INTRODUCTION TO PYTHON

A THESIS

Submitted by

Aiyyappan M R
(CB.EN.U4AIE22002)
Karthigai Selvam R
(CB.EN.U4AIE22025)
Rasha Sharma
(CB.EN.U4AIE22043)
Siddhaarth S
(CB.EN.U4AIE22051)

in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY
IN
CSE(AI)

Centre for Computational Engineering and Networking
AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE
AMRITA VISHWA VIDYAPEETHAM
COIMBATORE - 641 112 (INDIA)
DECEMBER - 2023

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE
AMRITA VISHWA VIDYAPEETHAM
COIMBATORE - 641 112

BONAFIDE CERTIFICATE

This is to certify that the thesis entitled **Intrusion detection system** submitted by Aiyyappan M R (CB.EN.U4AIE22002)**,** Karthigai Selvam R (CB.EN.U4AIE22025)**,** Rasha Sharma (CB.EN.U4AIE22043) **and** Siddhaarth S (CB.EN.U4AIE22051**)** for the award of the Degree of Bachelor of Technology in the "CSE(AI) " is a bonafide record of the work carried out by her under our guidance and supervision at Amrita School of Artificial Intelligence, Coimbatore.

Ms. Sreelakshmi K
Project Guide

Dr. K.P.Soman
Professor and Head CEN

Submitted for the university examination held on 21.12.2023

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE
AMRITA VISHWA VIDYAPEETHAM
COIMBATORE - 641 112

DECLARATION

We, Aiyyappan M R (CB.EN.U4AIE22002)**,** Karthigai Selvam R (CB.EN.U4AIE22025)**,** Rasha Sharma (CB.EN.U4AIE22043) **and** Siddhaarth S (CB.EN.U4AIE22051**)** hereby declare that this thesis entitled "Intrusion detection system", is the record of the original work done by us under the guidance of Ms. Sreelakshmi K, Assistant Professor, Centre for Computational Engineering and Networking, Amrita School of Artificial Intelligence, Coimbatore. To the best of my knowledge this work has not formed the basis for the award of any degree/diploma/ associate ship/fellowship/or a similar award to any candidate in any University.

Place: Coimbatore

Date:22-06-2023 **Signature of the Student**

Contents

Acknowledgement

We would like to express our special thanks of gratitude to our teacher (MS. SREELAKSHMI K ma'am), who gave us the golden opportunity to do this wonderful project on the topic (Intrusion detection system), which also helped us in doing a lot of Research and we came to know about so many new things. We are thankful for the opportunity given. We would also like to thank our group members, as without their cooperation, we would not have been able to complete the project within the prescribed time.

Abstract

The project focuses on combating a significant cybersecurity threat posed by Domain Generation Algorithms (DGAs), which serve as effective mediators for attackers to establish connections with command and control servers. DGAs have played a pivotal role in numerous major cyber attacks. The primary objective of our project is to enhance the detection of these malicious DGAs in real-time network traffic through the application of machine learning techniques. To achieve this, statistical and lexical attributes derived from domain names are made into a set of 48 features encompasses diverse characteristics, including the presence of dictionary words, pronounceability, entropy, and more. These features collectively provide a comprehensive representation of the underlying patterns associated with DGAs. The training dataset is curated meticulously by incorporating data from reputable sources, such as Alexa's top 1 million domains and DGA360 Netlab. In addition to the machine learning aspect, our project integrates an open-source codebase, leveraging the capabilities of Spacy for real-time DNS flagging in network traffic.

List of Figures

List of Table

LIST OF EQUATIONS,ABBRIVATIONS,SYMBOLS AND NOMENCLATURE

EQUATIONS

ABBRIVATIONS

C&C - Command and Control
DGA - Domain Generation Algorithm
IP - Internet Protocol
ML - Machine Learning
P - Probability
KNN - K-Nearest Neighbors
CART - Classification And Regression Trees
ACDT - Ant Colony Decision Tree
NP complete - nondeterministic polynomial-time complete

SYMBOLS

σ - Variance
X - Random variable
s - standard deviance
H(x) - entropy
$\bar{x}$ - mean
f - frequency
∑ - summation

1. INTRODUCTION

Malware is nothing but a piece of code which is designed to disturb, damage or gain unauthorized access to a computer system [1]. Attackers use to inject these malware families into host systems via emails, phishing websites, etc. After getting settled on a host system, the malware communicates with the attacker to get some instructions which may lead to gaining access or damaging a service, etc. The important part of communication is that the malware should know with whom it will be communicated or from whom it is getting the instructions. For this purpose, most of the malware had a hardcoded IP address. But adding a hardcoded IP address has made the security professionals job easier to perform reverse engineering on the malware executable.

The next advancement in the communication process between malware and their respective C&C is Domain Generation Algorithm (DGA). DGA implementations are based on Pseudo-random number generators where timestamp, currency exchange value between different countries, etc. are provided as the seed value to it (Fig 1). DGA may generate 'n' number of domain names in a day/week, among those a few domain names are the actual C&C address. It makes the security analyst job difficult because he/she might need to blacklist thousands of domain names every day. Even identifying the actual C&C domain name will be useful until the seed value is unchanged. But in real-time

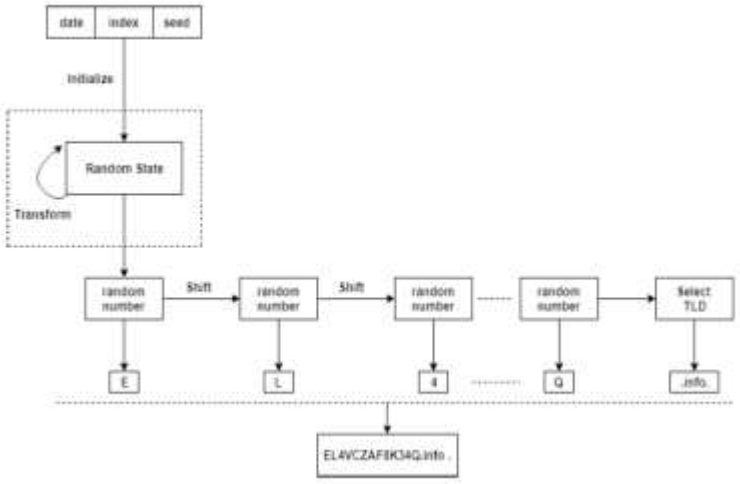

Figure 1: Character based Domain Generation algorithm structure

scenarios, attackers tend to change the seed value at regular intervals.

In this project we try to identify such DNS packets in real time network traffic. We use scapy to sniff packets real time from the network and extract the domain name from the request. Then we extract a set of 48 features from the domain name and vectorize it. Using random forest we will try to classify the Domain name. For training we use **alexa's top 1-million domain name** for benign domain names and **DGA 360** for DGA generated domain names [2].

2. DATA COLLECTION AND FEATURE EXTRACTION

We have collected static DGA malware domain names samples from 360 DGA NetLabOpenProject . There are around 19 different families of DGA based malware domains (Table1). Out of those 19 family samples , 15 families are time dependant and the remaining are time independent, which means that malware is not required to change their seed value in their lifetime (Plohmann)[3]. There are two types of DGA algorithms. One is character-based DGA where every character in the DGA domain name is independent of the other. The second type is word-based DGA where two or three random meaningful words from the English dictionary are concatenated and finally, a random TLD is added to get a domain name. In this approach, we are mainly focusing on character-based DGA that maybe time-independent or dependent.

2.1 FEATURE EXTRACTION

We have collected 50,600 samples from the domain names list out of which 25,300 are DGA based domain names and the remaining 25,300 are legitimate domain names. We extracted 48 features for each domain, include the top 15 features as specified by Jose (Selvi, 2019)[3] (Table 1) along with the other 29 features based on the character occurrence frequency of domain names (Table 1). Using these features, we try to capture

1. Random Character Sequences: DGA-generated domains often consist of random combinations of characters, making them look nonsensical and unrelated.
2. Non-Words: DGAs tend to generate domain names that are not

recognizable words or phrases in any language.

Table 1: DGA malware families

Family	*Number of domains per day*	*Time dependent*	*TLD*	*EXAMPLE*
Bamital	26	yes	org,˙ info, cz.cc, co.cc	cd8f66549913a78c5a8004c82bcf6b01.info
Conficker	250	yes	com, net, org, info, biz	gfedo.info
Cryptolocker	1000 per week	yes	com, net, biz, ru, org, co.uk, info	nvjwoofansjbh.ru
Gameover	1000	yes	com, org, biz, net	14dtuor1aubbmjhgup7915tlinc.net
Locky	6 per two days	yes	ru, pw, eu, in, yt, pm, us, fr, de, it, be, uk, nl, tf	lpfpdovapot.ru
Madmax	1 per week	yes	com, org, info, net	www.avuhtrgawe.org
Murofet	1020	yes	biz, info, org, net, com	uqiqvqylwlhutwvh.info
Necurs	2048 per three days	yes	pw, us, ug, ir, to, ga, com, net, org, biz, bit ..etc	mgvnbuxoab.su
Nymain	128	yes	com, org, biz, net, info, ru, in, xyz, pw	onrfza.info
Proslikefan	100	yes	eu, biz, se, info, com, net, org, ru, in, name	nuipkjqarq.in
Pykspa V2 Fake	800	yes	com, net, org, info	vgzmmusmr.org
Pykspa V2 Real	200 per 20 days	yes	com, net, org, info	kwukwsgcyemi.org
Qadars	**200 per week**	**yes**	**com, org, net**	**3slanc9aj4hy.net**
Ranbyus	**40**	**yes**	**in, me, cc, su, tw, net, com, pw, org**	**nslxbdyiofityx.com**
Symmi	**128 per month**	**yes**	**ddns.net**	**kuinechivuonlo.ddns.net**
Banjori	**2196 or 15372 (total)**	**no**	**Same as seed domain**	**earnestnessbiophysicalohax.com**
Dicrypt	**30 (total)**	**no**	**.com**	**mycojenxktsmozzthdv.com**
Fobber	**300 domain names (life time**	**no .**	**net**	**zzwzzqmihkfdevymi.net**
Ramit	**500 to 1000 depending on seed value**	**no**	**com**	**jrkaxdlkvhgsiyknhw.com**

3. Excessive Length: DGA domains can be longer than typical human-readable domains, containing numerous subdomains and characters.
4. Character Set: DGAs may use unusual characters, such as numbers, hyphens, or uncommon special characters, in domain names.
5. Repetition: DGA-generated domains might exhibit repetitive character sequences, which are uncommon in legitimate domains.
6. Lack of Pronounceability: DGA domains are often difficult to pronounce or remember, as they lack vowels or follow typical linguistic patterns.
7. Similarity to Hexadecimal or Binary: Some DGA-generated domains resemble hexadecimal or binary representations due to their random character sequences.
8. Dictionary Word Avoidance: DGAs avoid generating domain names that resemble common dictionary words, as this could raise suspicion.

Jose used masked domain name characters in his approach where a domain name is converted to consonant, vowel, numerical and special character pair. For eg: google.com can be converted as 'cvvccvscvc'. Later those masked strings are substituted to n-grams of 1,2 and 3 thus how Jose got c (Number of consonants), v (number of vowels) and other occurrences of masked 2-gram and 3-gram values of {cc, cv, vc, ccc, vcc, vcv, cvc } Along with those masked n-gram features, other statistical features of mean, variance, standard deviation are taken in to count for 1-gram and 2-gram occurrences.

In our approach, along with the 15 features mentioned by Jose, we considered another 29 features from Domain names which are entirely dependent on character frequencies of domain names (Born, 2010)[4] (Pereira, 2018)[5]. We considered top frequently used 1-grams, least frequently used 1-grams and the other top frequently used 2-gram,3-gram frequencies to improve to the classifier. According to statistics, characters {a,e,i,o,s,r,n,t} are the top most used 1-grams in the domain names where {f,k,w,v,x,j,z,q} are the least used 1-gram values. If we consider 2-gram features, {in,er,an,re,es,ar,on,or,te,al,st,ne,en}are the most seen ones and {ing,ion,ine,ter,lin,ent,the,ers,and,est,tio,tra,

S.N0	FEATURE	GOOGLE.COM
1.	Number of characters	10
2.	Number of Different Characters	7
3.	C	5
4.	V	4
5.	1-gram Mean	1.4285
6.	1-gram Variance	0.6190
7.	1-gram standard Deviation	0.7867
9.	2-gram mean	1
10.	2-gram variance	0
11.	2-gram standard deviation	0
12.	Cc	1
13.	Cv	3
14.	Vc	2
15.	Ccc	0
16.	Cvc	1
17.	Vcc	1
18.	vcv	0
19.	(TopFrequentusedlettersindomainnames)/(TotalNumberofcharactersindomainname)	
20.	(LeastFrequentusedlettersindomainnames)/(TotalNumberofcharactersindomainname)	
21.	shanon's entropy	
22-34	TopFrequentlyseen3-gramoccurrencesindomainnames	
35-48	TopFrequentlyseen3-gramoccurrencesindomainnames	

or,art} are the most seen 3-gram values in domain names.

Table 2: features extracted from the domain name

2.1.1 Python module

As per our course outcome we modularized all our code. Vectorize module is used for feature extraction and vectorization of the domain names. Let us see in detail about it's implementation.

2.1.2 Imports

```
from numpy import zeros
from math import log2
```

- We import zeros from numpy to store all the features as one dimensional vector which we can later add to the bootstrapped dataset
- Log2 is imported from math library to calculate the entropy of the domain names

2.1.3 Constructor

We define 5 instance variables for the class

- **Self.domain :** This variable contains the domain name as a string
- **Self.features :** it is a list of string where each entry is the feature name corresponding to the feature vector

```
class vectorize:

    '''Takes a domain name as input and generates a feature vector'''

    def __init__(self,domain:str):

        self.domain = domain
        self.features = ["NUMBER OF CHARACTERS","NUMBER OF DIFFERENT CHARACTERS","C","V","1-GRAM MEAN","1-
GRAM VARIANCE","1-GRAM STANDARD DEVIENCE","2-GRAM MEAN","2-GRAM VARIANCE","2-GRAM STANDARD
DEVIENCE","CC","VC","CV","CCC","CVC","VCC","VCV","MAX FREAQUECNY CHARTER/TOTAL NUMBER OF CHARACTER","MIN
FREAQUENCY CHARACTER/TOTAL NUMBER OF CHARACTER","SHANON
ENTROPY","IN","ER","AN","RE","ES","AR","ON","OR","TE","AL","ST","NE","EN","ING","ION","INE","TER","LIN","EN
T","THE","ERS","AND","EST","TIO","TRA","TOR","ART"]
        self.featureTable = dict.fromkeys(self.features ,[])
        self.vector = zeros((len(self.features)+1,))
        cons = 'bcdfghjklmnpqrstvwxyz'
        vow = 'aeiou'
        self.masked = ''
        for i in self.domain:

            if i in cons:

                self.masked += 'c'
            elif i in vow:
                self.masked += 'v'
            elif i.isdigit():
                self.masked += 'n'
            else:
                self.masked += 's'

        self.totalChar()
        self.differentChar()
        self.consonant()
        self.vowel()
        self.__freaquency,u = self.oneGramStat()
        self.twoGramStat()
        self.masked2gram()
        self.masked3gram()
        self.most_least_ratio()
        self.shanon_entropy()
        self.most_freaquent_twoGram_freaquency()
```

- **Self.vector :** it is a numpy array of float data type, which contains the value of each feature
- **Self.featureTable :** it is a dictionary containing feature name : value pair
- **Self.maked :** contains masked domain name (masked based on consonant or vowel)

The domain is masked with v if the character is vowel or c if the character is consonant, which is later used in multiple function. We have also called the functions defined under the class to extract the feature and add it to the vector

2.1.4 totalChar()

```
def totalChar(self)->int:

        '''Used to get the length of the word'''

        self.vector[0] = len(self.domain)
        self.featureTable[self.features[0]] = len(self.domain)
        return len(self.domain)
```

This function is used to get the length of the domain name

2.1.5 differentChar()

```
def differentChar(self)->int:
        self.vector[1] = len(set(self.domain))
        self.featureTable[self.features[1]] = self.vector[1]
        return self.vector[1]
```

It is used to get the number of unique characters present in the domain name

2.1.6 consonant() and vowel()

```
def consonant(self)->int:
        cons = 'bcdfghjklmnpqrstvwxyz'
        count = 0
        for i in self.domain:
            if i in cons:
                count += 1
        self.vector[2] = count
        self.featureTable[self.features[2]] = count
        return self.vector[2]
```

```
def vowel(self)->int:
        vow = 'aeiou'
        count = 0
        for i in self.domain:
            if i in vow:
                count += 1
        self.vector[3] = count
        self.featureTable[self.features[3]] = count
        return count
```

It is used to get the number of vowel and consonants from the domain name respectively

2.1.7 oneGramStat() and twoGramStat()

Let us consider google.com, in order to calculate 1-gram mean, variance and standard deviance we need to get the frequency of each character, in this case we have g : 2, o: 3, l : 1, e:1, . : 1, o : 1, m : 1. The formula for calculating mean is

Equation 1: Mean for n-grams

$$\overline{x} = \frac{\{\sum_{k=0}^{n} \mathrm{f_k}\}}{N}$$

So 2 + 3 + 1 + 1 + 1 + 1 + 1 divided by 7. The formula for variance and standard deviation are

Equation 2: Variance and standard devience of n-grams

$$\sigma = \frac{\{\sum_{k=0}^{n} (\overline{\mathrm{x}} - \mathrm{f_k})\text{^}2\}}{N} = s^2$$

The same formula can be obtained for 2 gram except we take the frequencies for go, oo, og, gl, le , e., .c, co, om. (code in next page)

2.1.8 masked2gram() and masked3gram()

As we discussed earlier the consonants in the domain name is substituted with c and vowels are substituted with v. We will get the frequency of some 2-grams and 3-grams that are highly corelated with the class

2-grams cc cv vc

3-grams ccc, cvc, vcc, vcv

2.1.9 most_least_ratio()

we take ratio of the least frequent character to the total number of characters and most frequent character to the total number of character

2.1.10 shanon_entropy()

Shannon entropy, named after Claude Shannon, is a measure of uncertainty or information content in a set of possible outcomes. It is often used in the context of information theory to quantify the amount of information or surprise associated with a random variable.

The formula for Shannon entropy, denoted as H(X) for a random variable X with probability distribution P, is given by:

Equation 3: Shanon's Entropy

$$\mathrm{H(x)} = \sum_{i=0}^{n} P(x_i) \times \log_2 P(x_i)$$

2.1.11 most_freaquent_twoGram_freaquency() and most_freaquent_threeGram_freaquency()

We get the frequency of most frequent 2-gram and 3-gram mentioned earlier.

```
def oneGramStat(self)->dict:
        sequence = dict.fromkeys(self.domain,0)
        for i in self.domain:
            sequence[i] += 1
        self.vector[4] = self.vector[0]/self.vector[1]
        for i in sequence.values():
            self.vector[5] += (i-self.vector[4])**2
        self.vector[5] = self.vector[5]/(len(sequence) - 1)
        self.vector[6] = self.vector[5]**0.5
        self.featureTable[self.features[4]] = self.vector[4]
        self.featureTable[self.features[5]] = self.vector[5]
        self.featureTable[self.features[6]] = self.vector[6]

        return sequence,{'mean':self.vector[4],'variance':self.vector[5],'standard variance':self.vector[6]}




    def twoGramStat(self)->dict:
        i = 0
        sequence = {}
        while i + 2 < len(self.domain):
            if self.domain[i:i+2] in sequence.keys():
                sequence[self.domain[i:i+2]] += 1
            else:
                sequence.update({self.domain[i:i+2]:1})
            i += 1
        self.vector[7] = i/len(sequence)
        for i in sequence.values():
            self.vector[8] += (i-self.vector[7])**2
        self.vector[8] = self.vector[8]/(len(sequence) - 1)
        self.vector[9] = self.vector[8]**0.5
        self.featureTable[self.features[7]] = self.vector[7]
        self.featureTable[self.features[8]] = self.vector[8]
        self.featureTable[self.features[9]] = self.vector[9]



        return {'mean':self.vector[7],'variance':self.vector[8],'standard variance':self.vector[9]}
```

```
def most_least_ratio(self)->dict:


        max_char = max(self.__freaquency, key=self.__freaquency.get)
        min_char = min(self.__freaquency,key=self.__freaquency.get)


        self.vector[17] = self.__freaquency[max_char]/len(self.domain)
        self.vector[18] = self.__freaquency[min_char]/len(self.domain)


        self.featureTable[self.features[17]] = self.vector[17]
        self.featureTable[self.features[18]] = self.vector[18]
```

```
def shanon_entropy(self)->float:


        total_chars = len(self.domain)
        probabilities = [count / total_chars for count in self.__freaquency.values()]
        entropy = -sum(p * log2(p) for p in probabilities if p > 0)
        self.vector[19] = entropy
        self.featureTable[sel
```

```
def most_freaquent_twoGram_freaquency(self)->dict:


        values = dict.fromkeys("in,er,an,re,es,ar,on,or,te,al,st,ne,en".split(','),0)
        i = 0
        while i +2 < len(self.domain):
            if self.domain[i:i+2] in values.keys():
                values[self.domain[i:i+2]] += 1
            i += 1
        for i,j in zip(range(20,32),values.values()):
            self.vector[i] = j
            self.featureTable[self.features[i]] = j


        return values
```

```
def most_freaquent_threeGram_freaquency(self)->dict:
    values = dict.fromkeys('ing,ion,ine,ter,lin,ent,the,ers,and,est,tio,tra,tor,art'.split(','),0)
    i = 0
    while i + 3 < len(self.domain):
        if self.domain[i:i+3] in values.keys():
            values[self.domain[i:i+3]] += 1
        i += 1
    for i,j in zip(range(33,46),values.values()):
        self.vector[i] = j
        self.featureTable[self.features[i]] = j
```

3. DECISION TREE

One of popular data classification methods are decision trees. Their application is particularly important due to the simple and effective (in terms of computational complexity) object classification process. They allow for simplifying the process of building an ensemble of classifiers: for example, decision forests, which are defined as sets of specific decision trees. This allows us to improve the quality of classification.

The process of constructing decision trees is a complex problem of multi-criterial decisions concerning the rules used in the optimal data split. Decision trees may easily represent complex concepts whose definitions can be described based on the feature space. In other words, trees can represent a function which maps attribute values (or values of the features) into a set of decision classes (or, in other words, class labels) which represent a permissible hypothesis (Kozak, 2017)[6]

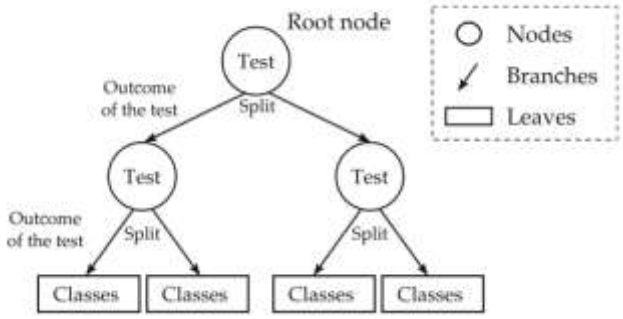

Figure 2: Binary decision tree

4. BAGGING

The bootstrap aggregating (bagging) was developed in 1994 by Breiman (L. Breiman) [7] to enhance the classification performance of ML models by combining the predictions from randomly generated training sets. The

author argued that perturbing the learning set could lead to significant modifications in the obtained predictor; hence bagging can improve accuracy (L. Breiman)[7]. Meanwhile, diversity is obtained in bagging by creating bootstrapped replicas of the input data, where several subsets of the input data are picked randomly with replacements from the original training set. Therefore, the various training sets are seen as diverse and used to train multiple base learners of the same ML algorithm.

Basically, the bagging method involves splitting the training data for each base learner using random sampling to generate *b* different subsets used to train *b* base learners. The *b* base learners are then combined using majority voting to obtain a strong classier (Song, 2022)[8]. Random forest is a popular implementation of the bagging technique Bagging enhances the performance of base learners more if the algorithm used in learning the model is unstable. An unstable algorithm significantly changes its generalization ability when slight modifications are made to its input.

Bagging focuses more on reducing the variance in the ensemble members than the bias. Therefore, bagging performs optimally when the ensemble members have high variance and low bias. An example of an unstable algorithm is the decision tree; hence, bagged decision trees usually perform better than the single decision tree. Meanwhile, k-nearest Neighbor (KNN) and naïve Bayes are examples of stable algorithms, and bagging does not perform well with these algorithms as base learners (C. Perales-González, 2022)[9].

A major advantage of bagging is that it efficiently decreases the variance without increasing bias. Other advantages of bagging include its ability to introduce diversity in the input data because of the bootstrapping approach. For large datasets, bagging has less computational time than most ML algorithms since it trains the model with a small sample size (Alelyani, 2021)[10].

Meanwhile, a limitation of bagging is that it enhances the model's accuracy without regard for interpretability. For example, if only one tree were applied as the base learner, a suitable and easy-to-interpret tree diagram would have been obtained; hence, the interpretability is neglected since bagging uses many decision trees. Also, the selected features during training are not interpretable

in bagging, so there could be situations where certain vital. (I. D. Mienye, 2022) [11]

5. IMPLEMENTATION OF DECISION TREE

The code implementation of decision tree classifier from scratch in Python is done using the CART (Classification and Regression Trees) algorithm. We will look in detail about construction of a decision tree using CART before looking at the implementation

5.1 BUILDING THE TREE

The classification and regression tree (CART) approach was developed by Breiman et al. in 1984 [7]. CART is characterized by the fact that it constructs binary trees. As the heuristic function of ACDT is based on CART, the latter algorithm is most relevant to our work. Splits are selected using the Twoing and Gini criteria. CART looks for splits that minimize the squared prediction error. The decision tree is built in accordance with a splitting rule that performs multiple splitting of the learning sample into smaller parts. Usually, a "divide and conquer" strategy is used to built the decision tree through recursive partitioning of the training set. This means we should be able to divide the problem, solve it, and then compile the results.

The process of building the decision tree begins with choosing an attribute (and a corresponding value of that attribute) for splitting the data set into subsets. Selection of the best splitting attribute is based on heuristic criteria. Construction of an optimal binary decision tree is an NP–complete problem, where an optimal tree is one that minimizes the expected number of tests required for identification of unknown objects . However, we should remember that we can also optimize the depth of decision trees, the number of nodes, or the classification quality. The problem of designing storage-efficient decision trees based on decision tables was examined in. They showed that in most cases construction of a storage-optimal decision tree is an NP-complete problem, and hence a heuristic approach to that task is necessary. Construction of an optimal decision tree may be defined as an optimization problem where, at each stage of decision making, an optimal data split is selected. (I. D. Mienye, 2022)[12]

5.2 SPLITTING CRITERIA IN THE CART ALGORITHM

There is no doubt that selection of the data division at every single node is the hardest and most complex phase of constructing a decision tree. For example, consider the CART algorithm. To evaluate the test, in most cases an impurity function i(m) (where m denotes the current node). The function enables estimation of the maximum homogeneity of the child nodes. Since the impurity function of the parent node m p is constant for every possible division a j ≤ a R j , j = 1, . . . , M (where M denotes the number of attributes, and a R j is the best possible division for attribute a j), the maximum homogeneity of the left and right descendants is determined by the maximum difference in the impurity function Δi(m) (diversity measure)

Equation 4: Impurity function

$$\Delta i(m) = i(m_p) - p_l i(m_l) - p_r i(m_r)$$

where:

Pl —probability of the object passing to node m l (left sub-tree),

Pr —probability of the object passing to node m r (right sub-tree).

The algorithm for constructing the decision tree solves the maximization problem at the phase of selecting the division for every node. It searches all possible values of attributes, which allows for finding the best possible division (the highest value of the diversity measure, i.e. of the difference in the impurity function)

Equation 5: Impurity optimization

$$argmax\ [i(m_p) - p_l(m_l) - p_r i(m_r)]$$

5.3 CODE

The code defines a Node class representing nodes in the decision tree. Each node contains information about the splitting feature, threshold, left and right child nodes, and a value if it is a leaf node.

The DecisionTree class is the main class that represents the decision tree model. It has methods for training the tree (fit), growing the tree recursively (_grow_tree), finding the best split at each node (_best_split), calculating information gain (_information_gain), splitting the dataset based on a threshold (_split), calculating entropy (_entropy), and predicting labels for new data (predict).

The fit method initializes the tree and triggers the recursive growth of the tree by calling the _grow_tree method. The _grow_tree method is responsible for building the tree structure based on the input features (X) and labels (y). The decision tree employs a set of stopping criteria: it stops growing the tree if the

maximum depth is reached (max_depth), if there is only one class in the data (n_labels == 1), or if the number of samples is below a specified threshold (min_samples_split)

.To find the best split at each node, the algorithm randomly selects a subset of features (n_features) and evaluates different thresholds for each feature to determine the one that maximizes information gain. The predict method traverses the tree for each input sample and returns the predicted class label based on the learned decision rules. Overall, the code demonstrates the construction of a basic decision tree for classification, with an emphasis on randomness in feature selection and recursive tree growth. This implementation serves as a foundation for understanding the inner workings of decision trees in machine learning.

```
import numpy as np
```

```
from collections import Counter


class Node:
    def __init__(self, feature=None, threshold=None, left=None, right=None,*,value=None):
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value

    def is_leaf_node(self):
        return self.value is not None


class DecisionTree:
    def __init__(self, min_samples_split=2, max_depth=100, n_features=None):
        self.min_samples_split=min_samples_split
        self.max_depth=max_depth
        self.n_features=n_features
        self.root=None

    def fit(self, X, y):
        self.n_features = X.shape[1] if not self.n_features else min(X.shape[1],self.n_features)
        self.root = self._grow_tree(X, y)

    def _grow_tree(self, X, y, depth=0):
```

```
        n_samples, n_feats = X.shape
        n_labels = len(np.unique(y))

        # check the stopping criteria
        if (depth>=self.max_depth or n_labels==1 or n_samples<self.min_samples_split):
            leaf_value = self._most_common_label(y)
            return Node(value=leaf_value)

        feat_idxs = np.random.choice(n_feats, self.n_features, replace=False)

        # find the best split
        best_feature, best_thresh = self._best_split(X, y, feat_idxs)

        # create child nodes
        left_idxs, right_idxs = self._split(X[:, best_feature], best_thresh)
        left = self._grow_tree(X[left_idxs, :], y[left_idxs], depth+1)
        right = self._grow_tree(X[right_idxs, :], y[right_idxs], depth+1)
        return Node(best_feature, best_thresh, left, right)


    def _best_split(self, X, y, feat_idxs):
        best_gain = -1
        split_idx, split_threshold = None, None

        for feat_idx in feat_idxs:
            X_column = X[:, feat_idx]
            thresholds = np.unique(X_column)

            for thr in thresholds:
                # calculate the information gain
                gain = self._information_gain(y, X_column, thr)

                if gain > best_gain:
                    best_gain = gain
                    split_idx = feat_idx
                    split_threshold = thr

        return split_idx, split_threshold
```

```
def _information_gain(self, y, X_column, threshold):
    # parent entropy
    parent_entropy = self._entropy(y)

    # create children
    left_idxs, right_idxs = self._split(X_column, threshold)

    if len(left_idxs) == 0 or len(right_idxs) == 0:
        return 0

    # calculate the weighted avg. entropy of children
    n = len(y)
    n_l, n_r = len(left_idxs), len(right_idxs)
    e_l, e_r = self._entropy(y[left_idxs]), self._entropy(y[right_idxs])
    child_entropy = (n_l/n) * e_l + (n_r/n) * e_r

    # calculate the IG
    information_gain = parent_entropy - child_entropy
    return information_gain

def _split(self, X_column, split_thresh):
    left_idxs = np.argwhere(X_column <= split_thresh).flatten()
    right_idxs = np.argwhere(X_column > split_thresh).flatten()
    return left_idxs, right_idxs

def _entropy(self, y):
    hist = np.bincount(y)
    ps = hist / len(y)
    return -np.sum([p * np.log(p) for p in ps if p>0])


def _most_common_label(self, y):
    counter = Counter(y)
    value = counter.most_common(1)[0][0]
    return value

def predict(self, X):
    return np.array([self._traverse_tree(x, self.root) for x in X])
```

```
def _traverse_tree(self, x, node):
    if node.is_leaf_node():
        return node.value

    if x[node.feature] <= node.threshold:
        return self._traverse_tree(x, node.left)
    return self._traverse_tree(x, node.right)
```

6.RANDOM FOREST

Random forest is an ensemble algorithm that applies the bagging technique to build multiple decision trees using boot-strapped samples. The bagging technique generates random samples with replacements from the input data and trains the decision trees from the samples (P. D. Caie, 2021)[13].

The decision tree is the main component in the random forest algorithm (Suthaharan)[14]. Meanwhile, the algorithm was first developed in 1995 by Ho using the random subspace method, and in 2001 Breiman developed an extended version of the algorithm. The random forest algorithm has been widely applied for numerous tasks because it is easy to implement, fast, and obtains excellent performance.

Two essential aspects of the random forest algorithm include the development of multiple decision trees during training and the combination of their predictions using majority voting. Since decision trees are prone to overfitting, the voting approach minimizes the random forest's chances to overt (P. D. Caie, 2021)[13].

The random forest algorithm follows the parallel ensemble learning block diagram in Figure 1, where the base learners are decision trees. Furthermore, this algorithm uses bagging and feature randomness in building a forest of uncorrelated decision trees. Meanwhile, feature randomness is achieved using the random subspace method that ensures the features are randomly selected for training each decision tree in the forest. The correlation between the decision trees that make up the forest is reduced since the trees are trained using a random feature subset rather than the whole feature set.

The random forest uses a random subset of k attributes, unlike the traditional bagging that uses all p attributes at each node of the trees. Also, the optimal partitioning rules for the

nodes are chosen from the specific random subset only. Furthermore, the random forest algorithm ensures the trees are diverse and uncorrelated by using only a subset of predictors. Furthermore, combining several uncorrelated trees reduces the model's variance, thereby enhancing the classification accuracy. (I. D. Mienye, 2022)[12]

Studies have shown that the performance of random forest is superior to the performance of the initial bagging algorithm (Krzywinski, 2017)[14]. A significant advantage of using this algorithm is its ability to solve the overfitting issue common in decision tree models. This is achieved through the random feature subset selection. Secondly, the random forest can handle missing data (Lynn)[15]. Additionally, random forests usually achieve excellent performance when the input data contains many features, i.e. high dimensional data .

Meanwhile, some limitations exist when using this algorithm; for example, more trees are needed to obtain a more accurate classification. However, too many trees would slow down the model training process. Also, as the number of decision trees increases, the random forest becomes slow in making predictions.

6.1 IMPLEMENTATION

The RandomForest class takes parameters such as the number of trees in the forest (n_trees), the maximum depth of each decision tree (max_depth), the minimum number of samples required to split an internal node (min_samples_split), and the number of features to consider when looking for the best split (n_features).

The fit method initializes an empty list of trees (self.trees) and iteratively grows decision trees. For each tree, it creates a new instance of the DecisionTree class with specified parameters and fits it to a random bootstrap sample of the input features (X) and labels (y). The bootstrap samples are generated using the _bootstrap_samples method, which randomly selects samples with replacement. The time taken to fit each tree is printed for monitoring purposes.

The _most_common_label method is used to determine the majority class in a set of labels, and it will be used later for aggregating predictions.

The predict method generates predictions for new data by collecting predictions from each tree in the forest. The predictions are obtained by calling the predict method of each

individual decision tree. The final prediction for each data point is determined by a majority vote, where the most common class label among the predictions is selected.

In summary, this code implements a random forest classifier, which leverages the diversity of multiple decision trees to improve the model's robustness and generalization performance. The randomness in both sample selection and feature selection during tree construction contributes to the ensemble's ability to handle various aspects of the data, reducing overfitting and enhancing predictive accuracy.

```
from decision_tree import DecisionTree
import numpy as np
from collections import Counter
import time

class RandomForest:
    def __init__(self, n_trees=10, max_depth=10, min_samples_split=2, n_feature=None):
        self.n_trees = n_trees
        self.max_depth=max_depth
        self.min_samples_split=min_samples_split
        self.n_features=n_feature
        self.trees = []

    def fit(self, X, y):
        self.trees = []
        for _ in range(self.n_trees):
            start = time.time()
            tree = DecisionTree(max_depth=self.max_depth,
                                min_samples_split=self.min_samples_split,
                                n_features=self.n_features)
            X_sample, y_sample = self._bootstrap_samples(X, y)
            tree.fit(X_sample, y_sample)
            self.trees.append(tree)
            end = time.time()
            print(f"{_} --> {end-start}")
```

```
    def _bootstrap_samples(self, X, y):
        n_samples = X.shape[0]
        idxs = np.random.choice(n_samples, n_samples, replace=True)
        return X[idxs], y[idxs]

    def _most_common_label(self, y):
        counter = Counter(y)
        most_common = counter.most_common(1)[0][0]
        return most_common

    def predict(self, X):
        predictions = np.array([tree.predict(X) for tree in self.trees])
        tree_preds = np.swapaxes(predictions, 0, 1)
        predictions = np.array([self._most_common_label(pred) for pred in tree_preds])
        return predictions
```

7.TRAINING

```
import sys
sys.path.append('../DGA_detection/feature_extraction')
sys.path.append('../ml_algorithm')
import random_forest
import vectorize
import numpy as np
import pandas as pd
import re
from random import random
from sklearn import datasets
from sklearn.model_selection import train_test_split
import joblib

if __name__ == '__main__':
    with open('../data/DGA/begnin/top-1m-domain.csv') as cl0, open('../data/DGA/dgaGenerated/360_dga.txt')
as cl1:
        malicious,begnin = [],[]
        for i,j in zip(cl1,cl0):
            val = re.search(r'\w+(\.\w+)+',i)
            malicious.append(val.group())
            begnin.append(j.split(',')[1])

    i = 0
```

```
    features = np.zeros((60000,48))
    classs = np.zeros((60000,),dtype='int64')

    while i < 30000:
        mal = vectorize.vectorize(malicious[i])
        beg = vectorize.vectorize(begnin[i])
        features[2*i] = mal.vector
        features[2*i+1] = beg.vector
        classs[2*i] = 1
        i += 1
df = pd.DataFrame(features)




X_train, X_test, y_train, y_test = train_test_split(
    df, classs, test_size=0.2, random_state=1234
)
print(type(X_train),X_train.shape,type(y_train),y_train.shape)

def accuracy(y_true, y_pred):
    accuracy = np.sum(y_true == y_pred) / len(y_true)
    return accuracy

clf = random_forest.RandomForest(n_trees=20)
clf.fit(np.array(X_train), np.array(y_train))
predictions = clf.predict(np.array(X_test))

acc =  accuracy(y_test, predictions)
print(acc)
joblib.dump(clf,'save.joblib')
```

Data Preparation:

Two files, top-1m-domain.csv (representing benign domains) and 360_dga.txt (representing malicious domains), are read simultaneously line by line.

Regular expressions are used to extract domain names from each line.

Malicious domain names and their corresponding benign counterparts are stored

in the malicious and begnin lists, respectively.

Feature Extraction:

A loop iterates over the first 30,000 domain pairs from both malicious and benign datasets.

For each domain, the vectorize function from the vectorize module is called to obtain a feature vector. This function is assumed to be defined in the vectorize module.

The feature vectors for malicious and benign domains are concatenated to form the feature matrix features.

Class labels are assigned to the classs array, with a value of 1 indicating a malicious domain.

DataFrame Creation:

A pandas DataFrame df is created using the feature matrix features.

Train-Test Split:

The data is split into training and testing sets using the train_test_split function from scikit-learn. 80% of the data is used for training (X_train and y_train), and 20% is used for testing (X_test and y_test).

Model Training:

An instance of the RandomForest class is created with 20 trees.

The fit method of the random forest classifier is called with the training data (X_train and y_train).

Prediction and Evaluation:

The trained model is used to predict the labels for the test data (X_test).

The accuracy function is defined to calculate the accuracy of the model by comparing the predicted labels with the true labels (y_test). The accuracy is printed, showing the proportion of correctly predicted labels.

Model Saving:

The trained random forest model (clf) is saved using the joblib.dump function.

7.1 MODEL EVALUTION

Model parameters :

Number of trees = 20
Number of features = 48
Max depth = 10
Minimum sample split = 2

```
0 --> 22.735360860824585
1 --> 22.7080180644989
2 --> 24.11452579498291
3 --> 21.520118474960327
4 --> 21.931385040283203
5 --> 21.171620845794678
6 --> 21.59576964378357
7 --> 20.824811935424805
8 --> 19.47300624847412
9 --> 21.517483949661255
10 --> 20.583974361419678
11 --> 21.54222321510315
12 --> 21.241304397583008
13 --> 20.847256660461426
14 --> 20.9639835357666
15 --> 20.27738308906555
16 --> 19.24436092376709
17 --> 20.79279398918152
18 --> 21.40284252166748
19 --> 21.397450923919678
0.9608333333333333
```

Figure 3: Time taken to create individual trees and accuracy of the trained model

8.PUTTING EVERYTHING TOGETHER

We take an open source python code that is used for capturing DNS packet using scapy to get the data from Realtime network traffic. We edit the open source code to extract only the domain name and use our trained model to classify the DNS from real time traffic as safe and potentially malicious DNS

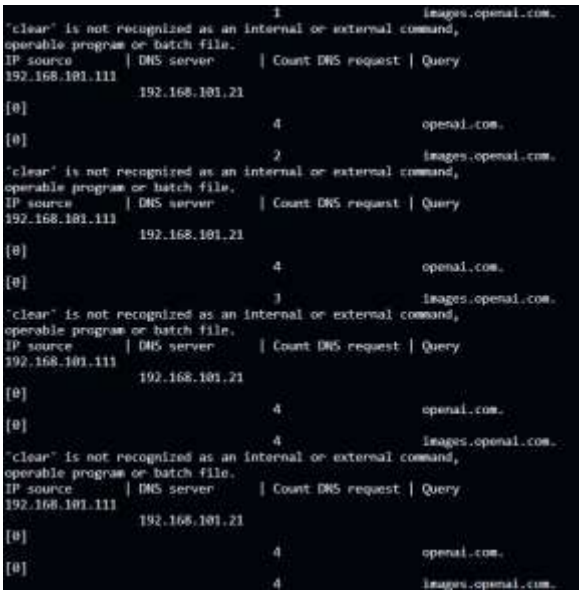

Figure 4: When openai.com is searched in the browser while the python code is running. this output is generated where 0 represents that it is a safe DNS

Figure 5: When we send a DGA generated DNS query attached to the openai request our model gives output as 1 denoting a potential threat

Bibliography

[1]MargaretRouse. (2019, April). *Malware Definition by SearchSecurity .techtarget*. Retrieved from techtarget.com: https://searchsecurity.techtarget.com/definition/malware

[2]P.Mohan Ananda, T. K. (2020). An Ensemble Approach For Algorithmically Generated Domain. *Procedia Computer science*.

[3] Selvi, J. R.-O. (2019). Detection of algorithmically generated malicious domain names using masked. *Expert Systems with Applications*, 156-163.

[4] Born, K. a. (2010). Detecting dns tunnels using character frequency analysis . *arXiv preprint arXiv:*.

[5] Pereira, M. e. (2018). Dictionary extraction and detection of algorithmically generated domain names in passive DNS traffic. *Symposium on Research in Attacks, Intrusions, and Defenses.*

[6] Kozak, J. (2017). *Decision Tree and.* Poland: Springer.

[7] L. Breiman. (n.d.). `Bagging predictors.

[8] Song, Q.-F. L.-M. (2022). High-performance concrete strength predic1441. *Construct. Building Mater*

[9]C. Perales-González, F. F.-N.-R. (2022). *IEEE* .

[10]Alelyani, S. (2021). Stable bagging feature selection on medical data. *Big data*.

[11] I. D. Mienye, Y. S. (2022). Survey of Ensemble Learning: Concepts, Algorithms, Applications, and Prospects. *IEEE*.

[12] Plohmann, D. a. (n.d.). A comprehensive measurement study of domain generating malware. *25th USENIX Security Symposium (USENIX)*.

[13]P. D. Caie, N. D. (2021). Precision medicine in digital pathology via image analysis and machine learning. *Artificial Intelligence and Deep Learning in Pathology*.

[14]Suthaharan, S. (n.d.). *Handbook Statist.*

[15]Krzywinski, N. A. (2017). Ensemble methods: Bagging and random forest. *Nature Methods*.

[16]Lynn, S. H. (n.d.). Accuracy of random-forest-based imputation of missing data in the presence of non-normality, non-linearity, and interaction. *BMC Med. Res. Methodol.*

.