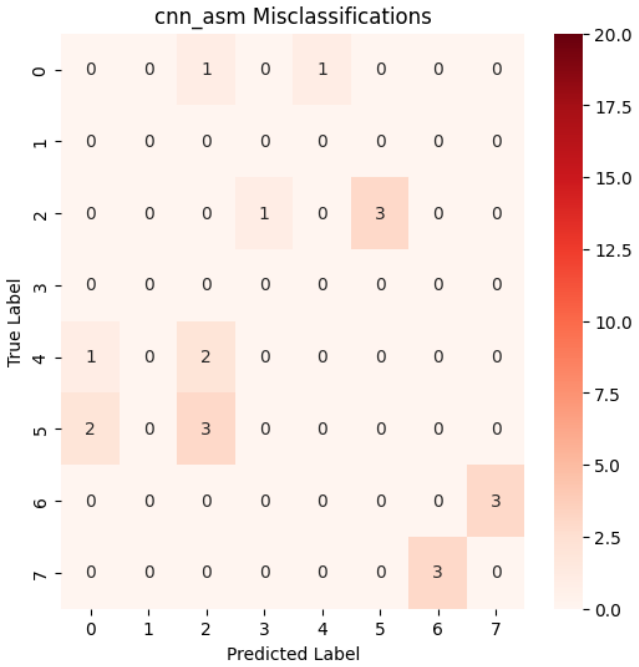

rnn_asm Misclassifications
True Label
Predicted Label
0 2 0 1 3 1 3 0
14 0 0 0 0 0 1 1
0 0 0 2 0 0 1 2
0 0 0 0 1 1 0 2
6 0 0 0 0 0 1 0
0 1 0 1 0 0 0 0
7 0 0 0 3 0 0 0
2 1 0 0 1 0 0 0
0 1 2 3 4 5 6 7
20.0 17.5 15.0 12.5 10.0 7.5 5.0 2.5 0.0

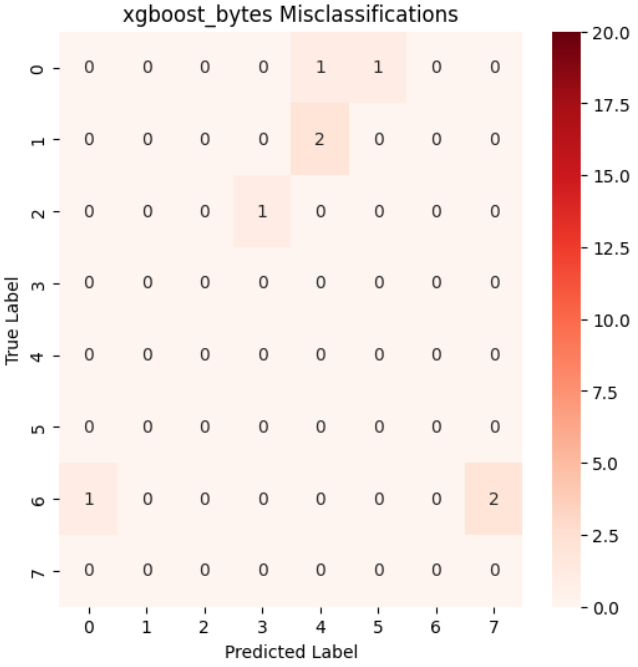

rnn_bytes Misclassifications
True Label
Predicted Label
0 7 0 1 2 0 4 2
1 0 0 0 1 0 1 2
0 0 0 0 0 1 1 1
1 1 0 0 3 2 0 0
1 0 1 1 0 5 1 6
0 0 0 4 4 0 3 0
0 3 1 1 1 2 0 3
3 1 0 3 1 1 8 0
0 1 2 3 4 5 6 7
20.0 17.5 15.0 12.5 10.0 7.5 5.0 2.5 0.0

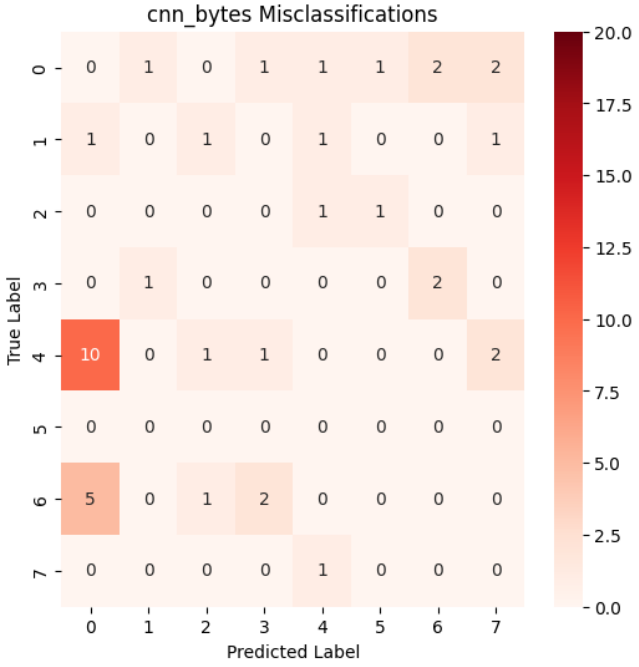

rnn_asm_bytes Misclassifications
True Label
Predicted Label
0 5 0 2 1 1 0 1
3 0 0 0 1 0 0 0
1 0 0 1 0 1 0 0
1 0 1 0 1 0 0 1
7 1 0 0 0 0 0 0
0 1 0 1 0 0 0 0
5 1 0 2 0 0 0 1
2 1 0 0 1 0 0 0
0 1 2 3 4 5 6 7
20.0 17.5 15.0 12.5 10.0 7.5 5.0 2.5 0.0

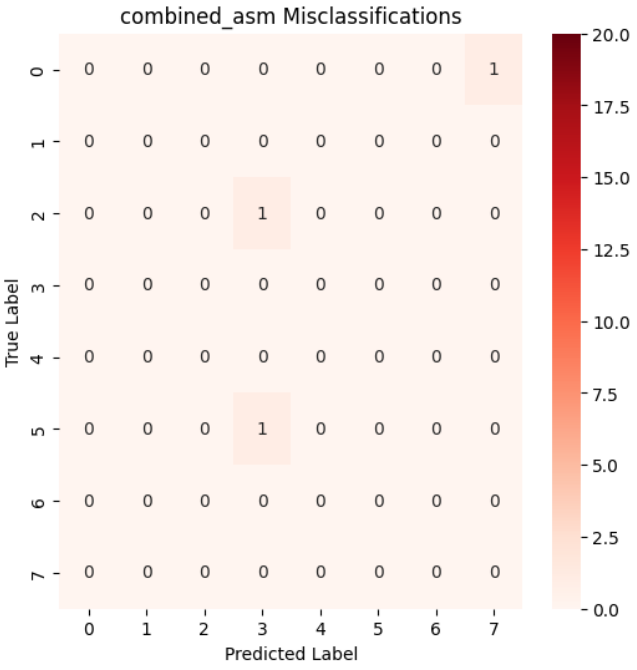

xgboost_asm Misclassifications
True Label
Predicted Label
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7
20.0 17.5 15.0 12.5 10.0 7.5 5.0 2.5 0.0

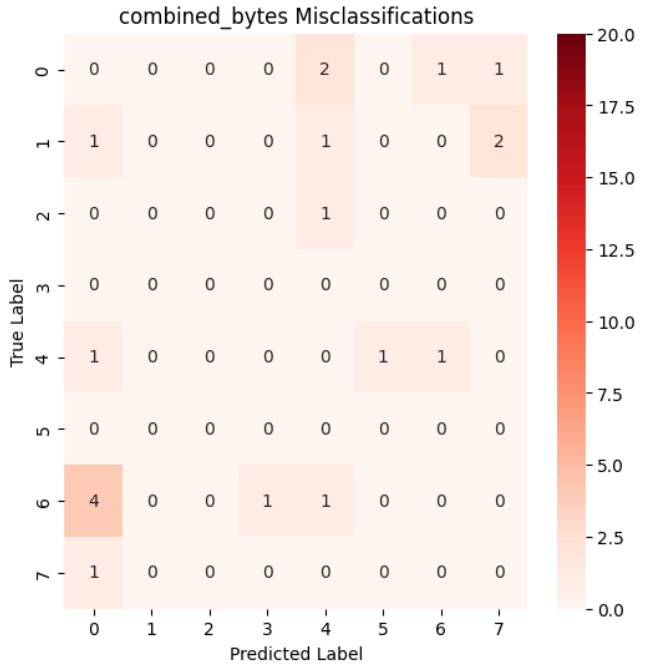

xgboost_asm_bytes Misclassifications
True Label
Predicted Label
0 0 0 1 0 1 0 0
0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7
20.0 17.5 15.0 12.5 10.0 7.5 5.0 2.5 0.0

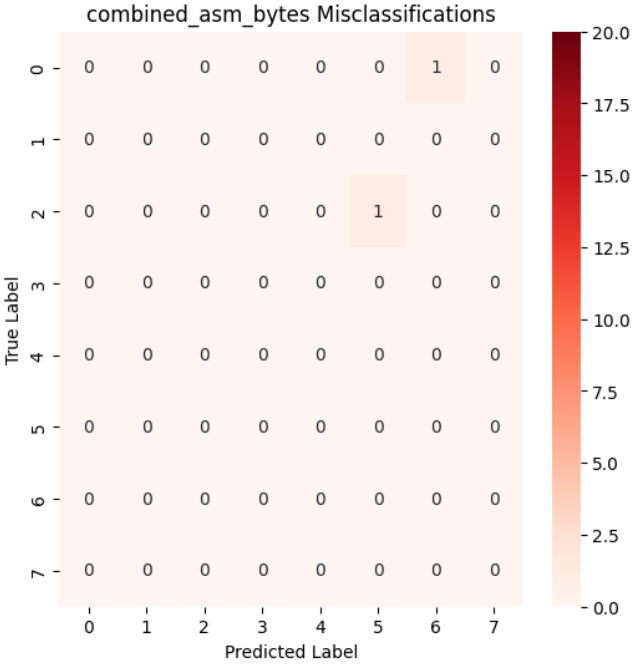

cnn_asm Misclassifications
True Label
Predicted Label
0 0 1 0 1 0 0 0
0 0 0 0 0 0 0 0
0 0 0 1 0 3 0 0
0 0 0 0 0 0 0 0
1 0 2 0 0 0 0 0
2 0 3 0 0 0 0 0
0 0 0 0 0 0 0 3
0 0 0 0 0 0 3 0
0 1 2 3 4 5 6 7
20.0 17.5 15.0 12.5 10.0 7.5 5.0 2.5 0.0


xgboost_bytes Misclassifications
True Label
Predicted Label
0 0 0 0 1 1 0 0
0 0 0 0 2 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 2
0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7
20.0 17.5 15.0 12.5 10.0 7.5 5.0 2.5 0.0


cnn_bytes Misclassifications
True Label
Predicted Label
0 1 0 1 1 1 2 2
1 0 1 0 1 0 0 1
0 0 0 0 1 1 0 0
0 1 0 0 0 0 2 0
10 0 1 1 0 0 0 2
0 0 0 0 0 0 0 0
5 0 1 2 0 0 0 0
0 0 0 0 1 0 0 0
0 1 2 3 4 5 6 7
20.0 17.5 15.0 12.5 10.0 7.5 5.0 2.5 0.0


combined_asm Misclassifications
True Label
Predicted Label
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7
20.0 17.5 15.0 12.5 10.0 7.5 5.0 2.5 0.0


combined_bytes Misclassifications
True Label
Predicted Label
0 0 0 0 2 0 1 1
1 0 0 0 1 0 0 2
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
1 0 0 0 0 1 1 0
0 0 0 0 0 0 0 0
4 0 0 1 1 0 0 0
1 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7
20.0 17.5 15.0 12.5 10.0 7.5 5.0 2.5 0.0


combined_asm_bytes Misclassifications
True Label
Predicted Label
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7
20.0 17.5 15.0 12.5 10.0 7.5 5.0 2.5 0.0