TÉCNICO LISBOA

Detection of Botnet Activity via Machine Learning

Diogo Aires Jerónimo

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Pedro Miguel dos Santos Alves Madeira Adão

Examination Committee

Chairperson: Prof. Rui Filipe Fernandes Prada
Supervisor: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Member of the Committee: Prof. Miguel Filipe Leitão Pardal

November 2018

Acknowledgments

The development of this thesis was only possible of the support of people that helped me in keeping this project moving forward. I want to express my thanks with a few words.

Firstly, I want to thank Prof. Pedro Adão for the orientation and the opportunity he gave me in working in a project that I enjoyed.

I also want to thank all my friends and colleagues that provided ideas, incentive and motivation for the problems faced along the way.

Lastly, a special thanks to my mother, brother, and sister for always believing in me and by providing me full support since the beginning of my studies.

Abstract

Botnet attacks have always been a serious problem for companies and organizations to deal with since they can engage in all sorts of malicious activity with the purpose of causing damage and stealing information from compromised machines. In this work we analyzed network trace logs from botnet connections and created a system that is capable of detecting this kind of activity leveraging a supervised machine learning approach. This created tool not only detects abnormal network behavior but it can be also used to aid forensic analysts in the detection of other related botnet attacks such as data exfiltration and DDOS attacks. We created two scenarios mixing normal and abnormal activity in the network in order to evaluate the system. In the first scenario, we tested the detector with only one botnet present, since it is the scenario that is closest to reality. In the second scenario, we mixed normal traffic with multiple types of botnets to see if it is viable for a machine learning system to detect so many patterns at once. We also use a Cross Validation strategy that respects the temporal order of the packets of each connection in the logs so that the detector does not use future information that can create bias in the classification. In the end, we present an example of a practical application of the detector with a module called Decider. The Decider chooses the machines that are most likely compromised in a network basing its decision on the results from the classifier algorithm. The solution's architecture was made to be extended in order to use information from other data sources such as file access and login logs to create clusters of information that allows the improvement of overall detection results.

Keywords

Botnets; Machine Learning; Information Security; Data Exfiltration

Resumo

Ataques de botnets têm sempre sido um grande problema para empresas e organizações lidarem uma vez que estas estão envolvidas em variados tipos de atividades maliciosas com o propósito de causar danos e roubar informação. Neste trabalho, analisámos logs de rede de conexões de botnets e criámos um sistema capaz de detetar esta atividade maliciosa utilizando técnicas de aprendizagem supervisionada. Esta ferramenta criada não só serve para detetar estes comportamentos maliciosos na rede como também é útil para analistas forenses usarem na deteção de outros ataques relacionados com bonets como a exfiltração de dados e ataques DDOS. Criámos dois cenários onde misturamos conexões normais e maliciosas que serão usados para avaliar o sistema. No primeiro cenário, testámos o sistema com apenas uma botnet presente na rede visto que, é o cenário mais próximo da realidade. No segundo cenário, misturámos conexões normais com múltiplas botnets para testar a viabilidade de uma abordagem de machine learning na deteção de vários padrões diferentes ao mesmo tempo. Usamos também a estratégia de validação cruzada que respeita a ordem temporal de cada pacote nas conexões dos logs para que o classificador não utilize informação futura, criando um enviesamento na classificação. No final, apresentamos um exemplo de uma aplicação prática do sistema com um módulo chamado Decider. O Decider escolhe as máquinas que acha que têm mais probabilidade de estarem comprometidas na rede, baseando a sua decisão nos resultados provenientes do algoritmo de classificação. A arquitetura da solução foi feita para ser estendida de maneira a ser possível usar-se dados de outras fontes de informação como logs de acesso a ficheiros e de login que permitirá criar um aglomerado de informação útil que melhore os resultados da deteção.

Palavras Chave

Botnets; Machine Learning; Segurança de Informação; Exfiltração de dados

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

DDOS	Distributed Denial of Service
IDS	Intrusion Detection System
DPI	Deep Packet Inspection
DNS	Domain Name System
IRC	Internet Relay Chat
SaaS	Software as a Service
DST	Discrete Sine Transform
ROC	Receiver operating characteristic
TP	True Positives
TN	True Negatives
FP	False Positives
FN	False Negatives

1

Introduction

Contents

1.1 Motivation

One of the most dangerous and damaging attacks for organizations, nowadays, is the infection of machines that allow the creation of botnets. The attacker can have full access to these machines allowing the engagement of multiple damaging activities. These kinds of attacks are hard to prevent because, once a machine is infected, the attacker might issue commands to the zombie machine right away, causing immediate damage. Current defense mechanisms on the subject to detect botnets use tools like Deep Packet Inspection (DPI) to inspect the payload of packets to detect potential information carried inside the packet. If the packets are ciphered though, the tool becomes ineffective because we cannot inspect the payload anymore.

Other existing methods aim at producing mechanisms that prevent attackers breach into the systems and the network infrastructure of organizations. For example, the main focus of the detection defense mechanisms rely on tools such as Intrusion Detection Systems (IDSs) that alone, are simply not enough for this type of attacks. Botnets have ways to minimize their network fingerprint like communicating with the Command and Control server in random times of the day. Also, the use of obfuscation and encryption as already mentioned makes the task harder for the defender. Preventing the infection by malware is pretty complex too. Sometimes, the malware can be sent by email masked as a benign file or malicious insiders can have enough knowledge to plant a malware inside one of the machines, creating a botnet easily and quickly.

The work presented here will explore this topic, focusing on malware that can create botnets, more specifically, we are interested in detecting connections made by machines that were infected and are communicating with the attacker's machine. Botnets have been largely used by attackers in the past to do all kinds of malicious activities including compromising machines to monitor its activities by planting a keylogger or to sniff traffic, to launch Distributed Denial of Service (DDOS) attacks, to steal the identity of the machine, and even to exfiltrate data from the user's computer.

The main goal of the system created in this project is to detect via a supervised machine learning approach, malicious network activity by extracting generic features associated with the flows between the network machines in order to train a classifier and predict future connections. However, the system was designed to be used as an auxiliary or complementary defense mechanism to existing ones. For example, an organization logs multiple services, at various operating system levels. One could use this tool to gather information from file access and modification logs, web services logs or even login information and correlate this data with the results from our classification process to detect both botnet activity and data exfiltration.

1.2 Existing Solutions

In order to get a better understanding of the problem we are dealing with, we will point out some relevant work and solutions proposed in the past. We will present relevant work in the field regarding botnet detection as well as machine learning methods to do so. Also, as an example, we will explore the issue of data exfiltration since our created tool can be used to detect this specific attack. To propose our method, we created a brief survey on previous work about these two topics. The survey will provide insight into what was developed in the area so that we can propose a method that attempts to solve the challenges faced by previous work.

Existing research on data exfiltration is very broad since there are multiple methods to do so. A data exfiltration taxonomy can be found in [1] as well as a description of some exfiltration methods. In regards to network data exfiltration detection, Liu *et al.* [2] describe a framework to be used to detect inside jobs focusing, primarily, on the outbound traffic. Inside jobs are essentially crimes committed by someone working on a company or organization sometimes being in positions of trust and power. These inside jobs can range from installing malicious software so that an external party can access information or stealing information from confidential documents. The framework is able to filter all outbound traffic and make a fine-grained identification of applications used (for example Gmail), based on some features of the traffic like temporal patterns and the size of the packets. Also, the method uses signatures of the traffic to produce fingerprints of sensitive files companies might want to protect. If such a signature is found leaving the network, the system might send an alert to the administrators.

Sometimes, getting traffic signatures is not that simple because attackers might use anti-signature techniques such as encryption. At the same time, the covertness of data exfiltration increases making the detection much more complex. He *et al.* [3] discuss methods to detect data exfiltration when the attackers encrypt the communications. The identification of encrypted packets is done by calculating entropy where high entropy values might indicate encryption. Information entropy is basically defined as how much useful information a message is expected to contain. Then, the authors try to identify data exfiltration activities by profiling the connections and leveraging a machine learning approach to classify malicious users.

In a more recent work, Nadler *et al.* [4] present ways to detect low throughput data exfiltration via Domain Name System (DNS) protocol, using a machine learning approach. Since making a lot of DNS requests to exfiltrate data would raise alarms in the intrusion detection systems of any company, sophisticated methods include making very little requests throughout the day, in order to avoid the defense mechanisms. The authors attempt to identify these stealthy communications by extracting features from logs and, use a classifier to distinguish legitimate communications from malicious ones.

A couple of physical data exfiltration techniques are discussed in [5,6]. These authors try to address a common problem regarding the problem of the identification of malicious copying of files from a Windows

filesystem in order to steal them through a physical device like a USB stick. They attempt to analyze patterns on folder copying and timestamp analysis for the detection.

In terms of botnet data exfiltration research, Al-Bataineh *et al.* [7] analyze a popular data exfiltration botnet called Zeus and propose a method to detect its activities. This work analyzes exfiltration where botnets use HTTP POST requests. We can find recent work on stealthy botnets in [8, 9] where the authors attempt to study these behaviors. The first work attempts to detect data exfiltration by deploying detectors over the network and periodically, moving them to reduce the likelihood of exfiltration. The second work uses a decision-making algorithm called reinforcement learning to deploy optimal defenses to the network such as honeypots, that detect intruders and network-based detection mechanisms that enable a defender to locate compromised machines that persist in the network. The authors aim to reduce the likelihood of the attacker to persist in the network by identifying and taking down the bots.

Work by Zhao *et al.* [10] and Beigi *et al.* [11] talk about machine learning techniques to detect botnets. In the first work, the authors base the detection in traffic behavior that allows the defender to detect malicious connections without needing to inspect the payload of packets which, can be obfuscated or encrypted. The authors make use of a Decision Tree with previously selected features from the traffic. The features are extracted in a specific time window which yield very high detection rate with very low false positive percentage. On the other hand, Beigi *et al.* extend the work by Zhao *et al.*, discussing the selection of traffic features and created a whole new dataset aiming to solve the problem of generality. In the end, the authors test the dataset with a Decision Tree algorithm.

There has been work done on data exfiltration by botnets and the stealthy behavior separately, however, there is very little research that focuses on these two aspects at the same time. That is, none of the previous research specifically studies the behavior of stealthy data exfiltration by botnets that employ anti-signature techniques like encryption and obfuscation. We believe that, with our work, this can become a reality in the near future, combining all sorts of valuable information contained in logs to extend our system.

1.3 Contribution

The method we propose has two main parts. The first one has to do with the analysis methodology and classification of botnets via Machine Learning and network trace logs. This is done by the tool that we created for this purpose. The second one has to do with a practical module called Decider that will allow us to flag the most likely infected machines using information from the classifier results.

Also, we will analyze what are the characteristics of data exfiltration since it can be useful in the future when having in mind the extension of this system to other types of attacks. We will try to answer questions like, what is data exfiltration? How can data be exfiltrated? What are the current methods that

attackers use? What can we do to detect it and mitigate its damage? To do this, we will first present a survey on data exfiltration in order to better understand this problem.

In summary, our main contributions are:

- Create a survey on botnets and machine learning approaches as well as data exfiltration techniques and defenses.
- Study the characteristics and indicators that can help to detect data botnet activity.
- Create a prototype of a system that leverages machine learning approaches and is capable of identifying botnets.
- Evaluate the solution based on scenarios and metrics to compare classifiers
- Create a practical application of the system by describing a simple component called Decider

1.4 Outline

This document is organized as follows: in the next section, we will describe the previous research done regarding botnet detection and data exfiltration by detailing the relevance and contributions of each one.

Section 3 will contain the architecture of the solution we propose, aiming to extend existing work. It will feature a diagram and the main ideas behind the proposal. In section 4, we will show the implementation process that led us to present the final version of our system. In section 5 we will provide details about the methodologies and methods used to evaluate the solution. We will discuss the results gathered from testing our solution by presenting two scenarios: one where we test a single botnet at a time, and one that tests the whole dataset, composed by more than a dozen of different botnets. Finally, the last section will be used to conclude the this document and to discuss some ideas that could potentially be done in a future work.

2

Related Work

Contents

We will divide this section into four parts: Exfiltration Techniques, Exfiltration Detection Mechanisms, Botnet Data Exfiltration, and Machine Learning. The first will be used to present examples of explored exfiltration techniques. On the second part, we will describe the latest work on detection of data exfiltration. The botnet data exfiltration section will describe the state of the art when it comes to the detection of stealthy botnet behaviors that maximize their presence on organization's networks. Finally, we provide some background on machine learning since it is the core of our proposed solution.

Data exfiltration is the act of stealing information from compromised servers or computers. A typical scenario involves the infection of one or more servers by an attacker and then, establishing a communication channel between the infected machine and the attacker's machine in order to transfer the information he wants. This type of attack can also happen if the attacker is an insider in an organization. This way, the exfiltration is more effective since most defense mechanisms deal with threats from the outside of the organization network. To add to this, malicious insiders have more knowledge of the internals of the network and company policies and possess valid access credentials to several services, allowing them to steal information more easily.

The following subsections will contain a brief survey of past exfiltration and detection techniques researched. Since the nature of data exfiltration may be different, we will divide the techniques into four groups: Network techniques, that describe all the exfiltration methods executed via common network protocols such as HTTP; Physical methods, that are related to data theft via devices such as USB sticks or CD's; and then the more stealthy detection techniques like Covert Channels and Steganography. Although some of these categories are hard to differentiate, we will try to separate them to better organize ideas. For example, sometimes, the research on the detection of network techniques involve some kind of covert channel while some other articles only describe pure out of band techniques. At the end of this section, we will reserve a special subsection for Botnet Data Exfiltration with previous work on botnets and their stealthy nature and a subsection about Machine Learning.

2.1 Exfiltration Techniques

Many methods exist nowadays that allow a skilled attacker to exfiltrate sensitive information from companies. To understand them better and to study them more efficiently, we can classify each technique. Giani *et al.* [1] describe a simple taxonomy for data exfiltration channels which gives some insight on what type of exfiltration channels we have to deal with. Data can be exfiltrated through the Network with very well known protocols such as HTTP, FTP, and DNS; through Physical devices such as USB sticks and CD's and finally, Cognitive methods that may involve Social engineering techniques which are capable of leaking information about the internal services of a company.

Another aspect we must have in mind, and described in this same article, is the covertness of these

channels. Covert channels are essentially ways to transfer data through channels not intended for information transfer [12] in a manner that hides that such communication is taking place while minimizing the risks of detection. In these types of scenario, detection is extremely hard since attacks can assume many forms and the exfiltration can get very creative.

2.1.1 Network Techniques

There has been a lot of research in regards to the multiple network techniques to perform and detect data exfiltration, and the channels used to do so (including covert channels). This type of techniques is one that happens in the company's infrastructure through protocols like DNS, HTTP, and even email.

In terms of network techniques, a lot of the research focuses on DNS exfiltration. Even with a well-defined network security, all networks require common services to function properly [13] like the DNS protocol. This is a very common exfiltration channel since all companies must use it to resolve domain names into IP addresses to establish connectivity between services. We can identify some indicators that give us enough information that DNS is being used in an unconventional way. For example, DNS queries that contain multiple levels and composed of hexadecimal strings (e04fdbe587a1.f6c7.example.com), long DNS name lookups, queries on foreign and unknown domains on a short timespan, and queries to Dynamic DNS. Although these are good indicators, not all DNS exfiltration is like that. To mitigate this, the query logs must be activated in order to manually inspect the incident and with a large number of DNS records a company can generate in one day, that task can be very hard to do. In addition to this, companies can implement network sensors and other control systems but, ultimately, we must find better alternatives.

Born [14] describes a couple of browser-based DNS exfiltration techniques that are hard to detect.

The techniques described include DNS prefetching which is a way to create custom DNS queries by adding an element to prefetch in the head of a document and create a DNS channel. Although this does not work with every browser in real-time, it is still possible to take advantage of this method by anchoring elements on the document. Another approach does not use prefetching instead, it takes advantage of the "src" attribute of a dynamically created object. This way, DNS queries can be sent to a controlled domain that does not reply to the requests, fooling the intrusion detection system. One last technique is described and has to do with Storage and Timing channels. Timing channels are basically covert channels that convey information by the timing of events and require a receiving end to measure these time differences to deduce what the information is [15]. The timing channel discussed is based on encoding the data in the inter-arrival time between DNS queries. Alternatively, the server can create a storage channel through the clever use of DNS responses by checking the return of previous DNS responses. Storage channels involve the direct or indirect writing of information on a location and the reading of that location [16]. Since these techniques don't require any elevated user permissions, any

defensive mechanism that regulates and controls the user permissions on the system will be useless.

The greatest weakness of this work is not giving any defense mechanism to the algorithms presented, leaving it to future work. Also, to further extend this work, the author suggests combining the exfiltration techniques presented with popular website traffic in order to provide one more layer of covertness to the channel.

2.1.2 Physical Techniques

Physical techniques consist of transferring the data to physical devices like USB sticks, CDs, and even laptops [1]. It is also possible to print the information to increase covertness or steal the devices where the sensitive information is stored. This kind of attack is hard to prevent but can be mitigated with strong security policies inside organizations. However, it can be very common for companies to be susceptible to this due to carelessness from the employees [17].

"Dumpster diving" is another thing to take into consideration. If companies are not careful when destroying information, an attacker might consider scanning and searching dustbins to see if some kind of data is possible to gather [17].

2.1.3 Covert Channel Techniques

Besides network and physical techniques, there are also out of band techniques or techniques that use some type of covert channel that allow attackers to exfiltrate data in a stealthy way, even from secure and isolated computers from the network (air-gapped systems).

The previous work [18] on out of band channels describe a technique that uses inaudible sound frequencies to encode data. This technique can exploit isolated machines without even installing new hardware in the compromised host since it only uses a software solution. It is shown that with a simple pair of speakers and a microphone, it is possible to exfiltrate RSA cryptosystem keys, passwords, and even PDF documents successfully in a range of 1 to 5 meters between the two machines.

The authors demonstrate two methods to carry out the attack: the first one uses a one-directional channel from the target machine to the other and the second attack method uses a bi-directional reliable channel that makes use of error correcting methods (just like a TCP channel) in order to deliver the exfiltrated information correctly. These methods are slow in transferring data however, they can be done in a very covert way. The biggest downside of these techniques is the fact that to exfiltrate such data, the two machines need to be very close and the method can be difficult to execute if the microphone captures noise. The simplest way to eliminate this threat is to remove the audio output devices from air-gapped machines since they have no real use in a protected network scenario. For future work, one can investigate more complicated processing audio signal techniques to extend the range of the

communication using relay devices. Another interesting topic discussed is the possibility to explore other scenarios to disseminate information to multiple machines.

Another recent work on out of band techniques include exfiltration from hard drive noises [19] and exfiltration from router LED's [20].

2.1.4 Steganography

Finally, steganography is another technique which attackers may use to hide data and, subsequently, exfiltrate information. The definition of steganography is "the art of hiding information in ways that prevent the detection of hidden messages" [21]. In a way, this definition is not very different from a covert channel definition, the only difference being the covert channel allows the transferring of information.

Normally, an attacker hides information in images, videos and even text, however, most recently, steganography is being exploited in a variety of ways including hiding data in the TCP/IP stack protocol, peer-to-peer services like BitTorrent[1], social media, and cloud computing environments [22].

2.2 Exfiltration Detection Mechanisms

This section will focus on previous work about the detection, prevention and defensive mechanisms against data exfiltration. There is a lot of variability on these approaches, from machine learning approaches to statistical approaches, showing that there is not one simple solution to the problem.

2.2.1 Network Detection

The following paragraphs describe network detection mechanisms. For example, a prototype framework called SIDD [2] was developed aiming to detect and mitigate sensitive data exfiltration attacks coming from inside jobs, focusing on the outbound network traffic. The solution created makes sure that the data that leaves the network passes through a couple of phases to ensure that sensitive information does not leave the organization. This framework is composed of several components.

Firstly, sensitive information that one wants to protect must be stored in a Critical Data Repository. In turn, this will be used to generate file signatures to be stored on the Signature Store for later comparison. The first step of the algorithm is to retrieve traffic from the network with a Deep Packet Inspection tool. This traffic is then filtered and goes to the application identification system where features of traffic are extracted like temporal patterns and size of the packets. This allows the system to distinguish traffic that comes from different types of application and takes a prompt action to exit and raise an alarm if a potentially dangerous application is being used. These applications might be applications that could

[1] http://www.bittorrent.com/

steal information and access confidential information. On a second stage, the packets will pass through the Content Retrieval process and the content of the traffic will be analyzed. A Content Signature Generator is used to generate signatures for the data that is being analyzed. After that, a matching algorithm is employed to compare the signatures computed with the ones on the Signature Store. If a match is found, the system may, again, exit and take a response action. In the last phase, the system will try to detect the existence of a covert channel. Some statistical features inherent in media will be generated and used in a previously trained classifier, in order to perform Steganalysis.

The main contribution of this framework is a prototype to detect data exfiltration leveraging content signature and identification method to filter some types of communication. Also, the system is capable of doing a more fine-grained classification of protocols using time information, frequency information, and wavelet-based analysis for the case of steganography. Although the solution developed had encouraging results, the authors acknowledged that the framework covered a limited amount of applications to be identified. That is, the authors only distinguished traffic from few applications such as Gmail. Also, for covert communications, only the audio was considered but there are other types of media in which an attacker might hide information like video and images. The future work includes extending the framework in order to reduce these limitations.

Sometimes to increase covertness in the communication and to confuse systems like SIDD, encryption may be used to encrypt traffic when stealing data, making previous techniques less effective. He *et al.* [3] proposed a method to detect encrypted data exfiltration in a cloud-based scenario. To do this, the authors use network profiling techniques, DPI tools that inspect the insides of packets that allow identification of media or specific bytes for example, and entropy calculations to assess if exfiltration is occurring.

The main contributions of this work have to do with the algorithms devised to detect encrypted packets. The authors take advantage of DPI and entropy estimation to judge whether traffic is encrypted in real-time. This point is important because of the urgency to detect potential exfiltration as fast as possible to shut down the connection and mitigate any damage. Also, the described method is capable of solving the issue of entropy calculation when traffic is using both plain and ciphertext that could possibly confuse the detection system. This is solved by using a proposed split-window algorithm. The second part of the paper deals with assessing if the encrypted traffic is normal or, potentially malicious indicating that data exfiltration is ongoing. This is done by building profiles of the behavior of cloud users and then, a Multinomial Naive Bayes algorithm is used in order to classify the connections. The algorithm is fed with a couple of features selected by the authors like destination IP, destination port and the network and application layer protocols. In the end, in the evaluation section, the method was tested in a real world environment tested with four different encryption methods. (HTTPS, AES, HTTP with encrypted data, and a Private algorithm made by the authors)

The results show that encrypted traffic can be detected on average after 45 seconds and the detection rate is over 90%. The accuracy in determining the state of encryption is more than 80%. The main weakness of this work focuses on the 45-second delay to detect exfiltration. It is a promising result, however, that delay should be as short as possible in order to mitigate potential damages. For future work, the authors propose to confirm the detection result and to further reduce the false positives in order to enhance the method. Also, a method to reduce the delay of detection would be something to consider as well. This work is very relevant in order to detect encrypted communication and is worth using these ideas on our specific problem.

More recently, a machine learning approach was proposed to detect low throughput data exfiltration over DNS, using feature selection and classification of traffic [4]. This article aims to solve the issue of detecting low throughput data exfiltration since previous research focuses a lot on the DNS tunneling tools and techniques.

The algorithm presents two main contributions, the first one is the selection and collection of meaningful features to be used by the classification algorithm that allows detection with high accuracy. The second contribution is the creation of the algorithm used to detect if a specific channel is being used for exfiltration. The algorithm relies on a two-step process consisting of a classifier and a rule-based filter to eliminate false positives. The whole method is composed of four main stages.

The first stage has the objective to gather data. To achieve this, the authors rely on DNS logs and, once an hour, all of the DNS traffic is aggregated using the primary domain as key.

The next step extracts features from the previously collected data in order to train the classifier. The extracted features include entropy calculation; IP-Hostname Exchange RR Type Distribution which computes the rate of resource record on the data (A, AAAA, MX, etc...) for a domain-specific traffic; Unique Query Ratio, this ratio is important because if DNS is being used as a covert channel, a used message is not likely to be repeated; Unique Query Volume that computes the volume of unique queries; Query Length Average that computes the average of the length of a DNS query and finally, Longest Meaningful Word Length Average that decomposes traffic into substrings and makes a dictionary lookup searching for words.

Now that the features are extracted, the method enters phase 3, Covert Channel Detection. In this stage, the authors attempt to classify the data from phase 2 using anomaly detection techniques (one class classifiers).

In the end, the cases remaining from phase 3 are filtered with a rule-based filter. Several heuristics are used such as Unique Client Count which is based on unique IP's that accessed the primary domain, eliminating the vast majority of legitimate covert channels; HTTP Popularity Rank which uses Alexa[2] top sites as a whitelist filter and Non-Internet Primary Domains that uses IANA Root Zone Database[3] to

[2]https://www.alexa.com/topsites
[3]https://www.iana.org/domains/root/db

strike out international top-level-domains used for routing.

The main disadvantage of the proposed solution has to do with the ability to avoid detection by distributing the payload among different primary domains and name servers. Another issue is in the rule-based filter used. If an attacker uses a compromised domain it inherits its popularity, meaning that it can be used to avoid being not filtered. These two issues should be addressed and can be used to extend this work.

2.2.2 Physical Detection

In terms of physical detection methods, a technique was developed by Grier [5] to detect if files were copied from a Windows filesystem and later exfiltrated. The technique presented aims to solve a common problem that forensic analysts deal with on a daily basis: How to check if files were copied from a filesystem given only the filesystem? Unlike in Unix based filesystems, copying a file in a Windows system does not leave a trace since the MAC timestamps (modification, access and creation timestamps) don't change, making the file seemingly invisible. However, not all is lost. Copying folders in the filesystem have very clear patterns that the forensic analyst can use in his analysis.

The author distinguishes the act of copying folders and the act of accessing files in a routine way. While copying folders is nonselective, temporarily continuous, recursive and the directory timestamp is updated but not the inner files timestamps, in a routine access this does not happen at all. Nonetheless, MAC timestamps are ephemeral and unreliable meaning we cannot immediately apply this knowledge. Grier makes two observations to help solve this problem: the first one is that timestamps increase but they cannot decrease and the filesystem activity resemble heavy-tailed distributions such as the Pareto distribution, which means that a small number of files will account for a large part of the activity while the other files will remain with little to no activity. It is stated that copying creates a cutoff cluster artifact which means: "a point in time which no subfolder has an access timestamp prior to (hence a cutoff) and which a disproportionate number of subfolders have access timestamps equal to (hence a cluster)".

Based on this, the method was evaluated in a simulated environment containing two folders created at the start and the quantitative analysis was discussed. The article does not address the case where the copying of folders may be perfectly legitimate, which means, it didn't discuss any method to deal with this. For future work, the author suggests the improvement of the above method by testing it and evaluating it in a real world setting.

Patel *et al.* [6] attempt to address some of the issues presented by the previous work, proposing a technique that can differentiate copy operations with another type of operations that produce false positives. The methods described are based on the notion of fuzzy logic and fuzzy sets that assign a degree of truth to each proposition. That degree of truth can range from 0 (absolute falsehood) to 1 (absolute truth). It was shown that specific operations were distinguished from copying of folders in the

filesystem like compression and anti-virus scanning, leading to satisfactory results. This method is not perfect though. On some Windows versions like Windows Vista and Windows 7, the access timestamps are disabled by default, making it difficult to get good results using this technique. The authors were unable to distinguish some operations like file searching, leaving it for future work.

A statistical approach is described by Hu *et al.* [23] to detect possible data exfiltration by insiders focusing on the file accesses. The proposed methods allow profiling legitimate uses of file repositories. By using this analysis, it is possible to compare various users and assert if anomalous activity is happening. The model presented is based on the assumption that people have well-defined profiles for file usage. For example, a software developer might have access to one or more file repositories and work on one or more projects. If his actions deviate from the standard access to those repositories, the model might flag this behavior and an exfiltration attack could be happening.

The first thing considered by the authors is the concentration of file repository access data. The concentration of file repository access data is helpful for identifying whether there are any suspicious file repository accesses to some extent. For example, if one employee accesses a large number of files in a day while on most other days he only accesses a few files, this may indicate some suspicious activities. To measure this, the authors use the Herfindahl index that is a statistical score to measure concentration in this case, of files and documents. Next, file access statistics are considered for one and for multiple repositories. This is done by measuring the mean number of files accessed every working day and using a confidence interval to detect anomalous behaviors. Another method used in the model is regression to calculate file access statistics. They use regression relations between file repositories because they are likely related to each other. Therefore, it is possible to detect data exfiltration more accurately if patterns can be constructed using regression. Lastly, a hypothesis is tested for multiple users and multiple repositories using the measures described above. The file access patterns can be compared in order to identify anomalous activities (users).

So that the model could be validated, the experiments made use of software repositories related to the Unix desktop environment KDE. The authors chose random users on these repositories (that worked on multiple repositories) and the respective file accesses and commits as input data. It was found that, for legitimate users, the number of files accessed on each check-in does follow a certain pattern. Another thing that was found is that the number of files accessed each day does not alter much. At the end, the authors don't suggest any extension but this statistical method has some flaws in detecting data exfiltration. For example, a user can still exfiltrate data without being detected if he follows his normal behavior. The model will not work very well if the exfiltration is done in more covert ways.

2.2.3 Covert Channel Detection

A etection method for Timing Channels is presented by Berk *et al.* [24]. The article explores the concept of encoding data in network packet delays in order to exfiltrate data. For example, if the packet arrives with a delay between 1.5 to 3ms, it could indicate that a “0” is being encoded. Else, the symbol encoded is “1”. This work investigates the covert channel capacity and proposes a statistical approach to detect a specific case of network packet delay.

Two methods were proposed to detect inter-packet timing delay covert channel. The first one is based on the idea that an attacker tries to exfiltrate the data at the maximum bandwidth as limited by the channel. The probability distribution of interpacket delays that achieves the highest capacity is compared with the real distribution observed in the network. This is done by finding the input symbol distribution using the Arimoto-Blahut algorithm which is a method to compute the information theoretic capacity of a channel. This method has two main obstacles: the first one is that the optimal input delay distribution might not be unique and the second is that the matrix of probabilities is not constant over time and it has to be updated every time. The second method is only applicable to a two symbol bit stream and tries to identify two concentrations of inter-packet delay times. This technique is based on the assumption that the packet delays will concentrate on two very distinct values while in a normal communication, the delays are random. A couple of statistic comparisons are done based on this assumption.

The main drawback of this work is that these algorithms were not tested in a live network setting, leaving an uncertainty if they will work in the real world. For future work, the authors suggest the investigation of other variations of these Timing Channels, for instance, cases where the attacker redirects the traffic to two different machines and reconstructs the data this way.

2.2.4 Steganography Detection

A steganalysis method to prevent exfiltration of data is described in [25]. There, the authors propose an approach for detecting and removing hidden data in videos using passive and active steganalysis. It can be implemented with a Software as a Service (SaaS) with a form of silent proxy middleware that acts like an automated steganalyist. All video files which leave the network pass through this abstract middleware. The described approach uses Discrete Sine Transform (DST) and wavelet filters to remove hidden data from the video’s frames (active steganalysis phase). This is done before the video leaves the organization’s network, in an attempt to sanitize the media and actively prevent exfiltration. This way, exfiltration is prevented in situations where the passive steganalysis mechanism is unable to detect hidden data.

The results were encouraging in situations where the passive steganalysis did not correctly detect steganography. Although this seems a good thing at first glance, nothing is said about the quality of

the media after the sanitization of the data. This should be further explored and tested since these operations can cause damage to the files.

2.3 Botnet Data Exfiltration

We present here some background on the main problem that we want to tackle with this work. A botnet is basically a network consisting of compromised machines, usually called 'zombies', that are controlled remotely by one or more Command&Control servers that the attacker owns. These networks have been largely used in the past to engage in all sorts of malicious activities such as monitoring victim's activities by planting a keylogger on their machines or to sniff traffic, launching DDOS attacks, stealing the identity of the machine, and even to exfiltrate data from the victim's computer which is what we are interested to study. In this section, we will provide relevant work on botnets and botnet data exfiltration that focuses on the detection of such threats.

Based on Network behavior, Strayer *et al.* [26] presented a method to detect botnets that use Internet Relay Chat (IRC). The authors try to detect the threat only by analyzing the network flows and not relying on traffic content, port numbers or watching DNS Servers.

Strayer *et al.* approach has the following phases: Filters, Classifiers, Correlator, and Topology Analysis. The *FIters* phase aims to reduce the network data flows gathered initially. The filters used were: remove traffic that is not TCP-based flows, packets containing only SYN and RST flags since they indicate that communication was not established, remove traffic with bulk transfers, use a 300-byte size cutoff on the packets, and finally remove brief flows of traffic. At the end of this stage, the dataset was reduced greatly, leaving the malicious botnet flows on the data traffic.

The next phase called *Classifier* has the objective to leverage machine learning algorithms to classify the network flows. The authors tested three algorithms: J48 Decision trees, Naive Bayes, and Bayesian Networks after extracting features and characteristics of the network traffic like flow duration, average bytes per packet, and bits per second for flow.

The next *Correlation* stage aimed at finding relationships between the flows. Flows are said to be correlated if they exhibit one or more of the following properties: they are the product of similar applications, there is a casual relationship where an event on one flow causes an event to occur on another flow, and if there are one receiver and multiple receivers. A correlation algorithm is proposed as well as the characteristics of the flows. The results of the algorithm suggested clusters of correlated flows that needed further investigation with topology analysis.

Finally, *Topology analysis* allows one to pick the most correlated flows and, automatically, analyze the overall structure of the correlated flow pairs in a graph, being possible to identify the communication structure of the botnet. The method proposed showed very good results, showing that is possible to

check real-time packet traces and extract evidence to discover botnet controllers. We may use some ideas of this work and incorporate them into our solution.

Botnets use encryption and obfuscation techniques to hide their communications when exfiltrating data. An approach to detecting data exfiltration by botnets in web traffic is explored in [7] by Al-Bataineh and White, presenting an analysis of a botnet called Zeus. The focus of the work is to detect this type of attack in HTTP-POST requests where data can be encrypted or compressed since previous work didn't explore this fact.

To achieve this, the authors first constructed a couple of datasets containing benign and malicious connections in order to extract features and train a classifier. Some features were considered such as the packet's entropy which serves as a very good indicator of an encrypted packet; the packet size and the Byte Frequency Distribution (BFD). These features were then extracted with a Python application running a modified version of the dpkt[4] package. To classify the traffic, a couple of classifiers were tested but, the authors came to the conclusion that a J48 tree classifier performed best achieving very good results. Although this is promising, one should consider the processing time of analyzing the traffic when considering more practical approaches like implementing the method with an intrusion detection system.

For future research, the authors suggest various extensions: partial content inspection of packets to decrease processing time and power, implementing the classifier with an IDS, experimenting with different features, and to study other botnets and compare that analysis results with this work.

Recent work has been done on stealthy behaviors of botnets. According to Venkatesan *et al.* [8], these stealthy mechanisms can be classified into two types: anti-signature stealth and architectural stealth. The *anti-signature stealth* techniques are those that modify the characteristics of the traffic generated by the botnet to avoid detection. This can be done by encrypting the traffic, randomizing the number of packets per flow or by other kinds of obfuscation. The *architectural stealth* techniques, aim at building topology aware bots to reduce exposure of malicious traffic to network detectors. They exploit the fact that these detectors are normally placed on specific parts of the network, for example, a location where all the traffic can be monitored.

The work by Venkatesan *et al.* has the objective to detect P2P based botnets that utilize architecture stealth techniques by deploying multiple detectors along the network. The authors attempt to periodically change the placement of the detectors with the objective to increase the attacker's effort and likelihood of detection by creating uncertainty about the location of these detectors. The authors presented two metrics to evaluate the proposed methods: the minimum detection probability and the attacker's uncertainty. An algorithm to calculate the minimum detection probability was also created and the methods were tested via simulations. The tests showed that the proposed solution could reduce the likelihood of

[4] https://github.com/kbandla/dpkt

successful data exfiltration attacks. As for future work, the authors suggest introducing a probabilistic model to the solution to account for false negatives in the detectors.

Another work by Venkatesan *et al.* with the goal to detect stealthy behaviors by botnets is described in [9]. The authors try, once again, to minimize the time that the botnet is active by maximizing the number of bots identified and removed from the network with a limited number of resources. The solution described uses a reinforcement learning approach to optimally deploy a limited number of defensive mechanisms like honeypots and network-based detectors within the network. Honeypots are used to detect intrusions while network-based detectors identify bots that coexist with benign machines and persist on the network. The authors discuss the threat model for the attack and discuss the lifetime cycle of the botnet.

Firstly, they assume that to construct a resilient botnet, a new bot will scan the network looking for machines to attack by using a stealthy scan. The stealthy scan aims to minimize the risks for detection, enumerating the active connections for the underlying host and sending only discovery probes to those machines. After the enumeration, the bots compromise these machines and adds them to their list of peers. It is assumed a minimum number of bots to create a resilient botnet. If the minimum number of bots is achieved, the bots communicate with each other and inform the attacker of their status. Otherwise, if the number of peers drops below the predefined threshold, the bot returns to the scanning phase. Having this in mind, the reinforcement learning approach is devised with respect to this threat model, making it possible to model the actions of the bots and the goal to be achieved by the defender. The method was tested with multiple approaches in a simulation. The reinforcement learning approach was able to beat the other approaches by averaging only 3 bots at the same time in simulation and reducing the average maximum bot lifetime to 20 days.

In comparison with other approaches, the result is very good considering the limited resources to deploy. However, the bots still persist for 20 days in the network which could make some damage by steal information from the machines. The authors propose some extensions to their work like more extensive experiments and considering the implications of a detector's misclassifications.

2.3.1 Machine Learning

Since our proposed solution is based on a machine learning approach, this subsection of the document will present some background on the matter. Machine learning is a field in Computer Science that allows computers to learn how to solve tasks, make predictions, or classify sets of data without being explicitly programmed to do so. This is possible because we can teach machine learning systems to learn from previously obtained data relevant to the problem we are trying to solve. This previously learned data can be represented by values called features, distinguishing with more precision sets of data that we want to test and classify. For example, in a botnet scenario, if a machine in a network checks for updates with the

Command&Control host and downloads them every fixed amount of time, we can model these repetitive patterns with features and employ a classification algorithm that will tell us if the traffic we are seeing is malicious or not. Obviously, recent botnets try to be stealthy when exploiting the victim's network but we can study a family of these more covert botnets and try to come up with features that represent well the problem we want to solve.

As pointed out before, we are going to use machine learning in our solution in order to classify network traffic and connections. Allied with clear and relevant features, the learning algorithms could potentially, help us detect abnormal behaviors in the network raising an alarm. We should seek to model these features very carefully in order to, capture with more accuracy the stealthy communication that botnets exhibit. For example, the bots may try to exfiltrate data from the victim's machines in encrypted packets. Also, the exfiltration could have low throughput. These ideas are based on previous work in the area that uses this exact same notion. In [4] they use a one class classifier to detect low throughput data exfiltration in the DNS protocol. This research is very helpful as it solves a similar problem. We could model the features used in that work and transpose the idea to our solution. Another interesting work by Strayer *et al.* [26] provides some ideas we can use. As explained in the previous subsection, they try to identify botnets using a four-stage approach which one of them involves classifiers. Although they didn't necessarily used the classifiers in the end, we could pick up some of those ideas and attempt to extend that specific work in order to achieve our system's goals. In the end, we will follow more closely the work done by Zhao *et al.* [10] and Beigi *et al.* [11]. Both works approach the detection of botnets with machine learning using flow features of the traffic. We will use the dataset from the latter work that it is publicly available, and will try to expand the research done there.

Of course that in theory, machine learning seems to be able to solve any problem out there with the premise that well-defined features should be enough to classify and predict datasets or outcomes. It should be noted however that most of the times, the data that we use to train the systems might not be that good, that is, it may contain noise and uninteresting data that is irrelevant to the task at hand. Another thing to have in mind is that the feature selection might not represent with reliability the data we want to classify, hurting the accuracy of the classifier.

3

Architecture

Contents

3.1 Introduction

The system implemented for the thesis is a detector based on machine learning algorithms capable of identifying malicious connections engaged by botnets. The classification is supervised and because of that, we trained multiple classifiers with a publicly available dataset [11] that contains traces of more than a dozen of different types of botnets separated by a training and testing dataset. The dataset used consists of two network trace files (.pcap) each containing more than 2 GB of information about the packets exchanged that will serve as logs to be analyzed. The network traces are generated by the authors on real machines with botnets that simulate well the reality of these malicious attacks.

The system will be a useful tool not only to detect botnet activity, but to also aid analysts in identifying other types of attacks such as data exfiltration. The results from this system can provide strong correlations between logs maintained on the victim's computer such as logs that keep track of file accesses and modifications and the presence of malicious activity in the network. In the future, the detection of other threats could be explored by extending this exact architecture that we propose.

3.2 Overview

We present here the system's architecture implemented for the solution. Each component will be described with more detail in the next paragraphs as well as the requirements of the system that should be met along the way. Previous research done on this problem discusses the main defense and monitor mechanisms but very few create fully functioning implementations capable of detecting effective botnet activity detection. We want to be able to create a prototype that will serve as a base to a detection framework supported by machine learning and that could be integrated with multiple tools such as IDSs, anti-virus software, and firewalls. Figure 3.1 shows the architecture and it's main components as implemented in the solution. The main idea is to simulate how can this system work in the real world regarding detection of botnet activity consisted of several steps, and potentially exfiltration, beginning with the collection of log data and resulting in a full report detailing what machines are most likely to be compromised and controlled by an attacker. We want to be able to detail these reports and alerts with complete information and statistics so that forensic analysts can act upon these types of threats. The system consists of the following steps:

1. **Data Source**: The first step is the collection of log files of network traces, that is, pcap files that contain information about the connections and the packets exchanged in the network.

2. **Data Collector**: With the data source defined, we will pre-process this data with a module that will transform the data into flows and host information so that features can be extracted by the next model, and the classifier is trained.

3. **Feature Extractor**: The extraction of features is done next, by processing the output of the data collector. The features are selected by the analysis of the logs and by trying to establish patterns on the connections. This module just calculates the values necessary to construct the vectors to feed to the classifier.

4. **Classifier**: Next, the vectors will be used to train the classifier algorithms. After that, the classifier will predict test vectors extracted from new and unseen data.

5. **Evaluator**: The evaluator module will return a report of each classifier tested and trained giving information about what classifiers perform best, showing us multiple metrics such as accuracy and recall.

6. **Decider**: Finally, the last step of the system features a decider that will make sense of the results of the classifier and will tell us what are the most probable machines to be infected. Also, it uses data from the feature vectors extracted. Since we are classifying connections, we want to be able to point out the specific problematic IPs.

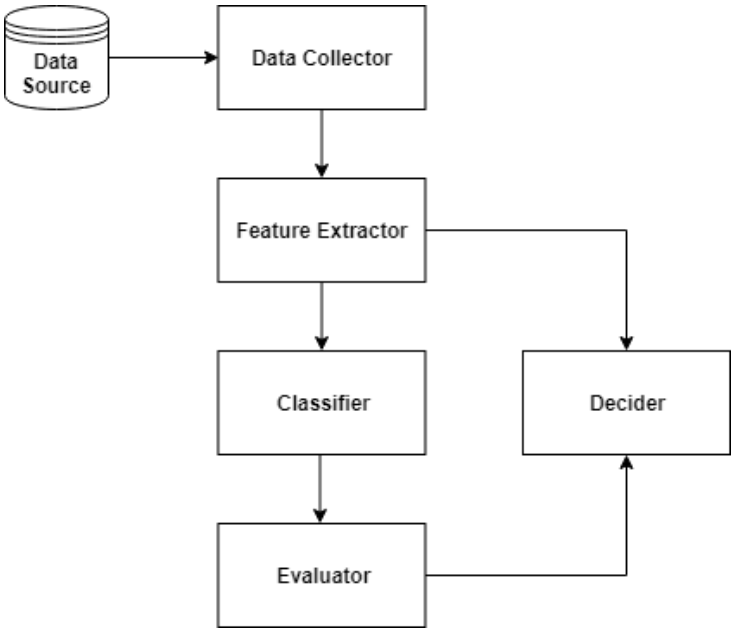


Figure 3.1: Diagram of the architecture implemented

3.2.1 Data Source

The *Data Source* is basically what will serve as the system's input. In this case, the data source corresponds to the network trace logs provided by the dataset previously mentioned. This is data that should be constantly monitored and captured so that the system can consume it right away.

The dataset is divided into two large .pcap files, one for training and one for testing, having 5.1 GB and 2.1 GB in size respectively. The two network traces consist of multiple connections made by more than a dozen botnets and to keep the problem close to reality, the authors of the dataset added a great number of normal connections. Tables 3.2 and 3.3 show exactly the distribution of the training and testing dataset in terms of botnets used, the number of flows, and the percentage of flows in the data. Notice that in the training data, the dataset is very unbalanced. We address this issue by cutting the IRC connections from the training. This is discussed in Chapter 5 where the testing environment is described.

Type of Flow	Number of flows	Flow Percentage
IRC	117537	57.8%
Neris	14444	7.1%
RBot	31527	15.5%
Virut	863	0.4%
Normal flows	39046	19.2%
Total	203417	100%

Table 3.2: Training dataset constitution

Type of Flow	Number of flows	Flow Percentage
IRC	15157	8%
Neris	19481	10.3%
RBot	166	0.1%
Menti	4539	2.4%
Sogou	46	0.02%
Murlo	8669	4.6%
Virut	33102	17.6%
IRCbot and black hole 1	37	0.02%
Black hole 2	19	0.01%
Black hole 3	122	0.1%
TBot	485	0.3%
Weasel	44687	23.7%
Zeus	283	0.2%
Osx_trojan	15	0.01%
Zero access	1234	0.7%
Smoke bot	66	0.04%
Normal flows	60006	31.9%
Total	188114	100%

Table 3.3: Testing dataset constitution

So that we can train our system later, we needed to know what are the malicious IPs, belonging to the botnet connections and those that are normal activity. Table 3.4 shows the IPs that we found on the logs and the respective labels.

All the other IPs that do not appear on this table will be considered non-malicious and will be treated as normal activity in the network. Anytime one of these botnet IPs appear on a flow, that flow will be

Botnet Name	IP Addresses
IRC	192.168.2.112 ->131.202.243.84 192.168.5.122 ->198.164.30.2 192.168.4.118 ->192.168.5.122 192.168.2.113 ->192.168.5.122 192.168.1.103 ->192.168.5.122 192.168.4.120 ->192.168.5.122 192.168.2.112 ->192.168.2.110 192.168.2.112 ->192.168.4.120 192.168.2.112 ->192.168.1.103 192.168.2.112 ->192.168.2.113 192.168.2.112 ->192.168.4.118 192.168.2.112 ->192.168.2.109 192.168.2.112 ->192.168.2.105 192.168.1.105 ->192.168.5.122
Neris	147.32.84.180
RBot	147.32.84.170
Menti	147.32.84.150
Sogou	147.32.84.140
Murlo	147.32.84.130
Virut	147.32.84.160
IRCbot and black hole1	10.0.2.15
Black hole 2	192.168.106.141
Black hole 3	192.168.106.131
TBot	172.16.253.130, 172.16.253.131, 172.16.253.129, 172.16.253.240
Weasel	158.65.110.24
Zeus	192.168.3.35, 192.168.3.25, 192.168.3.65, 172.29.0.116
Osx_trojan	172.29.0.109
Zero access	172.16.253.132, 192.168.248.165
Smoke bot	10.37.130.4

Table 3.4: Malicious IPs in the dataset

considered malicious and, therefore, will be labeled as such in the classifier.

The data will then be processed by a later module in order to extract relevant information for the classification of connections. This information includes connections made, IP addresses, port numbers, packet payloads, websites accessed and flow features such as packet sizes, flow duration and the inter arrival time of packets. This will allow us to infer some characteristics of the attackers and their behavior. All these features and characteristics will be explained in the next paragraphs when we discuss the other architecture modules.

3.2.2 Data Collector

The *Data Collector* component will serve as a parser for the input described above. This module will allow us to process the data so that we can extract the logs information and feed it to the classifier. Since the network logs have too much information, one of the great challenges of the Data Collector and the

system as a whole, is to process large chunks of data in the fastest way possible without compromising the effectiveness of the classification job. For example, we first need to load all the pcap file data into memory and that can take a lot of space in a machine. After that, we have to make sure to parse the packets in a way that the result gives us all the connections or flows from each specific network conversation. And for each of these connections, we have to keep values that will allow us to calculate the features to be used in the classifier, for example, packet sizes and arrival time of the packets. This is the main job of the data collector, to collect and parse the information present in the network traces.

3.2.3 Feature Extractor

The next component is the *Feature Extractor* where we have a module to extract relevant features from the collected information. This is done by going through each parsed connection and retrieving and calculating the data necessary to construct the feature vectors. The features extracted are not automatic and should be selected before extracting. The features selected in this module is the result of the analysis of the files as well as observation and patterns discovered by other works on the field. We tested a couple of features not found in the literature about botnet activity as well that we think that could perform well given the circumstances. This features were based on the Entropy of the hostname and the url GET request, in case a HTTP connection is made. Also, we added a feature regarding the biggest packet in the flow.

```
1 1312967063 147.32.84.180:137 147.32.84.255:137 UDP 27256 16981.285 110 ...
2 1312967064 147.32.80.9:53 147.32.84.180:1025 UDP 108898 16977.540 64 ...
3 1312967066 147.32.84.180:1027 74.125.232.195:80 TCP 1511 0.045 62 ...
4 1312967072 147.32.84.180:138 147.32.84.255:138 UDP 24562 16461.671 243 ...
5 1312967081 147.32.84.180:1035 239.255.255.250:1900 UDP 1050 6.006 175 ...
6 1312967183 147.32.84.180:1039 60.190.222.139:65520 TCP 11557 3761.725 62 ...
7 1312967196 147.32.84.180:1040 94.63.149.152:80 TCP 28496 0.216 62 ...
8 1312967198 147.32.84.180:1041 60.190.223.75:2012 TCP 61419 2.568 62 ...
9 1312967200 147.32.84.180:1042 195.88.191.59:80 TCP 43433 13.535 62 ...
```

Listing 3.5: First lines of the training .flow file

The output of this module in the system is a flow file (.flow) that for each line, it contains the information about the bidirectional conversation between two machines as well as values corresponding to features that we considered relevant and will be used for the classification task. The data parsed and saved in the flow file can be found in Table 3.6. Some of the information extracted such as Host name and URL GET Entropy were only considered in HTTP and HTTPS connections since it only makes sense there. Listing 3.5 shows the first lines of the training .flow file truncated for readability purposes.

Field	Notes
Start time	Time in which the beggining of the connection takes place
IP and Port of first machine IP and Port of second machine	IP and Port with the form IP:Port
Protocol	Protocol used in the Flow (ex: TCP, UDP)
Flow Size	Size in Bytes of the flow
Flow Duration	Duration of the flow in seconds
First Packet Size	Size in Bytes of the first packet of the connection
Average Packet Size per flow	Average size in Bytes of the connection
Number of Packets	Total number of the packets in the connection
Percentage of Outgoing Packets	Percentage of outgoing packets by the requester
Number of Small Packets	Total number of packets with size bigger than 63 and smaller than 400 bytes
Percentage of Small Packets	Percentage of small packets in the connection
Average Payload Size	Average size in Bytes of the packets payloads
HTTP GET Request URL	URL of the page that the machine requests (only HTTP)
Hostname	Name of the machine in which an HTTP request is made (only HTTP)
Average Inter Arrival Time	Average arrival of each packet in the machines
Biggest Packet Size	Size in Bytes of the biggest packet in the connection
Byte per second per Flow	Quantity of Bytes transfered per second on a connection

Table 3.6: Information parsed from the Data Source logs

The Pre-Processing phase of our system completes when the Data Collector and the Feature Extractor finishes execution. Algorithm 3.7 resumes below the steps necessary for this phase to run:

input : pcapList - List of pcap files
output: Processed .flow files
1 **for** $f \in pcapList$ **do**
2 packets $\leftarrow$ `readPcap(`f`);`
3 sessionList $\leftarrow$ `sessions(`packets`);`
4 **for** $s \in sessionList$ **do**
5 flow $\leftarrow$ `extract(`s`);`
6 **if** f $is\ training$ **then**
7 `writeToFile(`flow, trainingFlows`);`
8 **else**
9 `writeToFile(`flow, testingFlows`);`
10 **end**
11 **end**
12 **end**

Algorithm 3.7: Data Collector + Feature Extractor

On line 5, the extract function corresponds to the extraction of features from the connections loaded by the Data Collector. It basically consists in computations of values by iterating through the packets. For example, the size of the whole flow is calculated by adding the sizes of each packet. These features are stored in an array called *flow* that will be written and appended to the .flow training file if the input pcap is for training. Else it is appended to the testing flow file.

3.2.4 Classifier

After collecting the data, parsing it and extract the features, the Classifier is the next component of the system. Here, the module will have two important steps: train the classifier and test the classifier with new and unseen data. Since we had two different pcap files as a source of our input, we will use one for the training and one for testing. We also have to label each feature vector into two different classes. In our case, we used binary classification (0 or 1) to differentiate between normal and botnet activity.

This way, we will be able to automatically detect and distinguish malicious connections from normal and benign ones. In this step, we tested a lot of algorithms in order to, empirically, decide the best one. Table 3.8 shows the algorithms tested in the implementation of the solution. We also implemented a couple of features here to aid us with the testing of the classifiers so that we can get the best out of the algorithms offered by the machine learning library used. For example, we trained and tested the classifier with cross validation, implemented filters on the datasets so that we can assess what data was important in the training phase and even tested some feature selector algorithms, to help us decide what are the least important features for the detection job. In the end, we tried to vary the number of training vectors to see if, in reality, what was the least amount of data needed to detect well the malicious flows. All of these details will be explained later in Chapter 4 and 5 where we will discuss the implementation and the results obtained.

Classifier Algorithms
Linear Dsicriminant Analysis
Support Vector Classifier (SVC)
Linear SVC
Naive Bayes
Gaussian Naive Bayes
Logistic Regression
Neural Networks
Decision Tree
Random Forest

Table 3.8: Classifiers tested in the system

3.2.5 Evaluator

The *Evaluator* component provides the reports and results of the classifier. In our solution, we evaluate the system based on a lot of metrics such as accuracy, precision, recall, time spent in training, f1 score and we even look carefully at the confusion matrix which gives us the information about false positives, false negatives, true positives, and true negatives. In the end, the goal is to maximize the accuracy, precision, and recall of the classifier as much as possible. Obviously, in a machine learning problem and given the task at hand, we should try to obtain the best results possible given the input. The implemented system should be decent enough to distinguish botnet connections and we should be able to then analyze these results to see how can they be improved. The metrics that we will look to the most are accuracy and recall. Recall will give us an idea of how many connections did the classifier predicted correctly. On the other hand, the Recall metric will tell us the number of malicious flows we detected correctly.

The other goal of this module is to output the evaluation graphics and figures showed later on Chapter 5. We want to be able to analyze and review these figures so that we can make sense at the results of the classifier as a whole. This will allow us to choose the best parameters, algorithms, and features to use in the system in the future.

3.2.6 Decider

Finally, the *Decider* component complements the *Evaluator* in the sense that given the final results of the classifier step, we may now decide what IP addresses we should pay more attention first since they might have a higher probability of being compromised. For example, if the classifier algorithm says that 60% of the connections made by an IP address take part in the scheme of a botnet, the Analyzer will simply output that this machine should be further investigated as it is most certainly compromised.

So, in this final step of the system, we will use the information from the Evaluator results and the Feature Extractor module to decide what are, in reality, the malicious IPs in the network. Here we will use a combination of the best classifiers in order to make a decision. This is important because, each algorithm has its strengths in detecting one specific botnet, improving the overall results of the Decider.

This component is fairly simple as shown in Algorithm 3.9, presenting the logic behind the Decider module. First, from the classifier results, a dictionary is initialized with the correct and incorrect predictions for each IP address found (lines 2-20). A decision is made based on two thresholds, $threshold$ and $flowThreshold$. The $threshold$ variable represents the ratio that we do not want to consider. For example, if the classifier says that a specific IP address is malicious 80% of the time then, the Decider considers that machine. On the other hand, the Decider takes in consideration the number of connections classified with the $flowThreshold$ variable. An example of this is a machine that only com-

municated 20 times in a network. If we define the $flowThreshold$ as 300, those machines are not considered. In the end, the list of IPs is printed. Notice that this algorithm is executed for all classifiers chosen. The selection of algorithms is discussed with more detail in Chapter 5.

```
   input  : predictions, testLabels
   output: List of malicious IPs
 1 results ← dict();
 2 for i ← 0 to sizePredictions do
 3 |   sender ← senderIP;
 4 |   receiver ← receiverIP;
 5 |   if predictions[i] == testLabels[i] then
 6 |   |   v ← 0 ;
 7 |   else
 8 |   |   v ← 1 ;
 9 |   end
10 |   if sender not in results then
11 |   |   results[sender] = [0, 0];
12 |   |   if sender is malicious then
13 |   |   |   results[sender].append(1)
14 |   |   else
15 |   |   |   results[sender].append(0)
16 |   |   end
17 |   else
18 |   |   results[sender][v] += 1;
19 |   end
20 |   if receiver not in results then
21 |   |   results[receiver] = [0, 0];
22 |   |   if receiver is malicious then
23 |   |   |   results[receiver].append(1)
24 |   |   else
25 |   |   |   results[receiver].append(0)
26 |   |   end
27 |   else
28 |   |   results [receiver][v] += 1;
29 |   end
30 end
31 for res in results do
32 |   if results[res] == 1 then
33 |   |   if results[res][0] / total > threshold and total > flowThreshold then
34 |   |   |   print(res);
35 |   else
36 |   |   if results[res][1] / total > threshold and total > flowThreshold then
37 |   |   |   print(res);
38 end
```

Algorithm 3.9: Decider

4

Proposed Solution

Contents

4.1 Implementation

We will now detail and discuss all the implementation process of the architecture that we presented in Chapter 3. The solution created has 5 essential steps:

1. **Analysis** - Step where the data from the logs is analyzed. The goal here is to observe the threat's behavior with a forensic examination in mind.

2. **Pre-Processing** - This step will fundamentally have two goals, to transform the logs data into a better representation so that we can easily manipulate, and to extract features from the connections.

3. **Classification** - The classification is straightforward, it is where we are going to classify each connection by labeling each feature vector extracted in the previous step.

4. **Evaluation** - This next step has the goal of receiving the results from the Classification step in order to calculate the metrics that we decide that are useful to present. It will be in charge of creating the graphics and statistics that will help us in checking if the solution is good or not.

5. **Decision** - Finally, the Decider will be the last phase of the system as mentioned briefly in the previous Chapter. It will be used to determine what are the malicious IPs present on the network trace logs.

The next sections of this chapter will serve to discuss and explain the design choices made for the detection system by going through each of the steps stated above.

The solution was developed in the Python[1] language version 3.6 making use of the Sckikit-Learn[2] library that implements all the machine algorithms that we need. As stated in Chapter 3, the input of our system will be the network logs from the publicly available dataset[3] found at the University of New Brunswick website in order to train and test the system. To analyze the traces we used Wireshark[4] since it provides a nice graphical interface, and for the dataset processing, we created scripts using the Scapy[5] library in order to manipulate the data. In terms of structure, the project folder tree diagram can be viewed in Figure 4.1.

We will now explain each phase of the implementation in the following sections.

[1] https://www.python.org
[2] http://scikit-learn.org
[3] http://www.unb.ca/cic/datasets/botnet.html
[4] https://www.wireshark.org
[5] https://scapy.net/

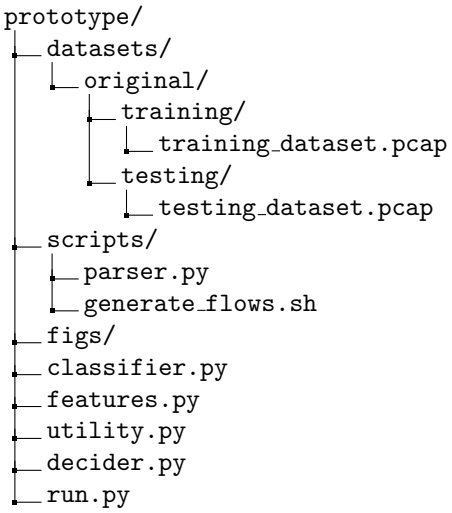


Figure 4.1: Folder tree diagram of the project

4.2 Analysis

Before implementing the system, we had to first analyze the network logs that we gathered from the datasets so that we could later, extract relevant information that we could use on the classifier phase. As stated in Chapter 3, both datasets were big and so, the task of analyzing the data in a thorough and manual way is a tedious and expensive one. Because of this, we developed a generic methodology to analyze the packets in a more efficient way, given the size of the datasets.

4.2.1 Analysis Methodology

In a first approach of the analysis, we decided to split the pcap into multiple smaller ones so that we could easily load them into our system's machine. We did this by making use of the editcap[6] tool. There was a necessity to split the files because of the memory they occupy. For example, the testing dataset occupied approximately 55 GB in memory. To analyze the packets and to open the files in Wireshark initially, we had to use this tool.

This particular tool lets us manipulate and edit network trace files in a variety of ways such as specifying how many packets we want to delete or exclude, shrink the capture file by truncating packets or even remove duplicate packets. In our case, we wanted to just do a simple analysis of the files by inspecting the inside the packets. To do this we edited the trace file by splitting it into various ones by indicating how many packets we wanted each file to have. We used the following command:

[6] https://www.wireshark.org/docs/man-pages/editcap.html

```
$ editcap -c <number of packets> <infile> <outfile>
```

Listing 4.2: Using the editcap tool to edit .pcap files

This way, the larger pcap file transformed into multiple ones, each one having a maximum of the number of packets specified with the -c flag. Another useful thing that we did initially, was to filter the data such that we only extracted malicious or non-malicious traffic in separate files. This way, we could analyze connections made by specific IP addresses. To do this, we used the thsark[7] tool. For example, thsark allows us to transform the original pcap into another one according to some sort of filter like saving only TCP or UDP connections. All filters available on Wireshark can be used with this tool. In our particular case, we wanted to filter the data by IP address so that the result only had the packet logs from a specific IP address. To accomplish this, one could use the following command:

```
$ tshark -r <infile> -w <outfile> -Y ip.addr=192.168.1.1
```

Listing 4.3: Using the tshark tool to filter for IP addresses

This gives us a brand new pcap with traffic filtered from the -Y flag, which gives us only connections from the IP address 192.168.1.1.

These two mentioned tools gave us, in a first approach, the means to analyze and process the data more efficiently. The problem with the approach of splitting the dataset though is that some connections might be split as well, meaning that we could lose some valuable information from the logs. However, it is certainly faster because, with the data broken down, we could potentially make use of multi-threading in our system to process the logs. In systems with fewer resources, splitting is the way to go because of the memory that the datasets occupy in the system. Processing the data in one shot not only is slower, but it also drains the memory resource pretty quickly and so, we have to keep in mind of the existing trade-offs, so that we can make the most out of the system.

With the help of the above tools, opening the files with Wireshark was easier. Figure 4.4 shows us a screenshot of the testing dataset when opened in Wireshark.

From the picture, we could see some HTTP as well as some TCP packets being exchanged. And we can see that it loaded around 5 million packets. Analyzing this by hand would be hard and time-consuming. The training dataset, on the other hand, was impossible to open on our machine as it occupied too much memory.

From this quick analysis of the trace, we gathered some information that could be useful in using on the classifier such as host names, http connection information and even patterns on the urls that the computers requested. This information allied with a couple of flow-based features tested in the literature

[7] https://www.wireshark.org/docs/man-pages/tshark.html

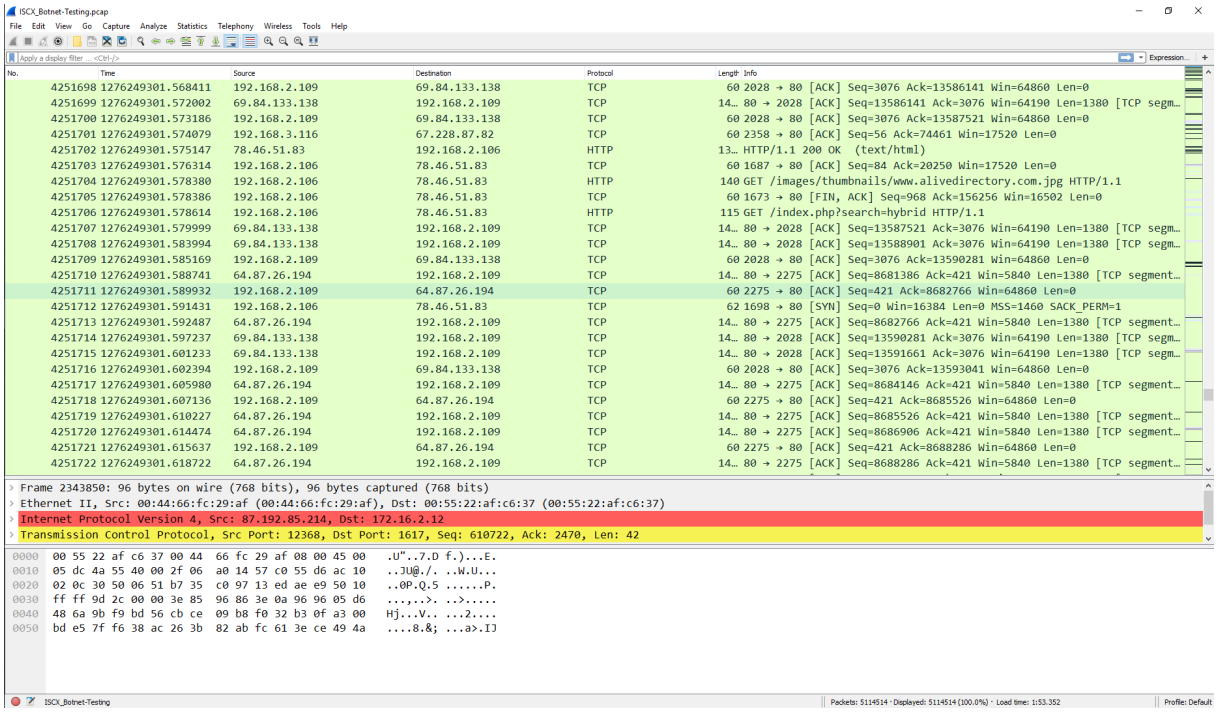


Figure 4.4: Some testing dataset packets as seen with Wireshark

gave us a good start and allowed us to begin constructing the rest of the system.

4.3 Pre-Processing

We will now begin to describe the pre-processing done on the datasets that allowed us to transform the data into a more standard representation that we could easily extract the information from the logs. The dataset used has two pcap files, one for training and one for testing containing a mix of benign traffic and malicious traffic. The malicious traffic contains more than a dozen of different botnets. In order to process and input the data on our framework, we made use of the Scapy library already mentioned since it converts all network packets into objects that we can access within the code, making it easier to manipulate.

On the analysis section above, we talked about specific tools that helped us analyzing the network traces. For the specific case of our final version of the system we tried not to split each dataset but that was impossible, given the resources we had. Although it is slower to load the pcap file at once, it is also the best way to not lose information. The training dataset, since it was bigger, was split into two while the testing dataset was all loaded at once.

After the analysis of the logs and after the files have been split or filtered as discussed in the paragraphs above, the next step is to convert the information in the dataset into bidirectional flows. Bidirec-

tional flows represent the connections that we will be classifying in the next phase of the system and consist of the packets exchanged by two particular machines in one session. Scapy does all the work here, saving all the connections in a Python dictionary that we can use. This is done using the sessions method present in the library. All training dataset and testing dataset flows will be saved in two .flow files, one for the training and another for the test, respectively.

To summarize, the pre-processing phase has the following steps:

1. Split the data or filter it using the tools thsark and editcap. This step could be potentially skipped if the machine can load all the dataset at once, but it is not efficient.

2. Load the data into our system and convert it into bidirectional flows.

3. Take the bidirectional flows and the information that we think is useful for the classifier and save it into a .flow file, where each line corresponds to a connection or flow. (Feature Extraction)

For all the pre-processing phase summarized above, we created a helper Python script to receive a pcap file as input and output the connection information to another file called *parser.py*. The script makes use of the sessions method within the Scapy library to process the packets and transform them into flows. After that, we only have to loop through all the sessions, extract the flows and save them in the respective .flow files.

4.3.1 Feature Extraction

For the machine learning part of our framework, we only considered flow features since they can capture botnet behavior without having to extract specific information of each machine. For example, we could train the classification algorithm with host characteristics like the IP address of the hosts or the ports with which they are establishing a link. What could potentially happen is that when we apply this system to other types of bonets or different IPs, these kinds of features would turn out to be useless since it captured similarities only on that specific network with those specific IP addresses. With that in mind, we will only consider features that allow us to classify not only previously seen and trained botnets but also new and unknown threats.

The features that were considered and evaluated in the framework can be found in Table 4.5. The features refer to flow characteristics of the traffic such as flow size and flow duration that identify a specific connection and not information that is relative to one machine or another.

Most features presented here were already tested in the literature in detecting centralized and decentralized botnets. More specifically, from the article of the dataset [11], the authors use some of these features and discuss their usage since they have brought good results in the past. On the other hand, we introduce here new features that could help in training the classifier.

Features
Flow Size
Flow Duration
First Packet Size
Average Packet Size
Number of Packets Exchanged
Percentage of Outgoing Packets
Number of Small Packets
Percentage of Small Packets
Average Payload Size
Hostname Entropy
HTTP GET URL Entropy
Biggest Packet Size
Average Inter Arrival Time

Table 4.5: Features tested with the multiple classifiers

- Hostname Entropy
- Entropy of the HTTP GET URL
- Biggest packet in the flow

The first two features are based on characteristics of HTTP connections with the idea that the host names that the machines exchange packets with can sometimes be easily identifiable. This is because the host name has the characteristic of being composed of random alphanumeric characters. From this idea, we can employ an Entropy function to the string and extract information from there. In information theory, entropy was first introduced by Shannon in [27] and can be defined as the expected number of bits of information in a message or string. It can be expressed by the following formula:

$$H(X) = -\sum_{i=0}^{N-1} p_i \cdot log_2\ p_i \tag{4.1}$$

Where p_i is the probability of each character in the string. If the string has characters that appear frequently, the entropy value will be much higher than a string with a more variety and uniqueness of characters. This is good for cases where the hostname does not repeat characters.

The second feature tested, rests on this idea too, but it is a lot more difficult to make it work well in practice. The entropy of the GET url could work because bots tend to mask information on the url such as using base64 encoding on the requests. The problem is, normal websites with usual requests might

have high entropy as well when the url arguments are cookies for example. Nonetheless, we test this feature too so that we can discuss it later in Chapter 5.

Finally, we added a last feature that is based on the biggest packet found on the connection. Sometimes, to update the botnet for example, the infected machine will download an executable file to install on the host with malware. Since these executable files can be big, this feature is useful for that type of cases.

The extraction of features is done with a helper Python script called features.py which basically receives as input the list of flows (.flow file), splits each flow into feature values and chooses the features to be evaluated constructing a list. In addition to the feature vector, we have to construct a vector that will store all the correct prediction for each flow. This is possible because we have knowledge about the specific IP addresses that are malicious in the network. For the labels, we will be using a binary classification that is, if the vector we are labeling is considered malicious, the label associated is 1. Else, the label will be 0.

This phase ends when the flow file is successfully transformed into the feature vectors that will be used in the classifier.

4.4 Classification

As stated in the sections above, we used the scikit-learn library on the implementation of our system since it simplifies our work by providing us with all necessary machine learning algorithms. Initially, to test out our code, we started by doing the pre-processing phase with only one kind of botnet. From there, we expanded the implementation to process all botnet data that we can find on the datasets.

We tested our framework with several popular classifiers as seen in Chapter 3 Table 3.8 such as Support Vector Classifiers,, Naive Bayes, Logistic Regresstion, Decision Trees, and Neural Networks.

For this phase of the system, we created a Python script called *classifier.py* which is the core of the framework. It takes the feature vectors created in the pre-processing phase and trains all the different classifiers that we want to test. After that, it tests the classifiers with new data and outputs the results to be evaluated. Also, the created classifier script implements the class that stores the feature data as well as a class that implements methods that wrap the scikit-learn package.

4.5 Evaluation and Decision

The last steps of the system involve **evaluating** the results of the classifier so that we can **decide** what are the IPs in the network that are compromised. We can then act upon this information and remove the threat of the botnet.

4.5.1 Evaluation

On this step of our detector, all the methods to output the results are implemented in the *classifier* script file and it wraps some scikit-learn methods that allow us to be able to calculate the metrics that we want to analyze. The metrics used will be discussed in the next chapter where we will support our results with graphics. In a first approach, we decided to test multiple classifier algorithms by analyzing their results on some main metrics such as precision, recall, f1-score, confusion matrix and accuracy. Then, we focused our experiments on the best classifiers. In our experience, Random Forests, Decision Trees, Linear Discriminant Analysis, and Neural Networks were the best algorithms tested. Forests and Trees performed well and had the fastest training and testing time, making them suitable for the detection of abnormal activities in the least time possible.

To test the classifiers, we used cross validation with a variable value for the folds to be trained. We also tested the classifiers without cross validation where we let an unseen set of feature vectors to be classified that belonged to the testing dataset.

The implemented methods for this component are the following:

- cross_validate_test: Does the cross validation, outputting the learning curve of the algorithm.
- plot_roc: Plots the Receiver operating characteristic (ROC) curve of the classifier and saves it into a file.
- plot_precision_recall_curve: Plots the Precision-Recall curve and saves it into a file.
- plot_confusion_matrix: Plots the Confuison Matrix of the classifier and saves it into a file.
- plot_feature_importances: Plots a graphic showing the features that most influenced the the classification. This is available for two particular classifiers, Random Forests and Decision Trees.
- print_results: Prints a report of each classifier tested by showing precision, recall, accuracy, f1-score as well as true/false positives and negatives.

4.5.2 Decision

The implementation for this last part of the system is actually quite simple. The information provided by the Evaluation component basically will serve to calculate the ratio of correctly predicted connections on each machine. This gives us an idea about what were the machines that were more flagged as malicious by the classifier. On the other hand, this ratio will give us the option to define a threshold on the Decider so that we can eliminate certain IP addresses that may not seem dangerous for the classifier. Also, machines with few flows might not be considered here because of the lack of information.

In the end, based on the above data, the system will output the IP addresses that seem like the most likely to be bots or engage in abnormal activities. This component is an example of a practical application of this type of detection system because, in a real world scenario, we want to be able to know how to apply the classification and the algorithms here described. The tool created with this work was made to be extensible. In this particular module, a lot of work can be done, as it is very simple as is. In the last Chapter of this document we will discuss ways on how to improve this, as well as give suggestions to future implementation options.

5

Evaluation

Contents

5.1 Evaluation Methodology

As part of the development of the project, we tested the system in iterations as we implemented the solution. We gradually executed more tests as more features would be created in order to understand if the solution could be viable in the real world and if it could be improved compared to other work already done on the field.

The tests were conducted in a machine with Ubuntu 18.04, 8GB of RAM, and an Intel® Core™ i5-7200 processor. The processing of files was executed in a Virtual Machine with 100GB of RAM because, as we will see in the next section, the data needs too much space to be stored on the original machine. We mainly tested the machine learning algorithms employed but we also tested the capacity of the system to perform as fast as possible and with fewer resources spent. The Evaluator component from the architecture, as seen in Chapter 3, carries out a lot of the calculations for the tests using the methods from the scikit-learn and scikit-plot[1] package. Other tests, such as resources needed and time elapsed, are executed simply with code written in Python and Bash.

The following sections will describe the tests that we thought crucial to the evaluation of each module of the system. In the end, we will discuss the results based on the metrics established, the objectives that we had for the project and the overall performance of the system and how could it be implemented in the real world.

5.2 Data Processing

One of the most important aspects of the framework is the way that we treat the input data of our system. If we think about it, in reality, analysts deal with a tremendous amount of logs that come out of multiple defense mechanisms in a not normalized way. We present here the results of our data processing in terms of speed and memory needed to transform the pcap files into objects that we could manipulate to our needs. Table 5.1 shows for each dataset file used, the time needed to load it and the memory occupied in the system. Notice that the training dataset was split in two. This is because, on our system, we did not have enough memory to store all the structures of the training data. Because of this, we had to split the file in half (approximately) using the editcap tool in order to process all data.

File name	Size	Time elapsed	Memory occupied
training1.pcap	2.6 GB	1h 32m 52s	$\approx$ 67.6 GB
training2.pcap	2.5 GB	56m 48s	$\approx$ 41.1 GB
original_testing.pcap	2.02 GB	1h 16m 48s	$\approx$ 55.1 GB

Table 5.1: Time and memory needed for each dataset

[1] https://github.com/reiinakano/scikit-plot

As we can see from the table, both the time and memory used is simply too high. The time metric is the most important here because, in order to make a viable analysis framework, we should be able to load the data and train it as fast as possible so that any threat could be stopped as soon as the framework finishes the classification.

To improve this results, one could use multi-threading. With multi-threading, we could break the data into multiple files and parse the packet information with multiple processor threads. This way, we could speed up the processing phase. And since the flows don't depend on each other when parsing, we do not need to worry about the order of the threads finishing. Also, another suggestion here is, if it takes too long to process, we could save the flows in multiple smaller files. With this idea, we could use the saved data right away, while the pre-processing module is finishing parsing the connections. In terms of memory consumed, there is not much we could do here since the Scapy package creates a lot of structures to parse the sessions into memory. This occupies a considerable size on memory so, the machines that do this task should have a lot of resources to deal with this problem. In some cases, splitting the dataset files here could be a solution too, but the data have to be processed in a sequential way.

5.3 Classifier Comparison

For Classifier selection, we conducted a couple of experiments to find out the best algorithm that suited the needs of the system, before even trying to make a decision about what machines should be further investigated. To do this, we tested the algorithms of Table 3.8 regarding the metrics already discussed such as accuracy and precision. We first wanted to know, if the features employed were valid and presented good results regarding the multiple botnets contained in the logs. To this end, a couple of botnets were tested separately first. We consider this experiment to be the most relevant and will discuss more around it because it is the one that is closest to what happens in real world setting. In a normal attack, we won't see a dozen of different botnets attacking. It is much more normal to have in a network, a big quantity of normal flows and then, only a couple of malicious flows, one usually from a single botnet.

On a second phase, we tested the whole dataset to see how the system would perform. Also, the majority of threats found on the testing dataset are not present in the training data. This will allow us to know if the system performs well when faced with unknown threats.

Since the classifiers will be compared based on the already discussed metrics, we need to define them here to understand the values that come out of the Evaluator component. First, lets define True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN). In classification, a true positive is when a positive result is correctly predicted. In our case, the classifier says correctly that

a flow is malicious. On the other hand, a true negative is when the classifier correctly predicts that a flow is non-malicious. As for false positives, the result of the classifier is positive when in reality, it should have been predicted as negative. False negatives are the opposite of this.

With that being said, the metrics used on the next sections to compare the algorithms are Accuracy, Precision, Recall and F1-Score as well as a mention of the Confusion Matrix which corresponds to a 2x2 matrix containing the predicted values explained above (TP, TN, FP, FN). Accuracy is essentially the proportion of correctly predicted values in all predicted results. It translates into the following formula:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.1}$$

Precision, Recall and the F1-Score are all defined in Powers work [28]. Precision (or Confidence) is the proportion of positive results that were correctly predicted as true positives.

$$Precision = \frac{TP}{TP + FP} \tag{5.2}$$

Conversely, the Recall metric (or Sensitivity) is the proportion of true positive values that are correctly predicted positive, in our case, the % of real threats that we were able to find.

$$Recall = \frac{TP}{TP + FN} \tag{5.3}$$

Lastly, the F1-Score combines both Precision and Recall to compute a score. It is defined by the following formula:

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{5.4}$$

With all the above-described metrics, we will try and choose the best machine learning algorithms for the system. The focus is on the number of correct predictions of each flow in the dataset (the Accuracy) but we cannot choose the algorithm based solely on a specific value. This is why we took into consideration other metrics. The following sections describe the results of each experiment. In regards to training of the Classifier, we used only three botnets: Neris, Virut and RBot, In addition to the normal connections. All features used in both approaches are the same and can be reviewed in Table 4.5. For the sake of realness, we did not consider the training IRC connections since they overpowered the rest of the flows in the dataset.

5.3.1 First Experiment - Analyzing a botnet at a time

In a first phase, as stated above, we tested the system in a more simple way by filtering the dataset flows to only a specific botnet at a time. More precisely, in the training dataset, we will use normal connections and three different botnets (Neris, Virut and RBot). In the test dataset, we will only leave flows of one specific botnet plus normal connections.

We present here the result for all botnets present on the training dataset as well as tests for two unknown botnets that do not appear in the training phase of the classifier. In terms of flow sizes in this set of tests, we tested the classifiers with exactly 60006 normal connections.

5.3.1.A Analysis of botnets that appear in the training dataset

Table 5.2 shows the results in detecting only the botnet Neris that uses IRC connections to communicate with the Command and Control.

Classifier Algorithm	Accuracy	Precision	Recall	F1-Score	Training	Total
Linear Discriminant Analysis	93%	90%	82%	86%	0.26s	0.43s
Linear SVC	70%	45%	94%	61%	31.24s	31.48s
Naive Bayes	40%	27%	82%	40%	0.09s	0.28s
Gaussian Naive Bayes	33%	24%	82%	37%	0.11s	0.31s
Logistic Regression	91%	83%	82%	82%	3.64s	3.87s
Neural Networks	75%	43%	11%	17%	8.43s	8.77s
Decision Tree	71%	72%	26%	13%	0.71s	0.90s
Random Forests	74%	33%	0.06%	0.1%	1.24s	1.48s

Table 5.2: Results for the different algorithms in detecting the Neris botnet

In Table 5.3 we have the results for another botnet that can be found in the training dataset, Virut. Finally, Table 5.4 displays the results for the RBot botnet.

Classifier Algorithm	Accuracy	Precision	Recall	F1-Score	Training	Total
Linear Discriminant Analysis	92%	94%	84%	88%	0.2s	0.47s
Linear SVC	90%	88%	82%	85%	31.15s	31.42s
Naive Bayes	49%	41%	91%	56%	0.09s	0.3s
Gaussian Naive Bayes	44%	38%	92%	53%	0.1s	0.34s
Logistic Regression	91%	90%	86%	88%	3.55s	3.81s
Neural Networks	92%	86%	91%	88%	4.03s	4.41s
Decision Tree	94%	87%	96%	91%	0.71s	0.93s
Random Forests	97%	93%	99%	96%	1.3s	1.54s

Table 5.3: Results for the different algorithms in detecting the Virut botnet

As we can deduce from the tables, the classifiers performed quite well regarding the chosen features, especially in the case of Virut. Both Accuracy and Recall are excellent in all botnets tested here, meaning that the system detects with effectiveness malicious behaviors. In the case of Neris, the best classifiers

Classifier Algorithm	Accuracy	Precision	Recall	F1-Score	Training	Total
Linear Discriminant Analysis	97%	0.3%	3%	0.5%	0.16s	1.1s
Linear SVC	83%	0.09%	6%	0.2%	22.2s	22.54s
Naive Bayes	26%	0.4%	98%	0.7%	0.09s	0.83s
Gaussian Naive Bayes	17%	0.3%	98%	0.6%	0.10s	0.9s
Logistic Regression	95%	0.2%	3.6%	0.4%	2.9s	3.6s
Neural Networks	97%	6.7%	87%	12%	5.2s	6.2s
Decision Tree	92%	3.4%	93%	6.6%	0.4s	1.2s
Random Forests	96%	6%	92%	11%	0.73s	1.5s

Table 5.4: Results for the different algorithms in detecting the RBot botnet

were the Linear Discriminant Analysis and the Logistic Regression, both surpassing an Accuracy of 90%. In the case of Virut, almost all classifiers performed well with the exception of the algorithms based on Naive Bayes. The great results from Virut might have to do with the quantity of the connections found in the testing dataset, producing great values in regards to the metrics. Finally, the last tested known botnet was RBot. The values of Table 5.4 are interesting but logical. Both the accuracy and recall are great in some cases like Random Forests and Decision Trees but the precision is awful. This is because, the number of connections found in the testing dataset for RBot is really small in comparison to the negative samples, even though almost all malicious flows were correctly detected as shown by the Recall values. The following figures will allow us to see this detail more clearly as it shows the Confusion matrices for these results. We will only show here the matrix relative to the best classifier of each algorithm for the sake of simplicity.

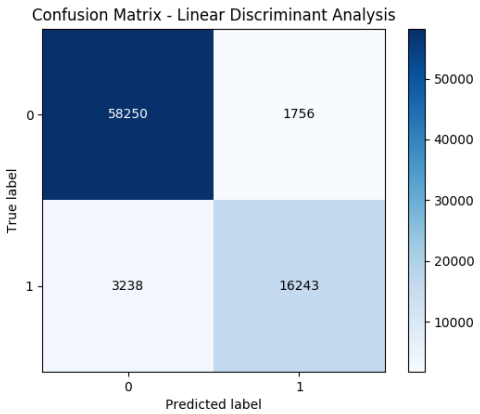


Figure 5.5: Confusion Matrix of Neris

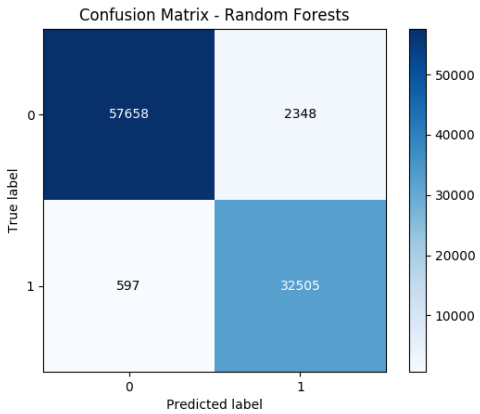


Figure 5.6: Confusion Matrix of Virut

As we can see from Figures 5.5-5.7, The classifiers all detect with success almost all negative (or normal connections) that can be found in the top left corner of the matrix. On the bottom right corner however, we can see the True Positives value. The other two squares correspond, respectively, to the False Positives (top right corner) and False Negatives (bottom left corner). Because the testing samples

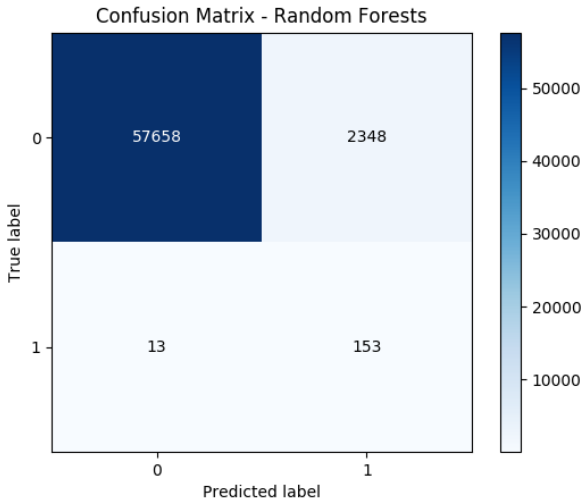


Figure 5.7: Confusion Matrix of RBot

from the RBot botnet were so few, it seemed like the classifier was performing badly. But, as we can see from the False Negatives, the total of malicious connections to classify were 166 (13 + 153). Confusion matrices are good because they summarize well the values of TP, TN, FP, and FN on a simple graphic.

5.3.1.B Analysis of botnets that do not appear in the training dataset

Regarding unknown threats, we tested two botnets: TBot and Menti. We chose TBot because it had a lower ratio of connections in relation to normal flows by appearing in only 485 connections. The Menti botnet had 4539 flows in the test. Table 5.8 and 5.9 show the results of the classifier in detecting these two threats.

Classifier Algorithm	Accuracy	Precision	Recall	F1-Score	Training	Total
Linear Discriminant Analysis	97%	10%	44%	17%	0.21s	0.41s
Linear SVC	83%	0.9%	19%	1.8%	31.12s	31.31s
Naive Bayes	27%	0.2%	16%	0.4%	0.09s	0.25s
Gaussian Naive Bayes	17%	0.5%	49%	0.9%	0.11s	0.27s
Logistic Regression	94%	7%	50%	12%	3.76s	3.94s
Neural Networks	92%	0.7%	6%	1.2%	12.06s	12.40s
Decision Tree	92%	3%	28%	5%	0.72s	0.87s
Random Forests	96%	3.4%	16%	5.6%	1.21s	1.40s

Table 5.8: Results for the different algorithms in detecting the TBot botnet

The results above are slightly different from the three two tables. From the TBot connections, we can see a significant drop in the Precision metric. This is because of the few connections to classify. For example, the best classifier (LDA) only detected correctly 212 out of the 495 connections, but the problem is the False Positives. In total, the classifier reported 1783 False Positives corresponding to

Classifier Algorithm	Accuracy	Precision	Recall	F1-Score	Training	Total
Linear Discriminant Analysis	97%	71%	98%	82%	0.21s	0.41s
Linear SVC	84%	30%	94%	45%	31.7s	31.9s
Naive Bayes	31%	9%	95%	16%	0.09s	0.26s
Gaussian Naive Bayes	23%	8%	99%	15%	0.11s	0.29s
Logistic Regression	95%	58%	98%	73%	3.72s	3.92s
Neural Networks	94%	55%	97%	70%	5.74s	6.02s
Decision Tree	92%	47%	98%	63%	0.73s	0.89s
Random Forests	97%	68%	98%	80%	1.21s	1.41s

Table 5.9: Results for the different algorithms in detecting the Menti botnet

3% of the flows in classification. The Accuracy remains high since almost all normal connections were correctly predicted. The situation of the Menti botnet is similar but with slightly better Precision and Recall.

Overall, the classification system did a good job of detecting a single botnet at a time. As expected, the detector performed better when the experiment involved a botnet present in the training dataset although the results were not bad in the second experiment, reaching high values in the Accuracy and Recall metric.

5.3.1.C Additional Metrics

We show here two more statistical graphics associated with the classification that allow us to analyze the results even further. One is the ROC curve that plots the True Positive rate in the Y axis and the False Positive Rate on the X axis. The ideal in this case is to maximize the True Positive rate while minimizing the False Positive rate.

In both Figures 5.11 and 5.10 we can see examples of optimal ROC curves, where the area below the lines are almost in the maximum value. These are very good results for these two botnets, meaning that we have a very low False Positive Rate for both of them.

The other metric we present here is the Precision-Recall curve. It plots in the Y axis the Precision and in the X axis the Recall. This is a good graphic to see the trade off between the two metrics discussed. High scores for both show that the classifier is returning accurate results. This happens when we observe Figure 5.13.

In Figure 5.12 the results are still good even though they are not optimal. We can see that the area below the precision-recall curve for the positive class is lower than in Figure 5.13, reflecting the lower Precision value found in Table 5.9.

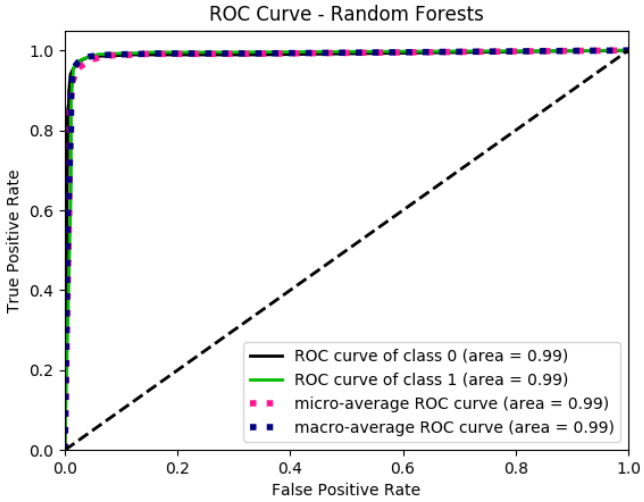


Figure 5.10: ROC curve of Virut with RF

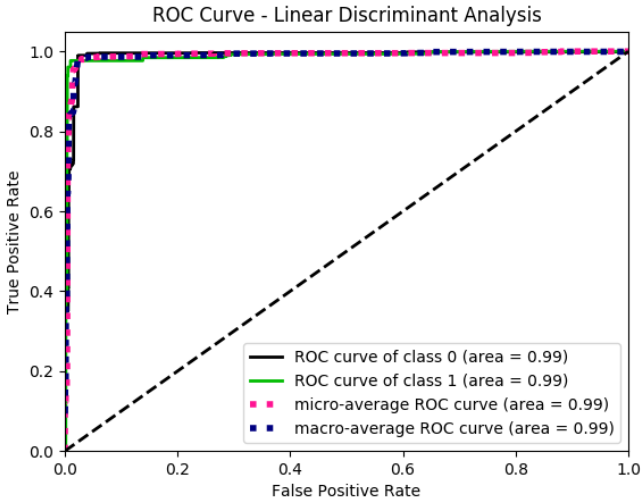


Figure 5.11: ROC curve of Menti with LDA

5.3.2 Second Experiment - Analyzing the entire dataset

On the second experiment, we tested the whole dataset to see how the system would behave when facing multiple threats at the same time. While this approach might be less real and practical in the real world, we thought it was worthwhile to present the results nonetheless. Table 5.14 shows the results in testing the whole dataset.

From the table, we can draw some conclusions. The first one is that the system does not perform very well when encountered with multiple botnets that operate very differently. For example, some botnets communicate via IRC and other through HTTP or UDP. It is extremely difficult to make a universal set of features that captures all of the multiple behaviors of the botnets present in the testing dataset. The

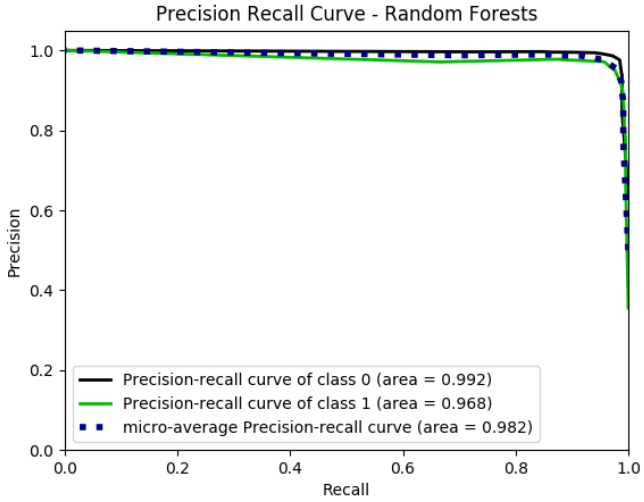


Figure 5.12: Precision-Recall curve of Virut with RF

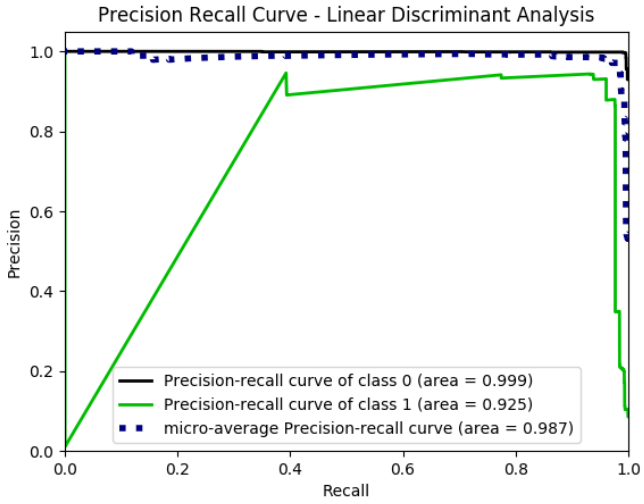


Figure 5.13: Precision-Recall curve of Menti with LDA

Classifier Algorithm	Accuracy	Precision	Recall	F1-Score	Training	Total
Linear Discriminant Analysis	51%	62%	68%	65%	0.3s	0.75s
Linear SVC	65%	67%	97%	79%	58.8s	59.3s
Naive Bayes	71%	72%	93%	81%	0.16s	0.66s
Gaussian Naive Bayes	68%	68%	99%	80%	0.18s	0.71s
Logistic Regression	51%	62%	68%	65%	3.2s	3.72s
Neural Networks	60%	84%	51%	63%	4.9s	5.6s
Decision Tree	56%	79%	47%	59%	1.1s	1.61s
Random Forests	57%	82%	46%	58%	2.5s	3.2s

Table 5.14: Results for the different algorithms when testing all dataset

second conclusion is that the large number of different patterns and communications in relation to the selected features, cause the algorithms to overfit. This is evident in cases where the algorithms tend to classify one label and not the other. For example, the Linear SVC detected almost all botnet connections. The problem is that it did not detect normal connections at all, detecting only 114 out of 60006 flows.

5.3.3 Cross Validation

In order to choose the best classifier algorithm, we employed Cross Validation on the dataset. Arlot *et al.* survey [29] describes this technique. Cross validation is a statistical technique usually used in machine learning to evaluate how good an algorithm will perform and thus, allowing us to choose the best classifier. The idea of cross validation is to split the dataset several times in a training sample used for training and in a validation sample used for estimating the risk of each algorithm. This strategy is good because, in theory, it avoids overfitting.

On this subsection, we will test the same classifiers with the same features and using the two already employed approaches: singular botnet at a time, and whole dataset but using the cross validation strategy. We will show the scores of each classifier in regards to the accuracy of the algorithms. We will do this by comparing the learning curves between each classifier. The learning curves have two purposes: to run the cross validation test on each algorithm and to show, by incrementing training samples, how the algorithm behaves with different quantities of information.

The learning curve shows us a green line corresponding to the mean accuracy score of the test samples while the red line corresponds to the mean accuracy score of the training samples. The green area and red area correspond to, respectively, the standard deviation of the testing and training scores computed in the various iterations of the cross validation.

With the exception of Linear SVC, the cross validation strategy was made with 10 splits, meaning that, the training dataset is split into 10 different training samples and 10 different testing samples. For each iteration, the classifier is trained with one of the training samples and tested with one of the testing samples. In the end, the accuracy score of the classification is averaged and this value is plotted on the figure. The cross validation is then repeated 5 times, with more training examples as seen in the X axis of the graphics.

In terms of the method to split the dataset, we used the strategy of time splits using the TimeSeriesSplit that scikit-learn provides. This strategy allows us to split the dataset into time intervals without shuffling the array of features. We want to do this because the training should not be done with future data. That is, the flows and the connections that appear on the network traces are not independent and using the flows that appear in the future might influence the results and create bias in the detector.

The learning curves are good tools in assessing if the testing scores of the algorithm are stabilized or not. They allow us to see if the number of samples in the dataset bring enough relevant information

to achieve optimal accuracy scores for that classifier.

5.3.3.A Analyzing a botnet at a time

From the experiment of analyzing a single botnet at a time, we will choose here only one botnet to show the results of the cross validation test since for each botnet, we have to show the graphics for all 8 classifiers. We decided to showcase the Virut botnet, because it had great results before with so little flows trained.

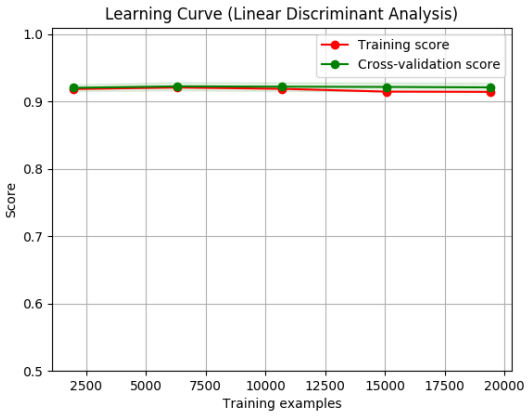


Figure 5.15: Learning curve for LDA - Virut

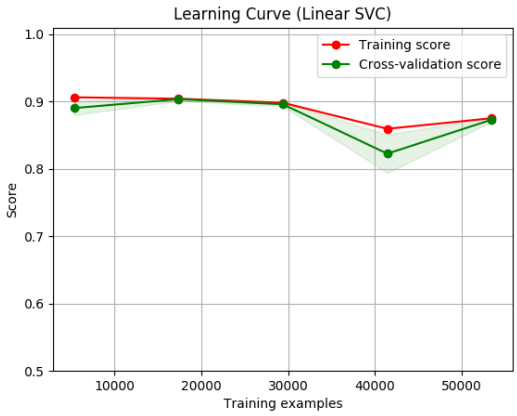


Figure 5.16: Learning curve for Linear SVC - Virut

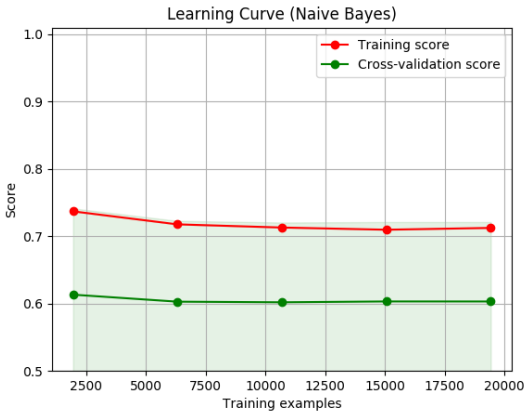


Figure 5.17: Learning curve for Naive Bayes - Virut

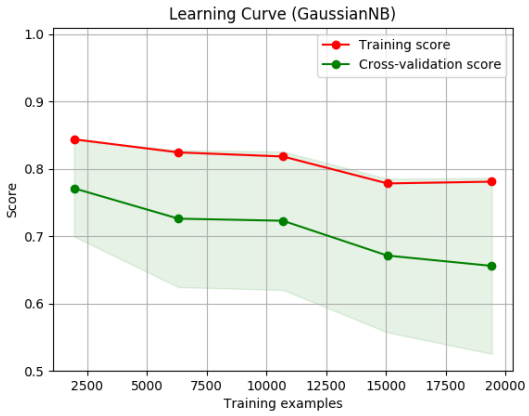


Figure 5.18: Learning curve for Gaussian NB - Virut

From the figures generated by the cross validation experiment, we can conclude that the results are very good and encouraging. Discounting Naive Bayes and the Gaussian Naive Bayes classifier, all algorithms performed very well and we can see that by the closeness of both lines in the figures. When both the test score and training score of each iteration of the cross validation (each point in the graph) are almost overlapping, we could say that we reached the optimal result for the classifier. For example,

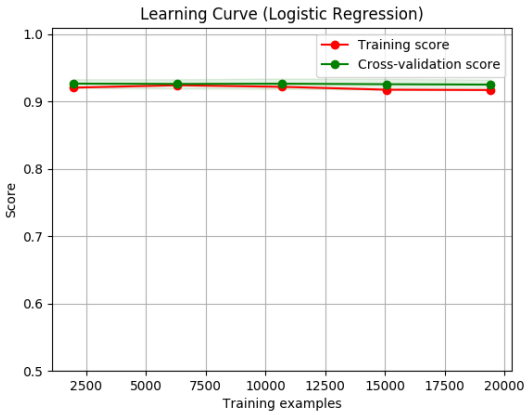


Figure 5.19: Learning curve for LR - Virut

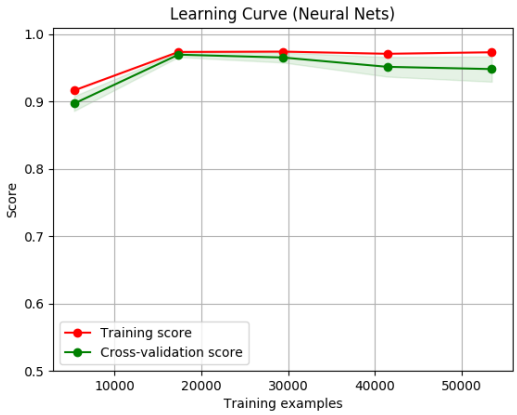


Figure 5.20: Learning curve for Neural Networks - Virut

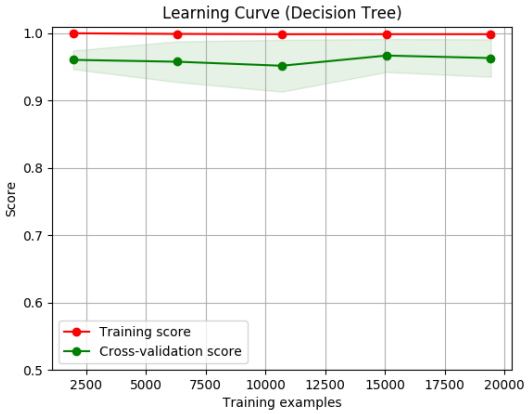


Figure 5.21: Learning curve for the Decision Tree - Virut

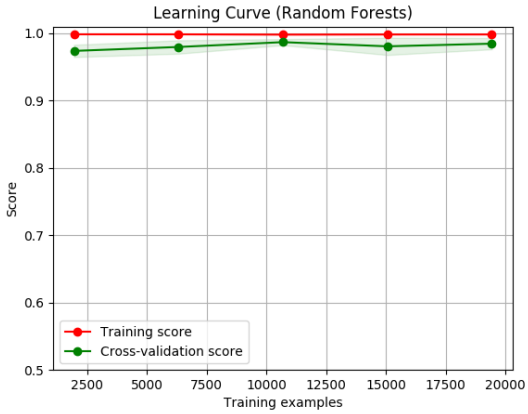


Figure 5.22: Learning curve for Random Forests - Virut

in the case of the Random Forests, the two lines are very close, where the Accuracy scores higher than 95%. These results only reinforce our conclusion that the system is very good in more practical scenarios, since it displays outstanding results in the detection of singular botnets. We can conclude that both Naive Bayes and Gaussian Naive Bayes are probably not suited for this type of classification, given the least desirable results shown in the respective figures.

5.3.3.B Analyzing the entire dataset

The following graphics show the learning curves for each algorithm where the dataset is composed of all testing botnets found in the dataset, mixed in with normal activity connections. This experiment is similar to what can be found in the work of the Belgi *et al.* [11] who created the dataset used in our system.

From the figures generated by the cross validation experiment, we can conclude that the results are more or less in line with the classification of the whole dataset as shown in the previous subsection.

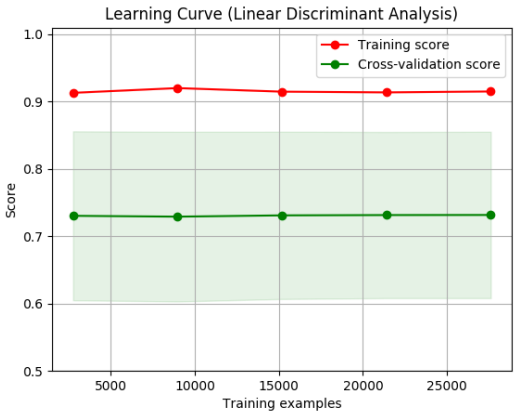


Figure 5.23: Learning curve for LDA

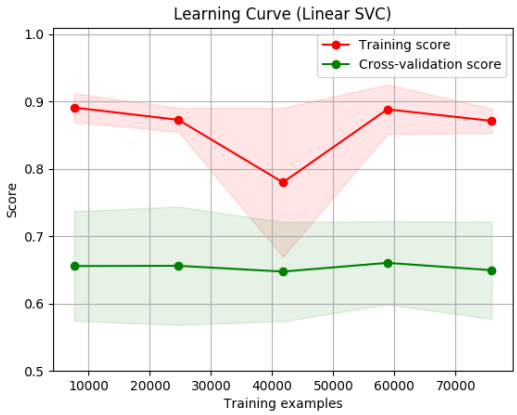


Figure 5.24: Learning curve for Linear SVC

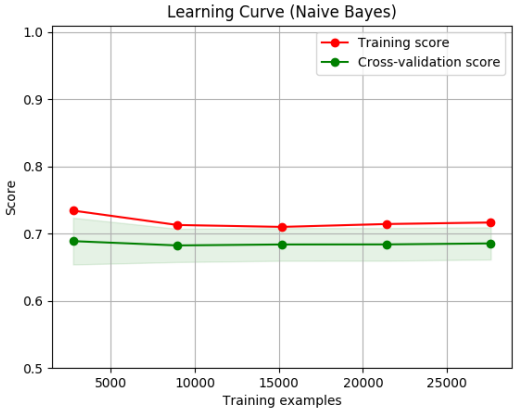


Figure 5.25: Learning curve for Naive Bayes

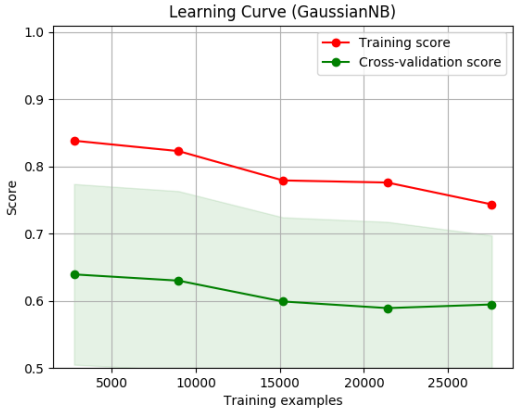


Figure 5.26: Learning curve for Gaussian NB

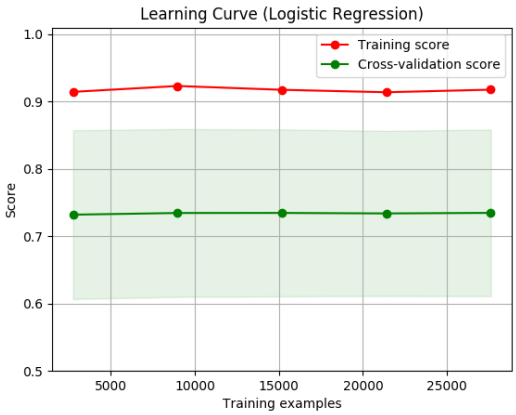


Figure 5.27: Learning curve for Logistic Regression

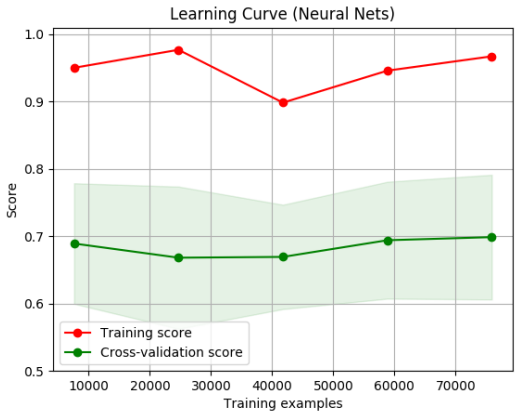


Figure 5.28: Learning curve for Neural Networks

And, unlike the previous experiment of a single botnet, the results are worse. However they seem to have improved slightly from the values seen on Table 5.14. In some cases, we can see that it could

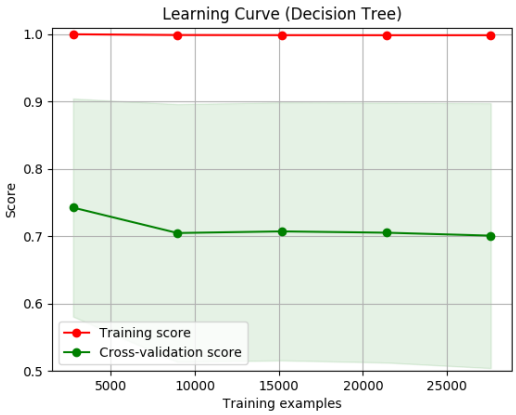


Figure 5.29: Learning curve for the Decision Tree

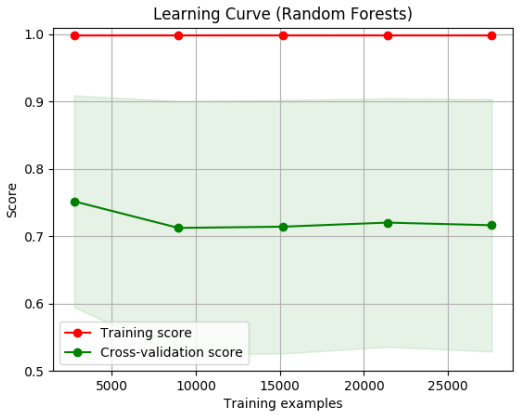


Figure 5.30: Learning curve for Random Forests

be possible to improve the results of the classification when adding more training examples. This is the case for all algorithms except Gaussian Naive Bayes and Naive Bayes, like we have seen on the previous cross validation experiment. The figures reflect that possibility when the training score is much higher than the testing score and it is not declining, meaning that, if we could execute more iterations of the cross validation with more training connections, the results would stabilize and improve a bit. However, the results could improve or decline, and this is deduced by the green area present on the figures. As stated before, the green area represents the range of values for the scores of the cross validation. So, in theory, the optimal scores could be in that area. To be completely sure and make conclusions, we needed to have more data to compute more iterations of the cross validation test.

Nevertheless, the results were still not very good with the best classifiers being LDA, Neural Nets, Logistic Regression, Decision Trees and Random Forests. All these classifiers scored above the 70% mark beating the results from the experiment made in Section 5.3.2. The detector will still have a lot of False Positives and False Negatives when faced with the detection of multiple and different botnets.

5.3.4 Final Remarks

The objective of this comparison was to choose the best classifier to for the system and, at the same time, check what initial results could be obtained when detecting single botnets or multiple botnets at the same time. In the case of a single botnet, we saw that detecting already known threats such as the Neris and Virut botnet got overall good results above the 91% accuracy for some classifiers. When classifying unknown botnets, the system had some trouble in distinguishing the connections, causing the algorithms to overfit if the data is almost non-existing, that is, if the number of flows seen in the test was very low in comparison with the normal activity.

From this experiment, it is possible to see that different algorithms have different results for different

botnets when trained with the exact same features. This could mean that we could try and tailor each classification to suit different patterns of botnets even if they are not the same.

In the end, we had to choose the best classifiers. If we are going to choose to detect all kinds of threats, the Naive Bayes might seem a winner but from the cross validation experiment, the score was decent but it showed no sign to improve with the current conditions. From a singular detection point of view, it is clear that an approach with the Naive Bayes family of algorithms does not work very well as the scoring is always lower than the other classifiers, although it did well in the final test containing the whole dataset. Some classifiers are slower than others too, which puts Linear SVC's in a worst position in regards to the others. The ideal is to use a hybrid combination of the algorithms that yielded the best results such as LDA, Neural Networks, Random Forests, and Decision Trees.

5.4 Decider Results

The last component of the system, the Decider, aims at making a sense of the results from the evaluation of the classifiers in order to pick the most suited candidates of being infected and executing malicious activities such as data exfiltration. Here we present the results for both scenarios, as the rest of the evaluation process.

In the case of a single botnet scenario, we showcase Menti here, a botnet that does not appear in the training dataset. For this experiment, we considered a value of 0.8 for the $threshold$ and 300 for the $flowThreshold$. Table 5.31 contains the results of running the Decider on a couple of chosen classifiers such as Neural Networks, Decision Tree, Random Forests and LDA.

Classifier	IP address	Correct Predictions	Wrong Predictions	True Label
LDA	147.32.84.150	4434	105	Botnet
	66.161.11.90	6	671	Not Botnet
Neural Netwoks	203.82.202.164	39	734	Not Botnet
Decision Tree	147.32.84.150	4335	204	Botnet
	195.228.245.1	49	596	Not Botnet
Random Forests	147.32.84.150	4539	180	Botnet

Table 5.31: IP addresses flagged by the Detector in the first scenario

As we can see from the 4 iterations (4 classifiers) of the decider, we notice that the Menti botnet is always correctly flagged given the thresholds we defined except in the Neural Networks case. From this result, we have got a running counter in an auxiliary dictionary that counts the times an IP was flagged during the algorithm. If we count the number of times the same machine appears, the conclusion is that the IP 147.32.84.150 has the strongest probability of being infected since it appears three times in the above table. This result is correct because that IP address belongs to the Menti botnet.

The second scenario is much more complex and hard to analyze. Since the datasets have more

than 500 different IP addresses, and given the results of the second scenario in other tests, it is clear that we are going to wrongly flag a lot of machines. With the same parameters as above, there are only 4 different malicious machines that are flagged while only 2 have a high running counter. But so does a lot other non malicious machines. If we lower the $threshold$ to 0.5, we flag a lot more bots but, at the same time, we catch more benign machines.

What we can conclude from this is that, the Decider component performs best in the scenario of a singular botnet in the traffic because the results of the classifier were very good there too. When the whole dataset is considered, we flag too many IP addresses to make this application viable.

5.5 Discussion

In regards to our evaluation process, we first evaluated the time and memory needed to transform the network log files into flows that we could manipulate and extract features from. The results did not seem encouraging but we have to remember that once the system is initially trained with some data, it can begin to classify connections right away. We discussed some methods as to how we could improve the time elapsed by employing multi-threading and even, delegating the broken down tasks to multiple machines.

In an attempt to choose the best classifier for the detection task, we made two initial experiments with the dataset. The first experiment came up with very good results in detecting botnets that are already known by the classifier mainly, Neris and Virut. On the other hand, detecting unknown botnets is harder and we saw that the accuracy of these cases depended on the features chosen, the number of samples in the testing set, and the similarities of patterns of previously seen botnets. Nonetheless, for the Menti botnet, the results were good when detected by the LDA or the Random Forests classifiers. Finally, we tried to classify the whole dataset. As expected, the algorithm has a lot of trouble when detecting all different connections going on. Given the complexity and great quantity of very diversified flows, the best classifier, Naive Bayes, only reached an accuracy of 71% which is a decent score.

To prevent overfitting of the classifiers and to test the system even further, the strategy of cross validation was tested by splitting the training samples respecting the timestamps on the connections. Although our method was a little different, we compare here our results regarding the work by Beigi *et al* [11]. With cross validation respecting the temporal order of the packets, our best algorithms (LDA or Logistic Regression) reached a score of $\approx$ 73%, a very similar score to their work. We believe that with more training samples throughout time, the classifiers could score a little higher because of the learning curves presented. On our work though, we tested a lot more classifiers while Beigi *et al*, only show results for Decision Trees. Another thing to point out here is the experiments made on both sides. We consider the approach in detecting the whole dataset weaker than to analyze various types of botnets

once at a time and try to mold the system for each different threat because it is more real in practice. Finally, we suggest an application of the system to use as a tool to detect other types of activity engaged by botnets.

In the end, we verified our Decider results. By combining the results from the best classifiers within the comparison, we applied a threshold of 0.8 in order to choose the most likely infected machines in both scenarios.The Decider presents very good promise when there is only one threat active in the network, which the scenario more true to the real world that we can model.

6

Conclusion

Contents

6.1 Conclusions

In an interconnected world like today, information is one of the most valued assets of any company. From all the attacks that exist and that threaten companies every day, the exfiltration of data is one of the most dangerous and damaging since aims at stealing sensitive information from the user's computers. One of the ways this can be achieved is by compromising vulnerable machines with malware and create a botnet that an attacker can control. Then, by simply issuing commands to these bots, the data can be exfiltrated in a stealthy manner.

In this work, we defined the botnet problem and discussed techniques that allow attackers to steal data as well as defense mechanisms that exist today. We discussed why some of the current defense mechanisms are not enough because there are multiple ways to execute botnet attacks and exfiltrate data. Also, organizations sometimes overlook the covert aspects of data exfiltration and use tools that can be outdated, not dealing directly with this type of problem.

We described a simple taxonomy for data exfiltration techniques found in [1] and created a survey. The survey was divided into four subsections: Exfiltration Techniques, Exfiltration Detection Mechanisms, Botnet Data Exfiltration, and Machine Learning. In regards to the exfiltration and based on the taxonomy, we explored four main categories: Network, Physical, Covert Channels, and Steganography. This division is quite flexible because some previous work mix network techniques with covert channels but it gives us a good starting point on the topic.

Then, we proposed a solution to the problem of detecting botnet activity using a machine learning approach. The goal is to observe what are the characteristics and behaviors of botnets and extract features allowing us to train a classifier. We showed that our system was composed of five main modules: Data Collector, Feature Extractor, Classifier, Evaluator and the Decider. The first two components allowed us to conduct the pre-processing phase on the pcap log files, transforming them into arrays that we could manipulate and parse features to use on the classification. In the next step, we used multiple classifiers and evaluated them in order to get the best results. The Evaluator module produced the report from the output of the classifiers that we discussed in Chapter 5 and gave us a good picture of the viability of the system in detecting single and multiple families of botnets. Finally, we saw that the Decider component analyzed the results from the Evaluator and gathered information from the Feature Extractor to choose the most likely infected machines.

In the end, we executed some experiments and discussed scenarios that the system could be faced with. In regards to the data processing, we discuss the long time needed to process the log files as well as the space occupied by the structures created by Scapy that could make the system less practical. On the other hand, the results from the classifier were encouraging when the system was faced with singular botnets (accuracy around 97%), specifically, those that we already had knowledge off. When we tested all the botnets found in the dataset, the detection rate was lower when using the cross validation strategy,

reaching 73% accuracy. Finally, we checked the results on the Decider component. The results were best when we executed the algorithm with a single botnet by marking the running botnet correctly, In terms of the whole dataset, the Decider had troubles in flagging correctly the machines, although that task is harder and and arguably unrealistic.

6.2 Future Work

The work on detecting data exfiltration and botnet activity, in general, is far from over, and there is still a lot to be researched and experimented in order to make these kinds of machine learning systems more efficient and effective in preventing attacks as fast as possible. Overall the results of our project were very good in detecting instances of singular botnets at a time. In the case of known botnets, the cross validation experiment showed us the potential for this system where we saw that the classifiers reached optimal Accuracy and Recall scores in some cases, all above 90%.

When testing with singular unknown botnets, we reached good accuracy results but when faced with very few malicious connections to classify, there is still a lot of false positives to go through and analyze. The classifier module of the system could be more fine tuned by testing different-features for example or, by employing a more sophisticated algorithm to make classifications, like a consensus style procedure that selects the best algorithm for each specific family of botnet.

Our system is slow on the data processing side although it could perform a lot better with the methods we discussed. In the future, the time spent in this task should be lowered to speed up large processing of network traces.

The problem of detecting data exfiltration by botnets was not directly solved by our solution in particular and it could be addressed in the future. The fact that a dataset meeting the specific criteria that we wanted did not exist, made the practical application harder to execute. In further research, it would be beneficial for the area to develop logs and data to create a dataset in data exfiltration or, to use already existing file access logs and feed them into our system as an extension to our tool.

Finally, the Decider module is a good bridge between the classifier results and a real world application. This component could be further improved by creating a better score to detect malicious machines in a network. Suggestions here could range from gathering information from multiple data sources to compute a score or to use some kind of heuristics on the features explored.

Bibliography

[1] A. Giani, V. H. Berk, and G. V. Cybenko, "Data exfiltration and covert channels," in *Defense and Security Symposium*. International Society for Optics and Photonics, 2006, pp. 620 103–620 103.

[2] Y. Liu, C. Corbett, K. Chiang, R. Archibald, B. Mukherjee, and D. Ghosal, "Sidd: A framework for detecting sensitive data exfiltration by an insider attack," in *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*. IEEE, 2009, pp. 1–10.

[3] G. He, T. Zhang, Y. Ma, and B. Xu, "A novel method to detect encrypted data exfiltration," in *Advanced Cloud and Big Data (CBD), 2014 Second International Conference on*. IEEE, 2014, pp. 240–246.

[4] A. Nadler, A. Aminov, and A. Shabtai, "Detection of malicious and low throughput data exfiltration over the DNS protocol," *CoRR*, vol. abs/1709.08395, 2017. [Online]. Available: http://arxiv.org/abs/1709.08395

[5] J. Grier, "Detecting data theft using stochastic forensics," *Digital investigation*, vol. 8, pp. S71–S77, 2011.

[6] P. C. Patel and U. Singh, "Detection of data theft using fuzzy inference system," in *Advance Computing Conference (IACC), 2013 IEEE 3rd International*. IEEE, 2013, pp. 702–707.

[7] A. Al-Bataineh and G. White, "Analysis and detection of malicious data exfiltration in web traffic," in *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*. IEEE, 2012, pp. 26–31.

[8] S. Venkatesan, M. Albanese, G. Cybenko, and S. Jajodia, "A moving target defense approach to disrupting stealthy botnets," in *Proceedings of the 2016 ACM Workshop on Moving Target Defense*. ACM, 2016, pp. 37–46.

[9] S. Venkatesan, M. Albanese, A. Shah, R. Ganesan, and S. Jajodia, "Detecting stealthy botnets in a resource-constrained environment using reinforcement learning," in *Proceedings of the 4th ACM Workshop on Moving Target Defense*, 2017, pp. 75–85.

[10] D. Zhao, I. Traore, B. Sayed, W. Lu, S. Saad, A. Ghorbani, and D. Garant, "Botnet detection based on traffic behavior analysis and flow intervals," *Computers & Security*, vol. 39, pp. 2–16, 2013.

[11] E. B. Beigi, H. H. Jazi, N. Stakhanova, and A. A. Ghorbani, "Towards effective feature selection in machine learning-based botnet detection approaches," in *Communications and Network Security (CNS), 2014 IEEE Conference on*. IEEE, 2014, pp. 247–255.

[12] B. W. Lampson, "A note on the confinement problem," *Communications of the ACM*, vol. 16, no. 10, pp. 613–615, 1973.

[13] S. Bromberger, "Dns as a covert channel within protected networks," *National Electronic Sector Cyber Security Organization (NESCO)(Jan., 2011)*, 2011.

[14] K. Born, "Browser-based covert data exfiltration," *CoRR*, vol. abs/1004.4357, 2010. [Online]. Available: http://arxiv.org/abs/1004.4357

[15] J. C. Wray, "An analysis of covert timing channels," *Journal of Computer Security*, vol. 1, no. 3-4, pp. 219–232, 1992.

[16] C.-R. Tsai, V. D. Gligor, and C. S. Chandersekaran, "On the identification of covert storage channels in secure systems," *IEEE Transactions on Software Engineering*, vol. 16, no. 6, pp. 569–580, 1990.

[17] P. Gordon, "Data leakage-threats and mitigation," *InfoSec Reading Room*, 2007.

[18] S. O'Malley and K. R. Choo, "Bridging the air gap: Inaudible data exfiltration by insiders," in *20th Americas Conference on Information Systems, AMCIS 2014, Savannah, Georgia, USA, August 7-9, 2014*, 2014.

[19] M. Guri, Y. A. Solewicz, A. Daidakulov, and Y. Elovici, "Acoustic data exfiltration from speakerless air-gapped computers via covert hard-drive noise ('diskfiltration')," in *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II*, 2017, pp. 98–115.

[20] M. Guri, B. Zadov, A. Daidakulov, and Y. Elovici, "xled: Covert data exfiltration from air-gapped networks via router leds," *CoRR*, vol. abs/1706.01140, 2017. [Online]. Available: http://arxiv.org/abs/1706.01140

[21] N. F. Johnson and S. Jajodia, "Exploring steganography: Seeing the unseen," *Computer*, vol. 31, no. 2, 1998.

[22] E. Zielińska, W. Mazurczyk, and K. Szczypiorski, "Trends in steganography," *Communications of the ACM*, vol. 57, no. 3, pp. 86–95, 2014.

[23] Y. Hu, C. Frank, J. Walden, E. Crawford, and D. Kasturiratna, “Mining file repository accesses for detecting data exfiltration activities,” *Journal of Artificial Intelligence and Soft Computing Research*, vol. 2, 2012.

[24] V. Berk, A. Giani, G. Cybenko, and N. Hanover, “Detection of covert channel encoding in network packet delays,” *Technical Report TR536 Darthmouth College*, 2005.

[25] C. Francis-Christie and D. Lo, “A combination of active and passive video steganalysis to fight sensitive data exfiltration through online video,” in *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, vol. 2. IEEE, 2016, pp. 371–376.

[26] W. T. Strayer, D. Lapsely, R. Walsh, and C. Livadas, “Botnet detection based on network behavior,” in *Botnet detection*. Springer, 2008, pp. 1–24.

[27] C. E. Shannon, “A mathematical theory of communication,” *ACM SIGMOBILE mobile computing and communications review*, vol. 5, no. 1, pp. 3–55, 2001.

[28] D. M. Powers, “Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation,” *Journal of Machine Learning Technologies*, pp. 37–63, 2011.

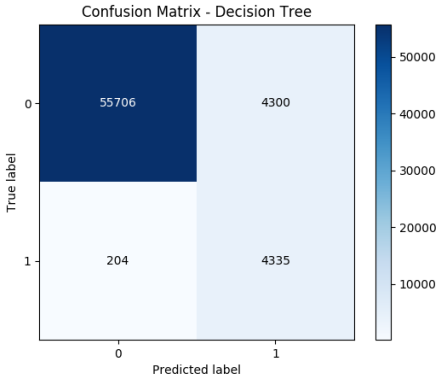
[29] S. Arlot, A. Celisse *et al.*, “A survey of cross-validation procedures for model selection,” *Statistics surveys*, vol. 4, pp. 40–79, 2010.

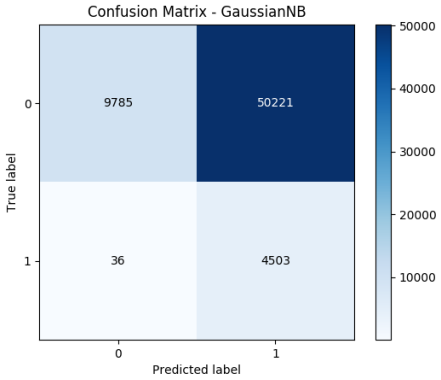
7

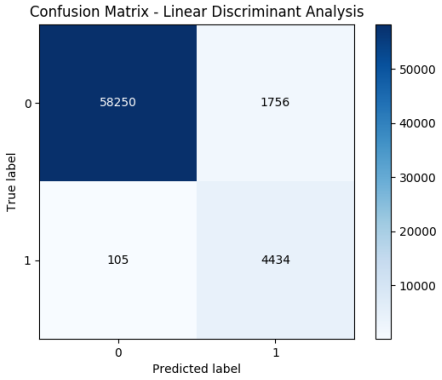
Additional Figures and Tables

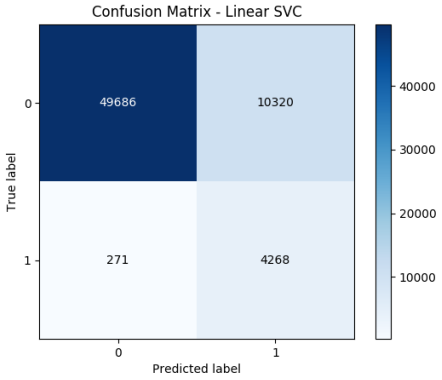
7.1 Classifier Comparison Figures

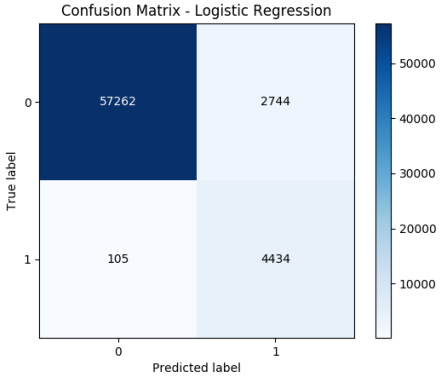
7.1.1 Menti

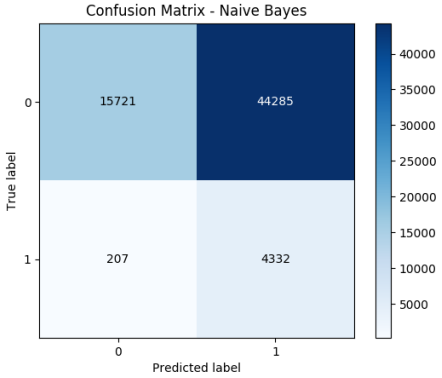


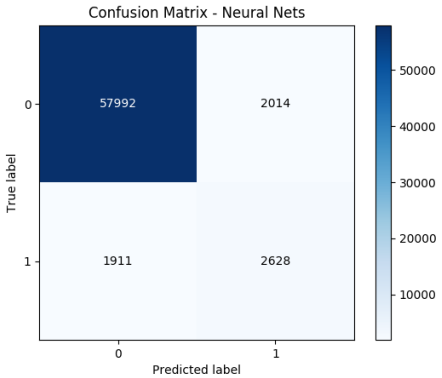


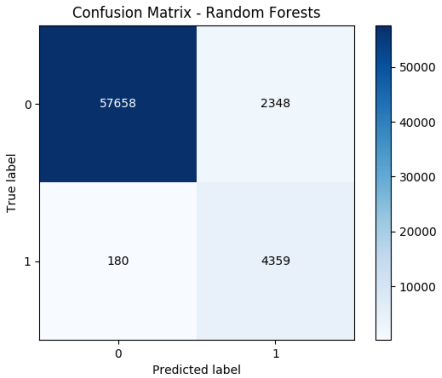

Confusion Matrix - Linear Discriminant Analysis
58250
1756
105
4434
True label
Predicted label
0
1
50000
40000
30000
20000
10000

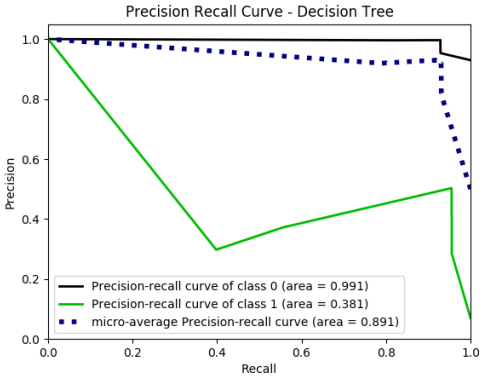

Confusion Matrix - Linear SVC
49686
10320
271
4268
True label
Predicted label
0
1
40000
30000
20000
10000

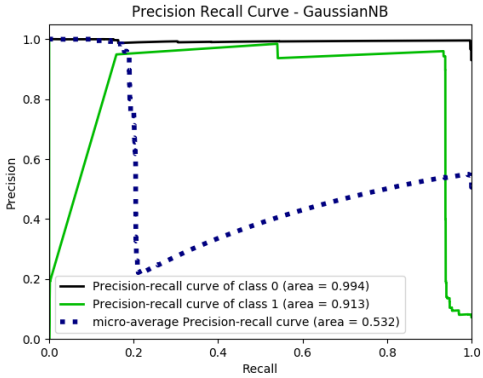

Confusion Matrix - Logistic Regression
57262
2744
105
4434
True label
Predicted label
0
1
50000
40000
30000
20000
10000

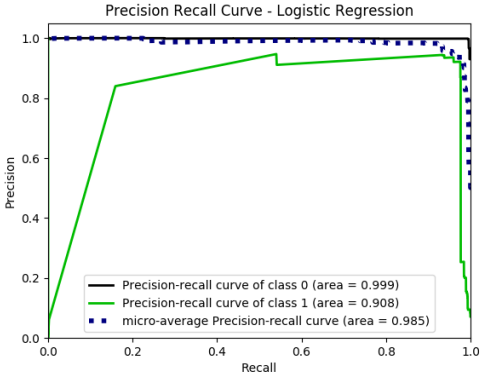

Confusion Matrix - Naive Bayes
15721
44285
207
4332
True label
Predicted label
0
1
40000
35000
30000
25000
20000
15000
10000
5000

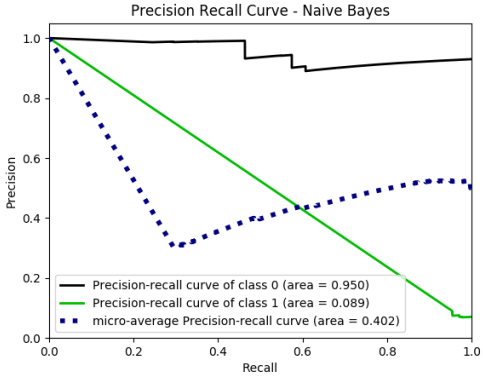

Confusion Matrix - Neural Nets
57992
2014
1911
2628
True label
Predicted label
0
1
50000
40000
30000
20000
10000

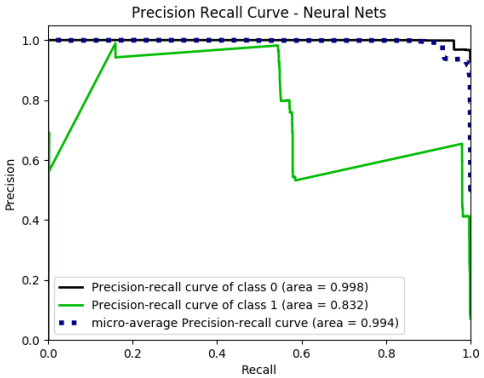

Confusion Matrix - Random Forests
57658
2348
180
4359
True label
Predicted label
0
1
50000
40000
30000
20000
10000

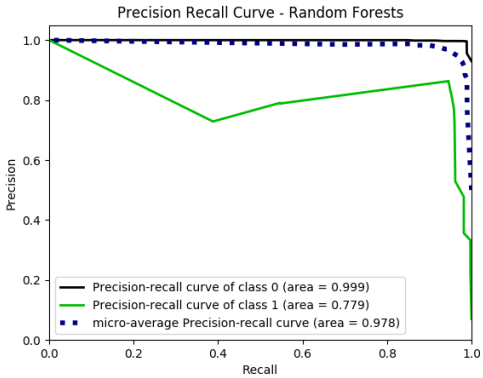

Precision Recall Curve - Decision Tree
Precision
Recall
Precision-recall curve of class 0 (area = 0.991)
Precision-recall curve of class 1 (area = 0.381)
micro-average Precision-recall curve (area = 0.891)

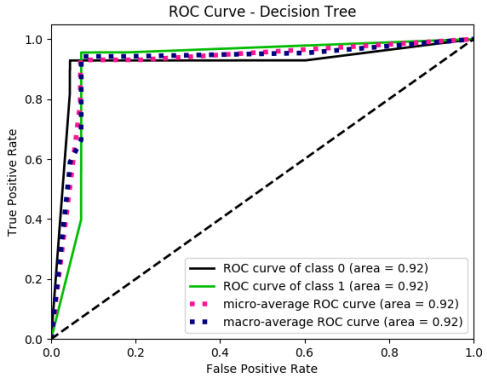

Precision Recall Curve - GaussianNB
Precision
Recall
Precision-recall curve of class 0 (area = 0.994)
Precision-recall curve of class 1 (area = 0.913)
micro-average Precision-recall curve (area = 0.532)

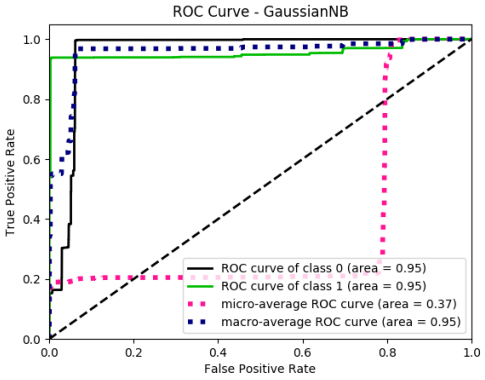

Precision Recall Curve - Logistic Regression
Precision
Recall
Precision-recall curve of class 0 (area = 0.999)
Precision-recall curve of class 1 (area = 0.908)
micro-average Precision-recall curve (area = 0.985)

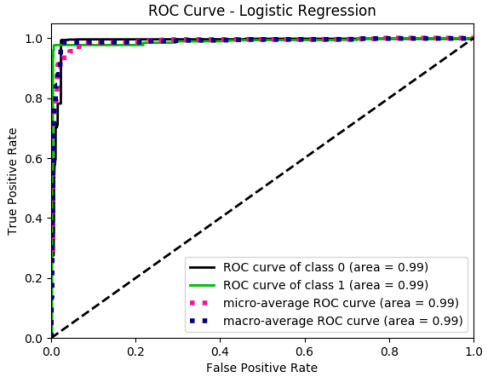

Precision Recall Curve - Naive Bayes
Precision
Recall
Precision-recall curve of class 0 (area = 0.950)
Precision-recall curve of class 1 (area = 0.089)
micro-average Precision-recall curve (area = 0.402)

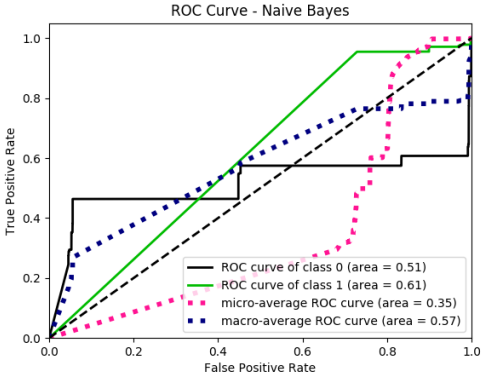

Precision Recall Curve - Neural Nets
Precision
Recall
Precision-recall curve of class 0 (area = 0.998)
Precision-recall curve of class 1 (area = 0.832)
micro-average Precision-recall curve (area = 0.994)

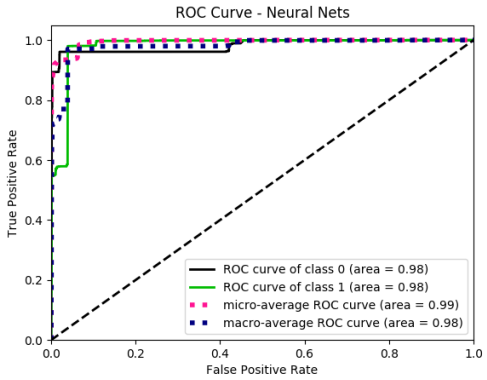

Precision Recall Curve - Random Forests
Precision
Recall
Precision-recall curve of class 0 (area = 0.999)
Precision-recall curve of class 1 (area = 0.779)
micro-average Precision-recall curve (area = 0.978)

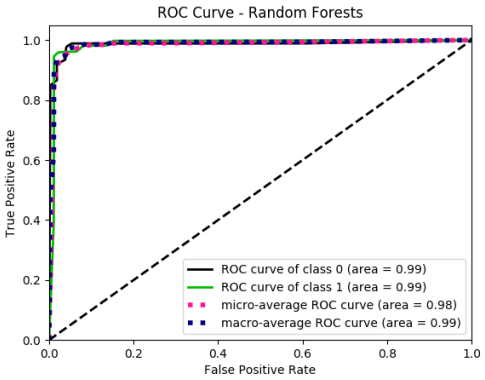

ROC Curve - Decision Tree
True Positive Rate
False Positive Rate
ROC curve of class 0 (area = 0.92)
ROC curve of class 1 (area = 0.92)
micro-average ROC curve (area = 0.92)
macro-average ROC curve (area = 0.92)


ROC Curve - GaussianNB
True Positive Rate
False Positive Rate
ROC curve of class 0 (area = 0.95)
ROC curve of class 1 (area = 0.95)
micro-average ROC curve (area = 0.37)
macro-average ROC curve (area = 0.95)

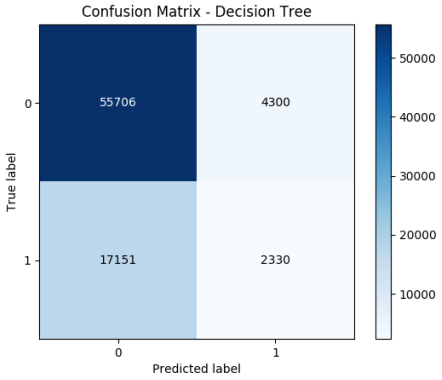

ROC Curve - Logistic Regression
True Positive Rate
False Positive Rate
ROC curve of class 0 (area = 0.99)
ROC curve of class 1 (area = 0.99)
micro-average ROC curve (area = 0.99)
macro-average ROC curve (area = 0.99)

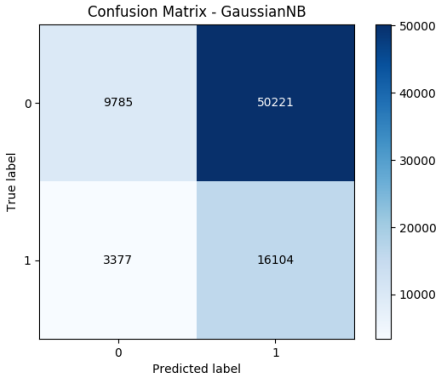

ROC Curve - Naive Bayes
True Positive Rate
False Positive Rate
ROC curve of class 0 (area = 0.51)
ROC curve of class 1 (area = 0.61)
micro-average ROC curve (area = 0.35)
macro-average ROC curve (area = 0.57)

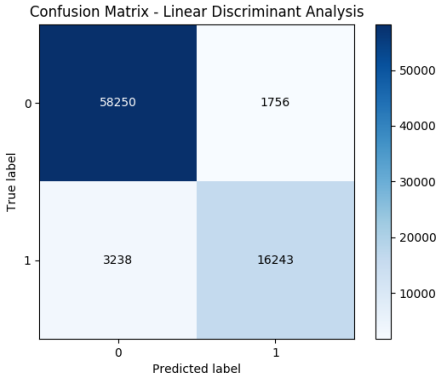

ROC Curve - Neural Nets
True Positive Rate
False Positive Rate
ROC curve of class 0 (area = 0.98)
ROC curve of class 1 (area = 0.98)
micro-average ROC curve (area = 0.99)
macro-average ROC curve (area = 0.98)

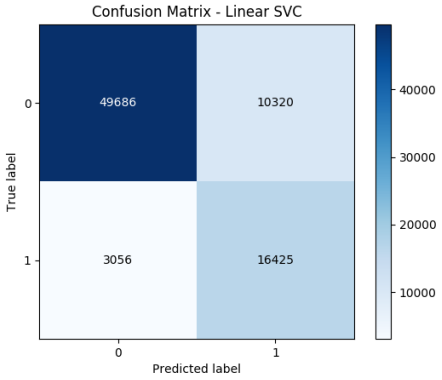

ROC Curve - Random Forests
True Positive Rate
False Positive Rate
ROC curve of class 0 (area = 0.99)
ROC curve of class 1 (area = 0.99)
micro-average ROC curve (area = 0.98)
macro-average ROC curve (area = 0.99)

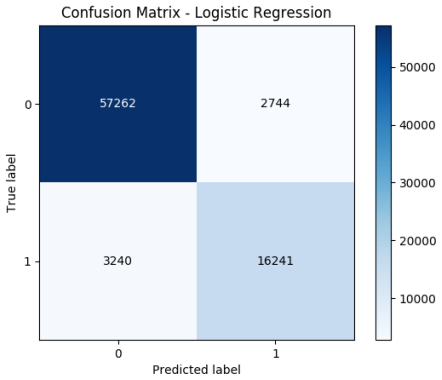
7.1.2 Neris

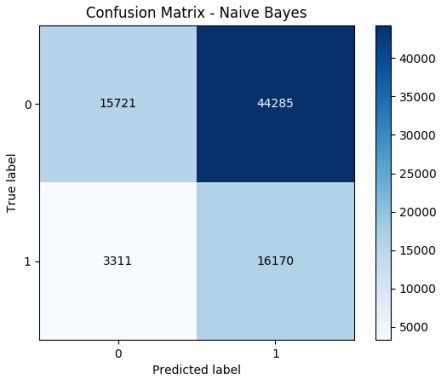

Confusion Matrix - Decision Tree
55706
4300
17151
2330
True label
Predicted label
0
1
50000
40000
30000
20000
10000

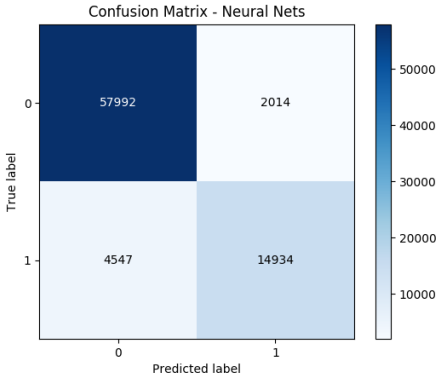

Confusion Matrix - GaussianNB
9785
50221
3377
16104
True label
Predicted label
0
1
50000
40000
30000
20000
10000

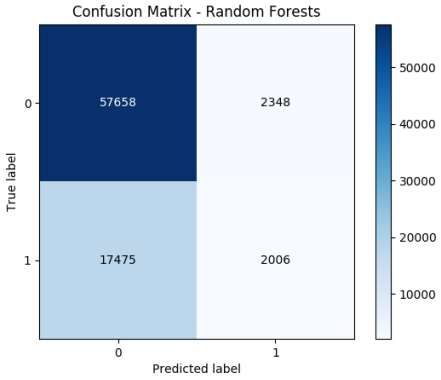

Confusion Matrix - Linear Discriminant Analysis
58250
1756
3238
16243
True label
Predicted label
0
1
50000
40000
30000
20000
10000

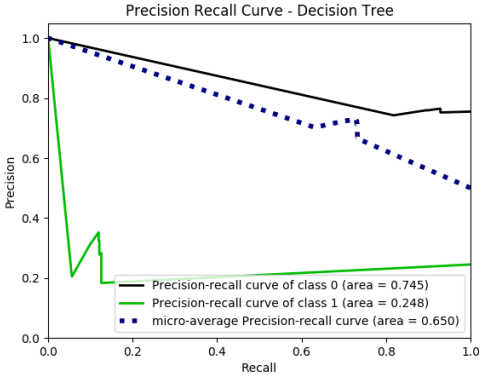

Confusion Matrix - Linear SVC
49686
10320
3056
16425
True label
Predicted label
0
1
40000
30000
20000
10000

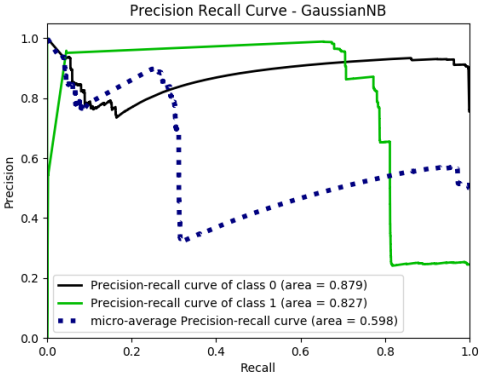

Confusion Matrix - Logistic Regression
57262
2744
3240
16241
True label
Predicted label
0
1
50000
40000
30000
20000
10000

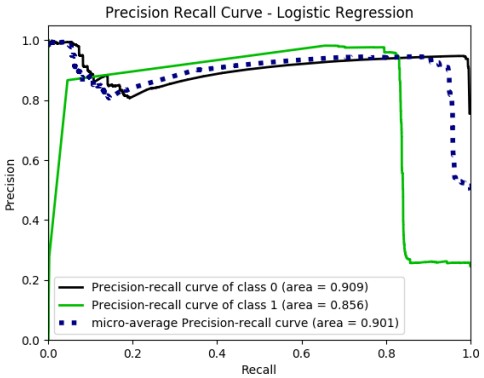

Confusion Matrix - Naive Bayes
15721
44285
3311
16170
True label
Predicted label
0
1
40000
35000
30000
25000
20000
15000
10000
5000

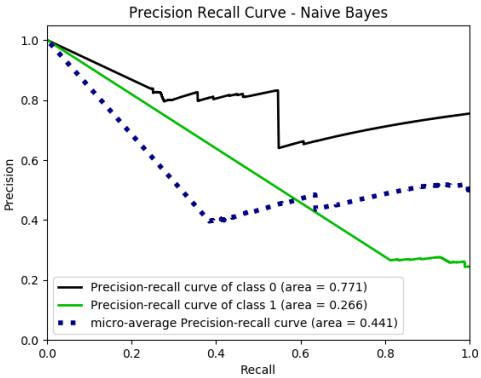

Confusion Matrix - Neural Nets
57992
2014
4547
14934
True label
Predicted label
0
1
50000
40000
30000
20000
10000

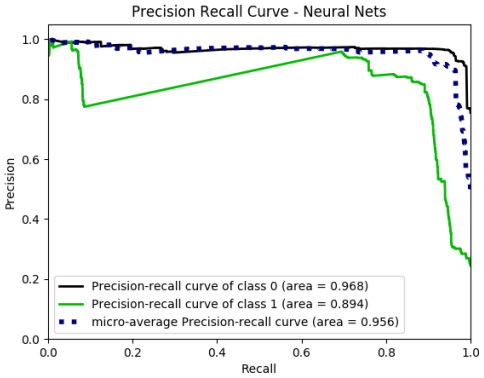

Confusion Matrix - Random Forests
57658
2348
17475
2006
True label
Predicted label
0
1
50000
40000
30000
20000
10000

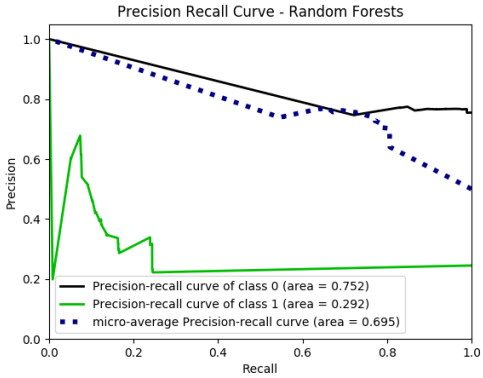

Precision Recall Curve - Decision Tree
Precision-recall curve of class 0 (area = 0.745)
Precision-recall curve of class 1 (area = 0.248)
micro-average Precision-recall curve (area = 0.650)
Precision
Recall

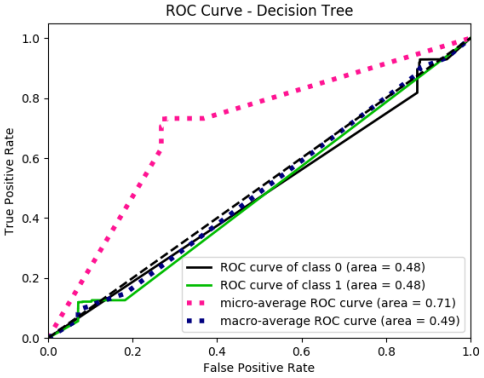

Precision Recall Curve - GaussianNB
Precision-recall curve of class 0 (area = 0.879)
Precision-recall curve of class 1 (area = 0.827)
micro-average Precision-recall curve (area = 0.598)
Precision
Recall

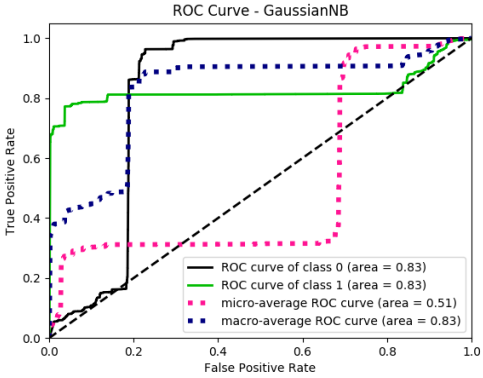

Precision Recall Curve - Logistic Regression
Precision-recall curve of class 0 (area = 0.909)
Precision-recall curve of class 1 (area = 0.856)
micro-average Precision-recall curve (area = 0.901)
Precision
Recall

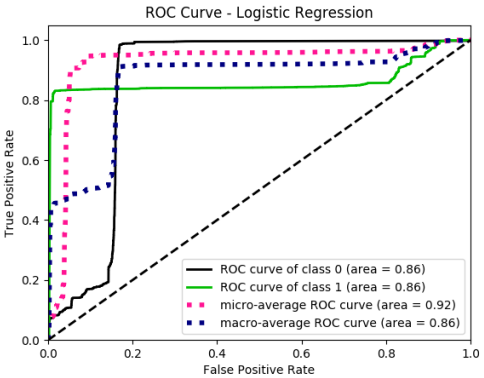

Precision Recall Curve - Naive Bayes
Precision-recall curve of class 0 (area = 0.771)
Precision-recall curve of class 1 (area = 0.266)
micro-average Precision-recall curve (area = 0.441)
Precision
Recall

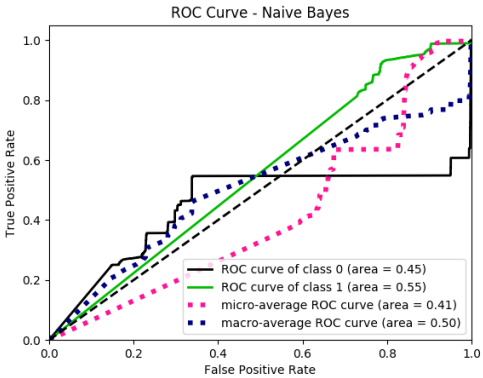

Precision Recall Curve - Neural Nets
Precision
Recall
Precision-recall curve of class 0 (area = 0.968)
Precision-recall curve of class 1 (area = 0.894)
micro-average Precision-recall curve (area = 0.956)

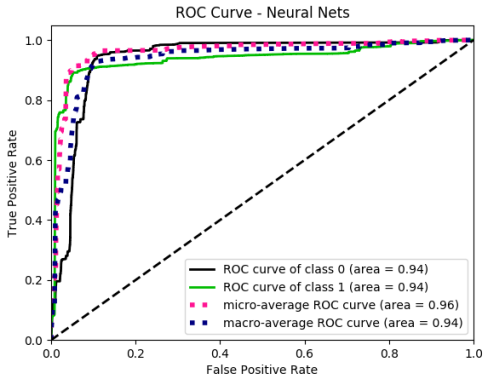

Precision Recall Curve - Random Forests
Precision
Recall
Precision-recall curve of class 0 (area = 0.752)
Precision-recall curve of class 1 (area = 0.292)
micro-average Precision-recall curve (area = 0.695)

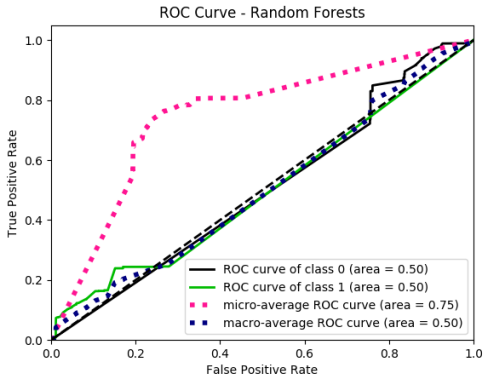

ROC Curve - Decision Tree
True Positive Rate
False Positive Rate
ROC curve of class 0 (area = 0.48)
ROC curve of class 1 (area = 0.48)
micro-average ROC curve (area = 0.71)
macro-average ROC curve (area = 0.49)


ROC Curve - GaussianNB
True Positive Rate
False Positive Rate
ROC curve of class 0 (area = 0.83)
ROC curve of class 1 (area = 0.83)
micro-average ROC curve (area = 0.51)
macro-average ROC curve (area = 0.83)

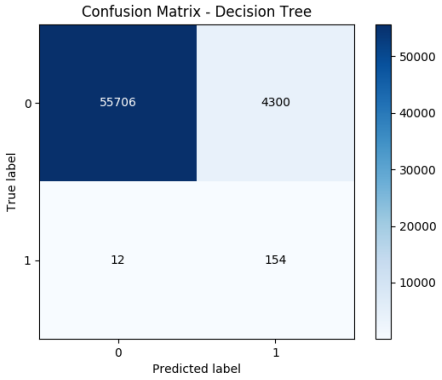

ROC Curve - Logistic Regression
True Positive Rate
False Positive Rate
ROC curve of class 0 (area = 0.86)
ROC curve of class 1 (area = 0.86)
micro-average ROC curve (area = 0.92)
macro-average ROC curve (area = 0.86)

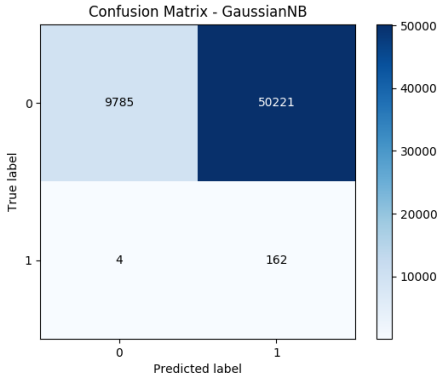

ROC Curve - Naive Bayes
True Positive Rate
False Positive Rate
ROC curve of class 0 (area = 0.45)
ROC curve of class 1 (area = 0.55)
micro-average ROC curve (area = 0.41)
macro-average ROC curve (area = 0.50)

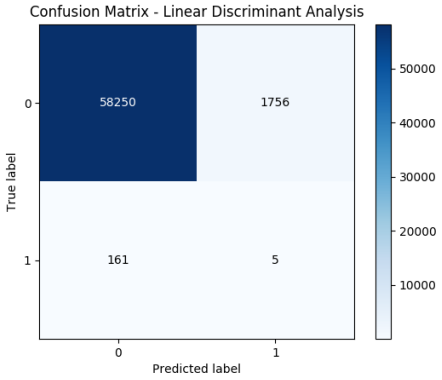

ROC Curve - Neural Nets
True Positive Rate
False Positive Rate
ROC curve of class 0 (area = 0.94)
ROC curve of class 1 (area = 0.94)
micro-average ROC curve (area = 0.96)
macro-average ROC curve (area = 0.94)

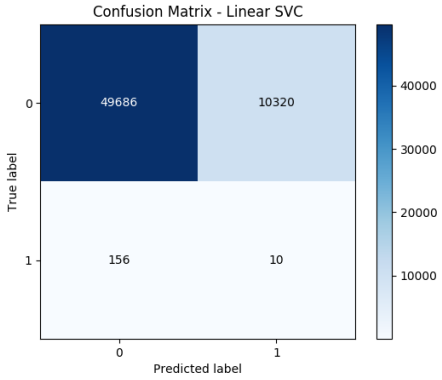

ROC Curve - Random Forests
True Positive Rate
False Positive Rate
ROC curve of class 0 (area = 0.50)
ROC curve of class 1 (area = 0.50)
micro-average ROC curve (area = 0.75)
macro-average ROC curve (area = 0.50)

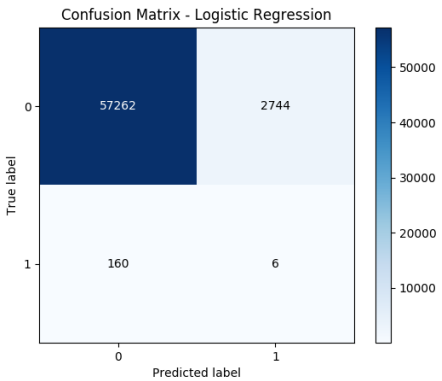
7.1.3 RBot

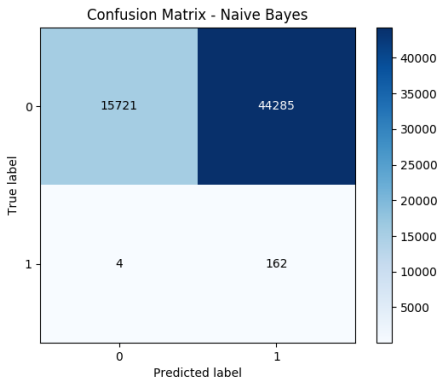

Confusion Matrix - Decision Tree
55706
4300
12
154
True label
Predicted label
0
1
50000
40000
30000
20000
10000

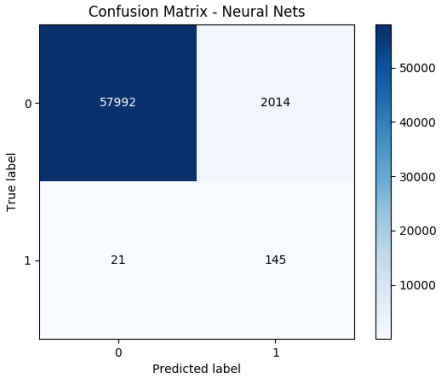

Confusion Matrix - GaussianNB
9785
50221
4
162
True label
Predicted label
0
1
50000
40000
30000
20000
10000

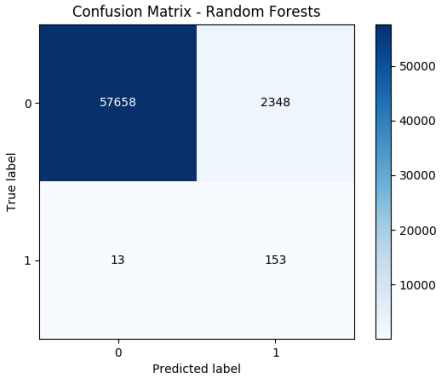

Confusion Matrix - Linear Discriminant Analysis
58250
1756
161
5
True label
Predicted label
0
1
50000
40000
30000
20000
10000

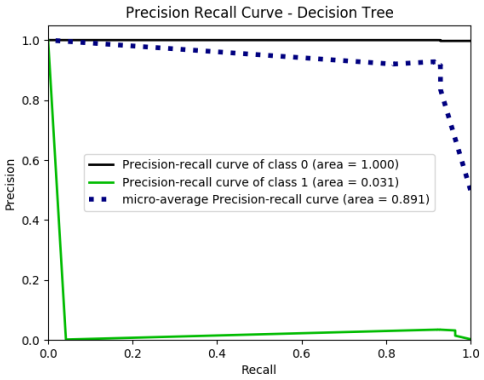

Confusion Matrix - Linear SVC
49686
10320
156
10
True label
Predicted label
0
1
40000
30000
20000
10000

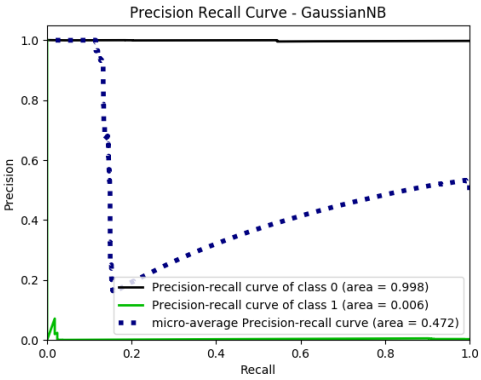

Confusion Matrix - Logistic Regression
57262
2744
160
6
True label
Predicted label
0
1
50000
40000
30000
20000
10000

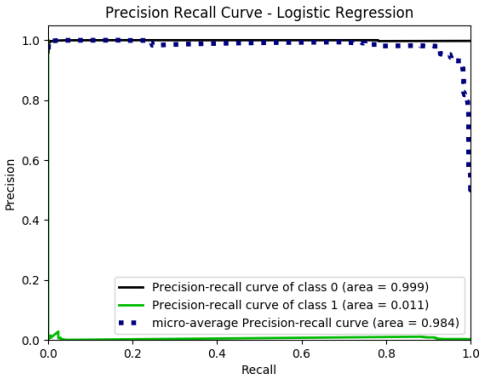

Confusion Matrix - Naive Bayes
15721
44285
4
162
True label
Predicted label
0
1
40000
35000
30000
25000
20000
15000
10000
5000

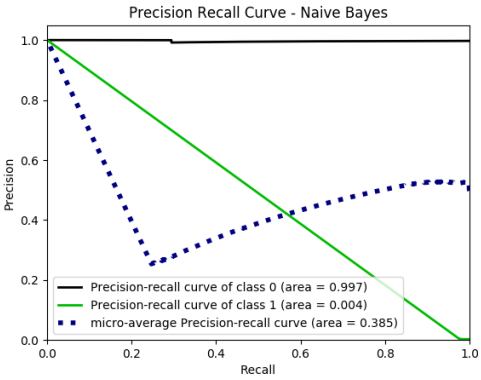

Confusion Matrix - Neural Nets
57992
2014
21
145
True label
Predicted label
0
1
50000
40000
30000
20000
10000

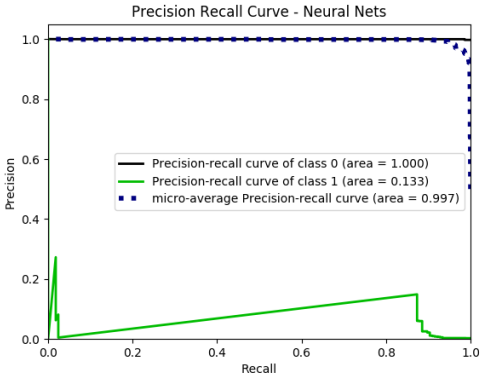

Confusion Matrix - Random Forests
57658
2348
13
153
True label
Predicted label
0
1
50000
40000
30000
20000
10000

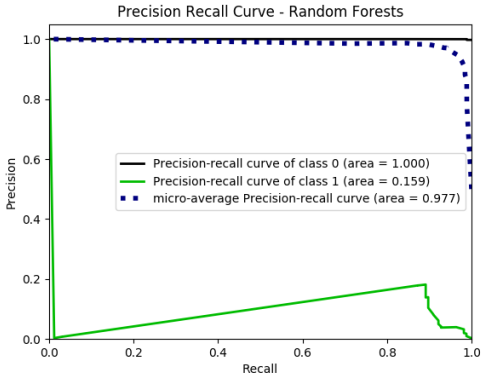

Precision Recall Curve - Decision Tree
Precision-recall curve of class 0 (area = 1.000)
Precision-recall curve of class 1 (area = 0.031)
micro-average Precision-recall curve (area = 0.891)
Precision
Recall

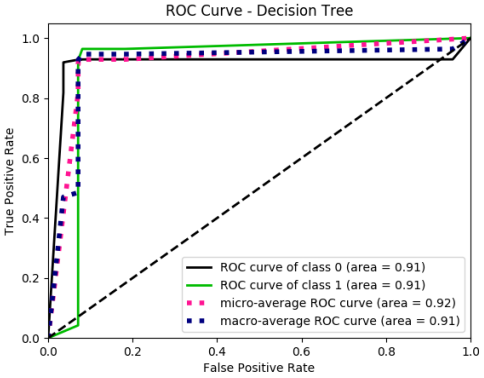

Precision Recall Curve - GaussianNB
Precision-recall curve of class 0 (area = 0.998)
Precision-recall curve of class 1 (area = 0.006)
micro-average Precision-recall curve (area = 0.472)
Precision
Recall

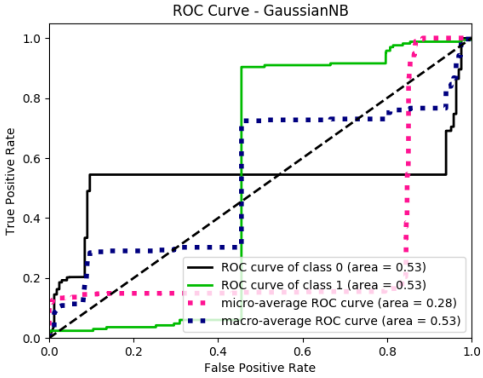

Precision Recall Curve - Logistic Regression
Precision-recall curve of class 0 (area = 0.999)
Precision-recall curve of class 1 (area = 0.011)
micro-average Precision-recall curve (area = 0.984)
Precision
Recall

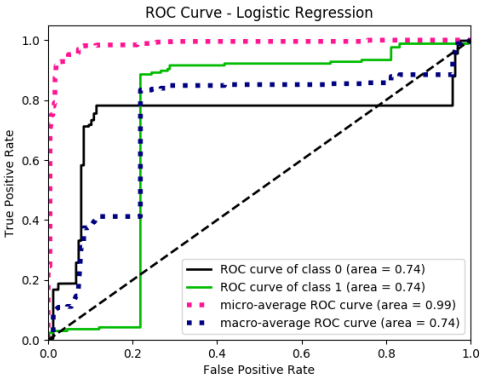

Precision Recall Curve - Naive Bayes
Precision-recall curve of class 0 (area = 0.997)
Precision-recall curve of class 1 (area = 0.004)
micro-average Precision-recall curve (area = 0.385)
Precision
Recall

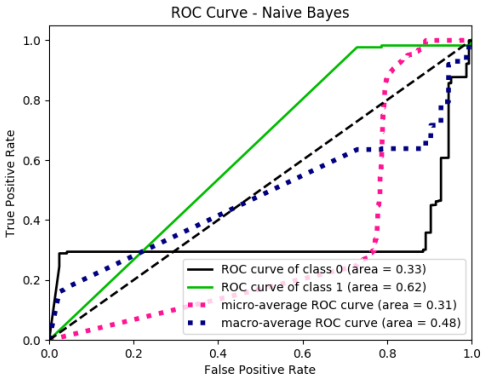

Precision Recall Curve - Neural Nets
Precision-recall curve of class 0 (area = 1.000)
Precision-recall curve of class 1 (area = 0.133)
micro-average Precision-recall curve (area = 0.997)
Precision
Recall

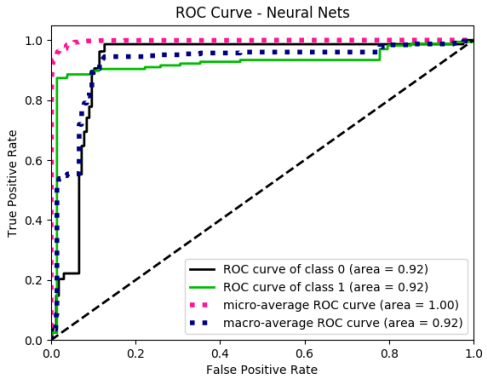

Precision Recall Curve - Random Forests
Precision-recall curve of class 0 (area = 1.000)
Precision-recall curve of class 1 (area = 0.159)
micro-average Precision-recall curve (area = 0.977)
Precision
Recall

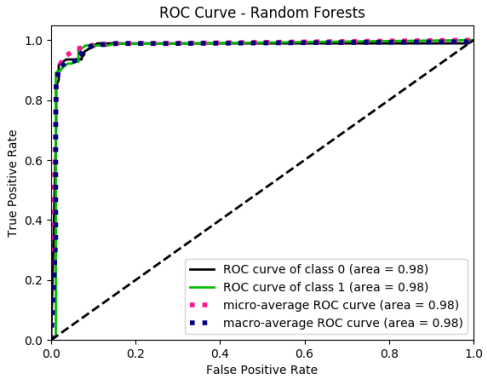

ROC Curve - Decision Tree
ROC curve of class 0 (area = 0.91)
ROC curve of class 1 (area = 0.91)
micro-average ROC curve (area = 0.92)
macro-average ROC curve (area = 0.91)
True Positive Rate
False Positive Rate


ROC Curve - GaussianNB
ROC curve of class 0 (area = 0.53)
ROC curve of class 1 (area = 0.53)
micro-average ROC curve (area = 0.28)
macro-average ROC curve (area = 0.53)
True Positive Rate
False Positive Rate

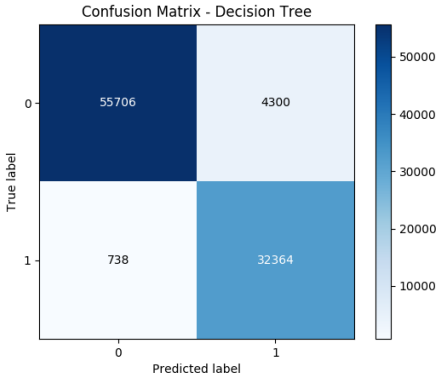

ROC Curve - Logistic Regression
ROC curve of class 0 (area = 0.74)
ROC curve of class 1 (area = 0.74)
micro-average ROC curve (area = 0.99)
macro-average ROC curve (area = 0.74)
True Positive Rate
False Positive Rate

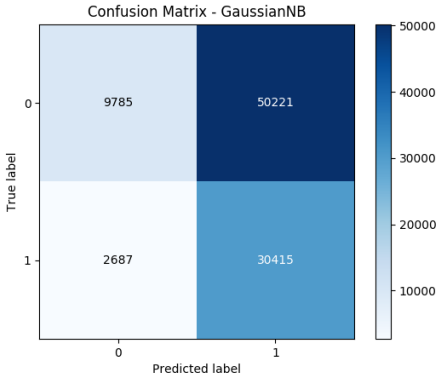

ROC Curve - Naive Bayes
ROC curve of class 0 (area = 0.33)
ROC curve of class 1 (area = 0.62)
micro-average ROC curve (area = 0.31)
macro-average ROC curve (area = 0.48)
True Positive Rate
False Positive Rate

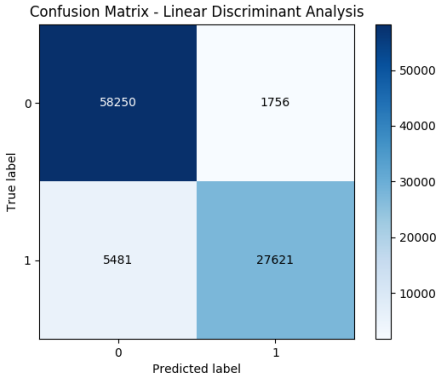

ROC Curve - Neural Nets
True Positive Rate
False Positive Rate
ROC curve of class 0 (area = 0.92)
ROC curve of class 1 (area = 0.92)
micro-average ROC curve (area = 1.00)
macro-average ROC curve (area = 0.92)

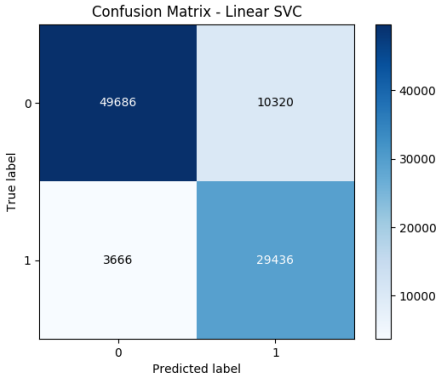

ROC Curve - Random Forests
True Positive Rate
False Positive Rate
ROC curve of class 0 (area = 0.98)
ROC curve of class 1 (area = 0.98)
micro-average ROC curve (area = 0.98)
macro-average ROC curve (area = 0.98)

7.1.4 Virut

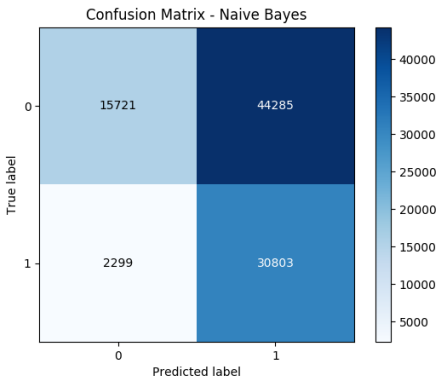

Confusion Matrix - Decision Tree
55706
4300
738
32364
True label
Predicted label
0
1
50000
40000
30000
20000
10000

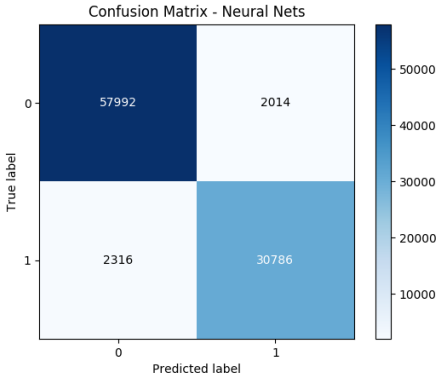

Confusion Matrix - GaussianNB
9785
50221
2687
30415
True label
Predicted label
0
1
50000
40000
30000
20000
10000

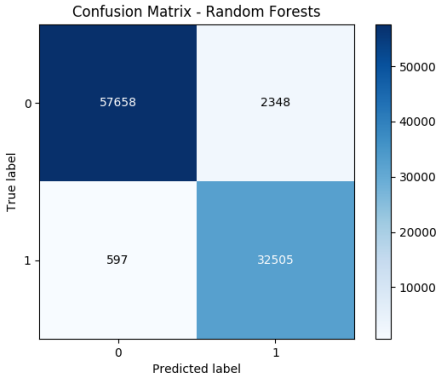

Confusion Matrix - Linear Discriminant Analysis
58250
1756
5481
27621
True label
Predicted label
0
1
50000
40000
30000
20000
10000

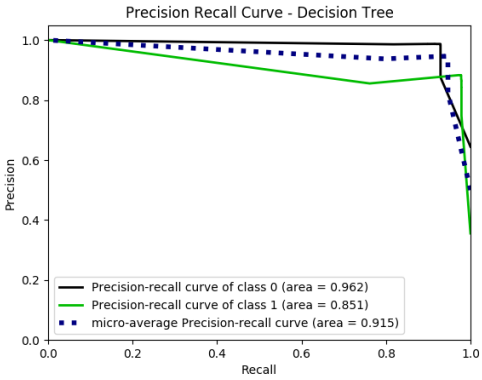

Confusion Matrix - Linear SVC
49686
10320
3666
29436
True label
Predicted label
0
1
40000
30000
20000
10000

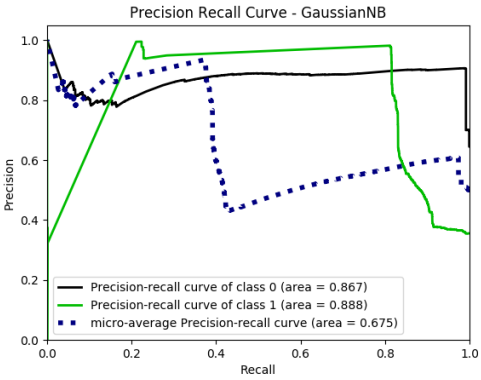

Confusion Matrix - Logistic Regression
57262
2744
5293
27809
True label
Predicted label
0
1
50000
40000
30000
20000
10000

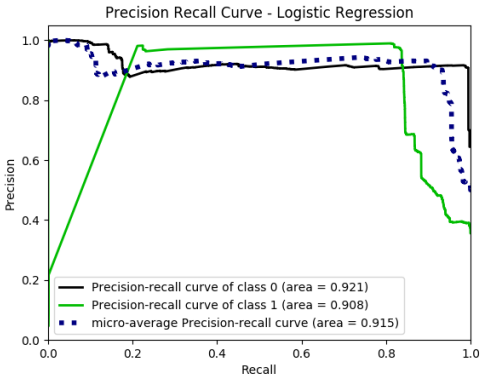

Confusion Matrix - Naive Bayes
15721
44285
2299
30803
True label
Predicted label
0
1
40000
35000
30000
25000
20000
15000
10000
5000

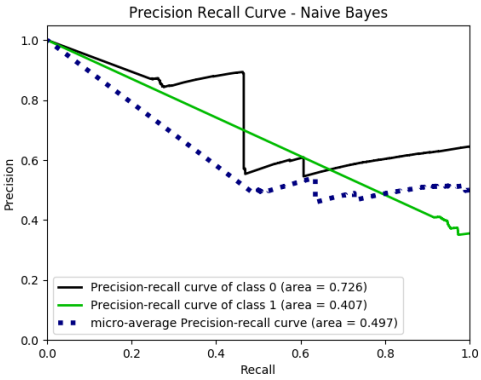

Confusion Matrix - Neural Nets
57992
2014
2316
30786
True label
Predicted label
0
1
50000
40000
30000
20000
10000

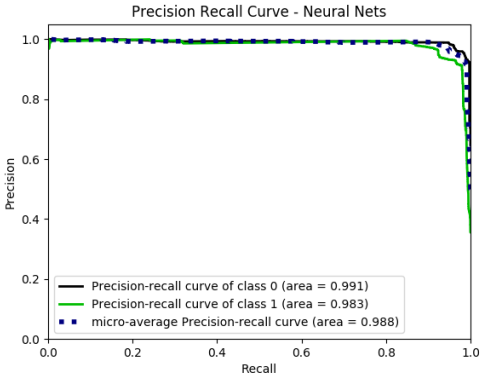

Confusion Matrix - Random Forests
57658
2348
597
32505
True label
Predicted label
0
1
50000
40000
30000
20000
10000

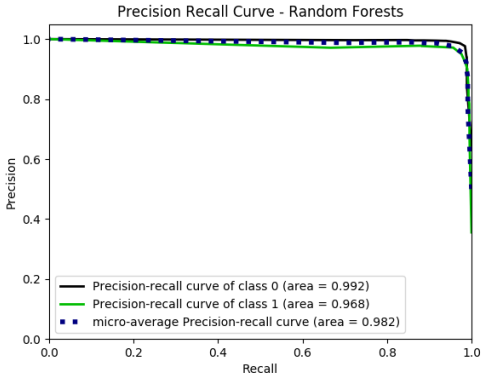

Precision Recall Curve - Decision Tree
Precision-recall curve of class 0 (area = 0.962)
Precision-recall curve of class 1 (area = 0.851)
micro-average Precision-recall curve (area = 0.915)
Precision
Recall

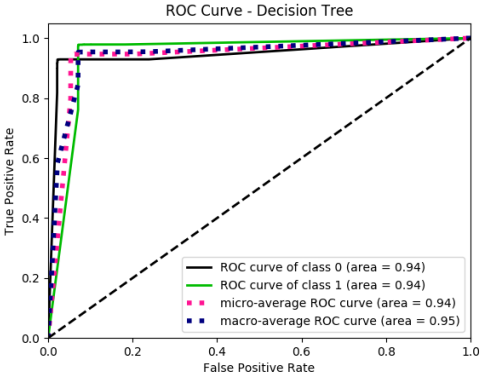

Precision Recall Curve - GaussianNB
Precision-recall curve of class 0 (area = 0.867)
Precision-recall curve of class 1 (area = 0.888)
micro-average Precision-recall curve (area = 0.675)
Precision
Recall

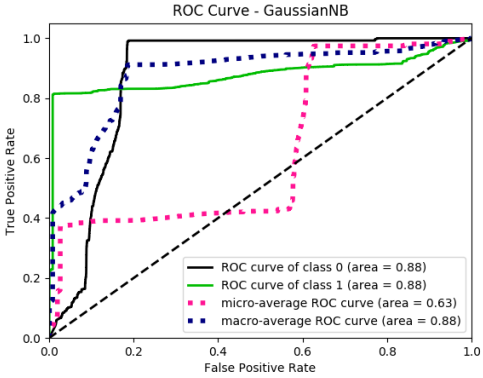

Precision Recall Curve - Logistic Regression
Precision-recall curve of class 0 (area = 0.921)
Precision-recall curve of class 1 (area = 0.908)
micro-average Precision-recall curve (area = 0.915)
Precision
Recall

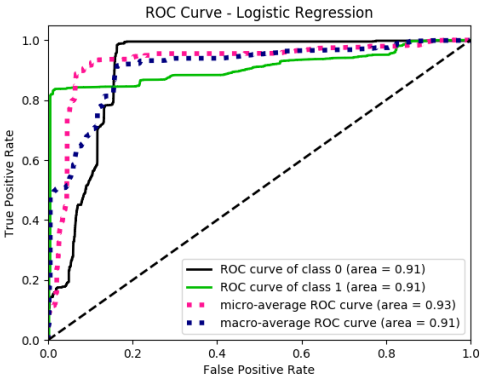

Precision Recall Curve - Naive Bayes
Precision-recall curve of class 0 (area = 0.726)
Precision-recall curve of class 1 (area = 0.407)
micro-average Precision-recall curve (area = 0.497)
Precision
Recall

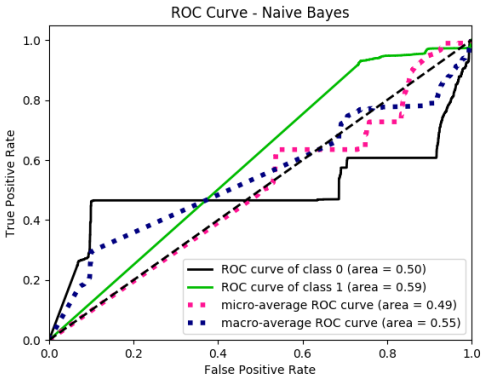

Precision Recall Curve - Neural Nets
Precision
Recall
Precision-recall curve of class 0 (area = 0.991)
Precision-recall curve of class 1 (area = 0.983)
micro-average Precision-recall curve (area = 0.988)

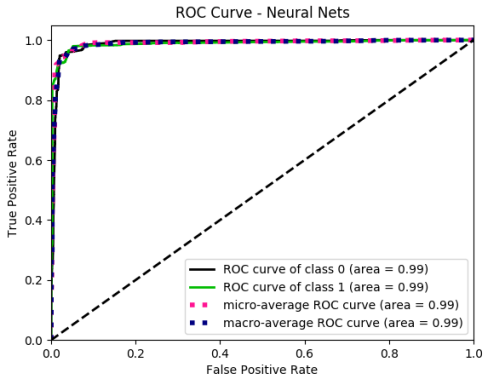

Precision Recall Curve - Random Forests
Precision
Recall
Precision-recall curve of class 0 (area = 0.992)
Precision-recall curve of class 1 (area = 0.968)
micro-average Precision-recall curve (area = 0.982)

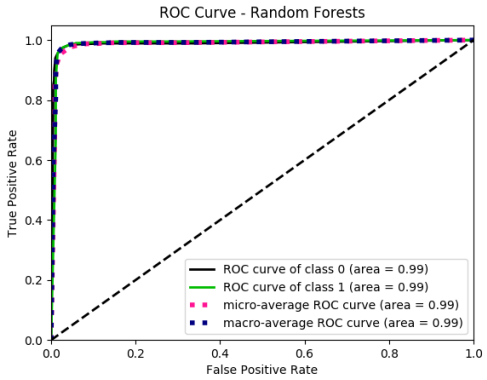

ROC Curve - Decision Tree
True Positive Rate
False Positive Rate
ROC curve of class 0 (area = 0.94)
ROC curve of class 1 (area = 0.94)
micro-average ROC curve (area = 0.94)
macro-average ROC curve (area = 0.95)


ROC Curve - GaussianNB
True Positive Rate
False Positive Rate
ROC curve of class 0 (area = 0.88)
ROC curve of class 1 (area = 0.88)
micro-average ROC curve (area = 0.63)
macro-average ROC curve (area = 0.88)

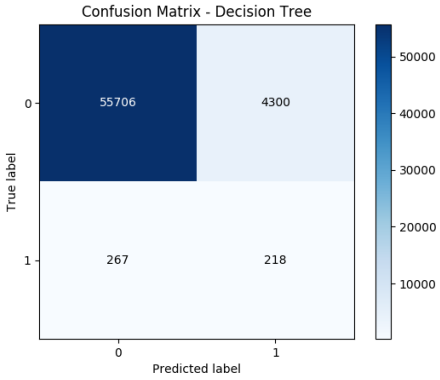

ROC Curve - Logistic Regression
True Positive Rate
False Positive Rate
ROC curve of class 0 (area = 0.91)
ROC curve of class 1 (area = 0.91)
micro-average ROC curve (area = 0.93)
macro-average ROC curve (area = 0.91)

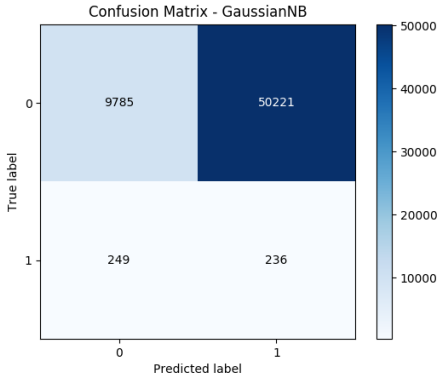

ROC Curve - Naive Bayes
True Positive Rate
False Positive Rate
ROC curve of class 0 (area = 0.50)
ROC curve of class 1 (area = 0.59)
micro-average ROC curve (area = 0.49)
macro-average ROC curve (area = 0.55)

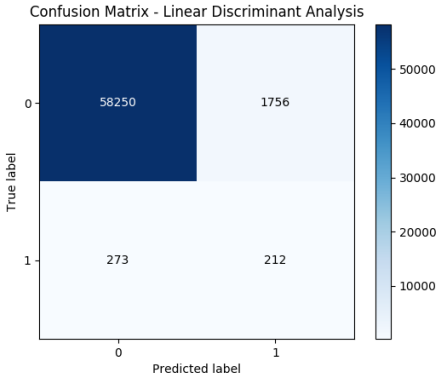

ROC Curve - Neural Nets
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve of class 0 (area = 0.99)
ROC curve of class 1 (area = 0.99)
micro-average ROC curve (area = 0.99)
macro-average ROC curve (area = 0.99)

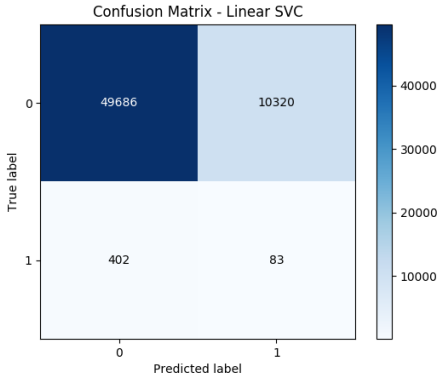

ROC Curve - Random Forests
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve of class 0 (area = 0.99)
ROC curve of class 1 (area = 0.99)
micro-average ROC curve (area = 0.99)
macro-average ROC curve (area = 0.99)

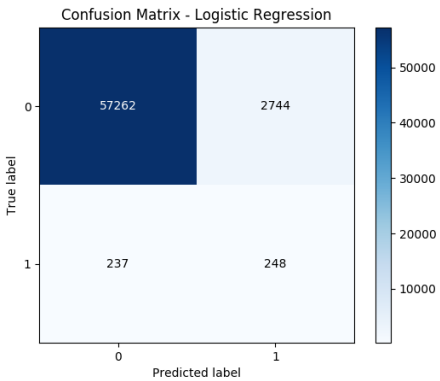
7.1.5 TBot

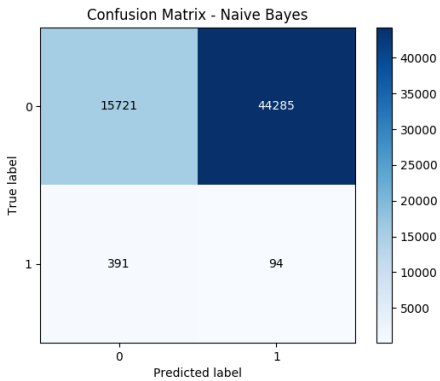

Confusion Matrix - Decision Tree
55706
4300
267
218
True label
Predicted label
0
1
50000
40000
30000
20000
10000

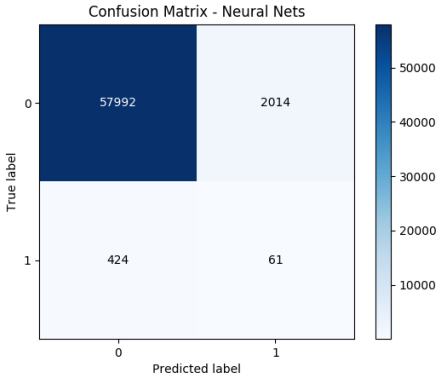

Confusion Matrix - GaussianNB
9785
50221
249
236
True label
Predicted label
0
1
50000
40000
30000
20000
10000

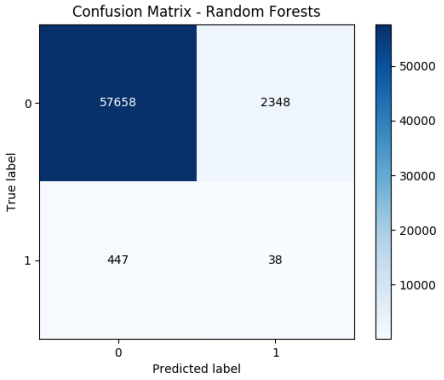

Confusion Matrix - Linear Discriminant Analysis
58250
1756
273
212
True label
Predicted label
0
1
50000
40000
30000
20000
10000

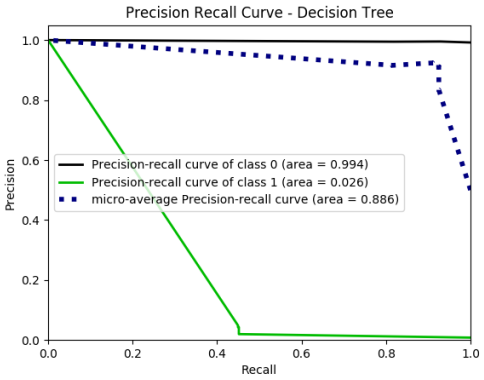

Confusion Matrix - Linear SVC
49686
10320
402
83
True label
Predicted label
0
1
40000
30000
20000
10000

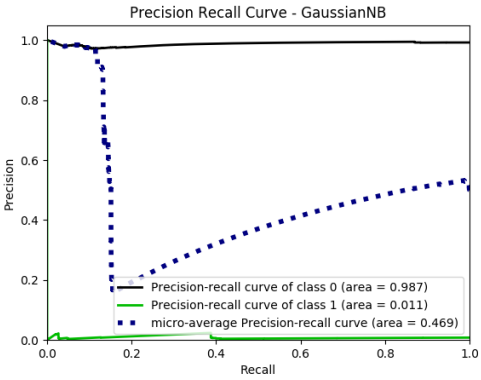

Confusion Matrix - Logistic Regression
57262
2744
237
248
True label
Predicted label
0
1
50000
40000
30000
20000
10000

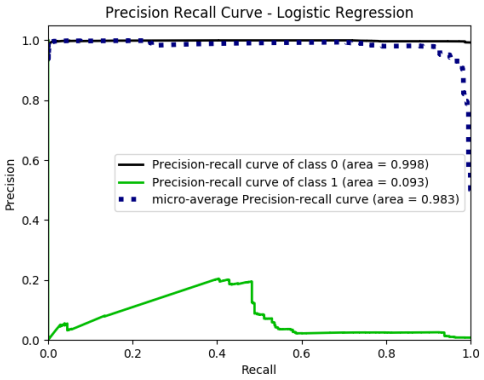

Confusion Matrix - Naive Bayes
15721
44285
391
94
True label
Predicted label
0
1
40000
35000
30000
25000
20000
15000
10000
5000

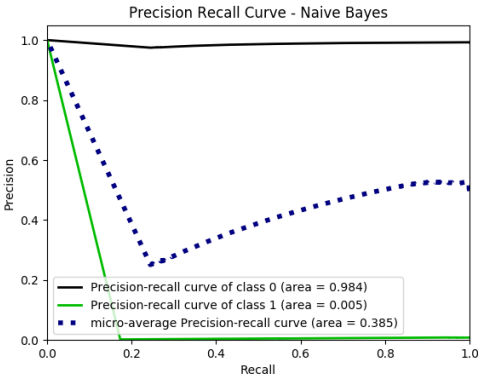

Confusion Matrix - Neural Nets
57992
2014
424
61
True label
Predicted label
0
1
50000
40000
30000
20000
10000

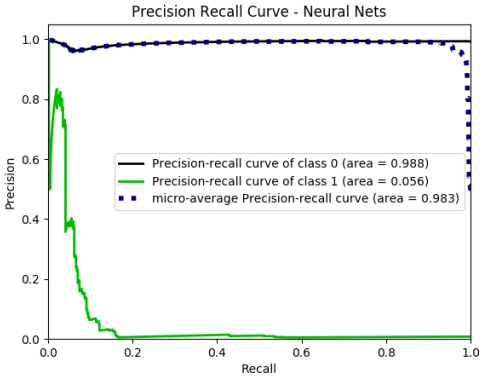

Confusion Matrix - Random Forests
57658
2348
447
38
True label
Predicted label
0
1
50000
40000
30000
20000
10000

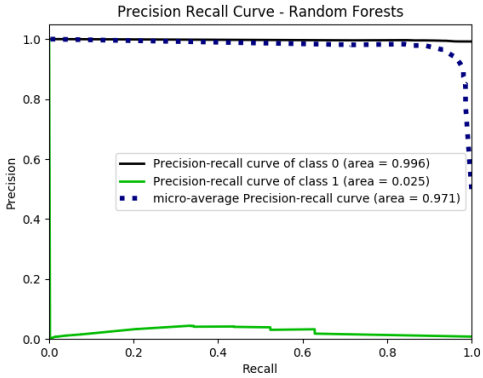

Precision Recall Curve - Decision Tree
Precision-recall curve of class 0 (area = 0.994)
Precision-recall curve of class 1 (area = 0.026)
micro-average Precision-recall curve (area = 0.886)
Precision
Recall

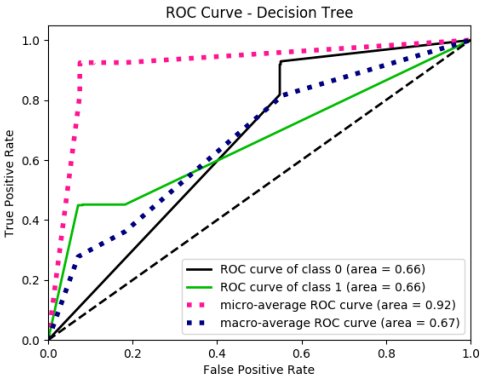

Precision Recall Curve - GaussianNB
Precision-recall curve of class 0 (area = 0.987)
Precision-recall curve of class 1 (area = 0.011)
micro-average Precision-recall curve (area = 0.469)
Precision
Recall

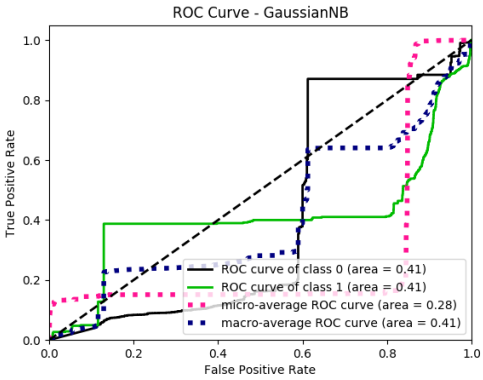

Precision Recall Curve - Logistic Regression
Precision-recall curve of class 0 (area = 0.998)
Precision-recall curve of class 1 (area = 0.093)
micro-average Precision-recall curve (area = 0.983)
Precision
Recall

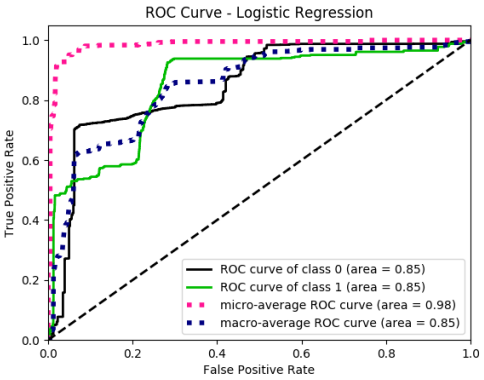

Precision Recall Curve - Naive Bayes
Precision-recall curve of class 0 (area = 0.984)
Precision-recall curve of class 1 (area = 0.005)
micro-average Precision-recall curve (area = 0.385)
Precision
Recall

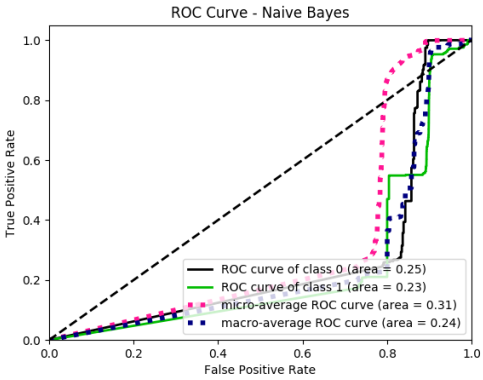

Precision Recall Curve - Neural Nets
Precision-recall curve of class 0 (area = 0.988)
Precision-recall curve of class 1 (area = 0.056)
micro-average Precision-recall curve (area = 0.983)
Precision
Recall

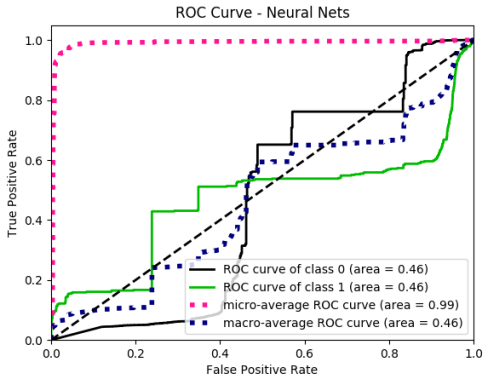

Precision Recall Curve - Random Forests
Precision-recall curve of class 0 (area = 0.996)
Precision-recall curve of class 1 (area = 0.025)
micro-average Precision-recall curve (area = 0.971)
Precision
Recall

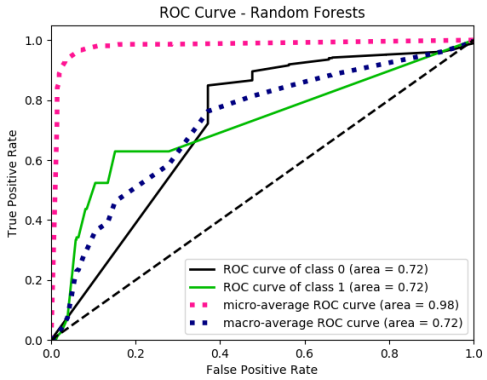

ROC Curve - Decision Tree
ROC curve of class 0 (area = 0.66)
ROC curve of class 1 (area = 0.66)
micro-average ROC curve (area = 0.92)
macro-average ROC curve (area = 0.67)
True Positive Rate
False Positive Rate


ROC Curve - GaussianNB
ROC curve of class 0 (area = 0.41)
ROC curve of class 1 (area = 0.41)
micro-average ROC curve (area = 0.28)
macro-average ROC curve (area = 0.41)
True Positive Rate
False Positive Rate


ROC Curve - Logistic Regression
ROC curve of class 0 (area = 0.85)
ROC curve of class 1 (area = 0.85)
micro-average ROC curve (area = 0.98)
macro-average ROC curve (area = 0.85)
True Positive Rate
False Positive Rate


ROC Curve - Naive Bayes
ROC curve of class 0 (area = 0.25)
ROC curve of class 1 (area = 0.23)
micro-average ROC curve (area = 0.31)
macro-average ROC curve (area = 0.24)
True Positive Rate
False Positive Rate


ROC Curve - Neural Nets
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve of class 0 (area = 0.46)
ROC curve of class 1 (area = 0.46)
micro-average ROC curve (area = 0.99)
macro-average ROC curve (area = 0.46)


ROC Curve - Random Forests
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve of class 0 (area = 0.72)
ROC curve of class 1 (area = 0.72)
micro-average ROC curve (area = 0.98)
macro-average ROC curve (area = 0.72)

7.2 Decider Results Tables

Classifier	IP address	Correct Predictions	Wrong Predictions	True Label
LDA	147.32.84.180	16253	3228	Botnet
	217.163.21.37	23	305	Not Botnet
	66.161.11.90	6	671	Not Botnet
Neural Netwoks	203.84.202.164	39	734	Not Botnet
Decision Tree	217.163.21.37	5	323	Not Botnet
	72.20.15.61	144	1233	Not Botnet
	193.23.181.44	330	2454	Not Botnet
	174.128.246.102	236	1167	Not Botnet
	67.19.72.206	214	1174	Not Botnet
	195.228.245.1	49	596	Not Botnet
Random Forests	217.163.21.37	0	328	Not Botnet
	72.20.15.61	144	1233	Not Botnet
	193.23.181.44	308	2476	Not Botnet
	174.37.196.55	201	1255	Not Botnet
	67.19.72.206	213	1175	Not Botnet

Table 7.1: IP addresses flagged by the Detector while testing only Neris

Classifier	IP address	Correct Predictions	Wrong Predictions	True Label
LDA	147.32.84.160	27621	5481	Botnet
	147.32.80.9	1	2733	Not Botnet
	173.192.170.88	53	390	Not Botnet
	66.161.11.90	6	671	Not Botnet
Neural Netwoks	147.32.84.160	30786	2316	Botnet
	203.84.202.164	39	734	Not Botnet
Decision Tree	147.32.84.160	32364	738	Botnet
	195.228.245	49	596	Not Botnet
Random Forests	147.32.84.160	32505	597	Botnet

Table 7.2: IP addresses flagged by the Detector while testing only Virut

Classifier	IP address	Correct Predictions	Wrong Predictions	True Label
LDA	66.161.11.90	6	671	Not Botnet
Neural Netwoks	147.32.84.170	145	21	Botnet
	293.84.202.164	39	734	Not Botnet
Decision Tree	147.32.84.170	154	12	Botnet
	195.228.245	49	596	Not Botnet
Random Forests	147.32.84.170	153	13	Botnet

Table 7.3: IP addresses flagged by the Detector while testing only RBot