Malware Detection with Neural Networks

Anthony Clavette, Youmeng Hin

Abstract:

In the vast world of cyber-attacks, one of the most relevant issues for the average computer-user is detecting the presence of malware. Technological advancement of malicious software has caused these attacks to become extremely stealthy and typically go unnoticed until damage is done. Pricey third-party services are marketed as the only solution to your malware detection problems, but this paper will outline why that may not be true. This problem is significant because the dangers of malware are infinite and ever-growing, but the safest solution to detecting threats on a personal PC may actually be within the system itself. Using a combination of machine learning algorithms and deep learning algorithms, we found that a system's memory dump at any given time can be used to determine the presence of malware with high confidence. These findings have limited implications because the dataset was retrieved from a formal source intended for machine learning.

Introduction:

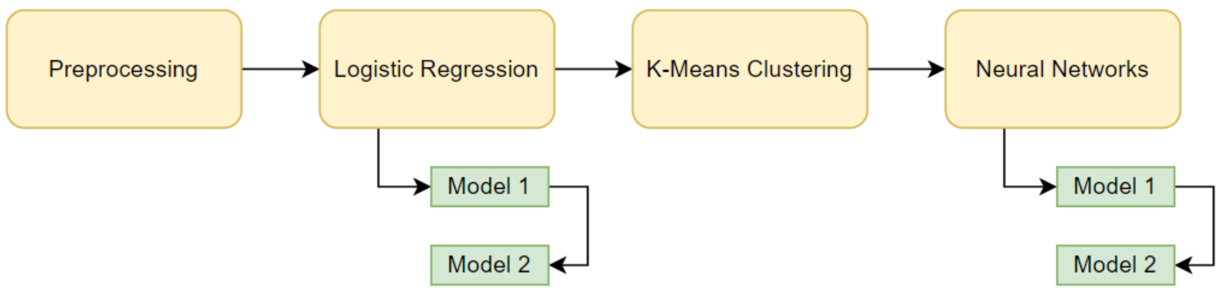
The average user of a personal computer stands no chance defending against a prepared cyberattack. A common approach to detection, often marketed as the only solution, is to install overpriced software which scans a system's files for malware. In an attempt to mitigate the clutter of add-on security, we want to show that there are built-in solutions to this problem within a system. In fact, an arbitrary memory dump file contains enough information to suggest that the system is infected, and analyzing that one file is a much simpler process than the typical method of examining each file in memory. The authors of this dataset proposed a malware detection model using Naive Bayes, Decision Tree, Random Forest, and Logistic Regression, but we wanted to explore any outcomes that may be different or improved as a result of analyzing the data with neural networking models.

Background:

Researchers from the Canadian Institute for Cybersecurity and Johns Hopkins constructed this dataset with optimal features for machine learning and proposed their most successful model. They used a stacked ensemble learner to analyze the features of the data, and their proposed combination included Naive Bayes, Decision Tree, and Random Forest as base-learners, and Logistic Regression as the meta-learner. The accuracy of this model came out to 99 percent.

Methodology:

Below is a flowchart that outlines the significant methodology of this project.



Preprocessing the data consisted of converting categorical features to numerical and transforming the data to array format for compatibility with the subsequent algorithms. The data was then used to make two logistic regression models, one which used all 55 features, and another that used 27. Next, a K-Means clustering algorithm was used on the same data. Finally, we made two neural network models, one using 55 features, and another using only 5 features.

Data:

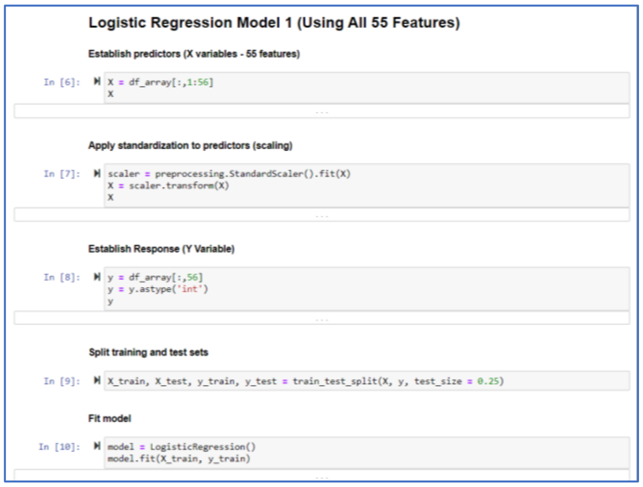
The dataset is a compilation of over 58,000 records containing features from memory dumps, half of which are from systems infected with obfuscated malware. This data was obtained by taking thousands of memory dump snapshots on a Windows 10 virtual machine. To ensure variety in the dataset, 2,916 different malware samples were used in the VM environments, all of which were Trojan Horse, Spyware, or Ransomware executables. Next, the feature extractor VolMemLyzer was used for the final feature extraction from the memory dumps before compiling the data in a CSV file. Since this data was already prepared for machine learning, our only additional data manipulation consisted of converting the 'Benign/Malware' column to numerical form and converting the data to an array for algorithm compatibility.

Experiments:

This experiment was hosted in an Anaconda Python environment, using Jupyter's notebook interface for execution. Various packages were used in this environment for application of algorithms: NumPy and pandas were used for preprocessing, scikit-learn for K-Means clustering and logistic regression, and TensorFlow for neural networks. For the final implementation of each of these algorithms, 75% of the dataset was used for training and 25% for testing. We tried several different training/test splits ranging from 75/25 to 25/75 and they all had extremely similar outcomes. Screenshots from the notebook interface are included below to show how the algorithms were applied.

i. Logistic Regression

Both models were created the same way, with the only difference being the number of predictors initially declared in the array of data. Methods from scikit-learn's 'preprocessing' module were used to scale the predictors in the data, and the LogisticRegression classifier was used to make the model.



Logistic Regression Model 1 (Using All 55 Features)

Establish predictors (X variables - 55 features)

```
In [6]:  X = df_array[:,1:56]
         X
```

Apply standardization to predictors (scaling)

```
In [7]:  scaler = preprocessing.StandardScaler().fit(X)
         X = scaler.transform(X)
         X
```

Establish Response (Y Variable)

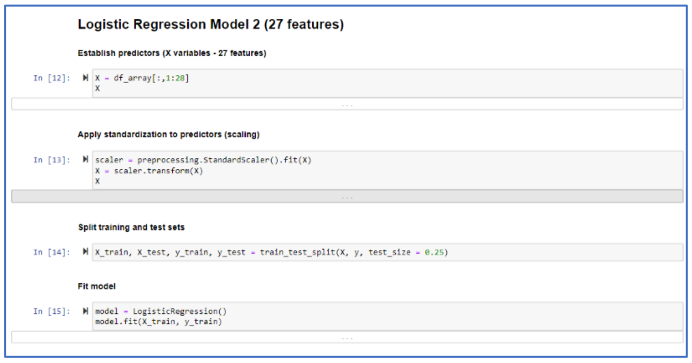
```
In [8]:  y = df_array[:,56]
         y = y.astype('int')
         y
```

Split training and test sets

```
In [9]:  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)
```

Fit model

```
In [10]: model = LogisticRegression()
         model.fit(X_train, y_train)
```



Logistic Regression Model 2 (27 features)

Establish predictors (X variables - 27 features)

```
In [12]:  X = df_array[:,1:28]
          X
```

Apply standardization to predictors (scaling)

```
In [13]:  scaler = preprocessing.StandardScaler().fit(X)
          X = scaler.transform(X)
          X
```

Split training and test sets

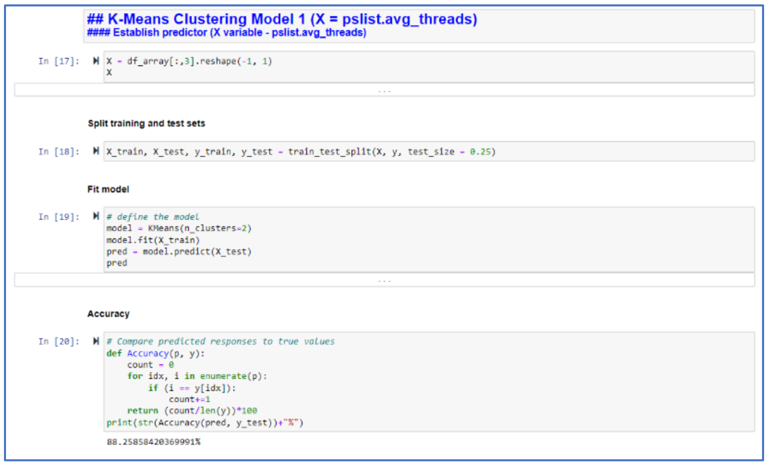
```
In [14]:  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)
```

Fit model

```
In [15]:  model = LogisticRegression()
          model.fit(X_train, y_train)
```

ii. K-Means Clustering

Scikit-learn's 'KMeans' class contains the methods used to run the clustering algorithm.



K-Means Clustering Model 1 (X = pslist.avg_threads)
Establish predictor (X variable - pslist.avg_threads)

```
In [17]:  X = df_array[:,3].reshape(-1, 1)
          X
```

Split training and test sets

```
In [18]:  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)
```

Fit model

```
In [19]:  # define the model
          model = KMeans(n_clusters=2)
          model.fit(X_train)
          pred = model.predict(X_test)
          pred
```

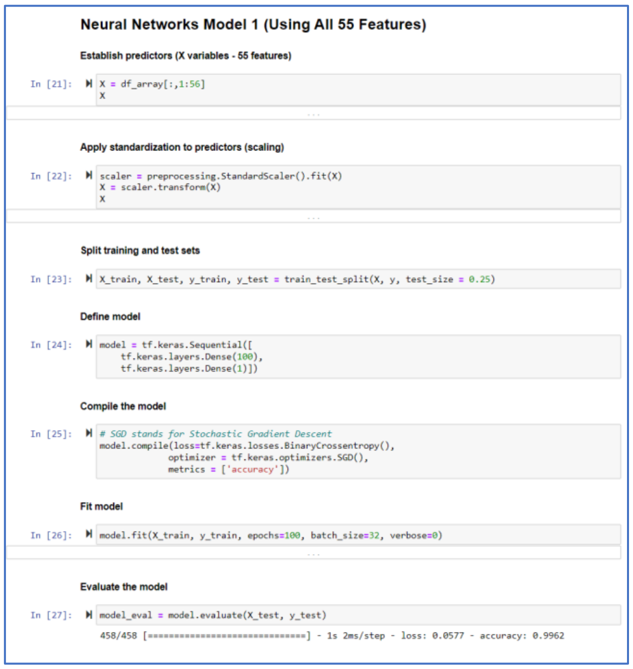
Accuracy

```
In [20]:  # Compare predicted responses to true values
          def Accuracy(p, y):
              count = 0
              for idx, i in enumerate(p):
                  if (i == y[idx]):
                      count+=1
              return (count/len(y))*100
          print(str(Accuracy(pred, y_test))+"%")

          88.25858420369991%
```

iii. Neural Network

To make and evaluate the neural network models, scikit-learn's data scaling tools were used again to prepare the data. The models were then built using tools contained in the TensorFlow package.



Neural Networks Model 1 (Using All 55 Features)

Establish predictors (X variables - 55 features)

```
In [21]:  X = df_array[:,1:56]
          X
```

Apply standardization to predictors (scaling)

```
In [22]:  scaler = preprocessing.StandardScaler().fit(X)
          X = scaler.transform(X)
          X
```

Split training and test sets

```
In [23]:  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)
```

Define model

```
In [24]:  model = tf.keras.Sequential([
              tf.keras.layers.Dense(100),
              tf.keras.layers.Dense(1)])
```

Compile the model

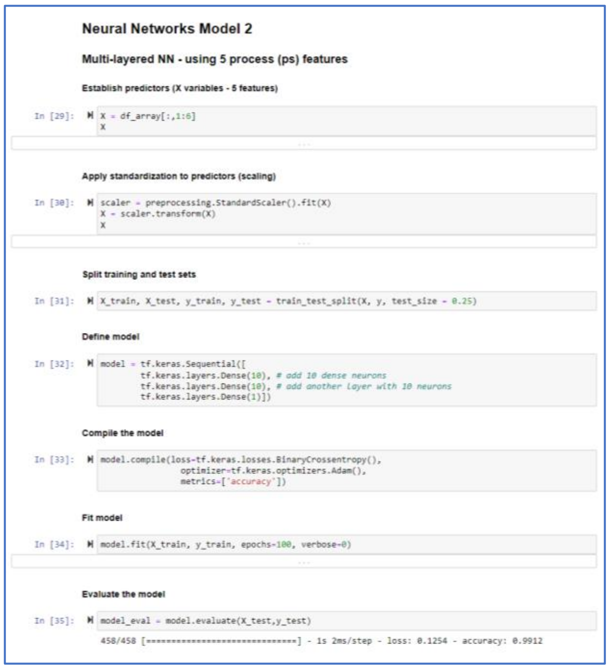
```
In [25]:  # SGD stands for Stochastic Gradient Descent
          model.compile(loss=tf.keras.losses.BinaryCrossentropy(),
                        optimizer = tf.keras.optimizers.SGD(),
                        metrics = ['accuracy'])
```

Fit model

```
In [26]:  model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=0)
```

Evaluate the model

```
In [27]:  model_eval = model.evaluate(X_test, y_test)
          458/458 [==============================] - 1s 2ms/step - loss: 0.0577 - accuracy: 0.9962
```



Neural Networks Model 2

Multi-layered NN - using 5 process (ps) features

Establish predictors (X variables - 5 features)

```
In [29]:  X = df_array[:,1:6]
          X
```

Apply standardization to predictors (scaling)

```
In [30]:  scaler = preprocessing.StandardScaler().fit(X)
          X = scaler.transform(X)
          X
```

Split training and test sets

```
In [31]:  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)
```

Define model

```
In [32]:  model = tf.keras.Sequential([
                  tf.keras.layers.Dense(10), # add 10 dense neurons
                  tf.keras.layers.Dense(10), # add another layer with 10 neurons
                  tf.keras.layers.Dense(1)])
```

Compile the model

```
In [33]:  model.compile(loss=tf.keras.losses.BinaryCrossentropy(),
                        optimizer=tf.keras.optimizers.Adam(),
                        metrics=['accuracy'])
```

Fit model

```
In [34]:  model.fit(X_train, y_train, epochs=100, verbose=0)
```

Evaluate the model

```
In [35]:  model_eval = model.evaluate(X_test,y_test)
          458/458 [==============================] - 1s 2ms/step - loss: 0.1254 - accuracy: 0.9912
```

Results:

Logistic Regression

i. The first logistic regression model performed well given the test data, yielding 99.9 percent accuracy. However, it does require all 55 of the features making it very computationally expensive; we can definitely sacrifice a bit of accuracy for a more optimized model.

ii. The second logistic regression model used only half the number of features needed for the previous model, and with only 27 features, the accuracy was 99.6 percent; only 0.3 percent of accuracy was lost. Not only does it reduce the computational power and storage required to run this model, but it also requires half the amount of data in order to identify malware.

K-Means Clustering

i. Surprisingly, the K-means clustering algorithm performed well for being unsupervised learning, returning an accuracy of 88.3 percent. However, one problem we encountered was not knowing when the model assigns 0 for Benign and 1 for Malware. We resolved this issue by setting a seed to gain consistent results when the model classifies Benign as 0 and Malware as 1.

Neural Networks

i. The first neural network model used all 55 features of the data. The model has 2 layers, 101 total neurons, and resulted in around 98.7 percent accuracy. However, using 101 neurons and 55 features requires a lot of computational power, and the accuracy can be improved.

ii. Using only 5 features, the second neural network model had 3 layers, 21 total neurons, and yielded an accuracy of 99.1 percent. This model excluded over 90 percent of the data's features and used only 20 percent of the previous total amount of neurons. Although, we used an extra layer and a different optimizer for the model, which was enough to make it more accurate than the first model at a much lower computation cost.

With a variety of models all performing at high accuracy scores, we are able to conclude that it is possible to cost-effectively detect malware on a system using features from a simple memory dump.

Conclusion:

This project shows a fascinating method of detecting malware at extremely high rates of accuracy. The research and experiments involved in this project cover the topic of detection, one of the essential elements of cybersecurity frameworks. As malware is an expanding threat every day, this project promotes alternative ways to perform threat detection on a system because add-on security is typically financially and computationally expensive. This project has many challenges and directions in which it could be advanced. The most beneficial advancement would be to continue optimizing for the best neural network model and solidifying a standard process for using this model to detect malware on any machine.

References

Tristan et al., 2022. Detecting Obfuscated Malware using Memory Feature Engineering. SCITEPRESS – Science and Technology Publications.