# Domain Generation Algorithm Detection Using Machine Learning Methods


**Moran Baruch and Gil David**

**Abstract** A botnet is a network of private computers infected with malicious software and controlled as a group without the knowledge of the owners. Botnets are used by cybercriminals for various malicious activities, such as stealing sensitive data, sending spam, launching Distributed Denial of Service (DDoS) attacks, etc. A Command and Control (C&C) server sends commands to the compromised hosts to execute those malicious activities. In order to avoid detection, recent botnets such as Conficker, Zeus, and Cryptolocker apply a technique called *Domain-Fluxing* or *Domain Name Generation Algorithms* (DGA), in which the infected bot periodically generates and tries to resolve a large number of pseudorandom domain names until one of them is resolved by the DNS server. In this paper, we survey different machine learning methods for detecting such DGAs by analyzing only the alphanumeric characteristics of the domain names in the network. We also propose unsupervised models and evaluate their performance while comparing them with existing supervised models used in previous researches in this field. The proposed unsupervised methods achieve better results than the compared supervised techniques, while detecting zero-day DGAs.

## 1 Introduction

A botnet is a network of private computers infected with malicious software and controlled as a group without knowledge of the owners. Cyberattackers use botnets for various malicious activities, such as stealing sensitive data, sending spam, launching Distributed Denial of Service (DDoS) attacks, etc. (Plohnmann et al. 2011). Botnets are controlled by a centralized Command and Control (C&C) server, which sends

M. Baruch (✉) · G. David
University of Jyvaskyla, Jyvaskyla, Finland
e-mail: moran.baruch@biu.ac.il

G. David
e-mail: gil.david@jyu.fi



M. Lehto and P. Neittaanmäki (eds.), *Cyber Security: Power and Technology*,
Intelligent Systems, Control and Automation: Science and Engineering 93,
https://doi.org/10.1007/978-3-319-75307-2_9

commands to its bots. There are several static mechanisms for blocking the communication between the bots and the C&C server. For example, by blacklisting the C&C server IP and domain, the bots are unable to establish command and control traffic with the server. In order to bypass such mechanisms, botnet operators are using more sophisticated techniques to hide the C&C server's fingerprints.

In this work, we will focus on *Domain-Fluxing* or *Domain Name Generation Algorithms* (DGA), in which the infected bot tries to communicate with its C&C server by periodically generating and trying to resolve a large number of pseudorandom domain names until one of them succeeds. Meanwhile, the attacker registers only one or a few generated domain names. The large number of potential rendezvous points makes it difficult for security vendors to pre-register and block the domain names. This technique is also popular among spammers for the purpose of avoiding detection. For instance, spammers advertise randomly generated domain names in their spam emails. Therefore, it is harder for security engines to detect and take down these domains with traditional techniques, such as blacklists of regular expressions representing malicious domain names. Moreover, even if one of the domains has been blocked, the attackers can register another domain in their domain list.

Several tactics have been proposed for detecting botnets. Most of them analyze network traffic to detect botnets by studying correlated activities between various clients. Applying these strategies to all networks is unfeasible and very performance-intensive. Other researchers suggest using reverse engineering (Rahimian et al. 2014) to understand the underlying pattern of the DGA. However, this technique is both resource- and time-consuming, which makes it impractical for real-time detection and protection.

A more practical approach would be to analyze DNS traffic to detect if domain names have been generated algorithmically. Once detected, the network administrator can disconnect bots from their C&C server by blocking these DNS queries.

Our methodology for detecting algorithmically generated domains focuses on *Machine Learning* and specifically *Anomaly Detection* methods, and is based on the assumption that botnet owners, who build and operate DGA botnets, have some constraints when generating the domain names. On the one hand, they need to avoid using real words in their hostnames, because these words are more likely already to be registered as real domain names. Hence, botnet developers would prefer to generate many domain names that are unpredictable to anyone, especially to security vendors. On the other hand, registering the C&C server under a purely random domain name will satisfy this need, but this name is unpredictable to the bots.

As a result, pseudorandom domain names are generated, which means that, as in encryption, they choose a seed that is known only to the C&C server and its bots. These constraints cause the DGA domains to significantly differ from human-generated domains. We would like to exploit this differentiation.

### *1.1 Related Campaigns*

Notable examples of DGA bots are Conficker, Kraken, Torpig, Murofet, and Srizbi, to name a few. Each one of them uses a different algorithm for generating random domain names while using a different random seeding.

**Conficker-A, B** (Porras et al. 2009): this worm infected millions of IP addresses from 206 countries. It causes various Windows operating systems (OS) to execute an arbitrary code segment without authentication. It generates 250 random domains every couple of hours. The randomizing process is seeded with the current Coordinated Universal Time (UTC) system date. In order to sync all the different bots to the same date and time, an HTTP_QUERY_DATE query is sent to one of six legitimate search engines, such as "*yahoo.com*" and "*answers.com*". This way, all clients try to contact the same set of domain names every day. Each domain contains four to ten characters, while the Top-Level Domain (TLD) is randomly selected from a predefined list of 110 TLDs. **Conficker-C** (Fitzgibbon and Wood 2009): this is a modified version, in which the number of randomly generated domains increases to 50,000 a day, while the server randomly chooses only 500 of them to register every 24 h. Some examples of the Conficker hostnames include *"viinrfdg.com", "qwnydyb.cc",* and *"wqqfbutswyf.info"*.

**Kraken** (**a.k.a. Oderoor and Bobax**) (Royal 2008): this uses social engineering techniques to infect machines, mostly through the distribution of a huge amount of spam emails. In order to communicate with the C&C server, the malware generates a random string of between six and eleven characters, and combines it with one of the second-level domains of the Dynamic DNS (DDNS) providers: *"yi.org", "dydns.org", "mooo.com"*, or "*dynserv.com*".

To explain the meaning of a second-level domain, we take a look at the following domain name as an example: *"mail.google.com"*. We refer to "*google.com*" as the second-level domain, and "*google*" as the second-level domain *label*.

Examples of such domains are "*quowesuqbbb.mooo.com*" and "*bayqvt.dyndns.org*".

**Torpig** (**a.k.a. Sinowal and Mebroot**) (Stone-Gross et al. 2009): this targets computers with Windows OS. It is designed to collect sensitive personal and corporate data, such as bank account and credit card data. In 10 days, Torpig retrieved 180,000 infections and recorded more than 70 GB of data from the victims. In order to generate a random domain, it uses the Twitter API to request trending topics (Sinegubko 2009) and takes the second character of the fifth most popular Twitter search on the calculated date in two specific hours on the same day as a seed. The algorithm rotates between the two generated domains twice a day. In a particular month, generated domain names will end with the same three characters. For example, domains generated on December will end with *"twe"*: *"wghfqlmw**twe**.com", "sfgtilbs**twe**.com",* etc.

**Murofet** (Shevchenko 2010): this is used to infect computer files and steal passwords. The virus attempts to download arbitrary executable files from the domains generated. The seed of the generated domains is calculated by taking the most sig-

nificant bytes from the current year, month, and day, and processing them into one seed value. This process repeats 800 times a day, and each time, the seed is incremented by one. TLD is taken from the set: {*biz, com, info, net, org, ru*}. Examples are *"zlcssltmlwknnk.com"* and *"kvovtsxogyqyvro.net"*.

**Zeus** (Dennis et al. 2013): this is a family of credential-stealing trojans. It generates domain names by calculating the MD5 hash function over the sequence of the year, month, day, and domain index. The DGAs concatenate one of the TLDs mentioned in Murofet. Every week a bot generates a list of 1000 unique domains and tries to connect to each of them until it receives a response. Example domains are *"aqm5ar1sa72cwhien6614rlwrr.com"* and *"13rmowp60nkcw1d0m2ceyatb1 f.com"*.

**Srizbi** (Wolf 2008): this sends spam emails from infected computers, which contain fake videos of celebrities and include a link to the malware kit. It takes the current date and transforms it in various ways in order to generate a seed for producing random domain names. The domain names include only the first 14 letters on the keyboard (*q, w, e, r…*). For example, *"yrytdyip.com"* and "*ododgoqs.com"*.

**CryptoLocker**: this is a family of ransomware Trojans (Yazdi 2014). It is designed to encrypt the victim's files on the computer and demand a ransom from the victim in order to recover them. The seed for generating domains is based on the current date (day, month, and year) (Panda Security 2015). The domains include 19–20 characters. The bot tries iteratively to reach the C&C server using the generated domains, and once it succeeds, it obtains the public key from the server. After retrieving the public key, it starts encrypting files on the computer. *"axoxypwociacbjw.com"* and *"jsjvitqhvvdnjlfn.com"* are examples of such domain names.

## *1.2 Paper Scope*

In this paper, we survey various methods of anomaly detection using supervised machine learning techniques and suggest several unsupervised approaches, which are based on DGA domain expertise. We focus on alphanumeric characteristics of the domain names and disregard other data, e.g., the IP address. We evaluate the performance and accuracy of the suggested techniques using a real dataset that contains four DGA botnets and compare them to the state-of-the-art methods.

## *1.3 Contributions*

We survey the existing machine learning based anomaly detection methods for DGA detection. We evaluate only techniques that are based on alphanumeric features of the domain names, without having to analyze any additional data, such as the IP address, the response packet to the DNS request, etc. This is more efficient and easy to implement than techniques that require this extra data.

Second, we introduce *unsupervised* anomaly detection methods for the first time in this field (to the best of our knowledge), which means that only a small portion of the data is required to be manually labeled in order to classify the domains as legitimate or malicious. Another important advantage of using unsupervised learning is the ability to detect DGAs that did not exist in the training data, as described in Sect. 5.4.1. We aim to prove the applicability of our unsupervised techniques by showing that the unsupervised *K-Nearest Neighbors* (KNN) method outperforms the previously used supervised ones.

### *1.4 Structure of the Paper*

This paper is organized as follows: Sect. 2 surveys related work in this field. In Sect. 3, we present some supervised techniques for detecting DGA, and in Sect. 4, we suggest unsupervised techniques for this task. Section 5 presents experimental results and Sect. 6 summarizes our work and discusses the results and limitations, as well as computation complexity and potential errors. To conclude, we suggest future work to improve our research.

## 2 Related Work

Mcgrath and Gupta (2008) examined several network features, such as IP addresses, "whois" records, and lexical features of URLs and domains classified to phishing and non-phishing websites. They observed that each class has a different alphabet distribution. Their conclusion was that malicious domain names are shorter than non-malicious domain names, mostly use fewer vowels, have a significant difference in alphabet distribution probability, and have more unique characters. Their motivation was to find useful heuristics to filter phishing-related emails and identify suspicious domain registrations.

**Cisco** (Namazifar and Pan 2015) developed an initial component of their DGA detection system. They presented a language-based algorithm for detecting randomly generated strings. The algorithm assigns a randomness score to each domain in order to decide whether it is algorithmically generated or not. To estimate this score, they first built a large set of dictionaries encompassing various languages, e.g., English, French, Chinese, etc., and also included English names, Scrabble words, Alexa 1000 domain names, and texting acronyms. Those dictionaries are used to find meaningful sequences in the domain names, which are unlikely to appear in a DGA-generated name.

For each inspected domain, all substrings are extracted and several features are calculated, such as the number of substrings appearing in the dictionaries, their corresponding length, and the number of different languages used. From the extracted

features, they built a linear model that calculates the randomness score. They showed a false negative rate between 0 and 2% on nine different DGAs.

Sandeep et al. (2010) presented a methodology for detecting domain fluxes by looking at the distribution of alphanumeric characters, both unigrams and bigrams. This methodology is based on the assumption that there is a significant difference between human-generated and algorithm-generated domains in terms of the distribution of alphanumeric characters. They first grouped together DNS queries via connected components, i.e., they share the same second-level domain, they are mapped to the same IP address, etc. Then, for each group, they computed KL divergence, Jaccard index, and edit distance (see Sects. 3.1–3.3) on unigrams and bigrams within the set of domain names. *The n-gram* of a string is a group of substrings of size n, which are extracted using a sliding window of length n from the beginning of the string to the end. For example, unigrams of the word "domain" would be *{"d","o","m","a","i","n"}*, and bigrams of that word are *{"do","om","ma","in"}*.

They evaluated their experiments on one day of network traffic of Tier-1 ISP in the Asia and South America dataset and detected Conficker, as well as some other unknown botnets. They claim to have a 100% accuracy rate with less than 6% false positives.

There are some weaknesses in their approach. Grouping domains by their second-level domain may give rise to false positives, since some legitimate domains have the same second-level domains as malicious ones. Furthermore, in grouping together domains mapped to the same IP address, one might group together many domains, both legitimate and malicious, that belong to the same IP. For example, Google Sites is a service that hosts many domains using the same second-level domain and IP, some of which might be malicious.

Another weakness is that the metrics suggested in their paper require a minimal number of domains in each group to achieve accurate results. However, many groups in real life data do not satisfy this requirement, resulting in many domains that would not be classified, since their groups were discarded.

One of the metrics applied for detection is the Jaccard index (JI) between a set of legitimate or malicious components and a test distribution. Here, the JI is implemented on sets of bigrams on a hostname or domain label. As long as the size of the legitimate sets of bigrams gets larger, a higher accuracy will be achieved. However, storing all the bigrams is both memory- and CPU-consuming.

Antonakakis et al. (2012) proposed a detection system called Pleiades. They assumed that the response to DGA queries will mainly be Non-Existent Domain (NX-Domain) and that infected machines in the same network with the same DGAs would receive a similar NX-Domain error response. They combined clustering and classification techniques to group together similar domains by the domain name string pattern and then define the DGA they belong to. After discovering the domains generated by the same DGA, they developed a method to resolve the C&C server. Over a period of 15 months, they found 12 DGAs, of which only half of them were known. They presented true positive rates of 95–99% and false-positive rates of 0.1–0.7%.

However, their system has some limitations. Their algorithm requires mapping of traffic data to the corresponding host, while in most organizations the hosts are connected to the internet via NAT, Firewall, DNS server, etc., and hence one IP might represent many different hosts. Also, most hosts use DHCP, so one host might have different IPs in a period of traffic investigation.

Detecting the C&C server would be very difficult and inaccurate in a case when the botnets use a combination of both domain-fluxing and IP-fluxing networks, in which the C&C domains point to a set of IP addresses. Even after the C&C server was detected, their system could not block all the IPs in real time.

Their proposed detection system uses a classification algorithm that is based on calculating the similarity between the inspected DGA and the existing DGAs using a machine learning technique. Therefore, when a new botnet generates an NX-Domain traffic pattern that is similar to an existing DGA, it might be wrongly classified as the existing DGA. Moreover, if a new variation of an existing DGA that has a different traffic pattern is tested against the existing DGAs, it would be wrongly classified as a new DGA.

Finally, in order to confuse the proposed system, malware developers can use the same DGA twice; the first one with one seed to generate the domains and the second one with a different seed to add noise with fake domains and thus receive a large amount of NX-Domain errors. During the learning phase, the noisy and the real NX-Domain would be clustered together, resulting in lower accuracy when learning the model for the domain name.

Abu-Alia (2015) presented a set of techniques for detecting DGAs without grouping together domains prior to classifying them. The techniques used are based on the areas of machine learning and statistical learning. They extracted alphanumeric features from the domain names and compared the performance between three machine learning classifiers: Support Vector Machine (SVM), Neural Network (NN), and Naïve Bayes. During the training phase, they generated traffic data consisting of previously known DGAs and a large set of legitimate domains through a process that determines the domain name associated with a given IPv4 address. This process is called reverse DNS lookup.

The extracted features per domain are detailed below:

*Number of vowels in the domain name*: it is expected that the number of vowels in legitimate domains is higher than that in malicious domains. For each domain, the ratio between the number of vowels and the domain length is calculated.

*Number of unique alphanumeric characters*: assume that random domain names will contain more unique characters.

*Number of dictionary words*: they obtained a list of words that appear in the English dictionary, and for every domain, they checked how many dictionary words it contains. They calculated the ratio between the number of dictionary words and the domain length.

*The ratio between the numbers of dictionary words appears in the domain name and the length of the domain name*

*Average Jaccard index on bigrams and trigrams*: the Jaccard index is between a test domain and every legitimate domain. Average is taken over the Jaccard index results. The Jaccard index metric is described in Sect. 3.2.

*Number of dictionary non-existent bigrams and trigrams*: they obtained a list of bigrams and a list of trigrams from the database of legitimate domain names. Then, they obtained two lists of all possible alphanumeric bigrams and trigrams and filtered the legitimate bigrams out of them. The presence in a particular domain of bigrams and trigrams that do not appear in the legitimate domains might imply that it is malicious.

Their experiments revealed that the SVM classifier showed the best results, with 5.97% false positives and 0.12% false negatives. The neural network classifier false positive rate was 2.7% and the false negative rate was 6.1%. This method suffers from the following drawbacks:

The neural network and the SVM are computationally intensive. They also have examined the Naïve Bayes classifier, which is better in regard to runtime, but its accuracy is much worse.

Another complexity limitation is the calculation of the Jaccard index for each new test case. Each test case is compared to each piece of labeled data, resulting in the need to compute the Jaccard metric 100,000 (the size of their dataset) times.

They mentioned in the paper that a small number of features had been extracted. In real-world scenarios, more features will probably be required. But again, adding more lexical features will result in more computational costs.

Nguyen et al. (2015) presented a large-scale botnet detection system that is dedicated to DGA-based botnets. This system has the ability to detect centralized architecture botnets, as well as their bots, by analyzing DNS traffic logs of the computer network. They claim that their system is able to detect new editions of a botnet, which is hard to find through reverse malware binaries. Their method is based on a Big Data platform and uses collaborative filtering and density-based clustering. Their algorithm relies on the similarity in characteristic distribution of domain names to remove noise and group similar domains. Their technique yielded a false positive rate of 18% and a false negative rate of 22%.

Their system architecture is focused only on botnets with centralized architecture. Therefore, their algorithm cannot detect Peer-to-Peer (P2P) botnets such as Zeus. Another limitation is that prior to analyzing the domains, they have to capture the entire DNS traffic from users' logs. Ideally, we would like to have a technique that only takes the domain names as input with no extra information.

## 3 DGA Detection Using Supervised Learning

In this section, we describe various methods for DGA detection, while in Sect. 5 we present their performance evaluation. These methods are considered to be supervised learning, since during the training phase, labeled data is required to build the models.

### 3.1 KL Divergence

Sandeep et al. (2010) suggested using the Kullback–Leibler (KL) divergence, a non-symmetric metric that is used to calculate the distance between probability P and probability Q in the following way (Kullback and Leibler 1951):

$$D_{KL}(P||Q) = \sum_{i=1}^{n} P(i) \log \frac{P(i)}{Q(i)}, \tag{2.1}$$

where n is the number of possible values for a discrete random variable. P represents the test distribution and Q represents the base distribution. In order to avoid singular probabilities, we use the symmetric form of this metric, which is given by

$$D_{sym} = \frac{1}{2} \cdot (D_{KL}(P \,\|\, Q) + D_{KL}(Q \,\|\, P)). \tag{2.2}$$

Assuming $g$ is a non-malicious domain name probability distribution given by unigrams or bigrams and $b$ is a malicious domain probability distribution of these features, the anomaly score for a test domain with distribution $d$ is calculated as

$$score = D_{sym}(dg) - D_{sym}(db). \tag{2.3}$$

If the anomaly score is greater than zero, the domain is classified as malicious; otherwise, it is classified as normal. In our experiments, we calculated KL distance on bigrams and unigrams of domain names.

The testing domain is compared both to the malicious distribution and to the legitimate distribution, since the testing domain might be different from both of the distributions. Therefore, combining the KL divergence scores helps to scale the results in a better way.

### 3.2 Jaccard Index

The Jaccard index (JI), which is also suggested in Sandeep et al. (2010), is also called the *Jaccard similarity coefficient* (Jaccard 1901). It is used to determine the similarity between two random variables. It is defined as the ratio between the size of the intersection of the samples over the size of their union. In our context, in which A and B are two sets of bigrams of two different hostnames, JI is calculated as follows:

$$JI = \frac{|A \cap B|}{|A \cup B|}, \; 0 \leq JI \leq 1 .$$

A low result implies that the bigrams of the two domains are different.

For example,

A = "thequ*ick*brownf*oxj*umps*ov*erthelazydog", number of bigrams = 35.
B = "*ickoxjsov*", number of bigrams = 8.

$$|A \cap B| = 6,\ |A \cup B| = 35 + 8 - 6 = 37 \rightarrow JI = 6/37 = 0.16.$$

Given a set $G$, which contains sets of bigrams of non-malicious domain name $g_i$, $g_i \in G$, $1 \leq i \leq |G|$, and a second set $B$, which contains sets of bigrams of malicious domain names $b_i \in B$, $1 \leq i \leq |B|$, a tested domain $d$ is classified by calculating $M_g = \frac{\sum_{i=1}^{|G|} JI(d,g_i)}{|G|}$ and $M_b = \frac{\sum_{i=1}^{|B|} JI(d,b_i)}{|B|}$, $b_i \in B$,

where $JI(d, g_i)$, $g_i \in G$ is the JI of the testing domain bigrams $d$ and a non-malicious domain bigram $g_i$, and $JI(d, b_i)$, $b_i \in B$ is the JI of $d$ and a malicious domain bigram $b_i$. $M_b$ and $M_g$ are the average of the JI results of the malicious and non-malicious, respectively. In this metric, as the JI score gets smaller, the domain is considered that much more malicious.

At last, the final score of JI on bigrams is obtained by

$$score = M_b - M_g.$$

As explained in KL divergence, the combination of distances to the malicious and the legitimate domains is used in order to scale the results. This result is the anomaly score of each domain, since we expect bigrams occurring in randomized (malicious) hostnames to be mostly different when compared with the set of non-malicious bigrams.

The input data used in Sects. 3.3 and 3.4, is a feature vector whose features are described in Sect. 2 in the description of Abu-Alia's research (Plohnmann et al. 2011).

### *3.3 Support Vector Machine (SVM)*

Abu-Alia (2015) used an SVM machine learning model for binary classification (Boser et al. 1992). Based on a training labeled dataset, it generates a model that assigns test samples into positive and negative categories. In our domain, the positive category is the malicious domains and the negative is the legitimate ones. The samples in this model are represented by points in space, with a clear wide gap between points belonging to the different categories. A new tested sample is determined to belong to one category or the other by checking on which side of the gap they lie. Mathematically, given a training dataset of $n$ points $(\vec{x}_i, y_i)$, $1 \leq i \leq n$, $\vec{x}_i \in \mathbb{R}^n$, and $y_i \in \{-1, 1\}$, where $\vec{x}_i$ represents a feature vector and $y_i$ is the corresponding label. r represents a vector notation. Our goal is to find the maximum margin hyperplane with $(n-1)$ dimensions that gives the best separation between the group of feature

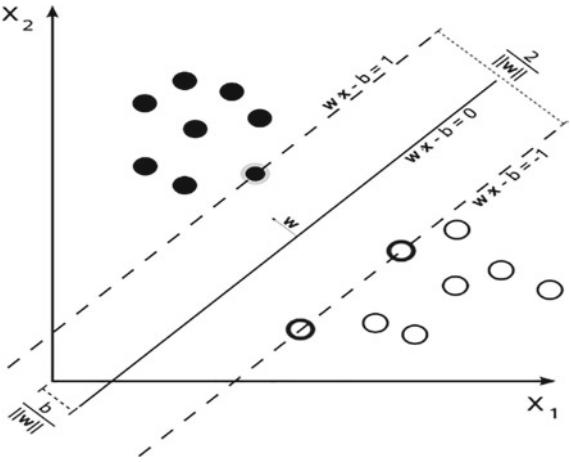


**Fig. 1** SVM classification model with samples from two classes

vectors belonging to label "+1" from the group of feature vectors belonging to label "−1", in order to minimize the *generalization error*, a measure that determines how accurately an algorithm is able to predict the value of a new unseen sample. The margins are considered from the nearest training points belonging to one class to the nearest training points belonging to the second class (the support vectors). More formally, a hyperplane can be described as a set of points $\vec{x}$: $\vec{w} \cdot \vec{x} - b = 0$ where $\vec{w}$ is the normal vector to the hyperplane and $b$ is the bias. Then, we need to find the suitable $\vec{w}$ and $b$ parameters satisfying the optimization problem:

$$\underset{\vec{w},b}{Min} ||\vec{w}|| \; s.t: \; y_i(\vec{w} \cdot \vec{x}_i + b) \geq 1, \; i = 1, \ldots, n \quad . \tag{2.4}$$

The classification equation of a new point is

$$\vec{x} \mapsto sign(\vec{w} \cdot \vec{x} - b) \tag{2.5}$$

Figure 1 illustrates an SVM classification model.

In a case in which the training data is noisy, SVM might not find a clear margin between the two sets of points. In that case, SVM with *soft margins can be used*. Soft margins means that some training points are "allowed" to lie on the wrong side of the margin (+1 point on the −1 side and vice versa), but a penalty is added to these points. We use a *hinge loss function*: $max(0, 1 - y_i(\vec{w} \cdot \vec{x}_i - b))$. Using this function, training points $\vec{x}_i$ that are misclassified will cause this function to result in a higher value, while well-classified points will get a 0 penalty.

The updated optimization problem is now minimizing the equation:

$$\left[\frac{1}{n}\sum_{i=1}^{n} max(0, 1 - y_i(\vec{w} \cdot \vec{x}_i - b))\right] + \lambda||\vec{w}||^2 \quad . \tag{2.6}$$

$\lambda$ is the trade-off between increasing the size of the margins and the size of penalty allowed for misclassified points. Usually, this parameter is drawn from the understanding of how noisy the ground-truth data is. For a small enough $\lambda$, the SVM soft margins model behaves the same as that of *hard margins,* whose optimization problem was described in Eq. (2.4) in this section.

Often, a linear classifier, with or without soft margins, is incapable of separating the data. To overcome this issue, SVM maps the feature vectors into higher dimensional space using a transformation function $\phi(\vec{x})$. Function $k\left(\vec{x}, \vec{x}'\right) = \phi(\vec{x})^T \phi(\vec{x}')$ is called the *kernel function*. The kernel function is used to measure the similarity between two feature vectors, $\vec{x}$ and $\vec{x}'$. This mapping into higher dimensional space is called the *kernel trick*, and it is applied by replacing every dot product in the algorithm with the kernel function. Thus, the decision function for a new point $\vec{z}$ is changed from Eq. (2.5) to

$$\vec{z} \mapsto sign(\vec{w} \cdot \varphi(\vec{z}) - b). \tag{2.7}$$

In Sect. 5.3.3, we describe the kernel function chosen for our experiments.

The anomaly score is obtained by the probability that a tested feature vector is a member of a malicious class.

## *3.4 Neural Network*

Neural network, also known as *Artificial Neural Network* (ANN), is another method tested in (Abu-Alia 2015). ANN is a machine learning model inspired by the human brain (Caudill 1989), which is based on many simple processing elements called "neurons" or "nodes", each calculating a simple function. The neurons are highly interconnected in a layered model. Typically, the neurons are organized such that the first layer is the input layer, the last one is the output layer, and the layers between them are the hidden layers. Each connection between nodes has weight, which is determined during the training phase. Figure 2 demonstrates a typical architecture of ANN.

The results of each neuron in the input layer are calculated layer after layer, until the output layer is reached. Each neuron in every layer (excluding the input layer) accepts the results obtained from the neurons in the preceding layer as input, $g_i(x)$, where $i$ is the index of the neuron in the layer. Then, each neuron output is computed according to the following steps:

Weighted sum: Each neuron calculates a weighted sum of its input neurons, $f(x) = \sum_i w_i g_i(x)$, where $w_i$ is the connection weight.

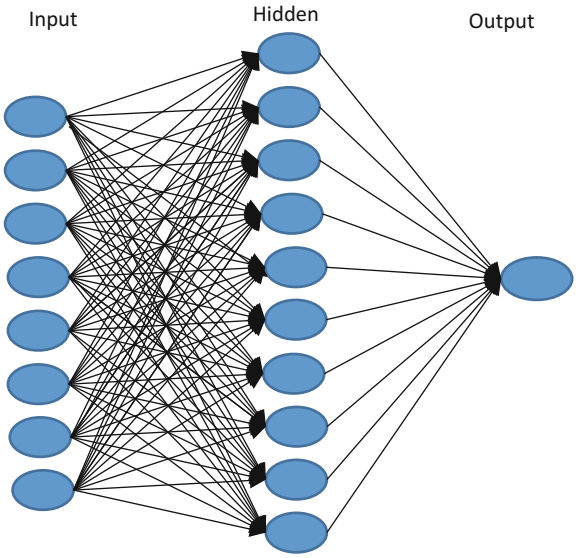


**Fig. 2** Neural network architecture

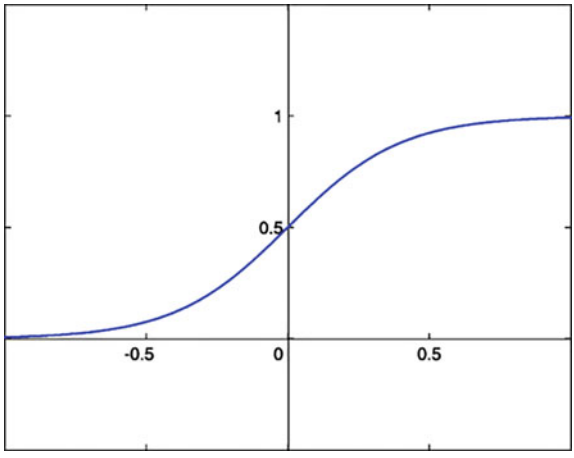


**Fig. 3** Sigmoid function

Nonlinear transformation on the weighted sum: An example of such a transformation would be a *sigmoid*, which is defined by the formula $\phi(f(x)) = \frac{1}{1+e^{-f(x)}}$. Figure 3 illustrates the sigmoid function.

Activation function: This is a decision function that uses a threshold to decide whether the result received from the nonlinear transformation is higher than the activation threshold or not and determines the output result accordingly. The formal representation of the activation function using a threshold of 0.5 is

$$y = g(f(x)) = \begin{cases} 1 \ \phi(f(x)) \geq 0.5 \\ 0 \quad otherwise \end{cases}. \tag{2.8}$$

In the testing phase, when trying to classify a new sample, we feed the input into the network and then calculate the output of each neuron, layer after layer from the input, until we reach the output layer and the classification result is returned. An example function for a nonlinear function for the output layer classification is called *Softmax*. Softmax takes all of the activation functions from the last output layer and transforms them to a vector of probability values, where the sum of probabilities of this vector is equal to 1. More formally, the softmax transformation for each neuron $x_k$ in the output layer is $\phi(f(x_k)) = \frac{e^{f(x_k)}}{\sum_{k=1}^{K} e^{f(x_k)}}$, where $K$ is the number of classes in the classification problem.

The common method of training ANN is called *Backpropagation* (Williams and Hinton 1986). This method strives to minimize the *loss function*, which is the difference between the calculated output and the known label, by adjusting the weights in the model. Each labeled sample is fed into the network, just as in the testing phase, but upon reaching the output layer, the algorithm calculates the difference from the known, desired output, and updates the weights of the model, so that the error will be minimal.

The strength of ANN is its ability to detect sophisticated similarities and patterns, using a combination of many simple operations.

The trained ANN model predicts the probability of a new sample vector belonging to class 1, which means this sample is malicious. The prediction is used as the anomaly score in our testing.

## 4 DGA Detection Using Unsupervised Learning

In this section, we propose several unsupervised machine learning methods for DGA detection. Unsupervised learning means that only a small portion of the dataset needs to be manually labeled as a preprocess step, in order to fix the parameters of a model. The process of labelling a large dataset requires much time and effort, so avoiding it is a major advantage. Two of the methods are based on the *K-Nearest Neighbors* (*KNN*) (Cover and Hart 1967) model. There are many variants to KNN, each providing a different result. The most common uses of a KNN model are classification and regression. We have chosen to use this model as an anomaly detection mechanism. In this approach, it is assumed that most of the training data belongs to the legitimate class and the few anomalous points are far away from the legitimate points in the feature space. Hence, a sample is said to have a high anomaly score if the distance to its $k$-nearest neighbor is too high, since if the point is legitimate, it would have many close points, while a malicious one would not have close neighbors, or only a few at most. Therefore, if the data contains $n$ samples, the anomaly score would be the

distance from a tested sample to the sample with the $k$th smallest distance ($k < n$); if that distance is higher than some trained threshold, the sample is determined to be anomalous.

The distance function between two samples can be any measure metric. We applied two metrics: Edit distance and Jaccard index, which will be described below.

## *4.1 Edit Distance*

Edit distance, which is the third metric suggested in (Sandeep et al. 2010), is used only on the test domains group without comparison to a database of distributions. Edit distance (Navarro 2001) takes two hostnames and computes how many transformations of single characters are required to transform one hostname into another. The rationale behind calculating the edit distance is that in a group of random domain names, the hostnames will be completely different from each other. Hence, when measuring the number of changes required to compare each domain within a group to the other domains in the same group, a high anomaly score is expected when the group includes DGAs. We used Levenshtein edit distance for the anomaly score. The Levenshtein distance between two strings $a$ and $b$, with lengths $i$ and $j$, respectively, is given by

$$lev_{a,b}(i,j) = \begin{cases} max(i,j) & if\ min(i,j) = 0, \\ min \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & otherwise. \end{cases} \tag{2.9}$$

$1_{(a_i \neq b_j)}$ is the indicator function equal to zero when $a_i = b_j$, being equal to one otherwise.

In our suggested implementation, we used this metric with a KNN classification model as follows:

During the training phase, for each training domain, we compute the edit distance to the rest of the domains and keep the distance to the first nearest neighbor. Then, an average and standard deviation of 1-nn distances is calculated. In order to have edit distance results in the range [0,1], the Levenshtein distance between two domains is scaled using the following equation:

$$scaled\_dist = lev_{a,b}(i,j)/max(i,j). \tag{2.10}$$

During the testing phase, we compute a KNN model with $k = 1$ between testing domains to the training domains using the edit distance metric. The anomaly score is obtained by computing how many standard deviations the edit distance of the domain

differs by from the average of the training set. For example, assuming the average of the training set is 0.32 and the standard deviation is 0.1, if the distance between a tested domain to its nearest domain is 0.52, its score would be 2, since 0.32 + 0.1*$\mathbf{2}$ = 0.52 (two standard deviations away from the average).

Calculating edit distance between a domain and the entire dataset increases the runtime costs significantly, since it compares each character of each domain against each character of the training domains. Therefore, we only compared domains with similar lengths, i.e., a domain name with 10 characters is only compared against domains with lengths between 6 and 14. We used the same ranges for calculating the edit distance for each domain. This approach is reasonable, since if we take, for example, two domains, one with eight characters and one with 30 characters, when comparing the edit distance between those domains, the result would be at least 22 (the number of missing characters), and 22/30 after scaling. As mentioned above, we are interested in the first nearest neighbor, and hence this result would probably not be the nearest.

## *4.2 Jaccard Index*

The Jaccard index (JI) metric on bigrams for supervised learning has been presented in Sect. 3.2. In this section, we define a new technique for detecting malicious domains through JI without using labeled data. Since there is no dataset of "legitimate" and "malicious" domains as before, we generated a set that contains lists of bigrams of the domains in the training set. A KNN model is applied to detect the anomalous domains with JI on bigrams as the similarity metric. The JI formula is defined in Sect. 3.2. The JI similarity is calculated between the bigrams of the tested domain and each bigram's list in the training set. The anomaly score is obtained by the $k$th JI result, which is the distance to the $k$th neighbor.

## *4.3 One-Class SVM for Anomaly Detection*

In Sect. 3.3, we have described the SVM model for binary classification. This model is a supervised learning method, used for classification. The traditional unsupervised version of SVM is called *One-Class SVM* (OC-SVM) (Schölkopf et al. 1999), which is mostly used for anomaly detection. In this model, a decision function is constructed from the extracted features to find the hyperplane with the maximum margin, which will contain most of the data in a relatively small region. This decision function assigns the value "+1" in the area where most of the samples reside, and "−1" otherwise.

The assumption is that normal domain names will have similar features, while the malicious domains would not share this similarity. Thus, the probability of being related to the main class would be small.

During the training phase, the model uses the training data to determine the hyperplane that best separates the majority of the data from the origin. For a new sample $x \in X^n$, if its feature vector lies within the hyperplane subspace, it is considered legitimate, being considered abnormal otherwise. Only a small portion of the dataset is allowed to lie on the other side of the decision boundary. Those data points are considered to be outliers.

The anomaly score given to a new sample is the distance of the sample from the separating hyperplane, if it is considered as an outlier. Otherwise, if the new sample is considered to be an inlier, the anomaly score is equal to zero.

The quadratic programming optimization problem for $n$ training samples is

$$\min_{\vec{\omega},\xi,\rho} \frac{1}{2} \|\vec{\omega}\|^2 + \frac{1}{\nu \cdot n} \sum_{i=1}^{n} \xi_i - \rho \ . \tag{2.11}$$

$$s.t : (\vec{\omega} \cdot \phi(\vec{x}_i)) \geq \rho - \xi_i, \ \xi_i \geq 0. \tag{2.12}$$

$\xi_i$ represents the distance of the sample $\vec{x}_i$ from outside the hyperplane, which means the sample is misclassified. It equals 0 if it falls inside the hyperplane. $\rho$ is the offset and $\|\vec{\omega}\|$ represents the size of the region containing most of the points.

We would like to minimize the size of this region while also minimizing the number of outliers. $\nu \in (0,1]$ is the trade-off between those two targets. It sets the upper bound on the proportion of outliers and it is a lower bound on the number of training samples contained by the hyperplane.

The decision function of a new point $\vec{z}$ is therefore

$$\vec{z} \mapsto sign(\vec{\omega} \cdot \phi(\vec{x}_i) - \rho). \tag{2.13}$$

In order to find the optimal $\vec{\omega}$ and $\rho$, this problem is solved using a kernel function (see Sect. 3.3) and Lagrange multipliers for the dot-product calculations.

OC-SVM technique can be a useful approach when the dataset is imbalanced, i.e., it contains far more legitimate domains than malicious ones.

## 5 Experimental Results

### *5.1 Dataset*

Our dataset contains 3473 legitimate domains and 131 malicious domains related to four different campaigns, as shown in Sect. 1.1. We obtained the legitimate domains by recording one day of network traffic in a big organization. The malicious domains were collected from various security reports and malware analysis reports, as well as from various source codes of the relevant malwares.

## 5.2 Results Evaluation

In this section, we describe our implementations of the different models described in Sects. 3 and 4. As mentioned in Table 1, we have four different labels for malicious domains. Therefore, we applied the algorithms four times, and on each iteration, we compared one label against the legitimate domains. The results are separated into the four different labels. We also applied the algorithm to the entire malicious dataset as one label against the legitimate dataset.

We developed our methods using the *scikit-learn* library for Python. It is an open-source library for machine learning algorithms.

We used the *k-fold cross-validation* model (Kohavi 1995) for training and testing the data. In this technique, the dataset is divided into $k$ (in our case, $k = 10$) different groups of the same size. In each phase, $(k-1)$ data groups are used for training and the remaining group is used for testing. This way, each part of the dataset is classified only once, and we end up having the whole dataset being classified. The cross-validation model is used to avoid over-fitting: if we use the whole dataset for training, we will get a higher classification rate, but when trying to predict with initially unseen data, the classification rate might decrease. The accuracy of this model is the ratio between the size of the data that is correctly classified and the size of the whole dataset.

In our tested dataset, the malicious domains had only two domain levels (i.e., "*sladfjhsaf.com*"). Thus, it is not possible to group the domains by second-level domains and it affects the way of implementing the metrics proposed in Yadav's paper.

As a consequence, we have computed the metrics over alphanumeric characters for each domain separately, without using grouping. We classified each domain separately by its alphanumeric characteristics and distributions against a database of malicious and legitimate domains.

There are various ways to evaluate the performance of the different classifiers. In the following sections, we describe the manner in which we evaluated the accuracy of the predicted anomaly scores.

**Table 1** Groups of malicious domain names

| Label No. | No. of domains | Group |
|---|---|---|
| 1 | 50 | Zeus |
| 2 | 27 | Conficker |
| 3 | 24 | Cryptolocker |
| 4 | 30 | Zero-day Hex domains |

### 5.2.1 True Positive Rate (TPR) and False Positive Rate (FPR)

Since we deal with a binary classification problem in which the prediction could be one of two classes, 0—legitimate, or 1—malicious, we use the TPR and FPR. TPR measures the proportion of the positives that are correctly identified as such, while FPR measures the proportion of negatives that incorrectly identified as positives: TPR = (True Positives/Positives) and FPR = (False Positives/Negatives).

Usually, there is a trade-off between the two measures. This trade-off can be represented by a *Receiver Operating Characteristic* (*ROC*) *curve* (to be explained below). A perfect predictor would be described as 100% TPR and 0% FPR; however, in practice, any predictor will possess a minimum error bound, which means that there would be some misclassifications by the classifier. Thus, when running our experiments, we choose the threshold that maximizes the TPR and minimizes the FPR.

### 5.2.2 Receiver Operating Characteristics (ROC Curve)

This is a graph that illustrates the performance of a binary classifier system as its discrimination threshold is varied (Fawcett 2006). The curve is created by plotting the TPR against the FPR at various threshold values. ROC analysis provides tools for selecting possibly optimal models and discarding suboptimal ones independently from the cost context or the class distribution. In Sects. 5.3 and 5.4, we evaluate our results using an ROC curve over the anomaly scores of the different models. Each curve represents the result of each class label, including the class that contains the entire malicious dataset as one-class label.

### 5.2.3 Area Under the ROC Curve (AUC)

This statistic metric is used in machine learning for comparing different models. Once the ROC is calculated, AUC measures the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one (assuming "positive" ranks higher than "negative"). It is calculated as follows:

$$A = \int_{\infty}^{-\infty} \mathrm{TPR}(T)\mathrm{FRP}'(T)dT = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} I\left(T' > T\right) f_1\left(T'\right) f_0\left(T\right) dT'dT = P\left(X_1 > X_0\right)$$

AUC results are in the [0, 1] range. In the optimal case, when a ROC curve gets perfect predictions, the AUC would be 1.

## 5.3 Supervised Learning Evaluation

### 5.3.1 KL Divergence

We computed this method twice: once with unigrams and once with bigrams. We separated the training data into legitimate and malicious, and for each of them, we created a list of bigram (or unigram) distributions. Then, we compared each test domain against the training groups, using the KL metric, as described in Sect. 3.1. KL results are shown in Fig. 4.

As shown in Fig. 4, the bigram results on our malicious dataset are very poor. To achieve TPR of 80%, the FPR is about 80%, and for lower FPR, the TPR is almost 0. When the KL divergence metric was used with unigrams instead of bigrams, the results were even worse.

The problem with KL divergence on unigrams is that it can only detect botnets with randomly generated domain names with uniform distribution. Also, for bigrams, the assumption that DGAs necessarily have different bigram distributions from normal domains is sometimes a mistake. There are algorithms that generate domain names with the same bigram distribution as English words, such as Kraken.

### 5.3.2 Jaccard Index

We obtained two databases (DBs), one for the malicious and a second for the non-malicious domain names; each contained lists of bigrams per domain. The lists were created as follows: For each domain from the training DB, we computed its bigrams and added them to the corresponding DB. When testing a new domain *d,* we calculated its bigrams and computed the JI against each legitimate bigrams list from the training. We computed the JI similarity only against domains from the training

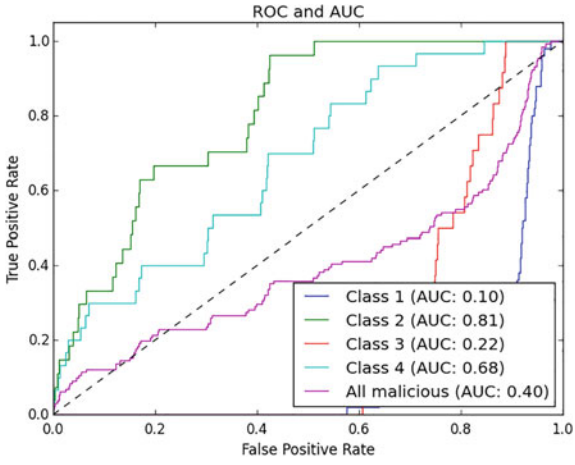


**Fig. 4** TPR and FPR of KL divergence on bigrams

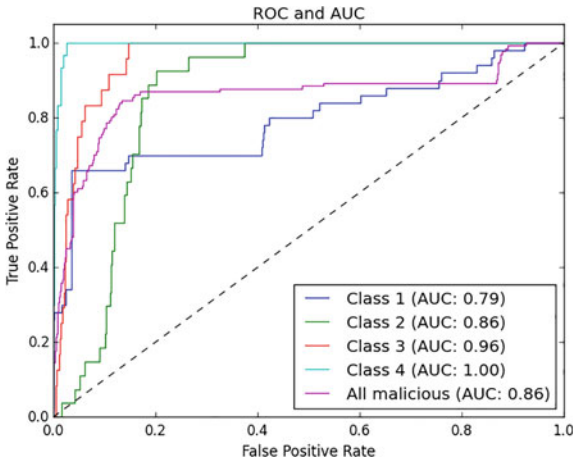


**Fig. 5** TPR and FPR of Jaccard index on bigrams

DBs that contain at least 25% of the bigrams presented in *d*. Then, we also discarded training domains that have less than 20% of the bigrams presented in the training, in common with *d*. For example, if a training domain contains 10 bigrams, we compute JI against the tested sample only if they have at least two bigrams in common. This threshold is useful in discarding words with less similarity and improves the accuracy of JI measurements.

After computing the JI against the DB of domains bigrams, we computed the average of the results. The same process was applied to the malicious DB. JI implementation for bigrams is discussed in Sect. 3.2. The results for each label appear in Fig. 5. As can be seen, JI achieves the best results when classifying class 4 and class 3, but only 0.79–0.86 AUC on the other classes.

### 5.3.3 Support Vector Machine (SVM)

We trained the classifier with $(k-1)$ groups of feature vectors, using the SVM classifier with *Radial Basis Function* (RBF) kernel (Chang et al. 2010) for the computations. This is a nonlinear kernel that maps samples into higher dimensional space. It has less complexity than other kernel functions and has fewer numerical challenges. The RBF kernel is defined as $k(\vec{x}, \vec{x}') = exp(-\gamma \left\| \vec{x} - \vec{x}' \right\|^2)$. $\gamma$ is a free parameter, which represents how far the influence of a single training example reaches. If we assign $\gamma$ with too high a value, it might result in over-fitting, since the radius of the area of influence of the support vectors includes only the support vector itself, no matter what the value of $\lambda$ is (see Sect. 3.3; Eq. 2.5). On the other hand, overly small $\gamma$ values might cause complications in handling complex data, since the radius of the area of influence of the support vector would include all of the training data.

As one can see, in order to achieve high accuracy for the classification of new test data, it is important to determine the optimal $\gamma$ and $\lambda$. Therefore, a 10-fold cross-

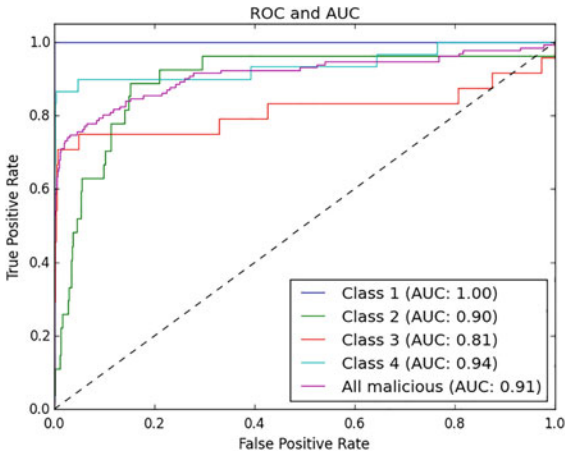


**Fig. 6** TPR and FPR of SVM

validation model has been applied and has revealed that the best values of $\gamma$ and $\lambda$ are 0.1 and 1, respectively.

The extracted features are described in Sect. 2. For testing a new domain, those features were extracted, and then the probability was calculated that the domain could be malicious by computing the distance from the margin using the RBF kernel, the result being used as the anomaly score. The results are shown in Fig. 6.

SVM completely succeeded in classifying class 1 domains, and with FPR of 0.2, it achieves TPR above 0.8 for classes 2, 4, and the "all malicious" class. For class 3, SVM achieves 0.4 TPR with 0.18 FPR.

### 5.3.4 Neural Network

The same process as in SVM has been applied using the Neural Network (NN) classifier. The architecture of the NN classifier is used as follows: eight nodes in the input layer (number of extracted features), one hidden layer with 10 nodes, and one node in the output layer. 64 iterations were used for optimizing the values and updating weights, with the learning rate being 0.001. The activation function in the hidden layer was "sigmoid" and "softmax" for the output layer. The architecture and the other parameters of the network, such as the activation function, were obtained according to Abu-Alia (2015). Using 10-fold cross-validation, we trained the network with malicious and legitimate domains in order to learn the weights. We tested the remaining domains on the NN using the learnt weights, and the NN determined whether a tested domain was malicious or not. In Sect. 3.4, we explained how the classification process is done.

The results are shown in Fig. 7.

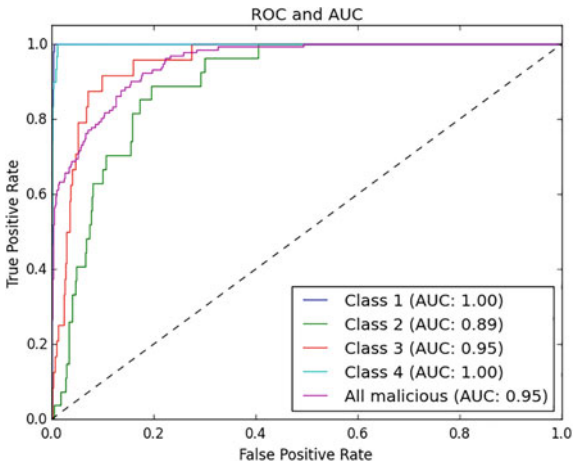


**Fig. 7** TPR and FPR of neural network

The neural network model provides the best results among the supervised classifiers examined. Again, class 1 achieved 100% success without false negatives, as well as class 4. The AUC of the rest of the classes is between 0.89 and 0.95.

## *5.4 Unsupervised Learning Evaluation Results*

For evaluation of the unsupervised methods, we did not use the labels in the training phase.

### 5.4.1 Edit Distance

During the training phase, we calculated the edit distance between each domain and its nearest neighbor, as described in the equations in Sect. 4.1. Then, an average and standard deviation of the first neighbor of the different domains was taken. Each domain is compared only against domains with similar length, i.e., the difference between the lengths is less than five.

Given a new sample, we calculated its edit distance to any other domain with a similar range and used the distance to the nearest neighbor. The anomaly score obtained by the number of standard deviations of the domain is far from the average of the group.

As shown in Fig. 8, using this method, we received AUC between 0.88 and 1 in all but class 2. After investigating the results for class 2, we found that the domains of class 2 contain 12–14 characters. On the other hand, our "legitimate" dataset contains many domains of length 12 that seem random. Therefore, running the algorithm on

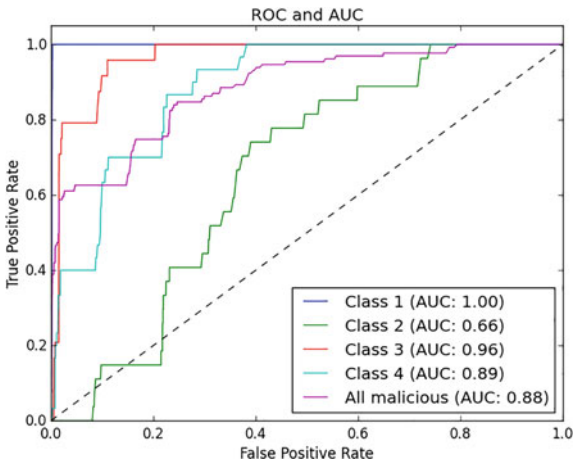


**Fig. 8** TPR and FPR of edit distance

**Table 2** Example of domains and their anomaly score, with respect to their length. The table shows malicious domains from class 2 with different lengths, as well as some legitimate domains that look random and have 12 characters. Those domains were part of an old botnet

| True class | Domain name | Edit distance anomaly score | Domain length |
|---|---|---|---|
| malicious-2 | sglzmfqmiw.com | 0.3887 | 14 |
| malicious-2 | iuobhicxny.com | 0.3883 | 14 |
| malicious-2 | pqkvaqpi.com | 0.3881 | 12 |
| malicious-2 | kmxnvuzy.com | 0.3852 | 12 |
| malicious-2 | xsbhnwdn.com | 0.242 | 12 |
| legitimate | twkfxcuk.com | 0.3829 | 12 |
| legitimate | nwgfgrml.com | 0.326 | 12 |
| legitimate | vscjttcu.com | 0.245 | 12 |

a small subset of domains with similar lengths emphasizes the noise and affects the results.

We verified those "legitimate" domains manually, and it seems that they were part of a botnet in the past. Using this unsupervised method, we were able to detect unknown malicious domains that were misclassified as legitimate in the organization's dataset.

An example of domains from class 2 with different anomaly scores affected by their length, and of legitimate domains that have random characteristics with 12 letters, can be found in Table 2.

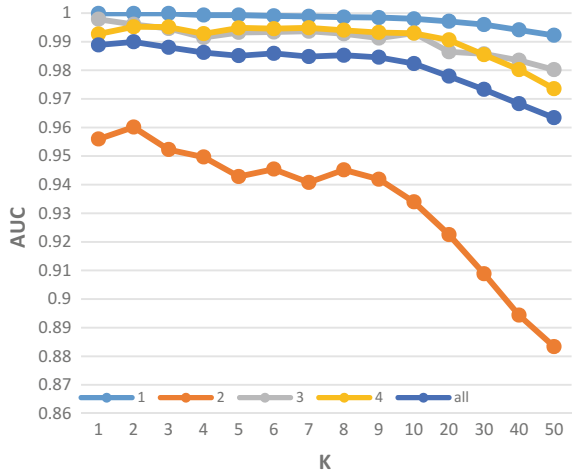


**Fig. 9** AUC of different k values for KNN with JI

### 5.4.2 Jaccard Index

We obtained a dataset of bigrams representing all the domains from the training set. When testing a new domain, we compute its bigrams and compare the JI measure to the $k$th nearest domain bigrams, using the KNN model. To predefine the right $k$ that yields minimal error with maximal accuracy, several values of $k$ were tested until we found the suitable value that solves the problem. In the anomaly detection problem, a small value of $k$ is chosen (usually less than 10), but not too small, i.e., $k = 1$, because such a model will not tolerate noise. We tested our model with different values of $k$ between one and 10. Figure 9 shows the AUC of the model for the different values. It is easy to see that for $k = 2-10$, the model is robust, and there are no major differences between the different AUC results.

According to Fig. 9, the best $k$ for detecting anomalies in the data correctly is 2. The results in Fig. 10 show the AUC of the classes when the value of $k$ is set to 2.

The results of this classifier show significant improvement compared to the supervised implementation of JI in Sect. 5.3.2. While in the previous implementation (Fig. 5) the ROC curve of the different classes was the same with AUC of 0.75, our new classifiers yield 100% TPR with almost zero FPR for classes 1 and 4, and less than 10% FPR for class 3 and "all malicious" classes. When applying the classifiers to domains of class 2, a TPR of 100% results in FPR of 25%, and for 80% TPR, the FPR would be 10%.

### 5.4.3 One-Class SVM

In this model, the features extracted from the dataset are the same as those used in the binary supervised SVM, which are described in Sect. 2.

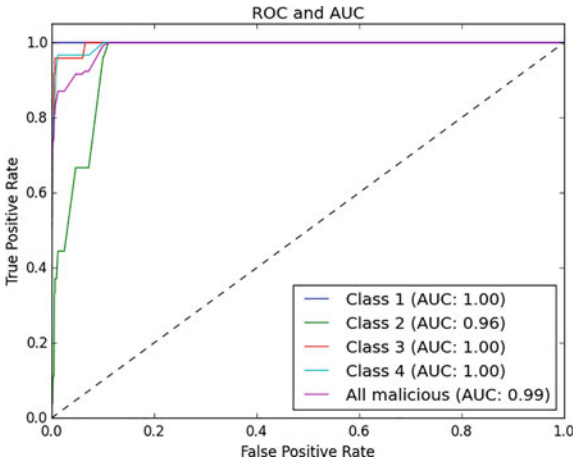


**Fig. 10** TPR and FPR of unsupervised Jaccard index on bigrams

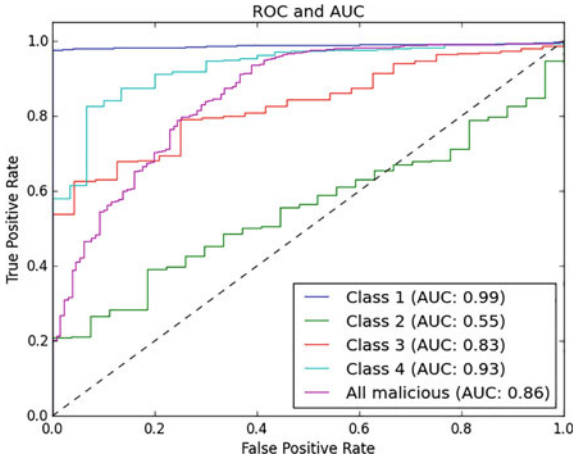


**Fig. 11** TPR and FPR of OC-SVM

We trained the OC-SVM classifier with RBF kernel (see Sect. 5.3.3), and used 10-fold cross-validation with several values of $\gamma$ and $\upsilon$ in order to choose the optimal parameters for the model and the kernel. We found that the best values of $\gamma$ and $\upsilon$ for this problem are 0.1 and 0.25, respectively.

We tested the remaining domains through a decision function, which determines the distance of a new point from the margins. The greater the distance, the higher the probability that the domain is malicious. The results are shown in Fig. 10 (Fig. 11).

OC-SVM also fails in classifying domains that belong to class 2, but succeeds in classifying domains from class 1 and 4 with 0.99 and 0.93 AUC. The reason for the failure in class 2 might again be because of the legitimate random domains that make the malicious domains with 12 letters fall close to the separating hyperplane, as explained in Sect. 5.4.1.

## 6 Discussion

In this paper, we have discussed and evaluated various machine learning methods for detecting DGA domains, by analyzing only the alphanumeric characteristics of the domain names in a monitored network. We compared the performance of these techniques when applied to four classes of DGAs, as shown in Table 3.

As can be seen, the KNN model that uses the JI metric achieves the best AUC among all different classes, even better than the supervised methods. Nevertheless, the complexity of computing the JI metric between the bigrams of the tested domain and the training dataset is high. If the runtime is of primary concern, the OC-SVM model is the best choice among the unsupervised methodologies, while the best detection rate among the supervised methods is achieved by the ANN classification model.

Although the KNN model that uses the edit distance metric achieved poor AUC when applied to domains from class 2, it detected zero-day domains that were part of the *legitimate* domains. The characteristics of the malicious domains of class 2 are similar to some domains from the legitimate dataset that are actually part of an old DGA.

The advantages of the proposed unsupervised approaches are that only a small portion of manually labeled training data is required in order to fix the parameters of the model for classification, a process that requires significant human labor. It is also useful for cases in which there is not enough malicious data to train the model, or when trying to detect a new unseen DGA.

Another advantage of our proposals is that by calculating only alphanumeric features of the domain names, without having to analyze the network characteristics, the detection process becomes easier to implement while still having a high rate of accuracy.

Third, since we do not analyze the traffic behavior or cluster domains offline, but rather analyze any new domain name independently, our generated model can

**Table 3** Comparison between the unsupervised and supervised methods by the AUC of the different classes. The best result of each class is marked in bold

| | Unsupervised methods | | | Supervised methods | | | |
|---|---|---|---|---|---|---|---|
| | KNN (JI) | KNN (edit distance) | OC-SVM | KL divergence | JI | SVM | ANN |
| Class 1 | **1** | **1** | 0.99 | 0.1 | 0.79 | 1 | 1 |
| Class 2 | **0.96** | 0.66 | 0.55 | 0.81 | 0.86 | 0.9 | 0.89 |
| Class 3 | **1** | 0.86 | 0.83 | 0.22 | 0.96 | 0.81 | 0.95 |
| Class 4 | **1** | 0.89 | 0.93 | 0.68 | 1 | 0.94 | 1 |
| All malicious | **0.99** | 0.88 | 0.86 | 0.4 | 0.86 | 0.91 | 0.95 |

be executed online to block suspicious domains in real time before the bots start to communicate with the C&C server.

After detecting a suspicious domain, the communication with its server can be blocked until further analysis of the domain's behavior can be applied by the security researchers.

In the future, we plan to take advantage of the knowledge about the process of generating DGA domains and try to develop a novel unsupervised approach for detecting DGAs in real time.

## References

Abu-Alia A (2015) Detecting domain flux botnet using machine learning techniques, Qatar University, College of Engineering

Antonakakis M, Perdisci R, Nadji Y, Vasiloglou N, Abu-Nimeh S, Lee W, Dagon D (2012) From throw-away traffic to bots: detecting the rise of dga-based malware. In: Presented as part of the 21st USENIX security symposium (Usenix Security 12), pp 491–506

Boser BE, Guyon IM, Vapnik VN (1992) A training algorithm for optimal margin classifiers. In: COLT 1992 Proceedings of the fifth annual workshop on computational learning theory. New York

Caudill M (1989) Neural nets primer, part VI. AI Expert 4(2):61–67

Chang Y, Hsieh C, Chang K, Ringgaard M, Lin C (2010) Training and testing low-degree polynomial data mappings via linear SVM. Mach Learn Res 11:1471–1490

Cover T, Hart P (1967) Nearest neighbor pattern classification. IEEE Trans Inf Theory 13(1):21–27

Dennis A, Rossow C, Stone-Gross B, Plohmann D, Bos H (2013) Highly resilient peer-to-peer botnets are here: an analysis of gameover zeus. In: 2013 8th international conference on malicious and unwanted software: the americas (MALWARE), pp 116–123

Fawcett T (2006) An introduction to ROC analysis. In: An introduction to ROC analysis, pp 861–874

Fitzgibbon N, Wood M (2009) Conficker. C: a technical analysis. Sophos Inc., SophosLabs

Jaccard P (1901) Distribution de la flore alpine: dans le bassin des dranses et dans quelques régions voisines. Rouge

Kohavi R (1995) A study of cross-validation and bootstrap for accuracy estimation and model selection. In: Proceedings of the fourteenth international joint conference on artificial intelligence. San Mateo

Kullback L, Leibler RA (1951) On information and sufficiency. In: The annals of mathematical statistics. Institute of Mathematical Statistics, pp 79–86

Mcgrath DK, Gupta M (2008) Behind phishing: an examination of phisher modi operandi. In: LEET

Namazifar M, Pan Y (2015) Research spotlight: detecting algorithmically generated domains. Cisco. http://blogs.cisco.com/security/talos/detecting-dga. Accessed 8 Aug 2015

Navarro G (2001) A guided tour to approximate string matching. ACM Comput Surv (CSUR) 33(1):31–88

Nguyen T-D, CAO T-D, Nguyen l-G (2015) DGA botnet detection using collaborative filtering and density-based clustering. In: SoICT 2015 Proceedings of the sixth international symposium on information and communication technology. New York

Panda Security (2015) CryptoLocker: What is and how to avoid it

Plohnmann D, Gerhards-Padilla E, Leder F (2011) Botnets: detection, measurement disinfection & defence, the european network and information security agency (ENISA)

Porras PA, Saidi H, Yegneswaran V (2009) A foray into conficker's logic and rendezvous points. In: LEET

Rahimian A, Ziharati R, Preda S, Debbabi M (2014) On the reverse engineering of the citadel botnet, In: Foundations and practice of security. Springer International Publishing, pp 408–425
Royal P (2008) Analysis of the Kraken Botnet. Damballa
Sandeep Y, Ashwath Kumar Krishna R, Narasimha RAL, Supranamaya R (2010) Detecting algorithmically generated malicious domain names. In: Proceedings of the 10th annual conference on internet measurement. New York
Schölkopf B, Smola WRCAJ, Shawe-Taylor J (1999) Support vector method for novelty detection. In: NIPS, vol 12, pp 582–588
Shevchenko S (2010) Domain name generator for murofet. http://blog.threatexpert.com/2010/10/domain-name-generator-for-murofet.html
Sinegubko D (2009) Twitter API still attracts hackers. http://blog.unmaskparasites.com/2009/12/09/twitter-api-still-attracts-hackers/. Accessed 09 Dec 2009
Stone-Gross B, Cova M, Cavallaro L, Gilbert B, Szydlowski M, Kemmerer R, Kruegel C, Vigna G (2009) Your botnet is my botnet: analysis of a botnet takeover. In: Proceedings of the 16th ACM conference on computer and communications security. ACM
Williams DRGHR, Hinton GE (1986) Learning representations by back-propagating errors. Nature 323:533–536
Wolf J (2008) Technical details of Srizbi's domain generation algorithm. FireEye
Yazdi S (2014) A closer look at cryptolocker's DGA, Fortinet. https://blog.fortinet.com/2014/01/16/a-closer-look-at-cryptolocker-s-dga. Accessed 16 Jan 2014