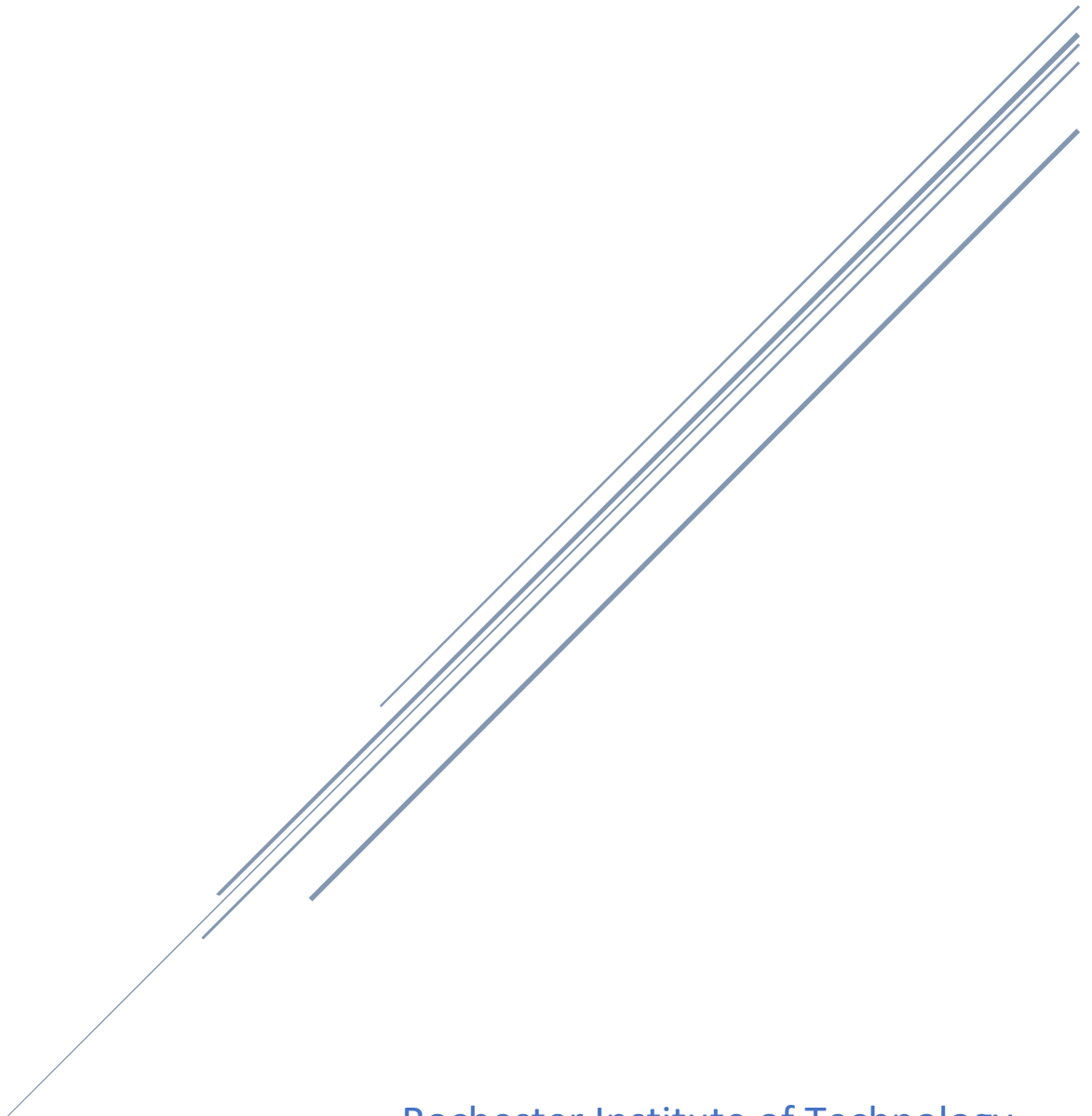


# Windows Infection Detector Model (WID)

By: Jack Hyland



Rochester Institute of Technology  
CSEC 759

## Table of Contents

Introduction .....	2
Features .....	3
Installation .....	5
Usage.....	5
Results.....	6
Model Feature Importance .....	6
Hyper-Parameter Tuning .....	7
Accuracy Vs Tree Depth .....	7
Accuracy Vs Number of Trees.....	8
Conclusion.....	8

# Introduction

I wanted to create a tool that was able to detect if a Windows machine was infected by a piece of malware using telemetry triaged from the inspected machine. The dataset I choose to implement was a Kaggle challenge that is composed of around 80 features which gives information about a specific Windows machine such as: Is anti-virus installed? Was anti-virus running? What version of Windows is running? Labels are also provided within the data which indicates if each host was compromised or not. This model is made to intake gathered information from a Windows host and give you a quick estimate of whether there is malicious activity. I will be making it open source as it could assist system administrators or other researchers within the field.

This tool detects malicious activity in general instead of one specific piece of malware. I think it is important to note my previous attempts at completing this project as they ultimately led me to my final decision of choosing a random forest. I first started with the dataset from the "Microsoft Malware Classification Challenge (BIG 2015)" on Kaggle. My original idea was to create a convolutional neural network that would be able to classify malicious binaries based on strings found within the program. Some of the logistic problems I faced included being able to decompress the half-terabyte dataset and impractical training times. These issues, as well as the fact that I have worked extensively with CNN's through previous research, led me to shift directions. I wanted to work with something that I previously had not worked with before, so I decided to use the Kaggle dataset, "Microsoft Malware Prediction" and attempt to classify if each machine was infected with malware based on the 80+ features given on each machine.

From my previous experience working in a security operation center, I know that when a machine gets infected with malware and is detected by EDR tech, or other security-monitoring tools, an image or triage package is created for that machine so that analysts can investigate how the malware got onto the machine and what it affected. The dataset I am using resembles information that could easily be added to one of these triage packages. The training dataset includes 83 features while the testing dataset contains 82. This is because the training set includes labels while the testing set does not. This was part of the challenge on Kaggle and how you would be rated against other participants. Once the training set is decompressed it is around ~4.2GB in size while the testing dataset is around ~3.7GB in size. I ended up deciding not to use the testing dataset since it did not have labels thus not giving me any way to train or test my model. The training dataset was large enough that I felt comfortable turning it into an 80%/20% split training/testing respectively.

# Features


As previously stated, there are over 80 features associated with this dataset. Detailing what each one is and what it means would take up a lot of space, so instead, I detailed a few that I felt were the most relevant to my model.

- **ProductName** - Defender state information e.g. win8defender
- **EngineVersion** - Defender state information e.g. 1.1.12603.0
- **AppVersion** - Defender state information e.g. 4.9.10586.0
- **AvSigVersion** - Defender state information e.g. 1.217.1014.0
- **AVProductStatesIdentifier** - ID for the specific configuration of a user's antivirus software
- **HasTpm** - True if machine has tpm
- **CountryIdentifier** - ID for the country the machine is located in
- **GeoNameIdentifier** - ID for the geographic region a machine is located in
- **LocaleEnglishNameIdentifier** - English name of Locale ID of the current user
- **OsVer** - Version of the current operating system
- **OsBuild** - Build of the current operating system
- **OsSuite** - Product suite mask for the current operating system.
- **OsPlatformSubRelease** - Returns the OS Platform sub-release (Windows Vista, Windows 7, Windows 8, TH1, TH2)
- **OsBuildLab** - Build lab that generated the current OS. Example: 9600.17630.amd64fre.winblue\_r7.150109-2022
- **SkuEdition** - The goal of this feature is to use the Product Type defined in the MSDN to map to a 'SKU-Edition' name that is useful in population reporting. The valid Product Type are defined in %sdxroot%\data\windowseditions.xml. This API has been used since Vista and Server 2008, so there are many Product Types that do not apply to Windows 10. The 'SKU-Edition' is a string value that is in one of three classes of results. The design must hand each class.
- **IsProtected** - This is a calculated field derived from the Spynet Report's AV Products field. Returns: a. TRUE if there is at least one active and up-to-date antivirus product running on this machine. b. FALSE if there is no active AV product on this machine, or if the AV is active, but is not receiving the latest updates. c. null if there are no Anti Virus Products in the report. Returns: Whether a machine is protected.
- **SMode** - This field is set to true when the device is known to be in 'S Mode', as in, Windows 10 S mode, where only Microsoft Store apps can be installed
- **SmartScreen** - This is the SmartScreen enabled string value from registry. This is obtained by checking in order, HKLM\SOFTWARE\Policies\Microsoft\Windows\System\SmartScreenEnabled and HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\SmartScreenEnabled. If the value exists but is blank, the value "ExistsNotSet" is sent in telemetry.
- **Firewall** - This attribute is true (1) for Windows 8.1 and above if windows firewall is enabled, as reported by the service.

One of the biggest challenges I faced was normalizing the data since many of the features contained strings instead of numbers. Random forests can only intake data that are all quantitative. I went through many iterations, most notably were 3 big changes. At first, I wanted to one-hot-encode each feature which contained strings. This worked well on the small sample I was using for quick testing. When trying to scale it to the full dataset at around a few hundred thousand samples the computer would run out of memory, this was because the one-hot-encoding turned my 82 features into ~3,000. This was not going to work with my full dataset which had around 9,000,000 samples. My next thought was to encode the strings into ASCII, this way they would be unique but also be numbers instead of letters. Quickly I found out a few of the features contained values with very long strings which when converted to one large ASCII number could not be held in memory.

I struggled with this problem for a while and thought that the only solution would be to delete those columns that contained strings. Losing this data could however be detrimental to the overall performance of my model. Eventually, I came up with a solution, I decided to iterate over the features which contained strings and store each value in the column correlated to a unique ID.

	SkuEdition
0	Pro
1	Pro
2	Home
3	Pro
4	Home
5	Pro
6	Home
7	Home
8	Pro
9	Home
10	Pro
11	Home
12	Pro
13	Pro
14	Home
15	Pro
16	Invalid



	SkuEdition
0	1
1	1
2	2
3	1
4	2
5	1
6	2
7	2
8	1
9	2
10	1
11	2
12	1
13	1
14	2
15	1
16	3

This was by far the best solution since it not only allowed me to easily use categorical data but it had the added benefit of reducing the size of my data making it easier to load and train on and keeping the original size of 82 features so the computer wouldn't run out of memory. Turning my original 4384966482 bytes → 4324181165 bytes. This was not a big downsize but did save some time loading in the data.

The overall design of my model was a random forest with the following hyper-parameters:

### **RandomForestClassifier**

```
(bootstrap=True, class_weight=None, criterion='gini', max_depth=10, max_features='auto',  
max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None,  
min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, n_estimators=101,  
n_jobs=None, oob_score=False, random_state=1, verbose=0, warm_start=False)
```

## **Installation**

I tried to make my tool as easy as possible to install using anaconda environments since I have been frustrated in the past trying to reproduce other researchers' code. I created a file called `environment.yml` which you can easily use to install an environment.

First, you need to install anaconda if you have not already:

```
https://www.anaconda.com/products/individual
```

Next just simply open a terminal in the directory where `environment.yml` is stored and type the following command:

```
conda env create -f environment.yml
```

This should start the installation process which might take a while.

Once complete you can check to make sure everything installed correctly by typing the command

```
conda env list
```

You should see an environment called `WID-Model` To use it type `conda activate WID-Model`

## **Usage**

To use the Windows Infection Detector is simple. It is composed of two python files, you should first open up the `WIDModel.py` file and scroll down to line 107 you should find a variable named `DATA` set this equal to the full path to your `data.csv` file I provided. That is the only change you will have to make, after that you should be able to run it by typing the command

```
python3 WIDModel.py
```

# Results

Overall In-Sample Accuracy: 63.144%

Overall Test Accuracy: 63.101%

Number of samples in X\_train: 7,137,186

Number of samples in y\_train: 7,137,186

Number of samples in X\_test: 1,784,297

Number of samples in y\_test: 1,784,297

False Positive Rate: 41.66%

True Positive Rate: 67.856%

The results are interesting since with a smaller sample size (1,000,000), the in-sample accuracy I can expect to see almost perfect but out of sample I average around ~65% accuracy. In this case when run against the entire data-set both the in-sample and test accuracies are very close at ~63%. I believe this to be due to my model overfitting to the data provided, this will be further investigated in the hyper-parameter tuning section. Below you can see that I extracted all the features and highlighted any feature that was rated as 5% important or higher.

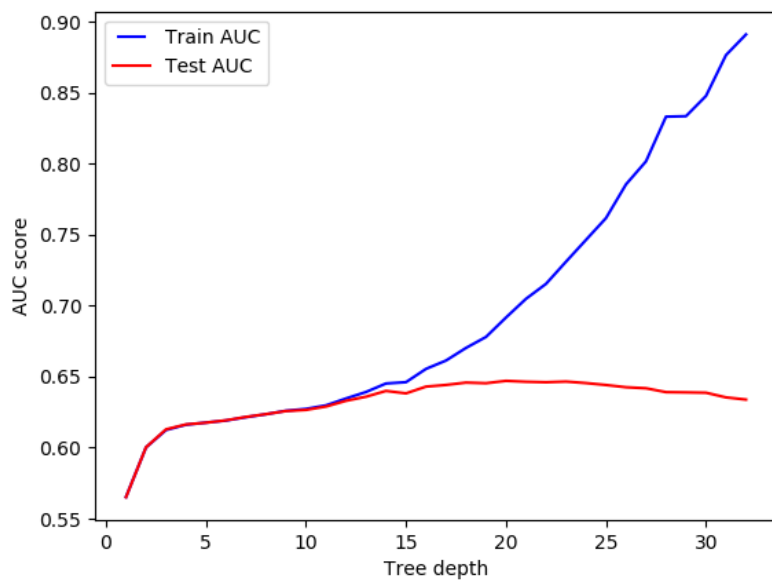
## Model Feature Importance

• ProductName	0.0006534879
• EngineVersion	0.0643321588
• AppVersion	0.0240642130
• AvSigVersion	0.0220686729
• RtpStateBitfield	0.0069220111
• IsSxsPassiveMode	0.0027218111
• DefaultBrowsersIdentifier	0.0032820957
• AVProductStatesIdentifier	0.1297997201
• AVProductsInstalled	0.1209559810
• AVProductsEnabled	0.0052399760
• GeoNameIdentifier	0.0013072552
• LocaleEnglishNameIdentifier	0.0035587468
• Platform	0.0011791984
• Processor	0.0087267379
• OsVer	0.0008865653
• OsBuild	0.0040384877

• IsProtected	0.0111832960
• SMode	0.0001507850
• IeVerIdentifier	0.0048431222
• SmartScreen	0.3449560388
• Census_ProcessorModelIdentifier	0.0042552860
• Census_PrimaryDiskTotalCapacity	0.0202832633
• Census_TotalPhysicalRAM	0.0275122182
• Census_InternalPrimaryDisplayResolutionHorizontal	0.0040613674
• Census_InternalPrimaryDisplayResolutionVertical	0.0115986986
• Census_OSVersion	0.0073070491
• Census_OSArchitecture	0.0137538725
• Census_OSInstallTypeName	0.0134397710
• Wdft_IsGamer	0.0135763751

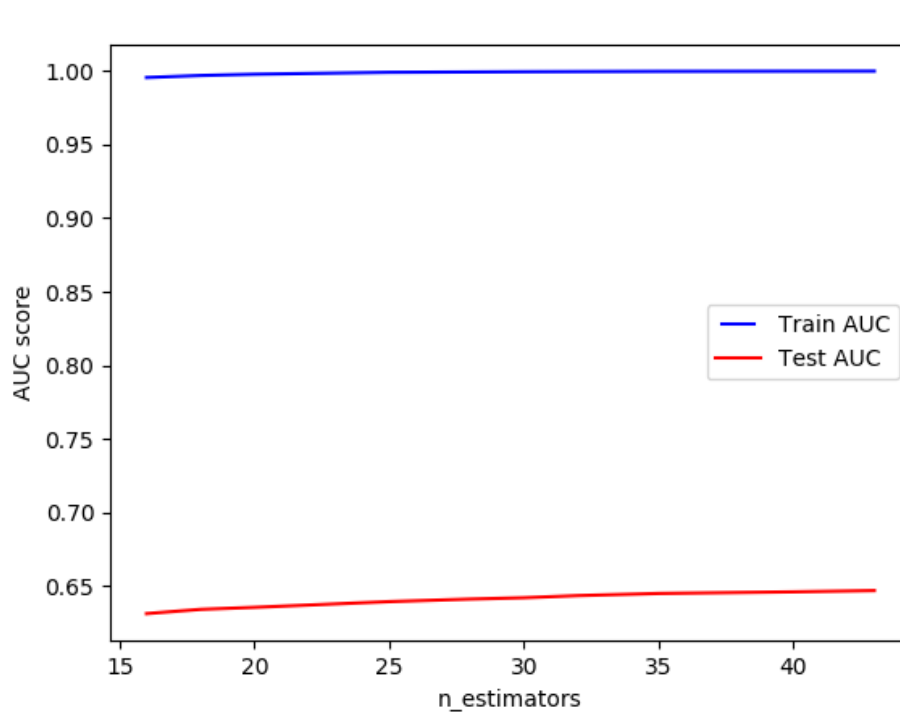
# Hyper-Parameter Tuning

Accuracy Vs Tree Depth





## Accuracy Vs Number of Trees



## Conclusion

Overall, the sample test accuracy of this model is not great (~65%) and further development and research would need to be conducted to increase it to the appropriate range. As is most evident by my results hyper-parameter tuning after a certain point when the random forest gets large enough it stops generalizing what an infected machine would look like and instead it memorizes the dataset, thus giving it the almost perfect accuracy. This can be seen at around the tree depth of 15. Also, we can see that by adding more trees to the random-forest we slightly increase the overall test performance of the model, this is a good sign that the model learns more with the more trees given, but the rate at which it learns is not ideal.