Intrusion Detection Systems

A machine learning approach for computer security

Francesco Terrosi

Università degli studi di Firenze

20 febbraio 2019

Table of content

INTRODUCTION

- Nowadays, we are seeing an increasing reliance on Artificial Intelligence
- Combined with the increasing knowledge on statistical techniques, such as Multivariate Analysis and Statistical Learning (which are translated in Computer Science as "Machine Learning") these tools are even more powerful
- Statistical Learning and Computer Security together to build a system capable of detecting users' and programs' malicious behaviours (e.g. an antivirus that detect malicious programs in a system)

- Our focus will be on the preprocessing and the analysis of data, in order to build such a system
- We need to classify users' (and programs') behaviours and to classify them in order to detect suspicious activities
- To do so, the Kddcup99 was used. It is a well known dataset in this field, consisting of web activities observed through a couple of months by a military company
- The dataset is *really* huge. This was the cause of some computational limitation for the research but, as you will see, good results have been achieved anyway
- The dataset also offer a set consisting of a non-contiguous 10% of the original data, that was used for the training phase

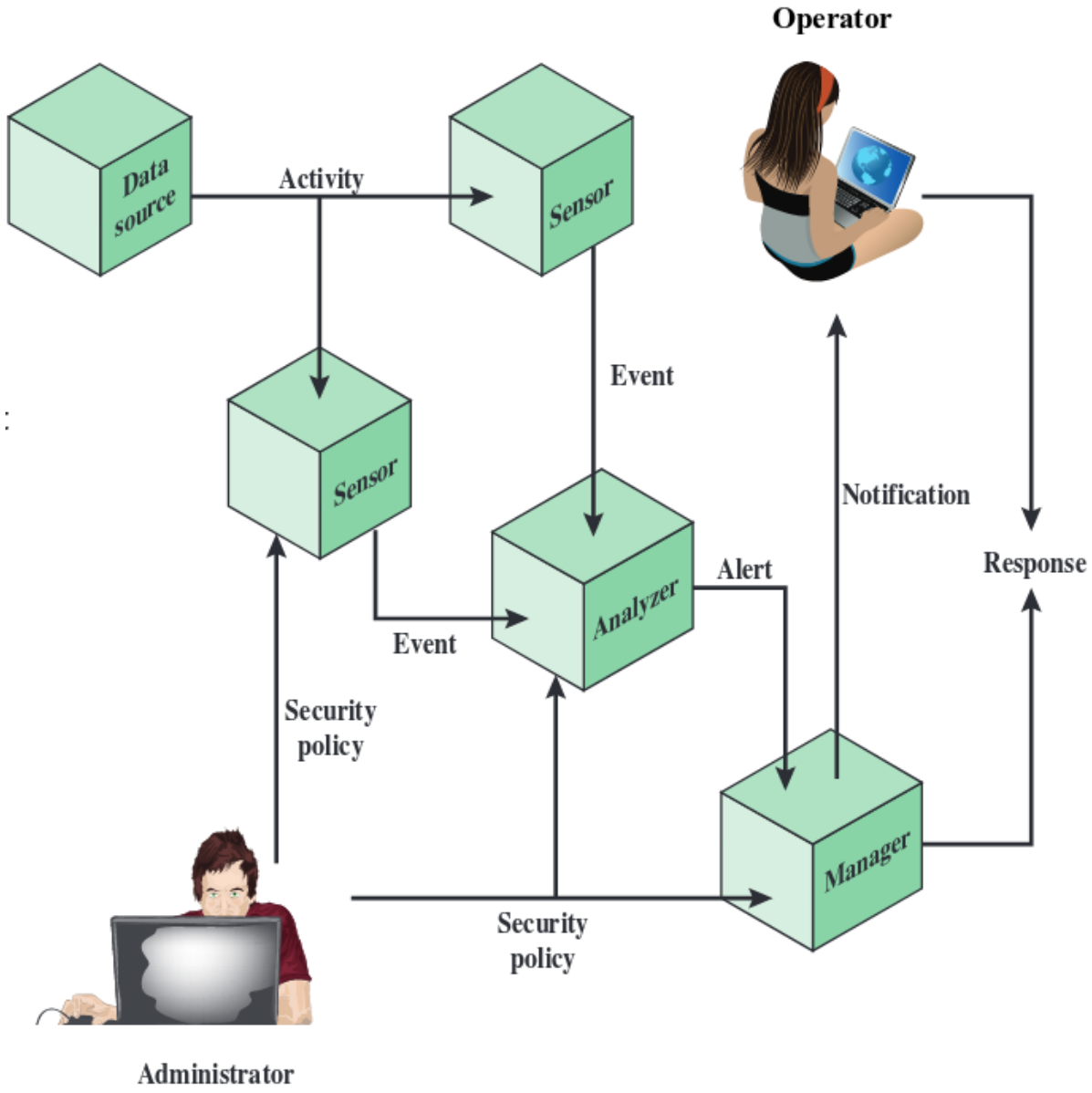
INTRUSION DETECTION SYSTEMS

Intrusion Detection System

An IDS is a hardware/software module capable of detecting *intrusions* in a computer system by analyzing information from various areas within a computer or a network

Intrusion

An *intrusion* is the *unauthorized* act of bypassing the security measures of a system


Operator
Data source
Activity
Sensor
Sensor
Event
Notification
Response
Alert
Analyzer
Event
Security policy
Manager
Security policy
Administrator

Intrusion Detection System can be characterized on the basis of 2 aspects: the *basing* of the system:

- Host-Based IDS
- Network-Based IDS
- Hybrid (Distributed) IDS

and on the techniques used for *detection*:

- Signature-Based IDS
  - $\rightarrow$ We define a set of *rules* that captures malicious behaviours
- Anomaly-Based IDS
  - $\rightarrow$ We define the *normal behaviours* and measure how close are the observed activities with respect to the model produced

Can you see where Statistical Learning comes in? :)

Anomaly-Based IDS can use techniques based on Multivariate Analysis or Statistical Learning to develop the model of users' *normal behaviour*

Decision Trees are a good candidate for predictions since empirical results showed that this technique is one of the most accurate for intrusion detection

Of course there are some drawbacks as:

- Data Gathering
  - → If you're not willing to use Stateful Protocol Analysis (i.e. you buy data from authorized vendors in order to build the model), you need sensors for data acquisition and you have to do some preprocessing
- False-Positive/False-Negative rates
  - → Since intrusions are considered *rare* (but often *catastrophic*) events, we want to achieve the maximum accuracy possible

ANALYZER IMPLEMENTATION

- An *analyzer* for IDSs is a (typically) software module that collects data and raises an alarm if an intrusion is detected
- In this work, an hybrid Python/R software was developed:
  1) All the Preprocessing was done in Python for comodity (I'm quite familiar with it) and for the numbers of libraries available for Machine Learning
  2) The Decision Tree was built with R, using the rpart package
  3) Since R can work as a server to communicate with other process, there's no drawback on splitting modules using different languages (actually, this is often a requirement in software engineering!)
- Data Analysis is a critical factor for achieving an high level of accuracy, thus multiple techniques were used for the process

Features' values in the dataset consist of heterogeneous types: we have integers, floats, strings...

The first step is to convert all the values to a numeric type in order to proceed with the analysis! This is a very simple procedure an can be implemented straightforward by checking all the strings in the dataset

```
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
```

The next step is features' standardization.

This step is mandatory since Principal Component Analysis was used as a method for features' selection:

If data are not standardized (i.e. scaled in order to associate to each feature a distribution with mean = 0 and variance = 1), there will be some features that appears to be the most relevant only because their values are of different scales

(e.g. error rates, or rates in general usually shrink [0,1], while durations, number of requests etc are positive natural numbers)

- × The very important thing in this step is that we need to standardize both the test and the train set, which consists of different features' values, using the same criteria
- ★ Luckily for us, there is a Python library that implements all these operations for us: the Pandas library

```
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)
```

After data have been standardized, we need to select only the relevant features in the dataset (since it consists of 40 variables!)

To do so, a Principal Component Analysis was conducted on the dataset.

PCA

- It is a technique that uses algebra calculations to reduce the number of variables in a dataset
- First of all, the variance-covariance matrix V must be computed
- Then, we need to calculate its eigenvectors and eigenvalues, that translates into solving the equation: $det(V - \lambda I) = 0$
- These vectors represent "new" features that explain the variance of the variables in the dataset
- You then choose the number of components (the vectors) you need to preserve the accuracy of the representation of the original dataset

Here we have the same problem we had with standardization: if we perform some transformation on the train-set, the *same* transformation must be done on the test-set and on the real data you will gather from sensors.

Luckily, again, the Pandas library comes in help:

```
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 are cleaned, standardized and the principal components were calculated, we can start building the Decision Tree!

Decision Trees are statistical models that can be used for classification and the analysis of variance.

In this research we are most interested in classification since we want to classify *bad* and *good* users. In the next block is briefly described the algorithm to build a DT

Decision Tree Construction

- For each variable we split the dataset in 2 groups (e.g. using the Gini Index metric)
  - $\rightarrow$ Since our variables are continuous, splits will be done on boolean conditions, that is: given a threshold t, we will check the features whose value is greater than t and the ones lower than t
- The variable (feature) that produced the "best" score will be the very first split (i.e. the root of the tree)
- The algorithm now proceeds recursively on each node, until a certain alt condition is met (e.g. all the variables have been chosen for splitting the dataset)

The DT was built using R, since it is faster than Python and, if our train-set is about 200 MB, our test-set is 2 GB, so considerations on performances led me thowards R

Decision Trees can be built using the package *rpart*. It offers many parameters to tell the algoirthm how to build it such as the variables that need to be considered and the complexity parameter (i.e. the tree *depth*)

It also provides a function for predictions over a dataset. The output of this predictions are used to compute the *Confusion Matrix*

	Actual values	
Predictions	True	False
True	True Positive	False Positive
False	False Negative	True Negative

Simple 2x2 confusion matrix example

The confusion matrix can be used to calculate the accuracy of predictions over a dataset. The formula is quite straightforward and is shown, together with the tree construction, in the figure below:

```
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
```

RESULTS & CONCLUSIONS

The Decision Tree was built using different three different representation of the same train-set.

The Kddcup99 dataset provides a file which consists of a 10% of the original dataset (the rows in this smaller dataset are not contiguous in the original dataset)

For technical reasons, this small file was used as a train-set (usually train-sets are about the 70/80% of the original file)

For complexity parameters equal to 0.01, 0.0025 and 0.001, the algorithm was given as input:

1) The preprocessed dataset (i.e. strings are replaced and values are standardized, all the features in input)
2) The features *selected* by PCA (this is, indeed, a non-standard use of PCA)
3) The new features *produced* by PCA

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

Standardized Kddcup

- The Decision Tree ran over the whole set of features, is the one with the lowest accuracy
- This Tree is also very susceptible to the complexity parameter: bigger trees mean lower accuracy (as you could see from the previous table)
- Trees built with this dataset are also very much unbalanced with respect to the others

Restricted Feature Set Standardized Kddcup

- This train-set consist of the features chosen by the PCA, but the values used are the ones of the *original* dataset
- Despite the fact that this is not the standard usage of PCA, these trees show pretty good accuracy (actually, greater complexity lead to higher accuracy with respect to PCA trees)
- However, this could be a consequence of using only the 10% of the original dataset as a train-set
- Regarding the Tree Structure, these ones are very similar to the PCA trees

PCA features

- The last train-set is the output of the PCA algorithm
- The drawback of this approach is that, since we changed the original features' set, we have to transform the test-set, as well as any other value that will be used for future predictions (remember we had to *fit* the PCA object in Pandas?)
- The accuracy of these trees is similar to the previous one, but the gain in accuracy is lower when we increase the complexity (i.e. decrease the complexity parameter)
- Just like the previous trees, these ones are very well balanced and do not suffer bigger structures

TREE PLOTS

THANKS FOR YOUR ATTENTION