Birkbeck MSc Data Science Project

Project Report September 2022

Identifying Malicious WiFi Traffic in the Aegean Wireless Intrusion Dataset 3 using Machine Learning

Laurence Tickell

Supervisor: Professor George Roussos

Github: https://github.com/Birkbeck/msc-data-science-project-2021_22-lazmac3

Abstract

WiFi has become an essential technology in the 21st century for internet users and an important surface for attackers. Attackers can attempt low level attacks on the WiFi protocol itself or application level attacks on networks using WiFi.

Identifying an attack using machine learning would be valuable as it would allow the identification of unseen attacks that are similar but not the same as previous attack frames to be identified.

This project looked at identifying attack frames for the 13 attacks in the Aegean Wireless Intrusion dataset. Using Decision Trees, it was possible to classify the dataset extremely well. A model trained on the whole dataset using only 10 of the 254 dimensions produced had a F1 score of 98.4%. The Random Forest used was very good at classifying the low level WiFi protocol base attacks and the SSDP attack. However the SSDP attack had a very large number of attack frames and is easy to identify which padded the results. The classification of the application level attacks required MAC and IP addresses of the researchers which would be unknown in a real attack. Without these MAC and IP addresses the results were very poor in most cases.

Contents

1. Introduction

a. WiFi and its vulnerabilities

Since its release in 1997 (ieeexplore.ieee.org) WiFI has become a key network transmission technology. While not as secure as Ethernet it is often preferred for both home and office use due to its lower set up cost and convenience. WiFi also allows mobile devices to stay online when mobile data is not available and is cheaper than cellular networks. As mobile computing becomes a more essential component of all our lives, WiFi's importance will continue to grow(Stallings, 2015).

WiFi's ubiquity and certain weaknesses make it a natural target for attackers. Most real world Intrusion Detection Systems (IDS) use signature based methods to identify attacks, where a new attack is matched to a previous known attack in a database. This is an example of misuse detection (Salmon, 2017). Using Machine learning is an example of anomaly detection, where abnormal behaviour is flagged. A good system would be able to identify attacks that are similar but not the same as those the algorithm has been trained from (Wang, 2004)

I used the skills I have learned on this Masters to try and classify attack packets in the Aegean Wireless Intrusion Dataset 3 (AWID3) (icsdweb.aegean.gr/awid/awid3). This is a collection of 13 simulated attacks which were sniffed and collated in 2021 by Chatzoglou,Kambourakis and Kolias (Chatzoglou, 2021).

b. Project Objectives

For my project I used machine learning algorithms to identify malicious WiFi frames in my chosen dataset. For reasons I will expand upon, all the algorithms I used were all types of decision trees.

My aim was to use binary classification where traffic is classified as suspicious or not suspicious, rather than using multiclass classification where I would be trying to identify the type of attack. This both maps to the real world scenario where a suspicious packet is flagged and a network engineer investigates further and is easier from a machine learning perspective. In a binary classification as we only have 4 boxes in our confusion Matrix:

		Predicted Condition	
		Positive	**Negative**
Actual	**Positive**	**True Positive (TP)** - An attack is correctly detected	**False Negative (FN)** - A malicious packet is incorrectly classified as normal
Condition	**Negative**	**False Positive (FP)** - A normal packet is classified as suspicious	**True Negative (TN)** - A normal packet is correctly classified as innocuous

Once I assessed the best algorithm to use I trained a model on my entire dataset which could be used for other purposes to classify suspicious traffic such as the REST API I built.

Using machine learning for Intrusion detection is a form of anomaly detection where the classes are unbalanced. Most networks are not under attack most of the time. A system that is too sensitive and flags up too many false positives can be irritating for a network administrator and may be ignored. A good algorithm should catch True positives but also not raise too many false negatives(Salmon, 2017, Kolias 2016).

The most common measures of machine learning performance can be poor choices for anomaly detection. An algorithm that classifies almost everything as Negative can get a good accuracy (TP+TN/TP+TN+FP+FN) score as it correctly classifies most frames as True Negatives. However such a system would be useless as an IDS.

c. Project Report Outline

The report consists of a description of the dataset and how it was preprocessed. I then detail the results of a grid search that identifies the most important dimensions on each attack individually. I then take this information and try various types of decision trees on a subset that contains packets from each attack to compare performance. Finally I train a model using the entire dataset and then deploy this model in Django REST API microservice.

d. Hardware and Software used

My initial plan for the project was to use Pyspark Mlib (spark.apache.org) to analyse the whole dataset doing relatively high dimension analysis using a cluster. I analysed each attack individually this way running Spark locally using a docker pyspark notebook (hub.docker.com). I did this on my gaming laptop which is an Asus with AMD Ryzen 7-4800H, Nvidia GeForce RTX 3050, 16GB RAM, 512GB SSD. The results on a per attack basis were much better than I expected and I ended up working with few dimensions.

I switched to scikit-learn for the later part of my analysis. While scikit-learn does not have the distributed capabilities of Spark it has far more libraries and is more user friendly. It has more machine learning algorithms (scikit-learn.org), and models can be saved and deployed and used in many more ways than Spark models. I did this analysis in the cloud using Google Vertex AI managed notebooks (cloud.google.com). I used the $300 free credits which have a three month expiry which is why I did my initial analysis locally. There are also limits on the number of CPUs available on the free tier. The n1-highmem-16 (16CPU 104GB RAM) was the most powerful setting available to me as the notebook has to be hosted in a single region. All my analysis was done using Jupyter notebooks so my results can be seen in the related GitHub Repository.

I built a Django REST API in a local Venv (docs.python.org), on my laptop using Ubuntu 22.04.1. I will demonstrate this working in a video.

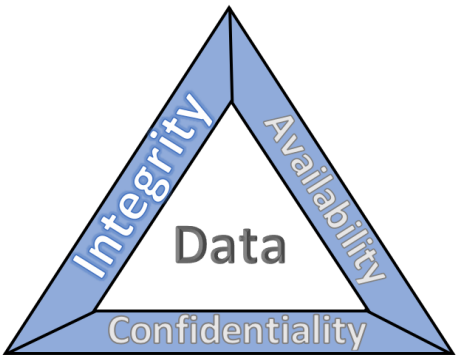
My datasets are too large to include in the Birkbeck GitHub repository, but I have them in GCP buckets and can share them on request. Alternatively they can be downloaded on request from the University of the Aegean Website.

2. Context

Before describing the AWID3 dataset I need to introduce some underlying concepts. The dataset contains 13 attacks which vary in what they try to alter and which type of WiFi frame they use to try and make these changes.

a. The CIA Triad

Information security has a model known as the CIA triad. The three concepts that make up the model are:



(nccoe.nist.gov)

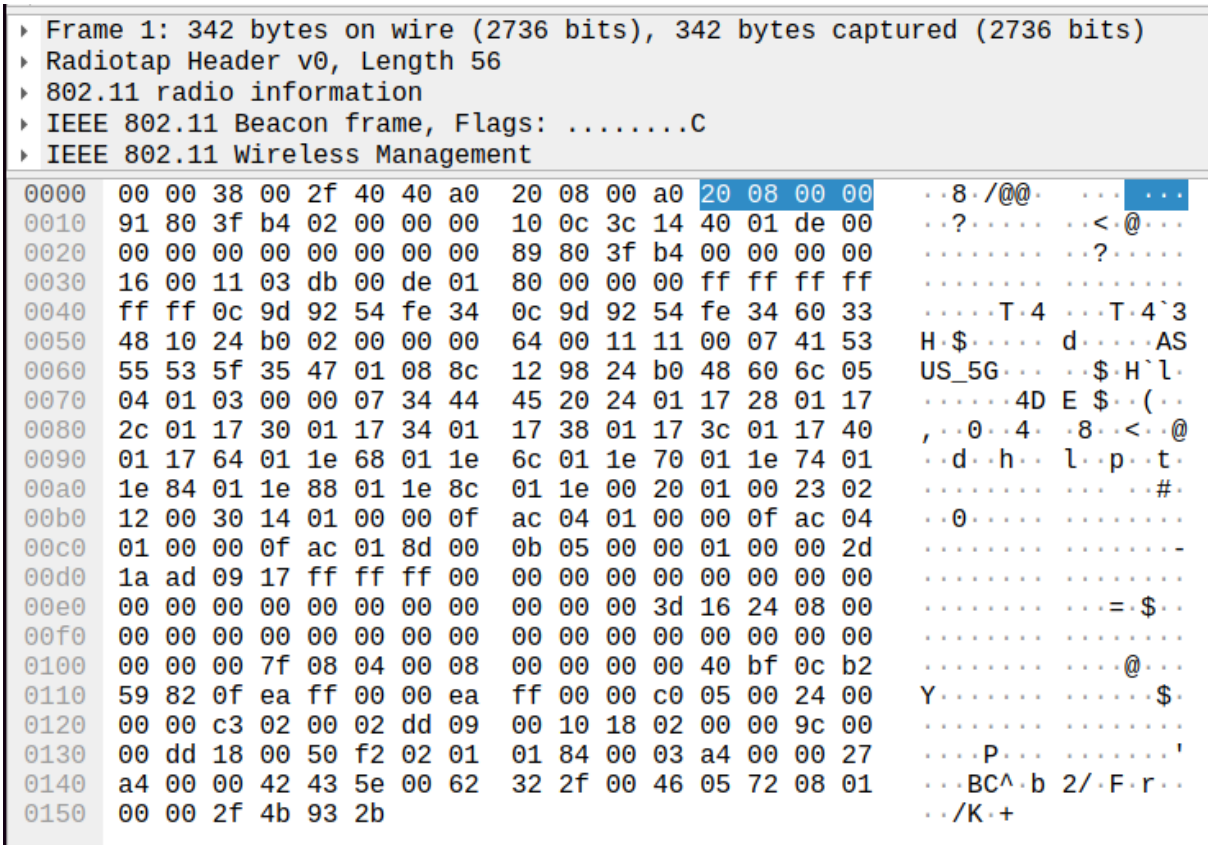
Confidentiality - data must only be visible to authorised users
Integrity - data can only be altered by an authorised user
Availability - data should be visible in a timely manner to authorised users
(Stallings, 2007)
For a system to be secure all three pillars of the triad must be maintained. Different attacks will target one or more of these pillars.

b. The nature WiFi Packets

Data sent over WiFi is sent in Frames, of which there are 3 types: Management, Control and Data. Each frame has a 30 byte header, a body which is usually encrypted and can vary in length from 0 to 2,312 bytes. Control packets do not have a body. At the end of each frame is a 4 byte Frame Check sequence (Kolias, 2016).

Attacks that try to use management and control packets are working on the lowest levels of the OSI (Open Systems Interconnection) model, affecting how information is sent. Attacks using data packets are working at higher levels and can be considered application layer attacks. The researchers highlight that attacks that target different layers in succession can be the most dangerous with modern WiFi. Older versions of WiFi were much more vulnerable to key based attacks. Upgrades to the standard have made these much harder, although there are two in the dataset Kr00k and Krack (wiki.wireshark.org, Chatzoglou, 2021).

Below is the first packet of the dataset (Deauthentication) viewed in wireshark as Hexidecimal. It is a beacon frame, which is a management frame.



3. AWID3

a. The Dataset

The AWID3 dataset is recent extension of AWID2 dataset. AWID2 (icsdweb.aegean.gr/awid/awid2) is from 2016 and is more widely studied. I chose to just use the AWID3 data set mainly as it was available in easier to use csv and pcap format. The original series are presented as less structured data files. When I started my project the researchers had not released detailed analysis of the dataset, but two papers were released over the summer (Chatzoglou, 2022, Chatzoglou,2022a)

Describing each attack in detail would use up too much of the word count but there are 13 attacks and they are grouped by the Authors in these categories (Each attack file is in **bold**):

- ***802.11 (WiFi) specific attacks*: Deauthentication**, **Disassociation** and **(Re)Association** use management frames, these can be used for Denial of Service (DOS). These flooding attacks reduce *Availability*. A Deauthentication attack is when an attacker sends a deauthentication frame, usually from a spoofed MAC address. The station will then disconnect and try to reconnect. The attacker may hope that the station then reconnects to a Rogue AP they are controlling instead. With a Disassociation attack a Disassociation frame is sent. These frames are for when the station goes out of range of the access point. As such the attack is similar to the Deauthentication attack, and a real attack is likely to include both frames. The (Re)Association is also similar but makes use of request and response frames sent between the access point and station after they have successfully connected through open authentication.

Rogue AP (access point) is a form of Evil Twin attack. The user connects to an access point controlled by the attacker. This access point will have the same channel, MAC address and SSID as the legitimate access point. The attacker can then read or alter the captured packets compromising *confidentiality* and *integrity*. This attack can be used in conjunction with the DOS methods listed above. A Deauthentication attack can take the legitimate WiFi access point offline, making the user more likely to connect to a Rogue access point.

Krack (krackattacks.com) and **Kr00k**(eset.com) are Man In the Middle (MITM) attacks that exploit faults in the WiFi standard allowing Key Reinstallation. Both attacks could allow encrypted packets to be read and for packets to be injected, compromising *confidentiality* and *integrity.* With a Krack attack the victim reinstalls a previously used key and the nonce is reset and reused. This can be exploited during the four way handshake. A Kr00K attack takes advantage of the session key being reset to zero after a disassociation. This attack would usually be run after a disassociation attack by the attacker.

- ***Attacks on Local Nodes:*** With these attacks a single node is compromised and then the attacker will cause damage from within the network. With **SSH** attack one node is compromised by using brute force to guess SSH credentials. Metasploit, available on Kali Linux, can be used for this purpose. Once the attacker has access to the network and *integrity* and *confidentiality* are lost.

  A **Botnet** attack uses malware to compromise a node so that it can be used as part of a group of computer nodes to achieve a nefarious task. The most common uses are to send spam or to run a Distributed Denial of Service (DDOS). A DDOS affects *Availability* on the target's machine. It is much harder to block than an attack from a single machine which can easily be blocked by a firewall. Botnets often infect Internet of Things (IOT) devices which are usually connected by WiFi. IOT devices often have weak security as the data they process is typically insensitive, so users are more interested

in availability rather than integrity or confidentiality. One of the largest Botnet DDOS was an attack by the Mirai Botnet where over a hundred thousand IOT devices were used to attack Dyn, the domain registration service provider in 2016.

Malware is short for malicious software and is software that delivers a payload that does harm to the target machine. The software varies, it could make the target become part of a botnet to help deliver DDOS attacks. It could be ransomware which encrypts the target's data. Or it could be spyware that records the users' interactions. Depending on the type of Malware *Availability, confidentiality* or *Integrity* could be compromised. The malware used by the researchers were WannaCry Plus, a ransomware crypto worm and TeslaCrypt a randomsomeware trojan. Both attacks encrypt data on the target's computer, with variations offering keys to decrypt the data if a ransom is paid (wikipedia.org, malware)

- ***Attacks on External Nodes:*** **SQL Injection** is where a successful attack through a webform allows the attacker to view or alter the underlying database that the webform writes to. This breaks *Confidentiality* and *integrity.* On a poorly coded website an attacker could enter a true condition after the data such as 1=1. This is then executed with the SQL SELECT * statement exposing the entire table. The attack may be able to manipulate data in the table. They could set the password field to password allowing them to log in as any user. In some cases they may be able to have a DROP statement execute deleting the database.

  An **SSDP** attack uses the universal plug and play protocol to send a large number of requests via a spoofed IP address. The attack is usually undertaken by a botnet and the request is for as much data as possible. The large amount of data being sent by each device overwhelms the network making it a DOS or DDOS attack. *Availability* is lost. The large number of frames can be seen in the table in the next section. This attack has by far the highest number of attack frames and it is the only data file where most of the frames are attack frames.

- ***Multi Layer Attacks:*** in these attacks genuine credentials are taken from a fake entry point and then used to attack. **Evil Twin** is a honey pot attack where fake details are entered into a wireless access point of the type that might be used in an airport. These details can then be used to log into the genuine access point compromising *Confidentiality* and *integrity.* The difference between this attack file and the Rogue AP attack file is that this attack uses open authentication and then a login page where the user enters their details. With the Rogue AP attack file the Access point uses the same MAC and SSID address as the legitimate point but is picked up due to the stronger signal. **Website Spoofing** uses a fake website to capture user details in the same way. A fake login page for a known website allows the attacker to capture the user's credentials for the legitimate site. An attacker could create a fake GMail or facebook page. The attacker then has the details stored to access the real site. This also compromises *Confidentiality* and *integrity.* Kali Linux has built in tools for both attacks, beef-xss and airgeddon.

(Stallings, 2007, Salmon, 2017, Chatzoglou, 2021, Kolias, 2016, cloudflare.com,kali.org)

Each attack has an individual file with the attack packets labelled with the attack name, and non attack packets labelled as “normal”. Looking at the whole dataset together the frames break down like this:

Normal	21,660,389
SSDP	5,499,851
Malware	131,611
Disas	75,131
Botnet	56,891
Evil_Twin	49,186
Deauth	38,942
Website_spoofing	25,923
Krack	16,009
SSH	11,882
(Re)Assoc	5,502
SQL_Injection	2,629
RogueAP	1,310
Kr00k	1,138
NA	21

Each frame is represented as a row with 254 columns with the final column being the label. As Frames vary in length and type many of the rows represent features that are not present in most frames. As a result the dataset is represented by a fairly sparse Matrix, this is one of the challenges in representing frame data as a vector and how I treat the blank, or in reality non existent data points will be important for my analysis.

4. ETL and Pipelines

GitHub:1 ETL

While I only used the csv files for my analysis I still needed to preprocess my data set before analysis. Spark is less flexible than the Pandas Dataframes that scikit-learn uses so I needed to specify the datatype of each dimension as well as putting the data in a format that was efficient for Spark.

a. Parquet

To use my data with Spark MLib I saved my files as parquet files. Parquet is a columnar data storage format that allows for much faster processing than CSV files. Spark is able to skip data that is not being processed and more advanced compression techniques are used (databricks.com). The format can also be used by scikit-learn so once I had the data stored in the format I only used my parquet files.

b. Column Names

The original column names map to the Wireshark documentation, so I would have liked to have kept them. However the “.” has to be escaped with backticks in Spark so I replaced them with “_”.

c. Target variable

Spark Mlib requires numerical labels for classification so I created a new column label where a normal packet is 0 and an attack packet is 1.

d. Unused columns

While I saved all the data in my files there were some columns I did not use in my analysis.

"frame_time" - This would have needed cleaning to be used as Spark TimestampType. It has the timezone but not in the correct position. It also duplicated the Unix time held in "time_frame_epoc". For a real attack the time of day could be very important. The attacker may live in a different time zone. However these are simulated attacks.
"Tcp_payload","udp_payload","ssh_packet_length_encrypted" - are encrypted so would have different values in the encoded vector even if the underlying value was the same.
"Frame_len","frame_number" -I removed these as the random forest was using them. In a multi attack analysis it is more useful to use a dimension that is inherent to the packet rather than length which correlates with that dimension.

After I had made these decisions Chatzoglou, (2022) released a paper which included a discussion on preprocessing. They recommended using the frame_len. It is pointed out that an unprotected frame from an attacker could have a shorter length than a protected frame from a legitimate user, although it also states that it is a field that an attacker would rarely be able to intentionally alter. If I had read this first I would have retained the length.

e. Schema

My dataset is primarily categorical, and many of the numeric values are in fact not ordinal. They are numeric as that binary number has a meaning for the frame but the numbers are not ordinal. For example

Wlan.fc.protected - is a flag 1 means protected, 0 means not protected

Wlan.fc.subtype - is an unsigned 1 byte integer where numbers have different meanings. 8 is a beacon, 12 is Deauthentication. This byte combines with the previous byte in the packet which describes the frame type. 0 is a management frame, 1 is a control frame and 2 is a dataframe. This number is not represented in the same way in my dataset. Ideally we would like these to be treated as categories and code them as strings. However I coded them numerically as it allows for much faster processing when we use the pipeline described next.

(wireshark.org, semfionetworks.com)

I used 4 Spark datatypes in my schema. StringType for the non numeric values and IntegerType, doubleType and LongType. (https://spark.apache.org,a).

f. Empty values

The absence of a feature is significant in my dataset so this data had to be encoded. For the numeric values I replaced the empty values with a 0 and with the string values I used “NA”. The best practice in spark is not to leave null values and while an empty string would be fine in most cases the String Encoder can not use an empty string.

g. Pipelines

As my data had many categorical features, pipelines were essential to my project and affected the type of Schema I had to use. I will describe the process in Spark as it is less abstracted from the user. I believe similar limitations would affect scikit-learn pipelines under the hood.

To process categorical variables in Pyspark we use StringIndexer, OneHotEncoder, VectorAssembler and Pipeline. The pipeline executes in this order

1	Create Arrays of Categorical and Numeric column names
2	New Column created for each categorical variables with a suffix OHE
3	StringIndexer gives each unique string a numeric value
4	OneHotEncoder stores a vector on each in the OHE column such as [0.0, 0.0,2.0]
5	Numeric and encoded data combined by VectorAssembler
6	Assembled data is used by classifier

The 4th part of this process can act as a bottleneck as a column with many different values can become huge and very sparse. The Array length will be the number of

unique values repeated on every row. This is why I kept some non ordinal numeric data as IntegerType or DoubleType, balancing processing time against the real world nature of the data. In the end I was able to process the data in a reasonable length of time as I became more experienced with pipelines. Were I starting the project again I would treat more of the dataset as categorical. Using numeric values for non ordinal values can reduce explainability (Chatzoglou, 2022).

Some of the TCP categories were primarily numeric, but occasionally had multiple data points. An example would be an occasional 01-01-01 instead of 1 or 693411408-1562281087-4292509808 instead of 693411408. I tried coding these as strings but the vector was too large to use so I used numeric data types and accepted the data loss.

The Spark VectorAssember and scikit-learn Classifiers usually convert all numbers to floats, so there is little performance advantage to using Integers rather than doubles. The VectorAssembler also can not use BooleanType so I did not use this even though the flag values and the labels were booleans (spark.apache.org,a, scikit-learn.org).

5. Analysing Individual Attacks

GitHub:2 Random Forest Gridsearch

a. Protocol Types

Chatzoglou(2021) groups the attacks into 4 categories, and we can see that there are differences in the type of frames that are used in the attack packets that follow these groupings.

AWID3 (https://icsdweb.aegean.gr/awid/awid3) provides wireshark queries and when we run these we see the attack packets fall into the following categories:

Attack Type	Management	Control	Data
Deauth	x		
Disass	x		
(Re)Assoc	x		
Rogue_AP	x		
Krack	x	x	x
Kr00k	x		
SSH			x
Botnet			x
Malware			x
SQL_Injection			x
SSDP			x
Evil_Twin	x		x
Website_spoofing			x

So the 802.11 specific attacks mainly use management frames, while the other type of attacks mainly use the data packets.

The data level attacks also use a wider variety of protocols. The list below shows the frame type followed by the description or protocol used:

Deauth

Management:*Deauthentication,Disassociation*

Disass

Management:*Deauthentication,Disassociation*

(Re)Assoc

Management:*Association Request,Reassociation Request,Beacon*

Rogue_AP

Management:*Beacon*

Krack

Management:*Authentication,Reassociation Request,Beacon, Action*

Control:*Request to Send,Clear to Send,Acknowledgement,*

Block ACK Request,VHT/HE NDP Announcement

Data:*LLC*

Kr00k

Management:*Disassociation*

SSH

Data:*TCP,SSH*

Botnet

Data:*TCP,HTTP,SMB2*

Malware

Data:*TCP,SMB2,NBSS*

SQL_Injection

Data:*TCP,TLS,HTTP,UDP,SSH*

SSDP

Data: *SSDP*

Evil_Twin

Management:*Beacon*

Data:*IPv4,TCP,TLS*

Website_spoofing

Management:*Probe Request,Probe Response*

Data:*TCP, TLS,ARP*

For the next stage of analysis I wanted to identify the dimensions that are most important in identifying attack packets. Working with the whole dataset would be computationally intensive and the simplicity of the Wireshark queries suggest unnecessary. Many of the dimensions are highly correlated as they are absent in almost all rows. An example is ARP (Address Resolution Protocol), there are 10 dimensions:

arp
arp.hw.type
arp.proto.type
arp.hw.size
arp.proto.size
arp.opcode
arp.src.hw_mac
arp.src.proto_ipv4
arp.dst.hw_mac
arp.dst.proto_ipv4

In almost all rows all these are empty as the protocol is not used, but when it is used all dimensions are likely to have a value. Using all 10 values would be unnecessary, but selecting which is a challenge. Ideally we could try all combinations, but this is not computationally practical. Reyes (2020) working on the AWID2 dataset used correlation methods to remove features which are correlated or have low variance. Chatzoglou's (2022) analysis of this dataset used expert knowledge to select features. My method was closer to the second using a grid search to find features as my knowledge of network traffic was limited before starting the project.

b. Trees

The only algorithms I used in the project were decision tree based. Decision Trees are well suited to categorical data, which is not the case for most machine learning algorithms (Ryza, 2017). I had also read a number of articles where network data

was analysed and trees tended to produce the best results. These included the researchers own work on the AWID2 dataset where Random Forest and J48 produced some of the best results (Kolias, 2016, Alotabi, 2016). If my results had been worse I would have tried other methods. I will give a more detailed explanation of how Tree algorithms work in the next section when comparing different types of trees.

Being unfamiliar with Pyspark I found an introductory Jupyter note from CERN which used Random Forest and Gradient boosted Trees. While the dataset was very different, looking for the Higgs Boson Particle with only large doubleType values, I used it as a guide to do teach myself how to do a grid search and extract feature importances (swan-gallery.web.cern.ch).

I needed to make some significant adjustments using pipelines to handle my categorical variables, and to consider how to measure the success of each model given that I have an unbalanced dataset where normal packets usually heavily out number attack packets.

c. Evaluating unbalanced datasets

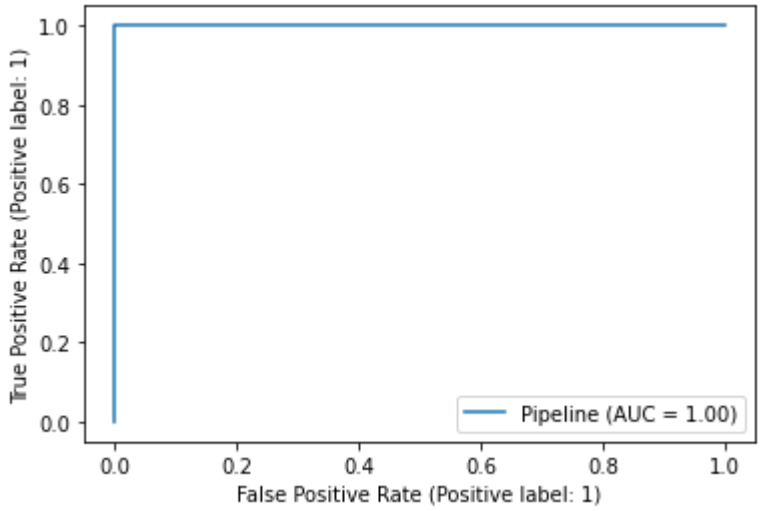
For most networks the vast majority of frames are not attack frames. This is one of the most challenging aspects of making a successful IDS. If we consider at Bayes theorem with A as an attack happening and B as the IDS altering for an attack:

$P(A|B)=(P(B|A)*P(A))/P(B)$

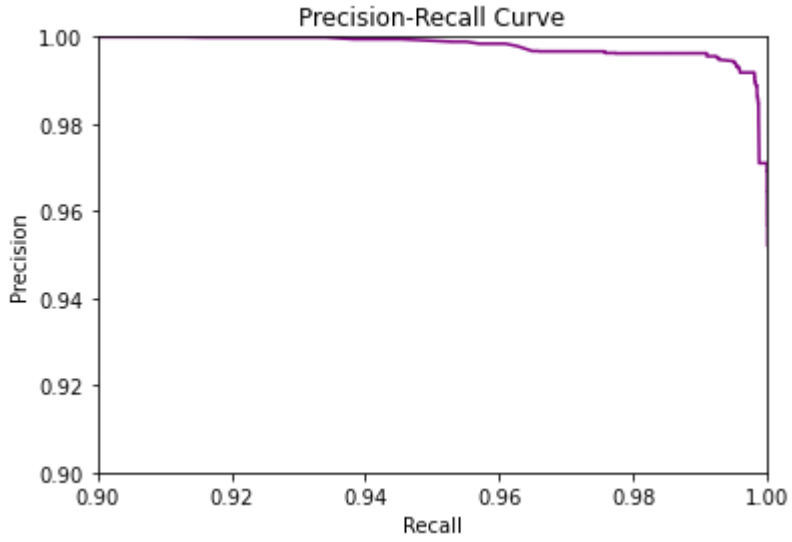
The probability of A will vary. It will be much higher on a network with a financial database or government secrets than an obscure blog. However it will always be a small number. Despite their rarity there is a large cost to False Negatives as it can allow an attacker to breach confidentiality, integrity or availability. A good detector must have high sensitivity (TP/(TP+FN)). However increasing sensitivity will inevitably increase the False positive rate too. While not as serious, an alarm that goes off too regularly for a scenario with a low base rate is also problematic. Such an

alarm can cause Network administrators to ignore it, and can be used as cover by an attacker. An attacker may trigger the alarm regularly using a simplistic attack and then penetrate the network using a more sophisticated attack which is ignored by the administrators (Haring, 2014, Salmon, 2017).

These considerations led to a couple of decisions with my work. The first was when using the Spark BinaryClassificationEvaluator. It has an attribute MetricName. This can take two values: areaUnderROC or areaUnderPR. The Area under the Receiver Operator Curve is the more common measure and is used in the CERN notebook. This curve plots True Positive Rate (Sensitivity) on the X Axis and False Positive Rate (FP/(FP+TN). A good classifier has a high sensitivity rate and a low false positive rate so the Area under the Receiver Operator Curve will be high. A perfect classifier is shown below, area is a square so 1*1=1.



I instead used the other metric area under the PR curve. This plots Precision (TP/TP+FP) on the X axis and Recall(TP/(TP+FN) on the Y Axis. This is preferred for unbalanced datasets. In my case a classifier that classified almost everything as negative would get a low false positive rate. The high number of True Negatives leads to a large denominator. Again a perfect classifier is a square with a score of 1*1=1 (scikit-learn.org,a).



The PR Curve can also be used to see how adjusting the sensitivity would optimise a classifier. The chart above is from my notebook pysparkMinimalDimensions.ipynb and was used for this reason. In some of my individual attack notebooks everything is classified as negative, however the classifier believes the attack packets are less likely to be normal than the normal packets. The Spark RandomForestClassifier has a default threshold of 0.5. In the case of the SSH attack the classifier I made has classified everything as negative. I have printed out some results for the attack and normal packets. Of those shown the classifier is more than 99% sure the first normal packet is normal [0.9962215012083501, 0.0037784987916499456]. For the first attack packet it is 85% sure it is normal [0.8529477152163364, 0.14705228478366364]. We could get a useful model by adjusting the threshold to somewhere between 0.99 and 0.85. Scikit-learn abstracts this part of the process from the user. Unfortunately the long run time producing these models meant that I was unable to rerun and make this adjustment when I better understood the process.

My literature review highlighted that the metric used to evaluate my models was an important consideration. Many of the most common metrics used to assess binary classifiers are problematic when used on an unbalanced dataset such as mine. Doshi (2018) looking at DDOS detection uses Precision, Recall, Accuracy and F1 score. Kolias (2016) studying the AWID2 dataset also gives these scores as well, FP Rate and Area under the ROC Curve. On the papers (Chatzoglou,2022) looking at the AWID3 dataset all these metrics are included with the F1 score emphasised.

Zolvanari(2019) studying attacks on IOT networks also uses Undetected Rate and Matthews Correlation Coefficient (MCC).

The formulas for some of these metrics:

Accuracy = (TP+TN)/(TP+FP+TN+FN)
Precision = TP/(TP+FP)
Recall = TP/(TP+FN)
F1 = (2*precision*recall)/(precision+recall)
False positive rate = FP/(FP+TN)
Undetected rate = FN/(FN+TP)
MCC=((TP*TN)-(FP*FN))/sqrt((TP+FP)*(TP+FN)*(TN+FP)*(TN+FN))

The accuracy score causes the most problems. An algorithm classifying almost everything as negative will result in a high True Negative score and also a high accuracy score, even if it is of no practical use.

I printed the precision, recall, F1 score and MCC. F1 and MCC seemed most suitable for my purposes, with MCC the best measure as it is suited to an unbalanced datasets.

d. Feature Importance

On each individual attack I ran a gridsearch testing some Hyperparameters but more importantly extracting the most important features. I used a subsample of the attack in question as a grid search is computationally expensive. I then used this information to create another random forest model only using the important features. This was a technique used in the CERN notebook and in most cases I got excellent results with a small number of dimensions, with the caveat of not understanding how to adjust the model threshold at the time.

featureImportances is an attribute of an RandomForestClassificationModel object in Spark. As I was using a pipeline it has to be extracted from the pipeline with the line:

bestModel.bestModel.stages[-1].featureImportances

If we look at the first notebook, *1.Deauth with extended Notes.ipynb* it is in cell 2, we see the output of the features used by the tree and their names:

[('wlan_seq', 1.9093512751738843e-05),
('radiotap_dbm_antsignal', 0.000307446777711239),
('wlan_fc_protected', 0.0046540080042853),
('wlan_fc_subtype', 6.92953668640615e-05),
('wlan_fixed_reason_code', 0.9949128115722106),
('wlan_radio_signal_dbm', 3.7344766177120 79e-05)]

The numbers are in scientific notation but this means that *wlan_fixed_reason_code* is responsible for 99.5 of the decisions the tree makes. This makes sense for a deauthentication attack. A *wlan_fixed_reason_code* gives a reason for the disconnection so is absent from most packets but has a value on the attack packets. This dimension is also key in the Disassociation attack.

When we look at the length of the featureImportances vector we get a large number, 6956 in the Deauthentication case (cell 26) far more than the 254 dimensions we started with. This is because every entry that the OneHotEncoder creates is counted as a separate feature, whereas the numeric features are only a single feature. This is useful, as can see the relevant category for the categorical variables. For the Botnet attack we can see the source IP address is ip_srcOHE_192.168.2.248. However it also accounts for the long training time for the models even on the single attack datasets.

Looking at the most important features for each attack:

		Reason Code and Importance: 1 means full responsible			
1	**Deauth**	wlan_fixed_reason_code, 0.995			
2	**Disas**	wlan_fixed_reason_code, 0.99			
3	**(Re)Assoc**	wlan_tag_lengthOHE_4, 0.871			
4	**Rogue AP**	wlan_country_info_fnmOHE_1, 0.636	wlan_daOHE_ff:ff:ff:ff:ff:ff, 0.363		
5	**Krack**	radiotap_channel_freq, 1.0			
6	**Kr00k**	wlan_fixed_reason_code, 0.999			
7	**SSH**	wlan_saOHE_00:0c:29:1d:70:f5, 0.43	wlan_daOHE_00:0c:29:1d:70:f5, 0.343	wlan_daOHE_04:ed:33:e0:24:82, 0.227	
8	**Botnet**	ip_srcOHE_192.168.2.248, 0.528	ip_dstOHE_192.168.2.248, 0.362		
9	**Malware**	wlan_saOHE_00:0c:29:cf:08:aa, 0.48417576824363523	wlan_daOHE_24:f5:a2:ea:86:c3, 0.16393184298732014	ip_dstOHE_192.168.2.130, 0.11818792086739371	wlan_daOHE_00:0c:29:cf:08:aa, 0.109
10	**SQL Injection**	ip_srcOHE_192.168.2.248, 0.588	ip_dstOHE_192.168.2.248, 0.388		
11	**SSDP**	udp_length, 0.987			
12	**Evil Twin**	wlan_bssidOHE_0c:9d:92:54:fe:35, 0.985			
13	**Website Spoofing**	wlan_saOHE_04:ed:33:e0:24:82, 0.85	wlan_daOHE_04:ed:33:e0:24:82, 0.145		

The feature importance reflects the average importance across all of the trees that are used in the random forest (spark.apache.org,b). The small number of dimensions came as a surprise to me. I was expecting some of the attacks to be easy to classify and others more difficult, but all the attacks had a small number of attacks that accounted for most of the choices the Tree made.

Chatazoglu (2022) lists papers that have studied the AWID2 dataset and which features are often used. The dataset had fewer dimensions and slightly different naming conventions than AWID3 set, but dimensions with wlan and radiotap dominate. The dimensions I used to classify the 802.11 attacks were in this category.

As my results meant that I was working in a lower dimension space that I was expecting I switched from Pyspark to scikit-learn. Spark's main advantage is its distributed processing ability and this was less of a benefit working with fewer dimensions. As well as being more user friendly scikit-learn has a larger number of tree algorithms that can be used, while Spark Mlib only has two Random Forest and Gradient Boosted Trees.

I also ran an all dimensions gridsearch on a 1/1000 subsample of the whole dataset (*2 Random Forest Gridsearch/wholeDataSet.ipynb*). Even this small sample had to train overnight. The dimensions that scored highly did not correlate especially with the individual file attack dimensions that scored highly. No features scored very highly. The most important features accounted for less than 10% of the feature importance vector. A very different result to the individual trees I ran. The higher scoring dimensions were the dimensions that I added to my medium dimension model. These are:

http_host
http_request_line
http_request_method
http_request_version
ssdp
http_request_full_uri
ip_ttl
udp_dstport
Udp_time_delta

Chatzoglou (2022,a) second paper of the summer created a model using WiFi and non WiFi features. These were chosen because of success in previous studies and their domain expertise. The ip.ttl was used in that model as was the udp length.

However the other non WiFi packets were ARP and TCP none of which I used. These are said to be better as they are less likely to be encrypted and unrelated to the specific application.

e. Mac and IP address

Github: 2 Random Forest Gridsearch/NoMacNoIP

With the WiFi specific attacks the features that score highly on the gridsearch relate to the type of packet, which shows the search is working well. With the non 802.11 attacks the Random Forest uses MAC (Media Access Control Address) and IP (Internet Protocol Address) addresses to identify the attacker. From the One Hot encoder feature name we can see which MAC and IP are being used.

Website Spoofing, Malware and SSH are using Wlan_sa and Wlan_da, these are the source and destination MAC addresses of the wireless local area network. SQL_Injection and Botnet use ip_src and ip_dst which are source and destination IP addresses. Evil Twin uses wlan_bssid which is the Basic Service set identifier of the access point and follows MAC conventions (wikipedia.org).

The Random forest identifying these signatures of the attacker is a good sign. They belong to the attacker so frames emanating or finishing from these addresses are attack packets. However these addresses belong to the researchers and are highly unlikely to be used by a real attacker. I trained my main model using these dimensions so it definitely suffers from this form of overfitting.

The one exception amongst the higher level attacks is the SSDP attack which is well identified by the udp_lengh. Looking at the attack data, all the attack packets seem to have the same payload (a long string) and also the same length (102) which is in the header of the User Datagram Protocol, indicating the size of the data (payload). The fact that both are the same suggests the tree favours numeric over categorical values, as both seem to have a one to one or near one to one mapping to the attack.

After creating my main model I returned to the single attack files. I tried to train models on these six attacks without using any MAC or IP address. The results were very poor in five of the six cases. I first trained the models using MaxDepth = None using the Random Forest in scikit-learn. This means that the tree can have unlimited splits. Even on the most powerful settings available to a free tier user, 16CPUs and 108GB RAM the resources were exhausted and crashed. I then reran the models using MaxDepth =256, a very deep tree. In the cases of Botnet, SSH and Malware and SQL_injection the models classified everything as normal. This causes a scikit-learn warning that the metrics are poorly defined. There are no True Positives so the numerator is zero and so precision, recall, F1 and MCC are all zero. A model that does this is useless.

The Evil Twin model classified almost all the frames as normal. This led to a good example of the problematic situation described in the evaluation section. The output is:

180175 true negatives
82 true positives
9739 false negatives
4 false positives

The precision is 0.9534883720930233
The recall is 0.008349455248956318
The fscore is 0.016553951751286967
The Matthews correlation coefficient is 0.08667704459364711

A large number of True and False Negatives, a small number of True Positives and a very small number of False Positives. This gives an excellent precision, a terrible recall and a bad F score and MCC. This is why Fscore and MCC are best suited for assessing unbalanced binary classifiers such as this dataset.

There was one model that was successful without MAC or IP address, the Website Spoofing model. While a little over-sensitive, the MCC for the model was 0.912, and it used a good variety of features. Features using the ARP (Address Resolution

Protocol) were the most used but only accounted for one third of the Feature importance score.

Decision Trees are unable to usefully classify the SSH, SQL_Injection, Botnet, Malware and Evil Twins attacks without using MAC and IP signatures. An IDS that used a model trained using decision trees would be unable to identify these attacks.

The first paper released over the summer by the dataset's compilers (Chatzoglou, 2022) makes this point that MAC addresses should not be used. Their second paper (Chatzoglou, 2022a) which attempts to classify the attacks into Normal, Flooding and impersonation attacks using Deep Neural Networks (DNN) produced much better results once an engineered feature was introduced. That engineered "insider" feature is given the attackers ip address. This supports the theory that ip addresses are needed for good classification of higher level attacks even when more complex algorithms like DNNs are used.

6. Assessing Types of Trees

Github:3 Tuning on a Subsample

a. Gridsearch on the whole dataset

After getting useful results when analysing each attack individually I ran a grid search using almost all the dimensions (wholeDataSet.ipynb). Despite only using a thousandth of the dataset the search ran over night and the feature importance was very messy. The highest scoring features were non WiFi features and did not correlate with the features that had given good results, http_host and http_request_line scored highest. This likely reflects that the higher level attacks have more attack packets and are harder to classify. This showed me that using fewer dimensions was essential not only for reducing run time but also for getting useful results.

b. A Subsample

As the grid search using all dimensions had taken so long, despite its small size, I created a stratified subsample to test algorithms on. My final model only took two hour to train using the maximum free GCP settings, so were I to redo this project I would use the whole dataset at this stage, but my knowledge was lower at this point.

My subsample parquet was just over 500mb and has 645314 packets, of which 94% are normal. It contains approximately 3,000 attack packets from each attack. Some of the attacks have fewer than this so all packets are included for some (SQL_Injection, Rogue_AP). For the largest attack, SSDP only 0.05% of the packets are used. This subsample is harder to classify than the whole dataset because of the smaller number of SSDP frames, which is most of the attack packets and correlates well with UDP length.

I ran three algorithms with Pyspark before moving to scikit-learn. I will not compare results using the different libraries as while the algorithms should be the same the parameters and defaults are different.

Algorithm	**F Score**	**MCC**
Minimal Dimensions	0.9067844926	0.9013799806
Medium Dimensions	0.9018583476	0.8987245415
Gradient Boosted Trees	0.9629412196	0.9613042595

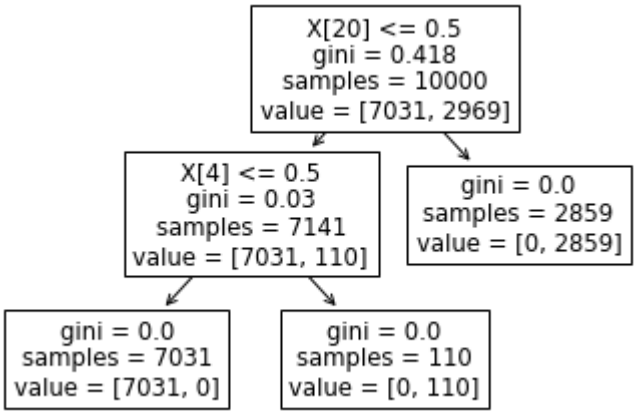
The Minimal dimensions used the 10 factors that my individual attack analysis identified. The Medium dimensions introduced another 9 dimensions that the all dataset gridsearch had used. Both these were untuned Random Forests, and produced good results. The extra dimensions resulted in slightly inferior performance, mainly due to more false positives. The Gradient Boosted Tree (GBT) used minimal dimensions and gave excellent results. The problem was the training time on this small subsample was 4.5 hours. This was much longer than the Random

Forest models. This is because the trees in a random forest can be trained in parallel, whereas GBT must be trained consecutively (databricks.com,a).

c. Types of Trees

While the main reason for switching from PySpark to scikit-learn was dealing with fewer dimensions, one of the other benefits was its larger library of Tree Algorithms. I built the rest of my models in the Cloud using GCP's Vertex AI managed notebooks. I only used the free account which had $300 dollars.The settings I used (16cpu 108gbram) cost $0.95 per hour in the Netherlands (europe-west4). However a paid account would be able to spin a more powerful instance even in a managed notebook. The maximum is 96cpus and 624gb of ram and costs $6.25 per hour. Using these higher settings with a paid account would allow parallelizable algorithms like the random forest to train faster.

scikit-learn contains a Decision Tree (DT) algorithm. Using a basic example which can be easily printed, we can see how the algorithm classifies a web spoofing attack which can be perfectly classified from Wlan.sa and Wlan.da dimensions. This is only using a small subset dataset of 10,000 so the numbers do not run over the box and are easy to read. The dimension names are not visible as we are using a pipeline.



The initial dataset has 7031 normal packets and 2969 attack packets, this gives a gini coefficient of 0.418, we are trying to minimise this so each leaf has as little variety as possible. The first split X[20] is the source MAC address of the attacker.

This gets a pure leaf with 2859 true positives. The reminding branch is mainly true negatives, but there are still 110 attack packets. The Gini is 0.03 but we want to minimise this number. A second split is made on the X[4] is made on the Wlan.da, the MAC address at the destination. This separates the attack packets so we get a perfect classification of the dataset. When dealing with more complex attacks, or removing MAC addresses the process becomes much more complex, hence the long run times or time outs. The feature importance vector reflects which dimensions have the biggest effect on the splits. Each Tree has parameters that can be altered to improve performance. DT is the simplest algorithm but still has some parameters that can be tuned. We can choose to use entropy, a measure of randomness instead of Gini, a measure of inequality and limit the number of features. A large number of features can cause long run times as the tree gets bigger and risks overfitting.

From the decision tree there are a number of Ensemble methods where multiple base estimators are used, in the hope of improving performance and reducing the chances of overfitting.

The first set of methods are averaging methods, this includes the Random Forest, the main algorithm used until now. Many trees are grown from a subsample, the data is then classified by each. We therefore get an averaging of classifications, making the tree less likely to overfit (stat.berkeley.edu). Scikit-learn also has an Extra Trees Algorithm which is a variation of the Random Forest. Here the thresholds for making splits are made at random, with the best performing selected. One of the advantages of this family of algorithms is that the trees can be grown separately allowing distributed computation. While scikit-learn does not have the distributed processing ability of Spark, we can spread the trees across the processors available using the n_jobs argument. If working in the paid tier we could use a very large number of CPUs.

The second family of Ensemble methods use boosting where trees are grown sequentially and then combined. The Gradient Boosted Trees included with Spark are in this family. The number of trees is set by the user using n_estimators, and a greedy algorithm is used where each new tree attempts to minimise the sum of losses of the previous tree. Scikit-learn also has a related form of the algorithm

called Histogram boosted trees with much faster run time. It seemed well suited to my dataset, but had been recently released and was not compatible with the sparse Matrices I was using. The other type of boosting Tree available on scikit-learn is the Adaboost. Here trees are grown sequentially first on the whole dataset and then a weighting applied depending on performance. The best performing trees are given a higher weighting, and more prominence if they are able to correctly classify the harder to classify observations.

(Provost(2013), Scikit-learn.org,b)

In the researcher's analysis of the AWID2 dataset, the J48 algorithm also performed well. This would have been interesting to attempt if I had had more time. It is a Java implementation of c.45 decision tree and not available on scikit-learn, or Pyspark. Spark is written in Scala which uses the Java Virtual Machine so it may become available in the future (weka.sourceforge.io, Kolias, 2016).

When running Tree algorithms there are trade offs. Deeper trees with more branches are better at classifying the data, however the training time will also be longer and there is diminishing returns with increasing tree size. A huge deep tree can become a lookup table for the dataset, and thus overfitting can be a problem if the algorithm is used on new data (databricks,a). This is related to the bias-variance trade off, a deep tree has high variance

I trained a model using each algorithm on the subset I had created. I used the medium number of dimensions (19). I did a basic parameter tuning before creating each model, but the performance can not be considered optimal for each tree. The results of the classification were:

Algorithm	**F Score**	**MCC**
Decision Tree	0.9184104995	0.9143841723
Random Forest	0.9191594922	0.915204972
Extra Trees	0.9189228636	0.9149641942
Ada Boost	0.8704858593	0.8634566932
Gradient Boosted Trees	0.9162943977	0.9123444003

The results for all the algorithms turned out very similar, with the exception of Adaboost, which produced more false positives than the other algorithms. Adaboost also performed poorly in the research carried out by the dataset's compilers (Chatzoglou, 2022).

The Gradient Boosted Tree performed worse than the Pyspark version on the same dataset and I do not know why. They both took a long time to train, although the scikit-learn one was faster on a more powerful machine. The Pyspark version had the parameters (maxIter=50, maxDepth=10) while the scikit-learn version had (n_estimators=100,learning_rate=0.1,max_depth=9). The estimators and iterations are equivalent. The Pyspark would also have used a learning rate of 0.1 as the default. There are several categories like *MaxBins* (Pyspark) and *min_samples_split* (scikit-learn) which do not have an exact equivalent in the other application. It is plausible that the different test-train split is the cause, but this seems less likely as performance was comparable for the random forests.

For my final model I discounted the boosting methods due to Adaboosts inferior performance and Gradient Boosted Trees long run time.

There was little difference between the other three algorithms, but I chose Random Forest due to having used it for most of the project and its lower tendency to overfit compared to Decision Tree and ability to be parallelized and make use of some of the 16CPUs available to me.

d. Hyperparameter Tuning

Hyperparameter tuning is adjusting a model's variables to optimise its performance (cloud.google.com,a). I have been doing a simple form of this on all the models using a basic gridsearch, but have been focusing more on dimension reduction and algorithm selection.

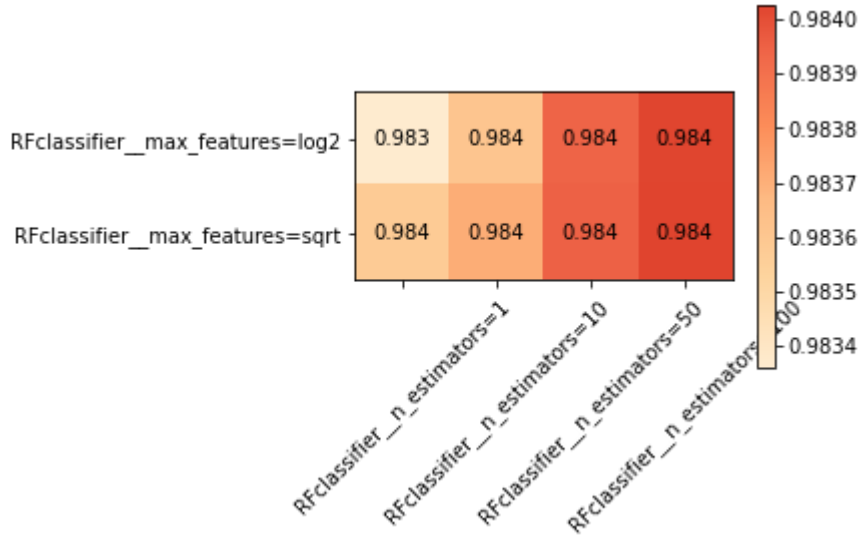
The scikit-learn Random Forest class has a number of parameters that can be altered. The number of trees can be changed using the n_estimators parameter. The criterion can be gini, entropy or log-loss. The maximum number of features used for splits can be set to none (all features), the square root or log2 of all features.

If training a model for production a lower depth may be preferred to reduce overfitting. Chatzoglou, (2022) set the max depth to 10 in one of their models for this reason. I left this at none though as I was looking at performance more than generalisability.

I used the halving gridsearch which reduces the time for the computationally expensive gridsearch. Even on the subsample the results suggested more trees would lead to slightly improved performance. Running the model with 600 trees gave a slightly better performance, but only at the fifth decimal point and the computation time scaled linearly taking 5.8 times longer.

Overall the performance improvement from Hyperparameter tuning was very small with very marginal gains at the expense of longer processing time. The graph below shows the change in performance, from using 1 to 100 trees and the output looks the same in almost all cases as the chart cuts off at the third decimal place (The attribute CV_results_ from the gridsearch class uses accuracy). This shows that correctly

classifying the majority of true Negatives is fairly simple. By adding more trees we get a small improvement.



7. Training on the Whole Dataset

Github: 4 Training and Deploying in the Cloud

To train my model on the whole data I used the information I had learned from my previous models. I used my minimal dimensions, both because increasing the number of dimensions had not improved performance and to make my REST API simpler to test.

The parameters used,

(max_depth=None, min_samples_split=2,max_features='log2',criterion='gini',n_estimators=100,random_state=0,n_jobs=-1)

This will result in a deep tree using all CPUs.

My instance crashed while loading the dataset, so I loaded each file, stored the used columns and deleted the variable. One of the advantages with Spark is that it would thrash when running low on memory and this would be visible in the Spark UI, allowing the user to accept slow performance or adjust the program.

The model trained on my 80% training set scored highly on my 20% test set. The model scored well:

F Score	MCC
0.9837094754	0.9793362616

I believe this improvement over the subsample models is mainly due to the large number of SSDP attack packets that make up a substantial number of the total dataset. I used Joblib to save my model, as scikit-learn recommends it over pickle. (scikit-learn,c).

8. Deploying the Model as a REST API with Django

Github: Django

Finally I created a simple Django application to deploy my model and show how a trained model can be useful. I also wanted to demonstrate that I can use functions and object-oriented programming which were almost entirely absent from my analysis.

Django is a web application framework which is fast, reliable and has excellent documentation and support on stackoverflow. I had used Django in Dr Stelios Sotiriadis' Cloud computing module to create a REST API for a social media application. I used the lecture notes and especially the Oauth 2.0 user authorisation as a starting point (Django, Birkbeck Cloud Computing, 2021). I have included a notebook that creates a new user and calls the API. I built the app in a local virtual

environment and intended to create a dockerfile but ran out of time getting the dependencies right.

If a user is registered and has a Oauth 2.0 token they are able to call the API using JSON data that corresponds to 10 dimensions that I used to train my whole dataset model:

JSON with prediction data sent with a POST request↦	serialized into bytes for DB↦	data saved to MySQL database as an object↦	data is passed to model, a prediction is saved as a related object↦	data can be consumed from the database

This is an example of a microservice (cloud.google.com,b) that performs a single service. It is language agnostic in that an application that calls the API can use any language. The application just needs to provide the data as JSON to make the POST request. This is equally true of another application consuming the data in the database. I have built a basic website that consumes data from the database *predWebpage.html,* but equally any other authorised application could consume data from the database by using a GET request and then displaying or manipulating the data.

This REST API is purely a proof of concept, it would not be useful in a real world setting. Stripping out these 10 dimensions from live packet data would be much more complex than my project, and writing the live packets to an SQL database would be too slow. However it does show how training a model is not purely useful for analysis but can be used on new real world data to make predictions.

Technologies allowing the analysis of live data, or building models from live data is an exciting and rapidly growing area of data science. Spark can be integrated with Apache Kafka to allow models to be trained on large scale stream data (Spark.apache.org, c). One of the functions of Google Vertex AI lab is deploying models which can then be sent large batches of data and return predictions. The examples given are for a bike sharing company, where a prediction is returned for

how long a current cyclist will stay on a bike or which movie a current viewer will watch next. For both the duration of the current activity is an important parameter, so could change the prediction (cloud.google.com, c).

9. Conclusions

Using Tree based Machine learning algorithms, was a very effective way of classifying attack packets in the AWID3 dataset in specific circumstances. A model run on a specific attack was able to identify attack packets using a very small number of dimensions, typically 1 to 3. This was true for all the 802.11 attacks. Deauthentication, Disassociation and (Re)Association which can be identified from the type of management packet used. It was also true for the Krack and Kr00k key reinstallation attacks which exploit faults in the WiFi standard and the SSDP attack which was identified by its use of the UDP as the underlying protocol.

The Random Forest was also able to identify attack packets for the other attacks, SSH, Botnet, Malware, Rogue AP, SQL_Injection, Evil Twin and Website spoofing. However the identification was only done using MAC or IP addresses that belonged to the datasets compilers. While knowing these details is beneficial, it would not be useful for identifying the application layer attacks in the wild. Such details would usually be unknown and can be easily spoofed by an attacker. If I was to continue studying this dataset, I would focus on these attacks. Chatzoglou, (2022) work on their dataset seems to run into this problem too even with their greater knowledge of WiFi, cyber security and the dataset itself. Using an engineered feature that knew the IP address substantially improved performance.

This does not mean that Machine learning is not useful in this domain. A real attack will often start with a WiFi specific attack, in order to encourage a user to then fall victim to a higher level attack. Giving a user more protection from lower layer attacks would be very beneficial (Chatzoglou,2021). Another problem with using trees for the higher level attacks is that they do not utilise the dataset as a time series. The attack packets are very varied in a higher level attack, which is difficult for the tree to use. An algorithm that was effective in identifying these attacks would better utilise the

order the packets are produced. An application layer attack may have packets that have to execute in a certain order. However the number of normal packets sniffed in between would nearly always vary. It is very hard for an algorithm to use this type of series as the rule for looking back an unknown number of times is very computationally intensive.

When classifying multiple attacks together the results from the Trees were also good, although I was using dimensions that included the researchers MAC and IP addresses. This situation is in some ways more artificial than studying an individual attack. A real attack is likely to include one or two types rather than 13. If a model was to be used in a real IDS it would be best to train it only on the WiFI specific attacks and SSDP, and use signature based methods to identify the higher level attacks.

If I were to continue this study I would also be more careful with how the SSDP was treated, the large number of packets start to move the exercise away from being a needle in a haystack like search. The dataset can be considered as three clusters that need to be treated differently, attacks that use the WiFi protocol, higher level attacks and SSDP. My final model had very high performance, but I think the improvement over my subsample models was mainly down to identifying the SSDP frames.

The other change I would make would be to use the whole dataset at an earlier stage. The training time of my final model was only one and a half hours and I finished the project with £130 of GCP credits. If I had had the level of knowledge I have now at the midpoint, I could have done all my tuning and algorithm testing on the whole dataset.

Bibliography

Articles

Alotaibi, 2016, B. Alotaibi and K. Elleithy, "A majority voting technique for Wireless Intrusion Detection Systems," 2016 IEEE Long Island Systems, Applications and Technology Conference (LISAT), 2016, pp. 1-6, doi: 10.1109/LISAT.2016.7494133. https://ieeexplore.ieee.org/abstract/document/7494133

Chatzoglou, 2021, E. Chatzoglou, G. Kambourakis and C. Kolias, "Empirical Evaluation of Attacks Against IEEE 802.11 Enterprise Networks: The AWID3 Dataset," in IEEE Access, vol. 9, pp. 34188-34205, 2021, doi: 10.1109/ACCESS.2021.3061609. https://icsdweb.aegean.gr/awid/Empirical-evaluation-of-attacks-against-IEEE-802.11-enterprise-networks-The-AWID3-dataset.pdf

Chatzoglou, 2022, E. Chatzoglou, G. Kambourakis, C. Kolias and C. Smiliotopoulos, "Pick Quality Over Quantity: Expert Feature Selection and Data Preprocessing for 802.11 Intrusion Detection Systems," in IEEE Access, vol. 10, pp. 64761-64784, 2022, doi: 10.1109/ACCESS.2022.3183597.

Chatzoglou, 2022a, Chatzoglou, E.; Kambourakis, G.; Smiliotopoulos, C.; Kolias, C. Best of Both Worlds: Detecting Application Layer Attacks through 802.11 and Non-802.11 Features. Sensors 2022, 22, 5633. https://doi.org/10.3390/s22155633

Doshi, 2018, R. Doshi, N. Apthorpe and N. Feamster, "Machine Learning DDoS Detection for Consumer Internet of Things Devices," 2018 IEEE Security and Privacy Workshops (SPW), 2018, pp. 29-35, doi: 10.1109/SPW.2018.00013

Kolias, 2016, C. Kolias, G. Kambourakis, A. Stavrou and S. Gritzalis, "Intrusion Detection in 802.11 Networks: Empirical Evaluation of Threats and a Public Dataset," in IEEE Communications Surveys & Tutorials, vol. 18, no. 1, pp. 184-208, Firstquarter 2016, doi: 10.1109/COMST.2015.2402161.

https://icsdweb.aegean.gr/awid/draft-Intrusion-Detection-in-802-11-Networks-Empirical-Evaluation-of-Threats-and-a-Public-Dataset.pdf

Reyes, 2020, A. Reyes, Abel, Francisco D. Vaca, Gabriel A. Castro Aguayo, Quamar Niyaz, and Vijay Devabhaktuni. 2020. "A Machine Learning Based Two-Stage Wi-Fi Network Intrusion Detection System" Electronics 9, no. 10: 1689. https://doi.org/10.3390/electronics9101689

Wang, 2004, Wang, K., Stolfo, S.J. (2004). Anomalous Payload-Based Network Intrusion Detection. In: Jonsson, E., Valdes, A., Almgren, M. (eds) Recent Advances in Intrusion Detection. RAID 2004. Lecture Notes in Computer Science, vol 3224. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-30143-1_11

Zolanvari, 2019, M. Zolanvari, M. A. Teixeira, L. Gupta, K. M. Khan and R. Jain, "Machine Learning-Based Network Vulnerability Analysis of Industrial Internet of Things," in IEEE Internet of Things Journal, vol. 6, no. 4, pp. 6822-6834, Aug. 2019, doi: 10.1109/JIOT.2019.2912022.https://arxiv.org/pdf/1911.05771.pdf

Books

Harang, R. (2014). Bridging the Semantic Gap: Human Factors in Anomaly-Based Intrusion Detection Systems. In: Pino, R. (eds) Network Science and Cybersecurity. Advances in Information Security, vol 55. Springer, New York, NY. https://doi.org/10.1007/978-1-4614-7597-2_2

Provost (2013), Provost, Foster,, Tom Fawcett, and an O'Reilly Media Company Safari. Data Science for Business. 1st edition. O'Reilly Media, Inc.

Ryza(2017), Sandy Ryza, Uri Laserson, Sean Owen, Josh Wills,Advanced Analytics with Spark, 2nd Edition, 2017, O'Reilly Media, Inc

Stallings(2007), William. Cryptography and Network Security: Principles and Practice. Seventh edition. Boston: Prentice Hall. 2007

Stallings, 2015, O'Reilly Media Company Safari. Foundations of Modern Networking: SDN, NFV, QoE, IoT, and Cloud. 1st edition. Addison-Wesley Professional

Salmon, 2017, Salmon, Arthur,, Warun Levesque, Michael McLafferty, and an O'Reilly Media Company Safari. Applied Network Security. 1st edition. Packt Publishing.

Web Resources

Birkbeck Cloud Computing, 2021, lecture and lab notes produced by Dr Stelios Sotiriadis

cloud.google.com, https://cloud.google.com/vertex-ai/docs/workbench/managed/, accessed 28/08/2022

Cloud.google.com, a, https://cloud.google.com/ai-platform/training/docs/hyperparameter-tuning-overview, accessed 04/09/2022

Cloud.google.com, b, https://cloud.google.com/learn/what-is-microservices-architecture, accessed 08/09/2022

Cloud.google.com, c, https://cloud.google.com/vertex-ai/docs/featurestore/serving-online, accessed 08/09/2022

Cloudflare.com, https://www.cloudflare.com/learning/ddos/what-is-a-ddos-botnet/, and https://www.cloudflare.com/learning/ddos/glossary/malware/, and https://www.cloudflare.com/learning/ddos/ssdp-ddos-attack/ accessed 29/08/2022

Databricks.com, https://www.databricks.com/glossary/what-is-parquet, accessed 29/08/2022

Databricks.com,a, https://www.databricks.com/blog/2015/01/21/random-forests-and-boosting-in-mllib.html, accessed 04/09/2022

Django, https://www.djangoproject.com/, accessed 08/09/2022

Docs.python.org, https://docs.python.org/3/library/venv.html, accessed 28/08/2022

Eset.com, https://www.eset.com/vn-en/kr00k/, accessed 28/08/2022

Ieeexplore.ieee.org, https://ieeexplore.ieee.org/document/654749, accessed 28/08/2022
icsdweb.aegean.gr/awid/awid2, accessed 28/08/2022

icsdweb.aegean.gr/awid/awid3, accessed 28/08/2022

hub.docker.com, https://hub.docker.com/r/jupyter/pyspark-notebook, accessed 28/08/2022

Kali.org, https://www.kali.org/tools/airgeddon/ and https://www.kali.org/tools/beef-xss/ and https://www.kali.org/tools/metasploit-framework/ accessed 28/08/2022

Krackattacks.com, https://www.krackattacks.com, accessed 28/08/2022

Scikit-learn.org, https://scikit-learn.org/stable/modules/ensemble.html, accessed 28/08/2022

Scikit-learn.org,a https://scikit-learn.org/stable/modules/model_evaluation.html, accessed 30/08/2022

Scikit-learn.org,b, scikit-learn.org/stable/modules/ensemble.html, accessed 04/09/2022

Scikit-learn.org,c,https://scikit-learn.org/stable/model_persistence.html accessed 08/09/2022

Semfionetworks.com, https://semfionetworks.com/wp-content/uploads/2021/04/wireshark_802.11_filters_-_reference_sheet.pdf, accessed 29/08/2022

spark.apache.org, https://spark.apache.org/docs/latest/ml-guide.html, accessed 28/08/2022

spark.apache.org,a,https://spark.apache.org/docs/latest/sql-ref-datatypes.html, and https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.ml.feature.VectorAssembler.html and https://spark.apache.org/docs/latest/ml-features, accessed 29/08/2022

Spark.apache.org,b, https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.classification.RandomForestClassificationModel.html, accessed 30/08/2022

Spark.apache.org, c, https://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html, accessed 08/09/2022

Stat.berkeley.edu, https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm, accessed 04/09/2022

Swan-gallery.web.cern.ch, https://swan-gallery.web.cern.ch/notebooks/apache_spark1/ML_Spark_MLlib.html, accessed 29/08/2022

weka.sourceforge.io,https://weka.sourceforge.io/doc.dev/weka/classifiers/trees/J48.html, accessed 04/09/2022

Wikipedia.org, https://en.wikipedia.org/wiki/Service_set_(802.11_network), accessed 30/08/2022

wikipedia.org, malware, https://en.wikipedia.org/wiki/TeslaCrypt and https://en.wikipedia.org/wiki/WannaCry_ransomware_attack, accessed 11/09/2022

Wiki.wireshark.org, https://wiki.wireshark.org/CaptureSetup/WLAN, accessed 28/08/2022

Wireshark.org, https://www.wireshark.org/docs/dfref/w/wlan.html, accessed 29/08/2022