
UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea Magistrale in Informatica

Multivariate Analysis & Statistical Learning

# INTRUSION DETECTION SYSTEMS AND MACHINE LEARNING

FRANCESCO TERROSI

6326113

Anno Accademico 2018-2019

# INDICE

# 1

# INTRODUCTION

Nowadays, we are seeing how much reliance is put into Artificial Intelligence, especially when statistical techniques come in.
The combination of AI and MASL (known as Machine Learning in computer science), gives us a set of very powerful tools in order to accomplish the hardest objectives. Most of the times, these techniques are used for financial business but what happens when we try to apply Machine Learning to Computer Security?
In this paper we will see how powerful can be statistical models when applied to computer security, how they can be applied and in what field and what are the techniques to be used and the steps to build a resilient and secure system.
The focus will be on Intrusion Detection and Prevention System (roughly, you can think to an Anti-Virus as an IPS) based on Anomaly Detection techniques.
Anomaly Detection relies on defining a model for the *normal* behaviour of a user into a system and comparing the observed behaviour of actual users (or programs) with the established model, in order to detect attacks and intrusion. for sake of understanding, you can think to a malware (e.g. a virus) as a "bad user" trying to gain control of your computer system.

In this paper we will go through these topics:

- A general overview of Intrusion Detection and Prevention Systems
- An overview of the techniques used for *Feature Selection*
- An overview of the techniques used for *Classification*
- A practical implementation

# 2

# INTRUSION DETECTION AND PREVENTION SYSTEMS

Intrusion Detection (and Prevention) Systems (from now on, IDS and/or IPS) are particular HW/SW systems developed to detect security threats in a computer system.
Historically, they were developed using signature-based techniques, that is: you define an attack pattern and, if ot recognize that exact pattern in the system's activity, it raises an alarm.

Since Machine Learning is one of the increasing trends, gaining more and more popularity, we began to think if, instead of defining the "attack pattern" (or signature) for a particular malicious activity, we could define the "good behaviour" for users and program to detect malicious behaviour and here is where anomaly-based IDS comes in.

### 2.0.1 *Machine Learning and IDS combined*

Let's get more into the interesting things now. The first question is: How do I define the "normal" behaviour for users and programs? We need to have a dataset representing these behaviours. We have two alternatives:

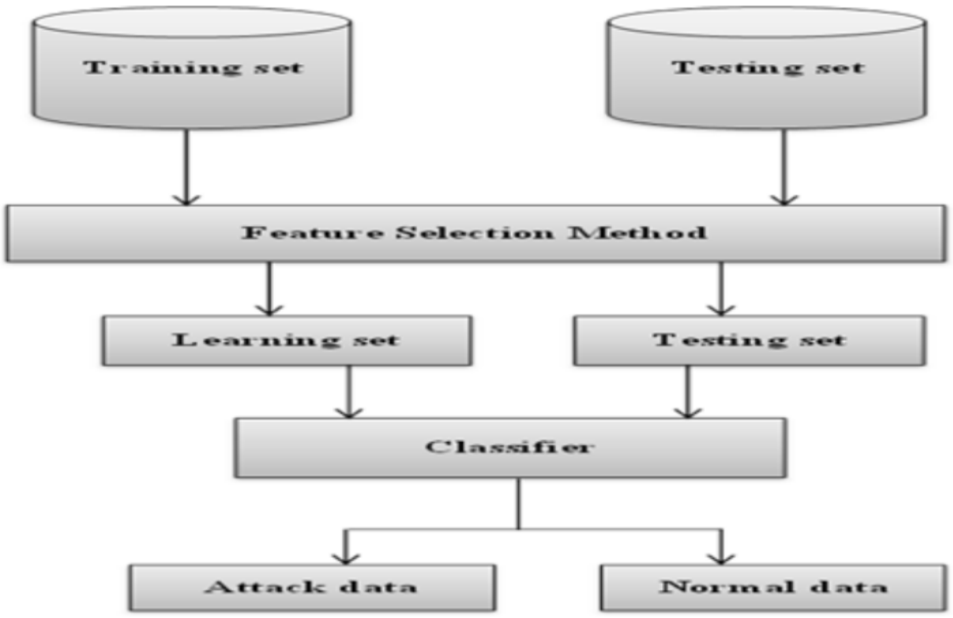
- Stateful Protocol Analysis
  - → We buy these datasets from authorized vendors. These data are usually very accurate and very little work is needed on them. The bad thing? The price is usually very high
- Collect data by ourselves
  - → This approach is usually less expensive but require much more work on the data (such as *feature selection*). For the purpose of this course we will have a better look at this approach, since it allows us to use some more statistical techniques

Data are collected by the IDS using a *sensor* (an HW or SW module). These data are sent to an *analyzer* that compares them with the ones collected in the training phase, in order to detect intrusions. However, the very big problem with these kind of systems is that we want a very small number of *false positives* and *false negatives*, otherwise the whole system is useless!

This is the main reason why we will put some emphasis to *feature selection* and *feature classification*

Roughly, the process used to build this kind of system is depicted in Figure 1:

1) First of all we need a *Train Set*, to tell the system what are the accepted behaviours
2) Now that we have only the relevant components in our train set, we can classify data. Empirically, it has been observed that *Decision Tree* provides the higher accuracy for anomaly detection
3) At this point we are able to test our system with a *Test Set*, to check the accuracy of our model of correctly detecting intrusions



# 3

# FEATURE SELECTION

Feature selection is, no doubt, the critical part when we need to prepare the dataset for the training phase.
If one collects data for the training on its own (i.e. you don't buy data from authorized vendors), tipically they will have a lot of redundant attributes, a lot of useless attributes (for intrusion detection) and what he/she wants is to extract from all the observed data the most representative attributes for attacks and behaviours.
In this section we will briefly describe the techniques that can be used and the pros and cons of each approach.

- *Principal Component Analysis* One of the most used techniques for determining uncorrelated attributes and the one that has been used in this implementation

- *Correlation Based Feature Selection Method* works under the hypothesis that: "Good feature subsets contain features higlhly correlated with the class, yet uncorrelated with each other"

- *Information Gain Based Feature Selection*: roughly, we combine entropy calculation and Information Gain to select the attributes that give us the more information about the class we want to predict

- *Minimum Redundancy Maximum Relevance* is an approach based on the assumption that the most redundant features are the less relevant ones.

## 3.1 PRINCIPAL COMPONENT ANALYSIS

Principal Component Analysis is a technique used for feature selection, which involves algebra techniques. The idea is to calculate the variance-

covariance matrix and to calculate its eigenvectors and eigenvalues, in order to select the biggest ones (i.e. the ones that explain most of the variance in the dataset).

$$V = \begin{pmatrix} \sigma_1 & \sigma_{12} & \sigma_{13} & \dots & \sigma_{1n} \\ \sigma_{21} & \sigma_2 & \sigma_{23} & \dots & \sigma_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \sigma_{n1} & \sigma_{n2} & \sigma_{n3} & \dots & \sigma_n \end{pmatrix}$$

Given this matrix, calculate $det(V - \lambda I) = 0$ to obtain all the eigenvectors and eigenvalues, and choose the "best" ones that is:

$$max\ |x|_{x\ eigenvalue\ \in V}$$

To use this techniques, it is *very* important to scale data: through standardization (mean=0, variance=1) or normalization (make values shrink [0,1]). This is because PCA produces a number n of eigenvalues and eigenvector, representing for each feature the impact it has on variance. If we have values with very different scales, we could give much more importance to integer values than to, for example, a rate values.

# 4

## CLASSIFICATION APPROACHES

A classifier is a tool that tries to categorize unseen patterns in suitable classes.
An accurate (or a "good") train-set is crucial to build a model that provides the best accuracy to Anomaly Detection.
The classifier uses the train-set to build the model that will be used to classify users (and programs) behaviour and to take decision about whether the observed actions are legitimate or not. Empirical studies will help in choosing the most accurate model (in terms of false-positives and false-negatives ratio)

Here there are the most used approached towards classification in Anomaly Detection:

- *Decision Trees* are certainly one of the most used and accurate tools for classification. It builds a model from a train-set that takes decision based on the comparison between the observed features and the nodes of the tree, starting from the root until reaching a decision (leaf) node

- *Nave Bayes* simply uses a form of Bayesian Network to infer intrusions

- *Support Vector Machine* is a technique in which we try to minimize the risk (but we sacrifice some accuracy) in order to optimize speed and scalability

- *Neural Network* is a "connectionist" approach that tries to simulate the human's brain with artificial (digital) neurons. The pros of this approach is, no doubt, its accuracy when the network is trained

for a long period, consisting in multiple generations of NN. The cons is that if an error is made in the first generations, it will be propagated to the next ones

- *K-Nearest Neighbours* is the simplest approach (but surprisingly one of the most accurate in the field of Intrusion Detection!) among all these Machine Learning techniques. The idea is to classify "patterns" instead of features

Collecting some results from empirical studies, here it is a comparison table from all these classification approaches combined with the feature selections technique. As can be seen, the *Decision Tree* classification technique is the one with the *overall* best accuracy.
On the other hand we got the best accuracy combining K-NN with Information Gain

| **Feature Selection technique** | **Classifier** | **Accuracy (in %)** |
|---|---|---|
| CFS | NB | 82.66 |
| | SVM | 76.61 |
| | DT | 98.99 |
| | NN | 83.8 |
| | k-NN | 97.65 |
| IGR | NB | 90.29 |
| | SVM | 94.39 |
| | DT | 97.83 |
| | NN | 97.7 |
| | k-NN | 99.07 |
| PCA | NB | 89.91 |
| | SVM | 96.78 |
| | DT | 98.95 |
| | NN | 97.5 |
| | k-NN | 98.87 |
| Minimum redundancy maximum relevance feature selection | NB | 87.56 |
| | SVM | 88.93 |
| | DT | 98.78 |
| | NN | 94.6 |
| | k-NN | 98.05 |

# 5

# IDS IMPLEMENTATION

Let's get serious now! An overview of IDS and how statistical techniques can be used with it has been proposed the next step is how to make all these things working together and what are the problems when you go from theory to practice!

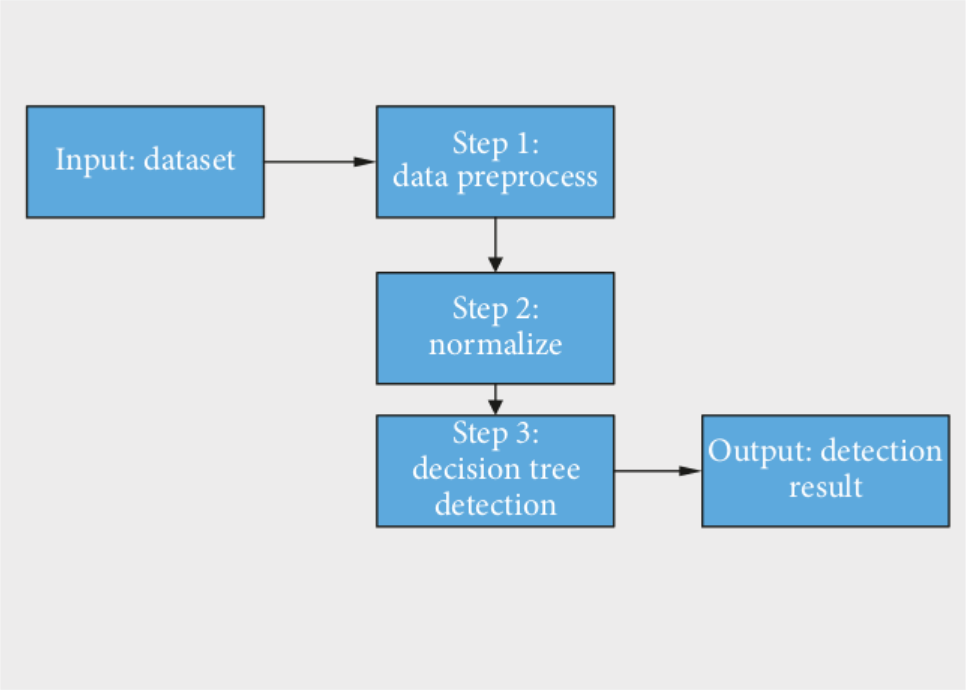
In this example, it has been used the KDDCUP99 dataset, a well known set of network data collected by a military company
The steps one must follow in order to build a good system, both in intrusion detection efficiency and computational efficiency are the followings:

1) **Data preprocess** - In this step the main issue is that most of the times data are numbers and strings mixed together. A numerical representation for all the strings in the dataset is mandatory for the next steps

2) **Data standardization/normalization** - At this point, if we observe the dataset D, it can be seen that the range of numbers in D is not uniform. What must be done here is to run an algorithm that produces an output consisting of a new dataset D' where the values are standardized (or normalized, according to your needs)

3) **Decision Tree method** Last step. The classification tool (the Decision Tree in this case) must be fed with the new dataset D' to build the data structure that will be then used to detect intrusions and malicious behaviours

A visual model for these steps is given in the next figure.




So, the first step is to preprocess data in order to work with them. This step is nothing but technical and requires really little skills

```
values = []
index = []
for i in range(0,len(data[0])):
    !isinstance(data[0][i], int) and !isinstance(data[0][i], float):
        index.append(i)

for i in range(0, len(data)):
    for j in range(0,len(index):
        if data[i][index[j]] not in values:
            values.append(data[i][j])

for i in range(0,len(data))
    for j in range(0,len(values)
        for k in range(0,len(index)):
            if data[i][index[k]] == values[j]:
                data[i][index[k]] = j
                continue
```

At this point, there is the need to standardize the data in order to be able to do the Principal Component Analysis. This is a very important step because, if wrongly implemented, it can really blast down the accuracy of predictions.
Standardizing data mean that the new dataset must have all the distribu-

tions associated to the features with mean = 0 and variance = 1.

Luckily, there is a python library that implements a lot of these features for us: the Pandas library.

The very important drawback here is that we can't standardize the train-set and the test-set separately (or the accuracy of prediction will be mined): the train-set must be standardized first and the same criteria must be used to standardize the test-set as well. In other words: the scaler must be "fit" using the train-set and then use the *same* scaler to apply the transformation to both sets

```
import pandas as pd
from sklearn.preprocessing import StandardScaler

test_set = pd.read_csv(test_set_name)
train_set = pd.read_csv(train_set_name)

# Separating out the features
test_set1 = test_set.loc[:, features].values
train_set1 = train_set.loc[:, features].values

# Separating out the target
y = test_set.loc[:,['target']].values
y1 = train_set.loc[:,['target']].values

scaler = StandardScaler()

scaler.fit(test_set1)

# Standardizing the features
test_set1 = scaler.transform(test_set1)
train_set1 = scaler.transform(train_set1)
```

At this point, Principal Component Analysis algorithm can be run in order to exclude those features that have little significance for our purpose:

```
from sklearn.decomposition import PCA

pca = PCA(.95)      # Notice the .95 parameter: this tells the PCA object
                    # to choose enough components to explain 95% of the data's variance

pca.fit(train_set)

trainSetComponents = pca.transform(train_set)
testSetComponents = pca.transform(test_set)

return (trainSetComponents, testSetComponents)
```

Now that data have been cleaned, standardized, and the features reduced those that are the most relevant, they can be used to feed the Decision Tree algorithm!

For the Tree construction the *rpart* package has been used, due to its acceptance in statistical/machine learning environments. It is also quite efficient and let the user specify a lot of useful parameters for the tree construction.
In the next few lines, the algorithm used to build a Decision Tree is briefly described:

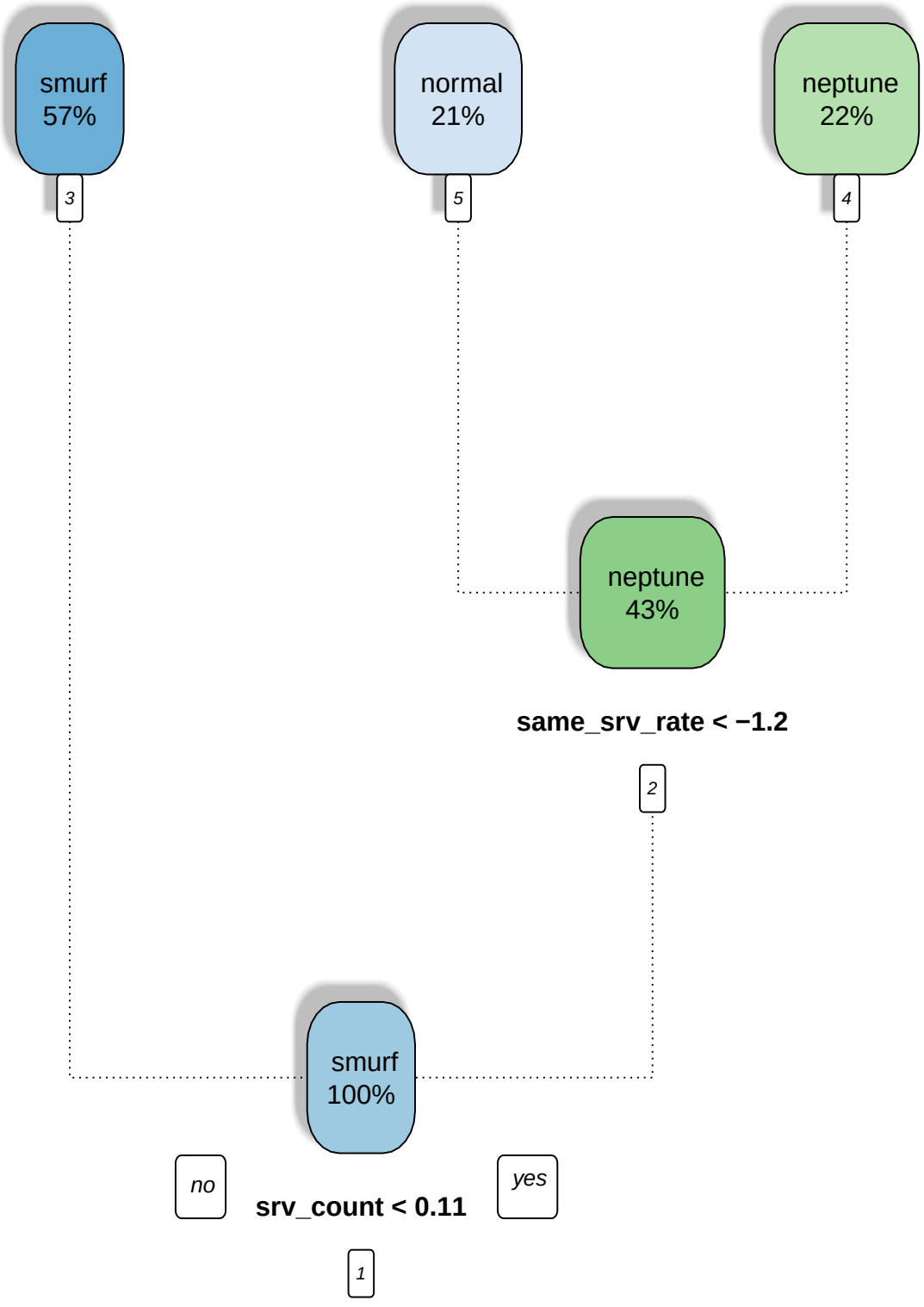
1) For each variable the whole dataset is split in 2 parts, using the Gini Index
   → Since the variables are somehow continuous, splits will be done on boolean conditions: given a *threshold* $t$ features will be split according to the ones greater than $t$, or lower than $t$
2) The variable (or feature) that produced the highest score (the lowest Gini Index) will be the very first split of the Tree (the root node)
3) The algorithm now proceeds recursively on each node, until there are no more variable to split

```
library(rpart)

fit <- rpart(V22 ~ .,                          # Building the tree
   data=train_set_principalComponents,
   control=rpart.control(cp=0.01),
    method="class")

t_pred <- predict(fit, test_set_principalComponents, type="class")  # Predicting on test-set
confMat <- table(test_set_principalComponents$V22,t_pred)           # Computing confusion matrix
accuracy <- sum(diag(confMat))/sum(confMat)                         # Computing accuracy of predictions
```

Small and simple tree example to be read bottom-up



# 6

## RESULTS & CONCLUSIONS

Now that that all the technical things are done we can run our algorithm to see its accuracy.

For empirical purposes, the algorithm have been run with different train-set and with different settings (e.g. complexity parameter), in order to see the differences between an analysis with and without principal components.
Before showing the results here there are some technical data:

The complexity parameter for the tree construction was set to 0.01, 0.0025 and 0.001 and train-sets were:

1) Standardized set, using all the features
2) Standardized set, using only the features *selected* by PCA (that is, indeed, a non-standard use of this technique)
3) The new set of features *produced* by PCA

Regarding test and train set:

- The train-set is the 10% of the Kddcup dataset (which is much much lower than the "standard" dimension, which usually is around 70/80%)
- The test-set is the *whole* Kddcup dataset, consisting of 42 features with almost 5 millions values
- When the PCA train-set was used, the test-set was transformed accordingly to the transformations done on the test-set (remember we had to "fit" the scaler with pandas?)

Results and some of the plots are shown in the next figures:

| Complexity parameter | 0.01 | 0.0025 | 0.001 |
|---|---|---|---|
| StandardizedKddcup | 0.9870385 | 0.9832191 | 0.9843356 |
| KddcupPrincipalFeatures | 0.9849946 | 0.9948412 | 0.9963878 |
| PCA eigenvectors | 0.9901552 | 0.9951242 | 0.9956888 |

If we observe data we can see how Principal Component Analysis raises the accuracy of our prediction. Moreover, the trees built using the selected components are much more balanced when we increase the complexity parameter (i.e. we build bigger trees), while using all the features we get an unbalanced tree.

Another thing to point out is that, with a c.p. of 0.001 we get more accuracy if we don't use the new features produced by PCA (but this could be a matter of the dimension of the train-set)

To conclude, it has been observed that PCA is indeed a useful technique to improve the overall accuracy of predictions in Decision Trees (and it also fasten execution time since we have less features!). For future works, considerations may be done regarding Random Forest (i.e. Distributed IDS) and how to combine it with PCA.

In the following pages there are some plots of the various Decision Tree in order to appreciate the differences between the ones that used PCA and the ones that didn't.

Figure 1 - This is the tree built with the whole features-set, with a c.p. = 0.0025. As can be seen it is very unbalanced. The more complex is the model, the more unbalanced is the tree

Figure 2 - This is the tree built using the new features produced by PCA. Compared to the previous one, it is much more balanced (and accurate!)

Figure 3 - Just for some fun, this is the (very big) tree, built using PCA's features with a c.p. = 0.001

Figure 4 - And here it is the DT built with the whole feature-set and a c.p. = 0.001. As can be seen, things exhacerbate when we try to build complex models, while using PCA we get better data structures

Figure 1

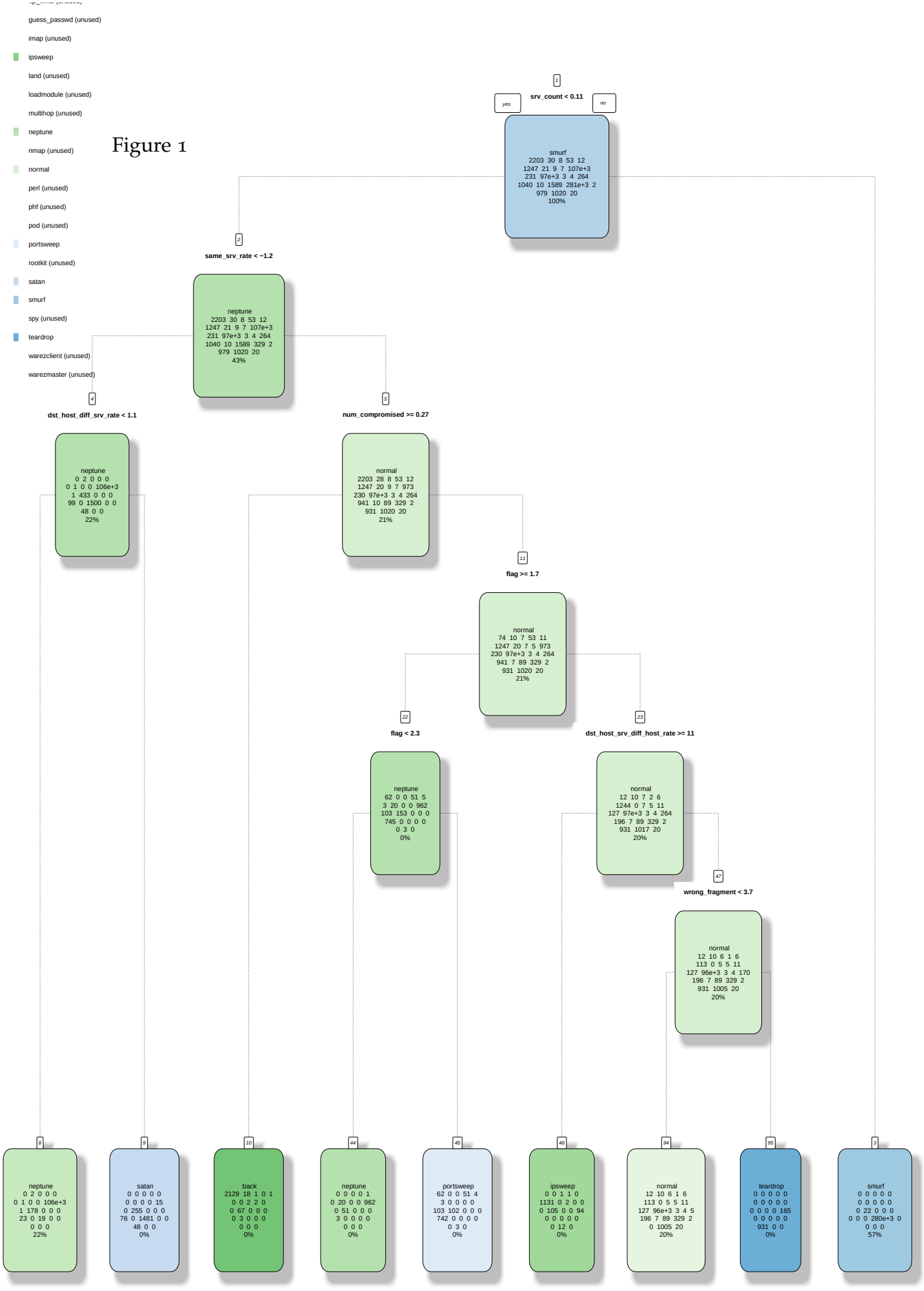

guess_passwd (unused)
imap (unused)
ipsweep
land (unused)
loadmodule (unused)
multihop (unused)
neptune
nmap (unused)
normal
perl (unused)
phf (unused)
pod (unused)
portsweep
rootkit (unused)
satan
smurf
spy (unused)
teardrop
warezclient (unused)
warezmaster (unused)
1
srv_count < 0.11
yes
no
smurf
2203 30 8 53 12
1247 21 9 7 107e+3
231 97e+3 3 4 264
1040 10 1589 281e+3 2
979 1020 20
100%
2
same_srv_rate < −1.2
neptune
2203 30 8 53 12
1247 21 9 7 107e+3
231 97e+3 3 4 264
1040 10 1589 329 2
979 1020 20
43%
4
dst_host_diff_srv_rate < 1.1
5
num_compromised >= 0.27
neptune
0 2 0 0 0
0 1 0 0 106e+3
1 433 0 0 0
99 0 1500 0 0
48 0 0
22%
normal
2203 28 8 53 12
1247 20 9 7 973
230 97e+3 3 4 264
941 10 89 329 2
931 1020 20
21%
11
flag >= 1.7
normal
74 10 7 53 11
1247 20 7 5 973
230 97e+3 3 4 264
941 7 89 329 2
931 1020 20
21%
22
flag < 2.3
23
dst_host_srv_diff_host_rate >= 11
neptune
62 0 0 51 5
3 20 0 0 962
103 153 0 0 0
745 0 0 0 0
0 3 0
0%
normal
12 10 7 2 6
1244 0 7 5 11
127 97e+3 3 4 264
196 7 89 329 2
931 1017 20
20%
47
wrong_fragment < 3.7
normal
12 10 6 1 6
113 0 5 5 11
127 96e+3 3 4 170
196 7 89 329 2
931 1005 20
20%
8
9
10
44
45
46
94
95
3
neptune
0 2 0 0 0
0 1 0 0 106e+3
1 178 0 0 0
23 0 19 0 0
0 0 0
22%
satan
0 0 0 0 0
0 0 0 0 15
0 255 0 0 0
76 0 1481 0 0
48 0 0
0%
back
2129 18 1 0 1
0 0 2 2 0
0 67 0 0 0
0 3 0 0 0
0 0 0
0%
neptune
0 0 0 0 1
0 20 0 0 962
0 51 0 0 0
3 0 0 0 0
0 0 0
0%
portsweep
62 0 0 51 4
3 0 0 0 0
103 102 0 0 0
742 0 0 0 0
0 3 0
0%
ipsweep
0 0 1 1 0
1131 0 2 0 0
0 105 0 0 94
0 0 0 0 0
0 12 0
0%
normal
12 10 6 1 6
113 0 5 5 11
127 96e+3 3 4 5
196 7 89 329 2
0 1005 20
20%
teardrop
0 0 0 0 0
0 0 0 0 0
0 0 0 0 165
0 0 0 0 0
931 0 0
0%
smurf
0 0 0 0 0
0 0 0 0 0
0 22 0 0 0
0 0 0 280e+3 0
0 0 0
57%

Figure 2

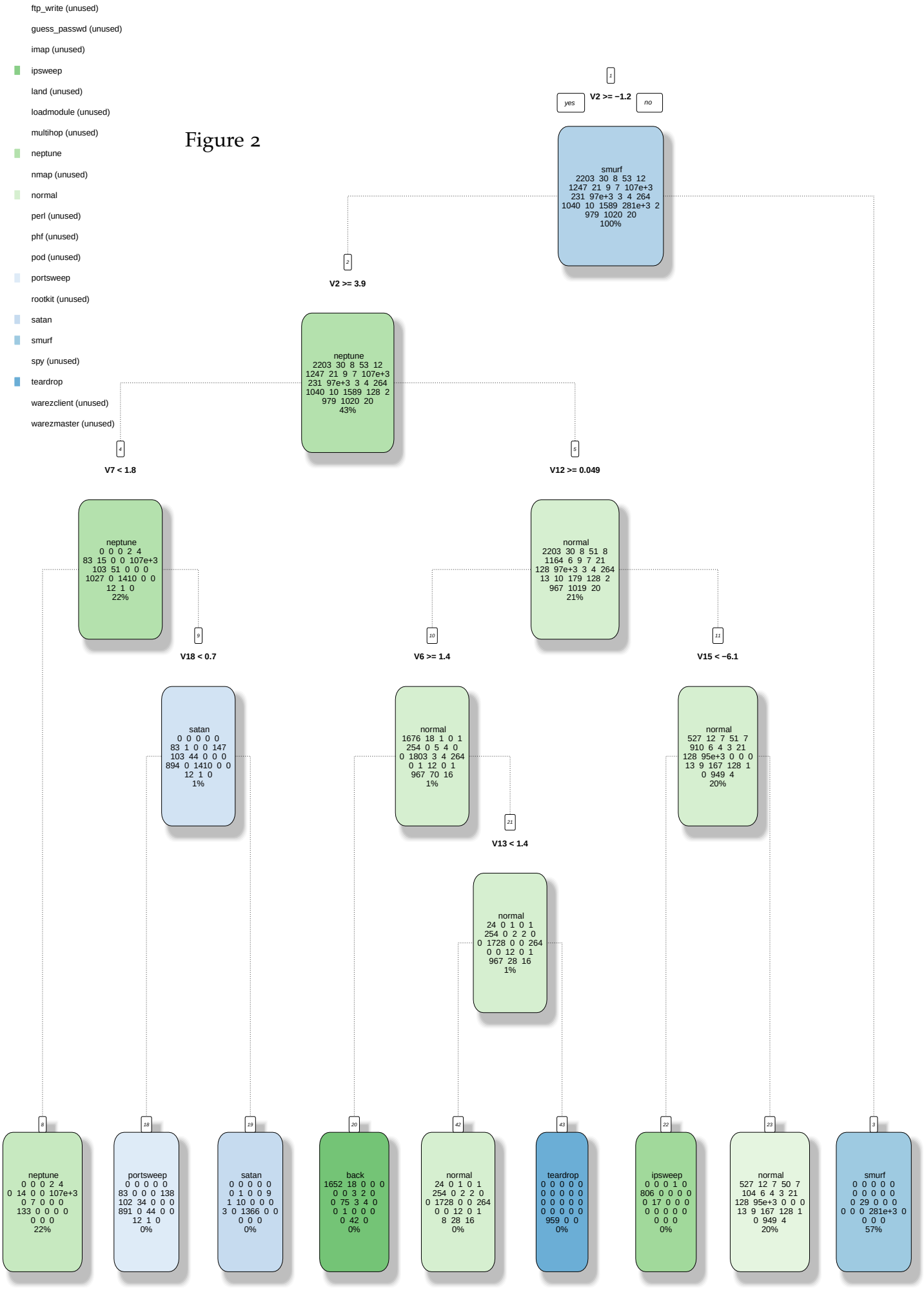

ftp_write (unused)
guess_passwd (unused)
imap (unused)
ipsweep
land (unused)
loadmodule (unused)
multihop (unused)
neptune
nmap (unused)
normal
perl (unused)
phf (unused)
pod (unused)
portsweep
rootkit (unused)
satan
smurf
spy (unused)
teardrop
warezclient (unused)
warezmaster (unused)
1
yes
V2 >= −1.2
no
smurf
2203 30 8 53 12
1247 21 9 7 107e+3
231 97e+3 3 4 264
1040 10 1589 281e+3 2
979 1020 20
100%
2
V2 >= 3.9
neptune
2203 30 8 53 12
1247 21 9 7 107e+3
231 97e+3 3 4 264
1040 10 1589 128 2
979 1020 20
43%
4
V7 < 1.8
5
V12 >= 0.049
neptune
0 0 0 2 4
83 15 0 0 107e+3
103 51 0 0 0
1027 0 1410 0 0
12 1 0
22%
9
V18 < 0.7
normal
2203 30 8 51 8
1164 6 9 7 21
128 97e+3 3 4 264
13 10 179 128 2
967 1019 20
21%
10
V6 >= 1.4
11
V15 < −6.1
satan
0 0 0 0 0
83 1 0 0 147
103 44 0 0 0
894 0 1410 0 0
12 1 0
1%
normal
1676 18 1 0 1
254 0 5 4 0
0 1803 3 4 264
0 1 12 0 1
967 70 16
1%
21
V13 < 1.4
normal
527 12 7 51 7
910 6 4 3 21
128 95e+3 0 0 0
13 9 167 128 1
0 949 4
20%
normal
24 0 1 0 1
254 0 2 2 0
0 1728 0 0 264
0 0 12 0 1
967 28 16
1%
8
neptune
0 0 0 2 4
0 14 0 0 107e+3
0 7 0 0 0
133 0 0 0 0
0 0 0
22%
18
portsweep
0 0 0 0 0
83 0 0 0 138
102 34 0 0 0
891 0 44 0 0
12 1 0
0%
19
satan
0 0 0 0 0
0 1 0 0 9
1 10 0 0 0
3 0 1366 0 0
0 0 0
0%
20
back
1652 18 0 0 0
0 0 3 2 0
0 75 3 4 0
0 1 0 0 0
0 42 0
0%
42
normal
24 0 1 0 1
254 0 2 2 0
0 1728 0 0 264
0 0 12 0 1
8 28 16
0%
43
teardrop
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
959 0 0
0%
22
ipsweep
0 0 0 1 0
806 0 0 0 0
0 17 0 0 0
0 0 0 0 0
0 0 0
0%
23
normal
527 12 7 50 7
104 6 4 3 21
128 95e+3 0 0 0
13 9 167 128 1
0 949 4
20%
3
smurf
0 0 0 0 0
0 0 0 0 0
0 29 0 0 0
0 0 0 281e+3 0
0 0 0
57%

Figure 3

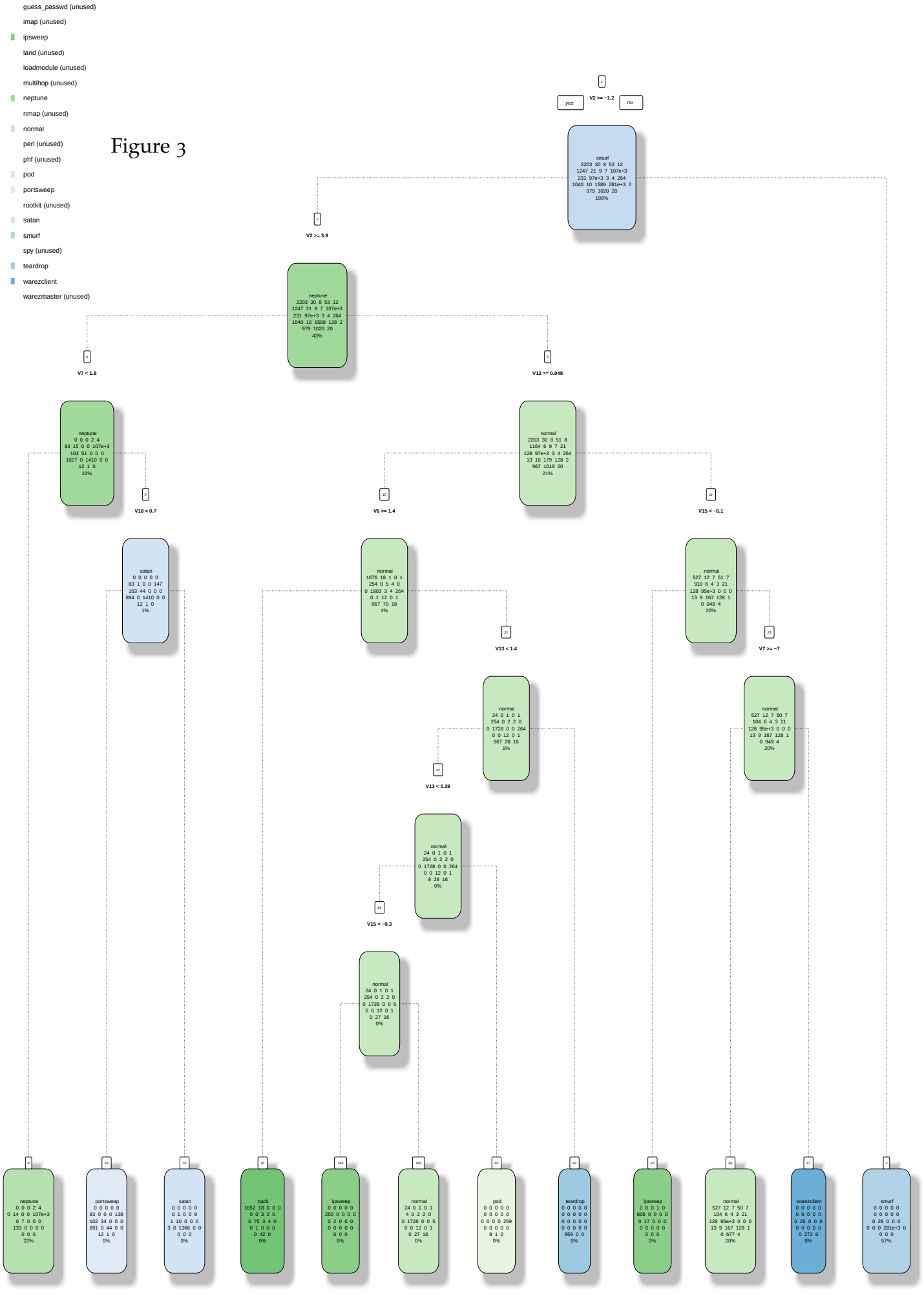
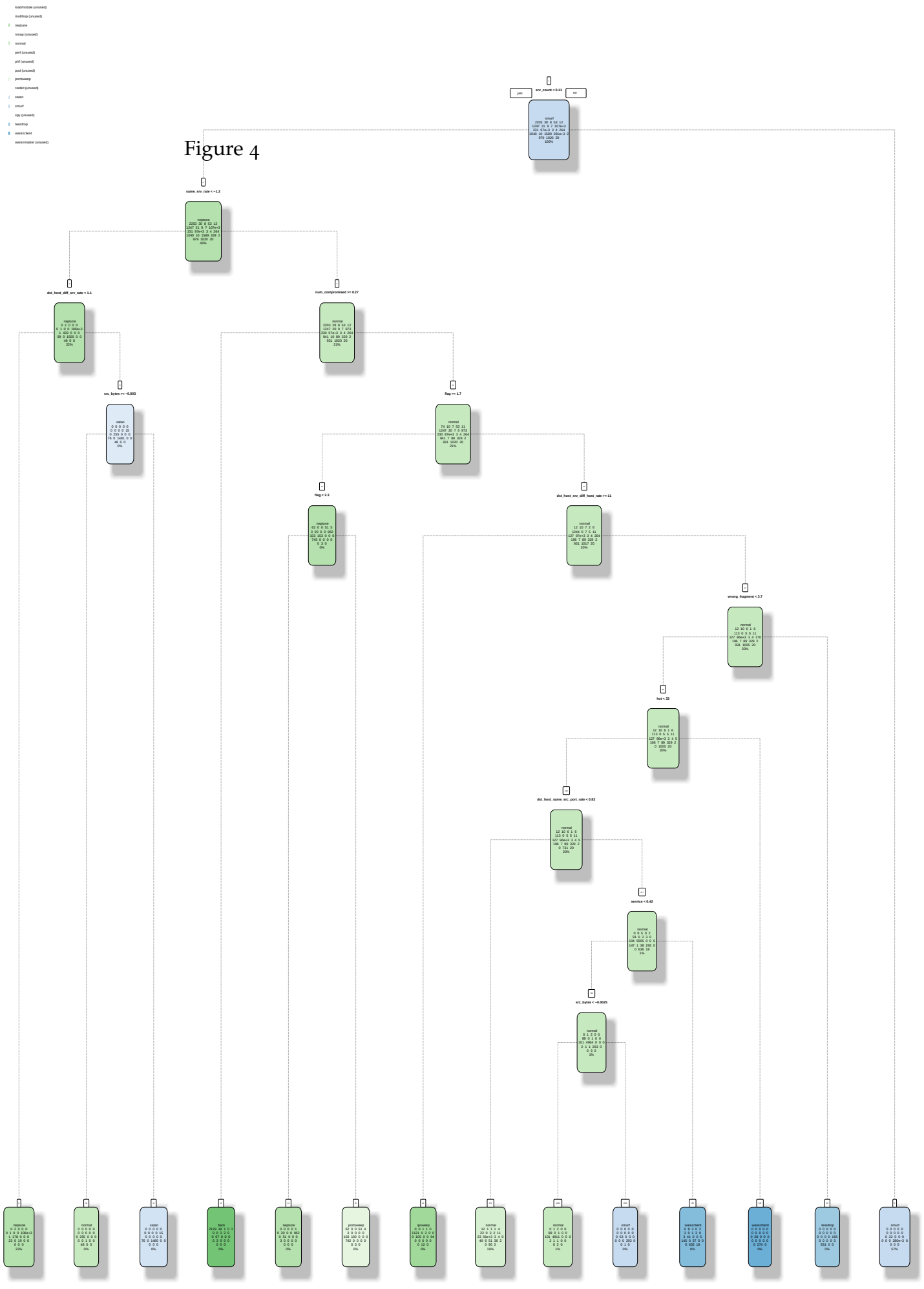

guess_passwd (unused)
imap (unused)
ipsweep
land (unused)
loadmodule (unused)
multihop (unused)
neptune
nmap (unused)
normal
perl (unused)
phf (unused)
pod
portsweep
rootkit (unused)
satan
smurf
spy (unused)
teardrop
warezclient
warezmaster (unused)
1
V2 >= −1.2
yes
no
smurf
2203 30 8 53 12
1247 21 9 7 107e+3
231 97e+3 3 4 264
1040 10 1589 281e+3 2
979 1020 20
100%
2
V2 >= 3.9
neptune
2203 30 8 53 12
1247 21 9 7 107e+3
231 97e+3 3 4 264
1040 10 1589 128 2
979 1020 20
43%
4
V7 < 1.8
9
V12 >= 0.049
neptune
0 0 0 2 4
83 15 0 0 107e+3
103 51 0 0 0
1027 0 1410 0 0
12 1 0
22%
8
V18 < 0.7
normal
2203 30 8 51 8
1164 6 9 7 21
128 97e+3 3 4 264
13 10 179 128 2
967 1019 20
21%
10
V6 >= 1.4
11
V15 < −6.1
satan
0 0 0 0 0
83 1 0 0 147
103 44 0 0 0
894 0 1410 0 0
12 1 0
1%
normal
1676 18 1 0 1
254 0 5 4 0
0 1803 3 4 264
0 1 12 0 1
967 70 16
1%
21
V13 < 1.4
normal
527 12 7 51 7
910 6 4 3 21
128 95e+3 0 0 0
13 9 167 128 1
0 949 4
20%
23
V7 >= −7
normal
24 0 1 0 1
254 0 2 2 0
0 1728 0 0 264
0 0 12 0 1
967 28 16
1%
42
V13 < 0.39
normal
527 12 7 50 7
104 6 4 3 21
128 95e+3 0 0 0
13 9 167 128 1
0 949 4
20%
normal
24 0 1 0 1
254 0 2 2 0
0 1728 0 0 264
0 0 12 0 1
8 28 16
0%
64
V15 < −9.3
normal
24 0 1 0 1
254 0 2 2 0
0 1728 0 0 5
0 0 12 0 1
0 27 16
0%
neptune
0 0 0 2 4
0 14 0 0 107e+3
0 7 0 0 0
133 0 0 0 0
0 0 0
22%
portsweep
0 0 0 0 0
83 0 0 0 138
102 34 0 0 0
891 0 44 0 0
12 1 0
0%
satan
0 0 0 0 0
0 1 0 0 9
1 10 0 0 0
3 0 1366 0 0
0 0 0
0%
back
1652 18 0 0 0
0 0 3 2 0
0 75 3 4 0
0 1 0 0 0
0 42 0
0%
ipsweep
0 0 0 0 0
250 0 0 0 0
0 2 0 0 0
0 0 0 0 0
0 0 0
0%
normal
24 0 1 0 1
4 0 2 2 0
0 1726 0 0 5
0 0 12 0 1
0 27 16
0%
pod
0 0 0 0 0
0 0 0 0 0
0 0 0 0 259
0 0 0 0 0
8 1 0
0%
teardrop
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
959 0 0
0%
ipsweep
0 0 0 1 0
806 0 0 0 0
0 17 0 0 0
0 0 0 0 0
0 0 0
0%
normal
527 12 7 50 7
104 6 4 3 21
128 95e+3 0 0 0
13 9 167 128 1
0 677 4
20%
warezclient
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 25 0 0 0
0 0 0 0 0
0 272 0
0%
smurf
0 0 0 0 0
0 0 0 0 0
0 29 0 0 0
0 0 0 281e+3 0
0 0 0
57%

Figure 4


loadmodule (unused)
multihop (unused)
neptune
nmap (unused)
normal
perl (unused)
phf (unused)
pod (unused)
portsweep
rootkit (unused)
satan
smurf
spy (unused)
teardrop
warezclient
warezmaster (unused)
srv_count < 0.11
yes
no
same_srv_rate < -1.2
dst_host_diff_srv_rate < 1.1
num_compromised >= 0.27
src_bytes >= -0.003
flag >= 1.7
flag < 2.3
dst_host_srv_diff_host_rate >= 11
wrong_fragment < 3.7
host < 33
dst_host_same_src_port_rate < 0.02
service < 0.42
src_bytes < -0.0025

# BIBLIOGRAFIA

[1] Saroj Kr. Biswas - *Intrusion Detection Using Machine Learning: A Comparison Study*

[2] Kai Peng, Victor C. M. Leung, Lixin Zheng, Shangguang Wang, Chao Huang, Tao Lin - *Intrusion Detection System Based on Decision Tree over Big Data in Fog Environment*