Preemptive Malware Prediction

Microsoft Malware Prediction

Matthew Kint
College of Computer Science
California State University, Sacramento
Sacramento, CA
mkint@csus.edu

ABSTRACT

The malware industry is an ever-growing, well-funded problem showing no signs of slowing down [1]. An infected host is at risk of personal data loss and hardware takeover. Companies such as Microsoft have been working to build software solutions to detect and remove malware from a host. Utilizing Microsoft's data set of nearly ten million personal computers, I set out to see if I can determine if a computer is infected, or likely to become infected, based on system information without the need to scan files. All antivirus solutions on the market scan files to determine if malware needs to be removed; If I can develop a flag to notify the software of the vulnerability I can scan less frequently, allowing the host machine to free resources.

The dataset provided is primed to be read into a dense neural network or a convolutional neural network. Utilizing both models will help us raise a flag when a host machine is nearing peak vulnerability.

My results determined an accuracy of roughly sixty-five to seventy percent that the host machine was likely to be infected soon.

1 Introduction

Microsoft manages more than one billion enterprise and consumer customers [1]. Each customer is susceptible to a plethora of malware infections with leave their machines in the hands of someone else. To provide an extra layer of protection, I will feed Microsoft's dataset of nearly ten million computers into several machine learning models—testing against whether a given machine is infected or not. This should provide insight into whether a given machine is likely to be infected soon.

1.1 Organization

2 Problem Formulation

The given inputs to my learning model can be found here [2].

Except for *MachineIdentifier, PuaMode, Census_ProcessorClass, Census_IsWIMBootEnabled, IsBeta, Census_IsFlightsDisabled, Census_IsFlightingInternal, AutoSampleOptIn, Census_ThresholdOptIn, SMode, Census_IsPortableOperatingSystem, Census_DeviceFamily, UacLuaenable, Census_IsVirtualDevice, Platform, Census_OSSkuName, Census_OSInstallLanguageIdentifier, Processor, OsBuildLab.* These columns were determined to have too similar information or too few information given the dataset. My input will be tested against *HasDetections*, which tells us whether a workstation is known to have a form of malware on their machine (either 0 for false or 1 for true).

3 System/Algorithm Design

3.1 System Architecture

Utilization of two model types: fully-connected, dense, neural network (DNN) and convolutional neural network (CNN). Both models implement a binary classification output. Each of these models will take my preprocessed data, except for CNN, which will reshape the data slightly before reading the input.

3.2 Data Preprocessing

The data provided has a total of eighty-three columns, eighty-two of which are usable as my input. As mentioned in *Problem Formulation*, the dropped columns were determined to have too many similar values to be relevant to an output without skewing results. Pandas, the python library I used to manage the data, has some built in functions which let us see unique values in a given column as well as their total counts.

Many of the columns are version numbers. One-hot encoding the version numbers seemed to make the most sense as versions are categorical. However, I determined that converting the version numbers to integers and normalizing them provided similar results with an exceptionally faster runtime, as one-hot encoding all version number columns resulted in thousands of extra columns. The rest of the columns are categorical and can be one-hot encoded.

Determining which columns are categorical came down to utilizing built-in Pandas functions. Microsoft has provided a detailed description of each category as mentioned in *Problem Formulation.* Many of the columns contained unreadable data for my models. If the values were missing, I replaced them with zero. If the values did not follow the general format of a given column, the entry was removed as an outlier.

3.3 Dense Neural Network

My DNN utilizes *ReLU* activation with an adaptive optimizer, *adam*, to speed up results through the large dataset.

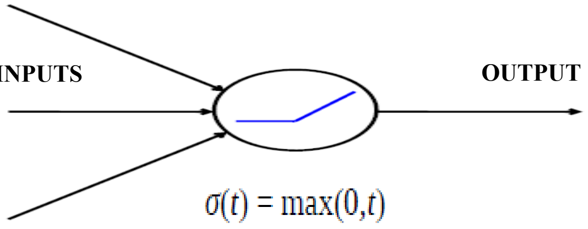
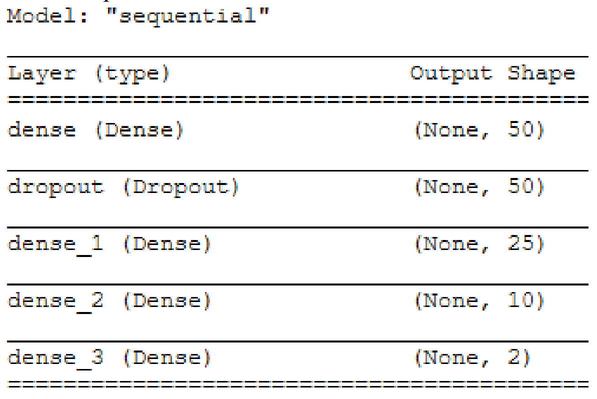


Figure 1: A given neuron with ReLU activation—accepting inputs and determining output based on formula

Each of the inputs are weighted through the *ReLU* function and sent as an output to the next set of neurons.

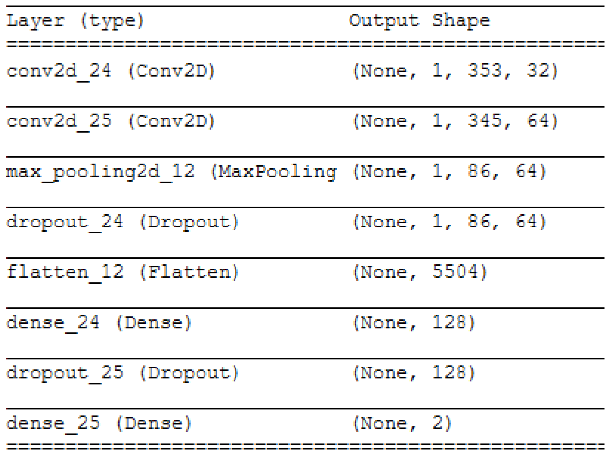


```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape
=================================================================
dense (Dense)                (None, 50)
_________________________________________________________________
dropout (Dropout)            (None, 50)
_________________________________________________________________
dense_1 (Dense)              (None, 25)
_________________________________________________________________
dense_2 (Dense)              (None, 10)
_________________________________________________________________
dense_3 (Dense)              (None, 2)
=================================================================
```

Figure 2: DNN model summary

3.4 Convolutional Neural Network

My CNN also utilizes the *ReLU* activation with the *adam* optimizer.



```
_________________________________________________________________
Layer (type)                 Output Shape
=================================================================
conv2d_24 (Conv2D)           (None, 1, 353, 32)
_________________________________________________________________
conv2d_25 (Conv2D)           (None, 1, 345, 64)
_________________________________________________________________
max_pooling2d_12 (MaxPooling (None, 1, 86, 64)
_________________________________________________________________
dropout_24 (Dropout)         (None, 1, 86, 64)
_________________________________________________________________
flatten_12 (Flatten)         (None, 5504)
_________________________________________________________________
dense_24 (Dense)             (None, 128)
_________________________________________________________________
dropout_25 (Dropout)         (None, 128)
_________________________________________________________________
dense_25 (Dense)             (None, 2)
=================================================================
```

Figure 3: CNN model summary

One major difference is the need to reshape the data in order to be red by my neurons. I are treating each row as an image, so I make my "image" the size of one by x, where x is the number of columns in each entry. CNNs require a kernel to "scan" the image and determine features. This kernel also has to be one by z, where z is the maximum kernel length; nine was chosen.

4 Experimental Evaluation

4.1 Methodology

The data supplied covers nearly ten million computers. Each row is an entry while each corresponding column containes information regarding said entry. I wanted to make the data as clean as possible. That is, eliminate any rows (computers) or columns (computer-specific information) where there were an exceptionally high number of missing values, as this would skew the results unfavorably. If a column contained a low percentage of missing values, I simply replaced those values with zero. Each of the categorical columns were then one-hot encoded for easier processing by my models. Any columns which contained a count of something, i.e. RAM size, were z-scored. I determined z-scoring columns which contained version numbers was the best course of action. I initially wanted to one-hot encode these columns, then through trial and error found it made no meaningful difference in results. Whereas z-scoring the values proved to be considerably more scalable as I increased inputs.

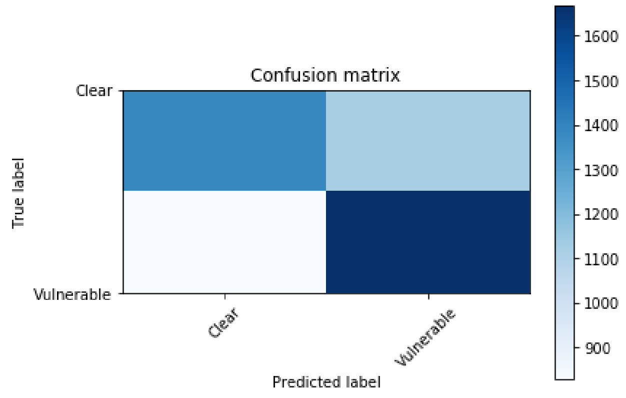
My experiments were conducted through a Jupyter notebook, allowing us to quickly view data changes and process results without re-running the entire process. Experiments were conducted using ROC curve, f1-score, recall, precision, and graphing my model loss. In addition, both my DNN and CNN models were tested against multiple activation functions and optimizers.

Training data was split up as twenty-five percent of the total dataset with a random seed.

4.2 Results

All following results utilize *ReLU* activation with the *adam* optimizer, as all others tested provided poor results.

```
[[1384 1117]
 [ 831 1668]]
Ploting confusion matrix
```



```
              precision    recall  f1-score   support

           0       0.62      0.55      0.59      2501
           1       0.60      0.67      0.63      2499

    accuracy                           0.61      5000
   macro avg       0.61      0.61      0.61      5000
weighted avg       0.61      0.61      0.61      5000
```

Figure 4: Dense Neural Network results, on a subset of 20,000 computers, utilizing *ReLU* activation with *adam* optimizer

Figure 4 shows results obtained in my DNN. Testing any other optimizer skewed results to a single category, proving worthless. Attempts to increase accuracy proved futile as exceeding sixty-five percent accuracy proved impossible given my preprocessed data.

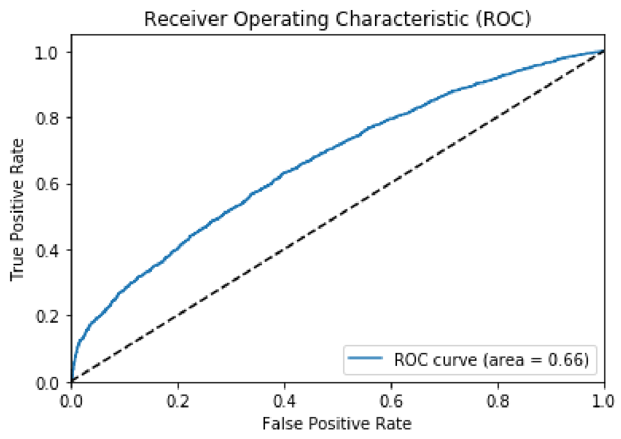
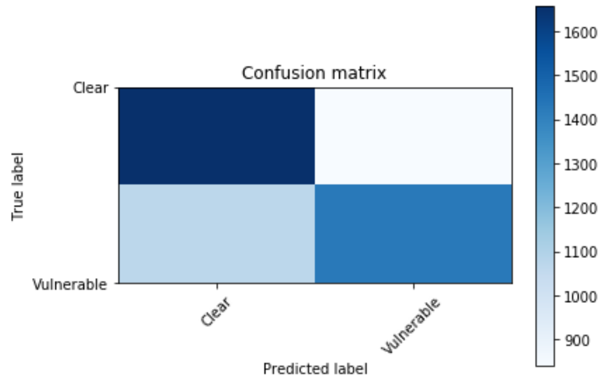


Figure 5: ROC curve for *Figure 4*

My ROC curve shows accuracy exceeding baseline. There is certainly room to improve as the area under the curve is sixty-six percent.

```
[[1659  842]
 [1070 1429]]
Ploting confusion matrix
```



```
              precision    recall  f1-score   support

           0       0.61      0.66      0.63      2501
           1       0.63      0.57      0.60      2499

    accuracy                           0.62      5000
   macro avg       0.62      0.62      0.62      5000
weighted avg       0.62      0.62      0.62      5000
```

Figure 6: Convolutional Neural Network results, on a subset of 20,000 computers, utilizing summary from *Figure 3*.

Results are like *Figure 4*. However, it appears the CNN favored false positives of “clear” computers, whereas the DNN seemed to favor false positives of “vulnerable” computers.

I applied L1 and L2 regularization to my CNN model to reduce underfitting or overfitting.

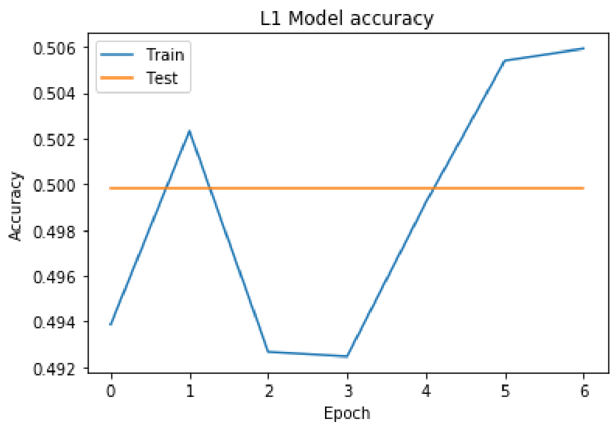


Figure 7: Model from *Figure 3* with L1 regularization applied

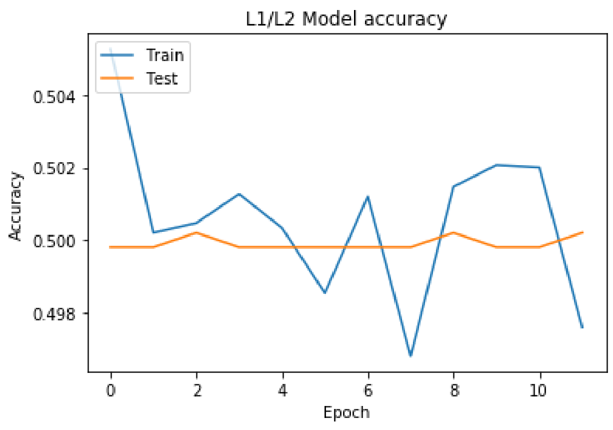


Figure 8: Model from *Figure 3* with L1/L2 regularization applied

Figure 7 and *Figure 8* show how poorly my accuracy became when applying L1 regularization as well as both L1 and L2 regularization.

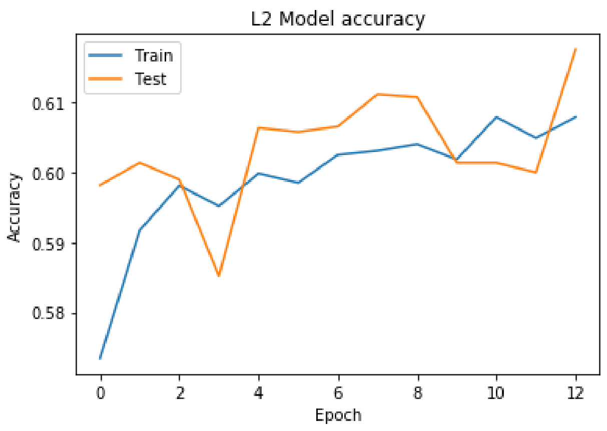


Figure 9: Model from *Figure 3* with L2 regularization applied

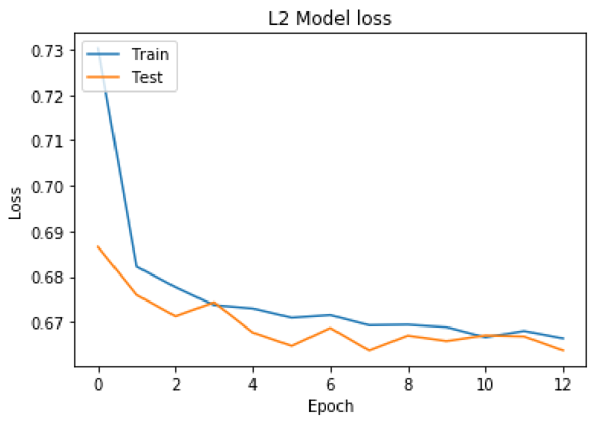


Figure 10: Model loss utilizing model from *Figure 3* with L2 regularization applied

Figure 9 and *Figure 10* show a much clearer accuracy with only L2 regularization applied. *Figure 10*, specifically, shows us that the model is neither overfitting or underfitting.

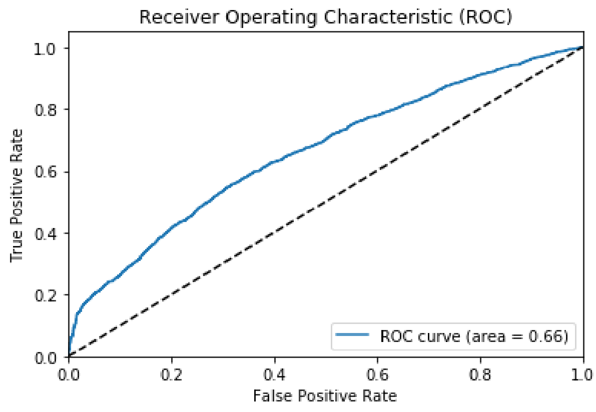


Figure 11: ROC curve utilizing model from *Figure 3* with L2 regularization applied

Like my DNN, the ROC curve in *Figure 11* shows an area of sixty-six percent.

5 Conclusion

My results proved to be responsive to both model types. However, achieving an accuracy higher than sixty-five percent was incredibly challenging and ultimately unobtainable. Perhaps utilizing a different form of data preprocessing I would be able to obtain an ideal accuracy. My time spent preprocessing, however, was the focus of the project and not without trial.

8 Learning Experience

I initially thought preprocessing would be the easiest part of the project. However, that proved to be wrong. Preprocessing determined whether my model would give us desired accuracy or even be processed at all by my model. Model tweaking was relatively simple as the changes in node count did not seem to change the output at all. Activation and optimizer, however, proved to have the largest effect on my models. Perhaps with more time I could preprocess the data in a way that would increase my accuracy.

REFERENCES

[1] https://www.kaggle.com/c/microsoft-malware-prediction/overview
[2] https://www.kaggle.com/c/microsoft-malware-prediction/data