Project Report

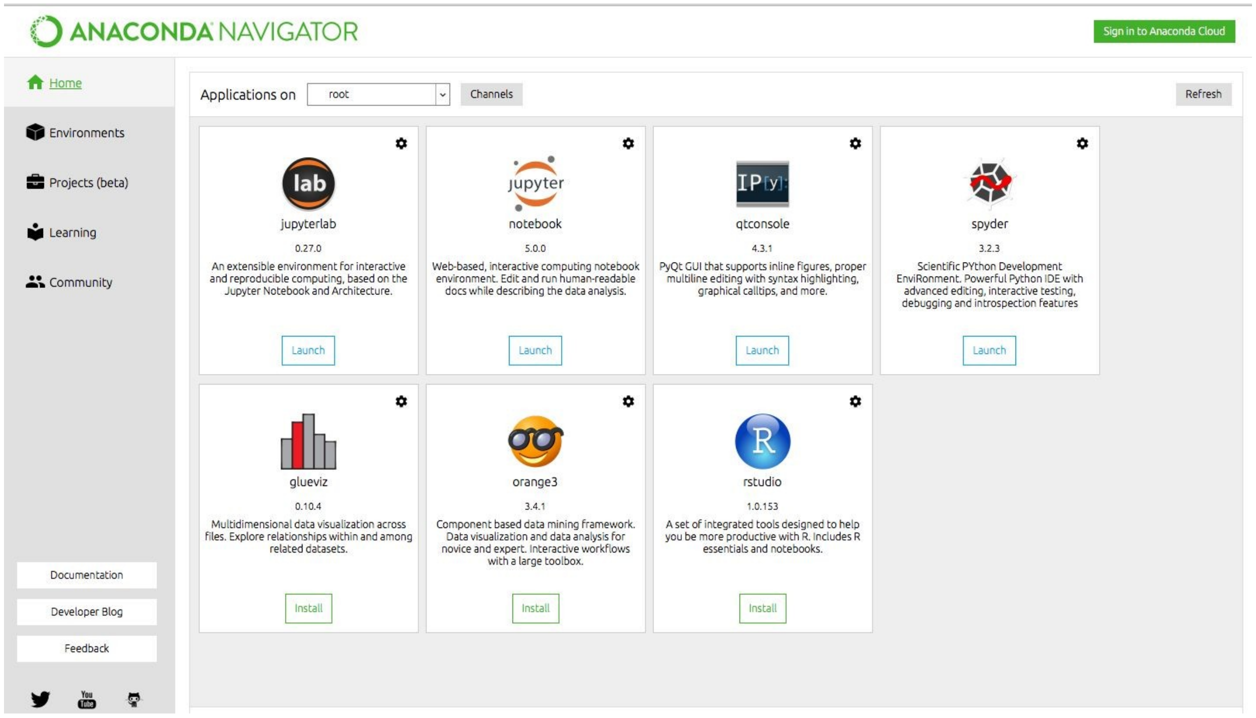
Data Mining For Networks

Prepared by: Abdelkarim HAFID : Ubinetter

Supervised by: Frederic GIROIRE

0. Prerequisites

To carry out this project I installed Anaconda. Which is an ecosystem of python. It contains the main tools I'm going to need in this project. In particular I'm going to use Jupyter to ease the maintenance and evolution of our codebase.



jupyterlab
0.27.0
An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.

notebook
5.0.0
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

qtconsole
4.3.1
PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

spyder
3.2.3
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features

glueviz
0.10.4
Multidimensional data visualization across files. Explore relationships within and among related datasets.

orange3
3.4.1
Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.

rstudio
1.0.153
A set of integrated tools designed to help you be more productive with R. Includes R essentials and notebooks.

1. Reading the paper

Karagiannis, T., Papagiannaki, K., Taft, N., and Faloutsos, M. (2007, April). Proling the end host. In International Conference on Passive and Active Network Measurement (PAM) (pp. 186-196). Springer, Berlin, Heidelberg.

As was required, I read the paper presenting end host profiling and the following are the main ideas at the core of this paper.

Profiling end hosts is:

- Used for diagnosis and security reasons
- design general end host profiles capable of capturing and representing a broad range of user activity/behaviour.

2. Understanding the paper

Profiling a behaviour means: observing measured data and extracting information which is representative of the behaviour or usage patterns.
They try to formalize the concept of profiling transport layer information and identify desirable properties.
Profiling mechanisms should meet the following goals:

- Be able to identify dominant and persistent behaviours of the end-system (repeatable behaviour over time)
- It should be a compact enough representation
- It should be stable over short time scales avoiding transient variability in the host behaviour.
- It should evolve by adding new behaviours and removing stale ones.
- It should be able to capture historical information to illustrate typical ranges of values for features.

Approach:

- Use graph based structure which they call graphlets; to capture the interactions among the transport layer protocols, the @IPs and port numbers.
- Two step method based on unsupervised learning: 1) Build & continuously update activity graphlets 2) Compress the large activity graphlet -> evolve the detained summary to reflect change over time.

Methodology :

1) Capturing host activity via graphs:
- Graphlet = graph arranged in six columns (@IPsrc, protocol, @IPdst, PORTsrc, PORTdst, @IPdst)
- Graphlet node = distinct entity from the set of possible entities of the corresponding column.
- Lines connecting nodes imply that there exists at least one flow whose packets contain the specific nodes.
- A flow creates a directed graph and might represent numerous packets.
- Destination @IP is redundant to observe all pairwise interactions between the most information-heavy fields of the 5-tuple header.
- In-degree (Out-degree) = number of edges on the left(right) side of the node.

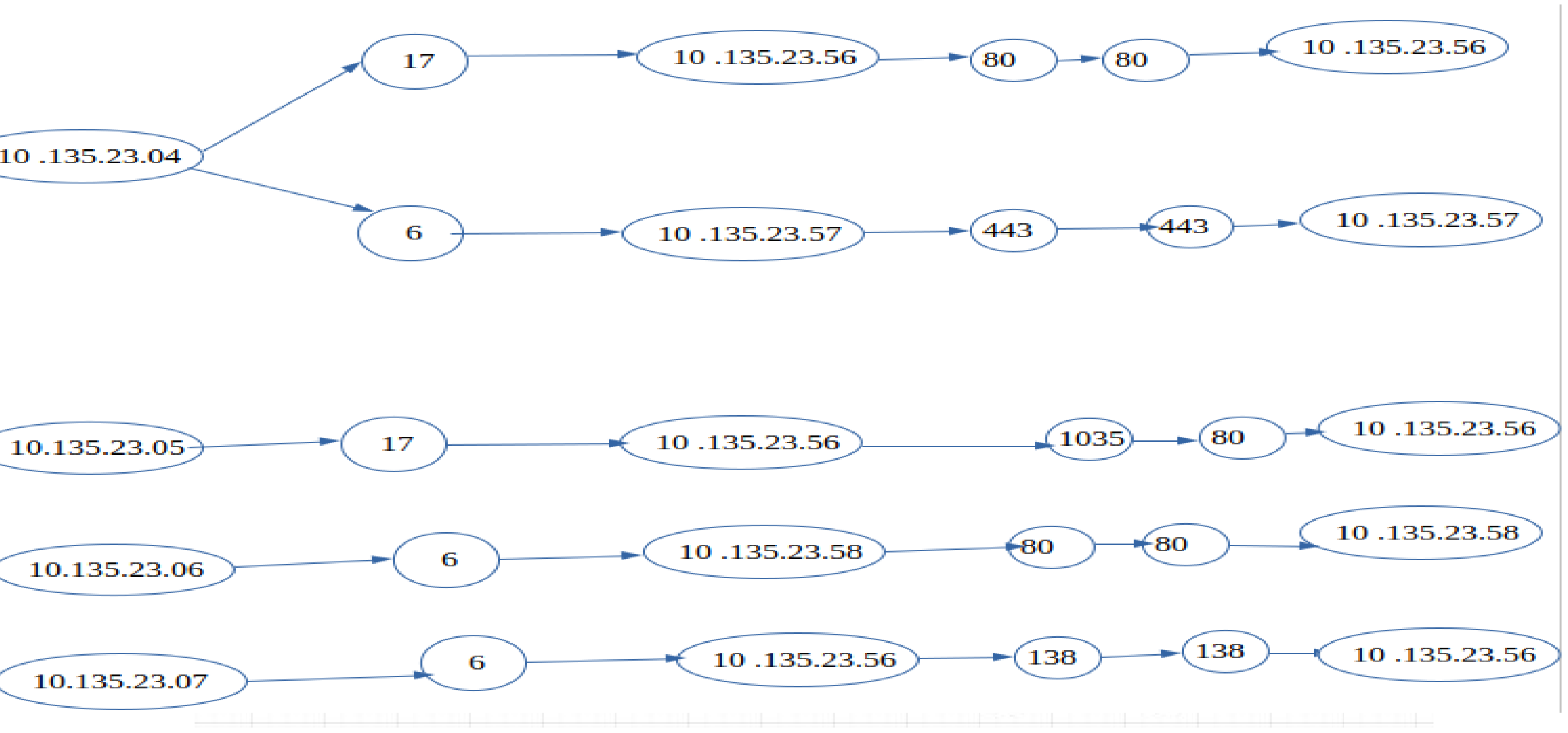
2) Advantages of graphlet profiling:
- Scanning behaviour can be easily noticed + scan space easily detected by out degree.

3) Building profiles:
- They define methods to convert activity graphlet into profile graphlet via policies of compression and adaptivity.

3. Building the graphlet

Now I'm going to showcase my understanding of the concept of graphlets through the following graphlet that corresponds to the pattern shown below:

format: srcIP protocol dstIP   sPort dPort
1O.135.23.04 17 10.135.23.56 80 80
10.135.23.05 17 10.135.23.56 1035 80
10.135.23.04 6   10.135.23.57 443 443
10.135.23.06 6   10.135.23.58 80 80
10.135.23.07 6   10.135.23.56 138 138



4. How to build model

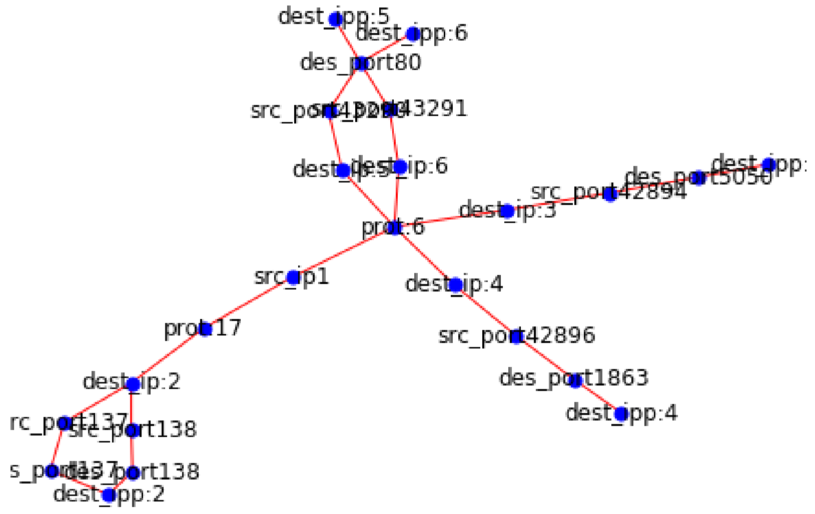
Now I’m going to use these concepts as a leverage to detect malicious traffic within a none annotated file given in the project. But first let's try to build the Activity graphlet and the Profile graphlet Of one Source address IP nammed “1” with the correspondant flows that I put in the data.csv file that you can find zipped with this Report. Then, I generalized the building of the Activity and the Profile graphlets for all the end hosts of the annotated-trace.txt file .

The first code and the execution shows the activity graphlet and the profile graphlet of one source IP: “1” given In the data.csv file with the correspondent flows to this address IP .
I benefit from the algorithm1 described in the Paper to build the Profile Graphlet of the annotated-trace.txt file .
As in the second picture below, the class Profile try to isolate the significant flows from the non-significant flows To draw the Profile graphlet, because the profile graphlet contain only the significant flows and by significant flov I mean the flow that contain at least two significant nodes .

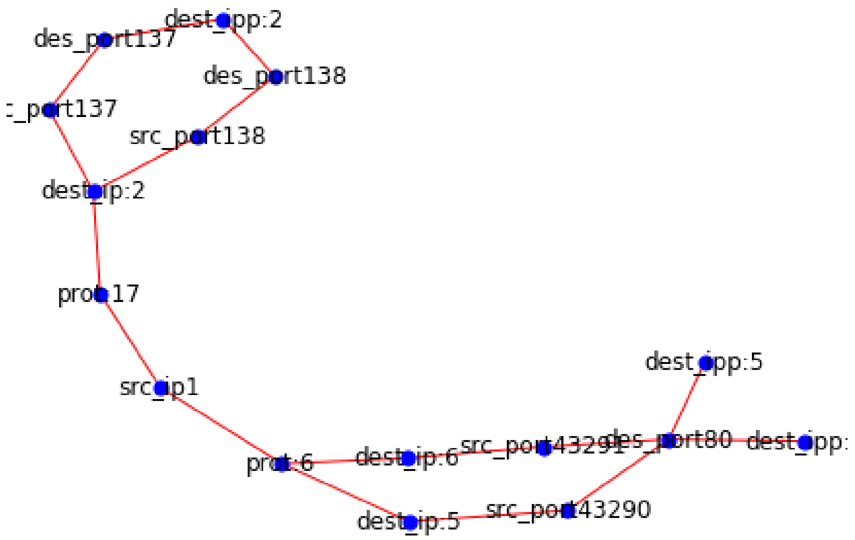
ntrée [2]:
```
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
Data = (pd.read_csv('data.csv',header=None)).iloc[:,0:5].drop_duplicates()
```

ntrée [3]:
```
def Activity(source):
    Specific_data = Data.loc[Data[0] == int(source),:]
    G=nx.Graph()
    G.add_nodes_from(["src_ip"+str(source)])
    for i in range(len(Specific_data)):
        G.add_nodes_from(["dest_ip:"+str(Specific_data.iloc[i,1]),"dest_ipp:"+str(Specific_data.iloc[i,1]),
                          "prot:"+str(Specific_data.iloc[i,2]),"src_port"+str(Specific_data.iloc[i,3]),
                          "des_port"+str(Specific_data.iloc[i,4])])
        G.add_edges_from([("src_ip"+str(source),"prot:"+str(Specific_data.iloc[i,2])),
                          ("prot:"+str(Specific_data.iloc[i,2]),"dest_ip:"+str(Specific_data.iloc[i,1])),
                          ("dest_ip:"+str(Specific_data.iloc[i,1]),"src_port"+str(Specific_data.iloc[i,3])),
                          ("src_port"+str(Specific_data.iloc[i,3]),"des_port"+str(Specific_data.iloc[i,4])),
                          ("des_port"+str(Specific_data.iloc[i,4]),"dest_ipp:"+str(Specific_data.iloc[i,1]))])
    return G
```

Entrée [4]:
```
def Profile(source):
    Specific_data = Data.loc[Data[0] == int(source),:]
    Specific_data.index = [i for i in range(len(Specific_data))]
    Significant_Flows=[]
    Non_Significant_Flows=[]
    for i in range(len(Specific_data)):
        for j in range(len(Specific_data)):
            if i==j:
                continue
            if Specific_data.iloc[i,1] == Specific_data.iloc[j,1] or Specific_data.iloc[i,3] == Specific_data.iloc[j
                Significant_Flows.append(i)
                break

    for i in Specific_data.index:
        if i not in Significant_Flows:
            Non_Significant_Flows.append(i)
    Specific_data = Specific_data.drop(Non_Significant_Flows)
    #Ploting the graph with networkx

    G=nx.Graph()
    G.add_nodes_from(["src_ip"+str(source)])
    for i in range(len(Specific_data)):
        G.add_nodes_from(["dest_ip:"+str(Specific_data.iloc[i,1]),"dest_ipp:"+str(Specific_data.iloc[i,1]),
                          "prot:"+str(Specific_data.iloc[i,2]),"src_port"+str(Specific_data.iloc[i,3]),
                          "des_port"+str(Specific_data.iloc[i,4])])
        G.add_edges_from([("src_ip"+str(source),"prot:"+str(Specific_data.iloc[i,2])),
                          ("prot:"+str(Specific_data.iloc[i,2]),"dest_ip:"+str(Specific_data.iloc[i,1])),
                          ("dest_ip:"+str(Specific_data.iloc[i,1]),"src_port"+str(Specific_data.iloc[i,3])),
                          ("src_port"+str(Specific_data.iloc[i,3]),"des_port"+str(Specific_data.iloc[i,4])),
                          ("des_port"+str(Specific_data.iloc[i,4]),"dest_ipp:"+str(Specific_data.iloc[i,1]))])
    return G
```



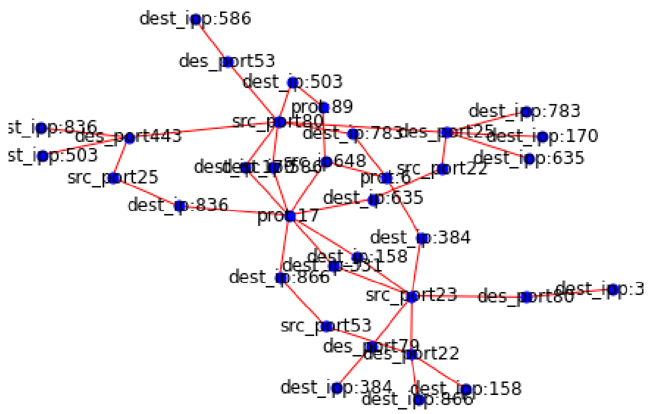
```
Entrée [6]: nx.draw(Profile(1), with_labels=True, node_color="b",edge_color="r",node_size=50)
```



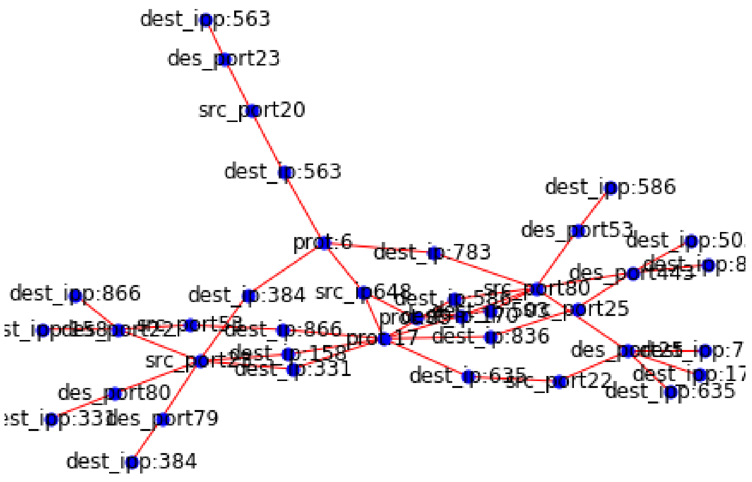
5. Building the end host graphlets

Now I'm going to change the dataset from data.csv to annotated-trace.txt and I will run the Functions Activity(source) And Profile(source) on the IP source addresses of the annotated-trace.txt like the First and the Second picture bellow show. Here I have tried the 648 like IP source Address and the results are given in the Two pictures screens.

```
Entrée [8]: nx.draw(Profile(648), with_labels=True, node_color="b",edge_color="r",node_size=50)
```

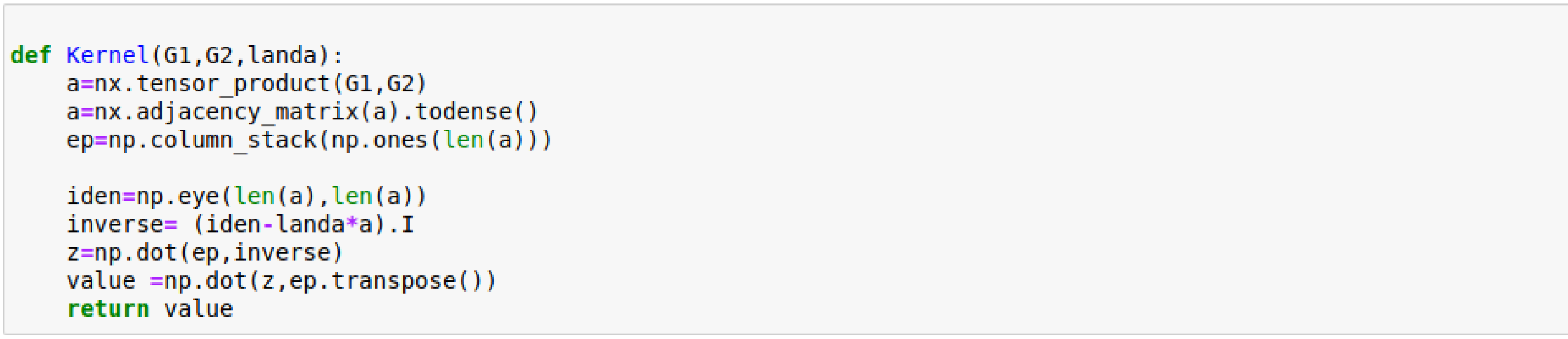


```
Entrée [10]: Data = (pd.read_csv('annotated-trace.csv',header=None)).iloc[:,0:5].drop_duplicates()
             nx.draw(Activity(648), with_labels=True, node_color="b",edge_color="r",node_size=50)
```



6. Building the kernel function

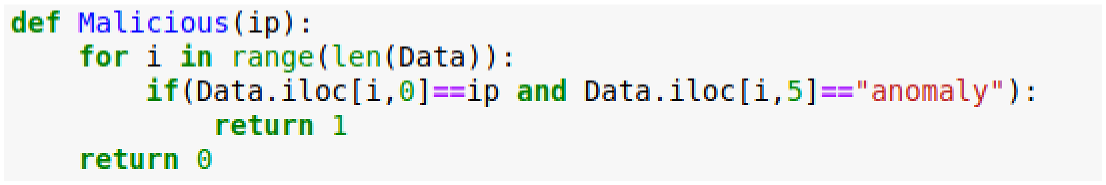
Now we get into the material we have gone through what we learned in class about random walk kernel. The following function takes as an input two graphs G1 and G2 and landa the decaying factor for the sum to converge that must be less than one .
Later, for me each address IpSource will represent one graph because for each address IPsource I will draw by the Profile() Function the correspondant graph. Then I will measure the similarities between the source IP addresses by calculating the kernel as the picture below show :



```
def Kernel(G1,G2,landa):
    a=nx.tensor_product(G1,G2)
    a=nx.adjacency_matrix(a).todense()
    ep=np.column_stack(np.ones(len(a)))

    iden=np.eye(len(a),len(a))
    inverse= (iden-landa*a).I
    z=np.dot(ep,inverse)
    value =np.dot(z,ep.transpose())
    return value
```

7. Use of SVM

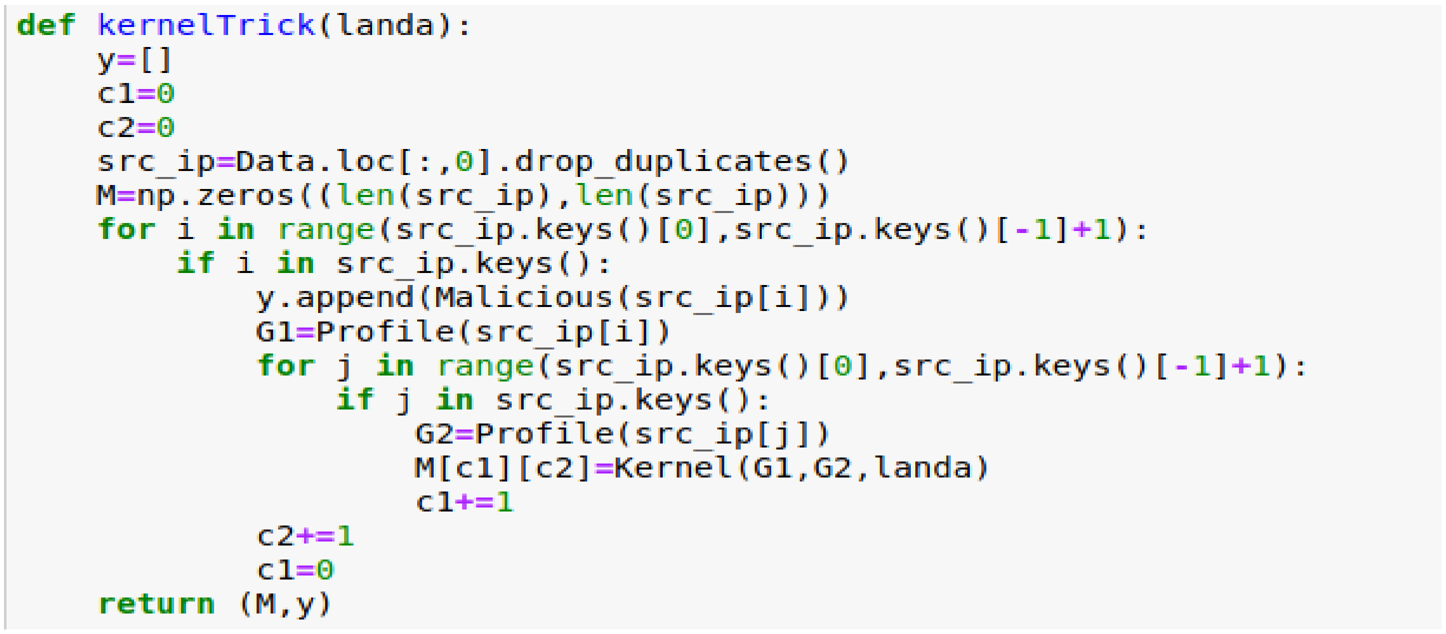
I'm going to try to use the Support Machine Vector classification algorithm to separate the normal activity, from the malicious one. I tried hardly to figure out a way to correspond the kernel values to the SVM fitting functions.
The challenge was however how to benefit from the kernel Matrice I built before to train the model And classify the IP sources to malicious and normal ones. So I created one function called Malicious to separate the malicious from the normal IP source addresses.



```
def Malicious(ip):
    for i in range(len(Data)):
        if(Data.iloc[i,0]==ip and Data.iloc[i,5]=="anomaly"):
            return 1
    return 0
```

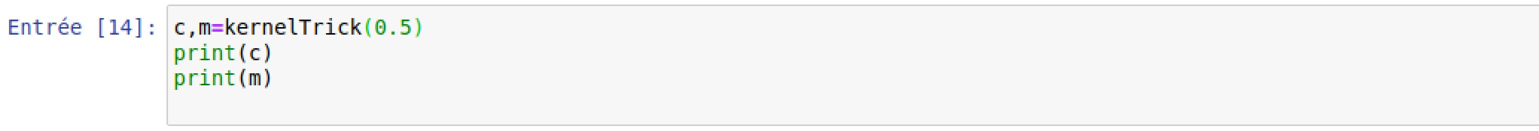
8. Kernel trick

In this Part, I will apply SVM on the annotated data directly to build the model. The following code gives one Function that calculate the kernel Trick of all the IP addresses. It takes as input the landa and it give us As output the Matrix M of all the kernels between the IP addresses and one target values on one vector y.

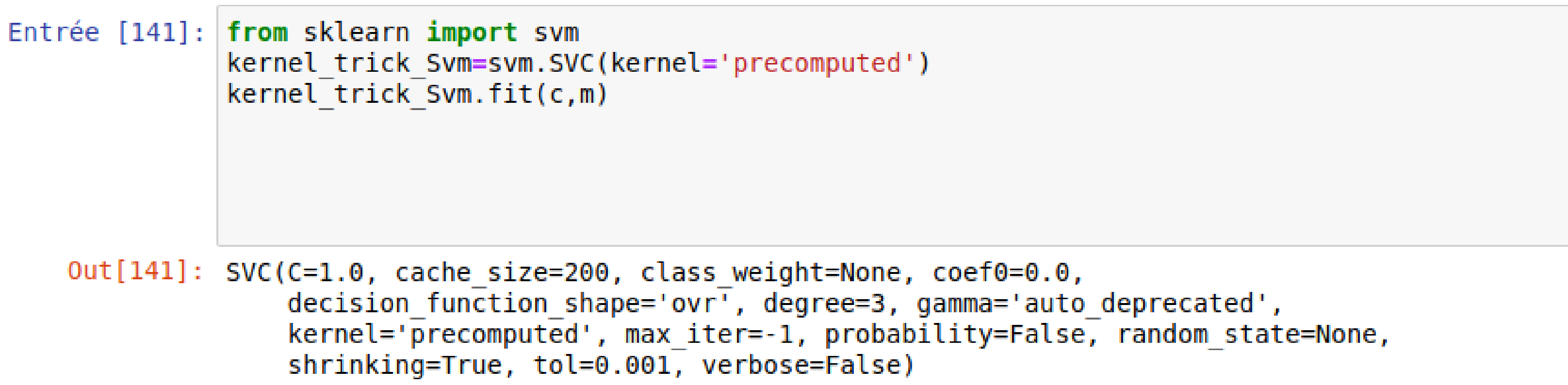


```
def kernelTrick(landa):
    y=[]
    c1=0
    c2=0
    src_ip=Data.loc[:,0].drop_duplicates()
    M=np.zeros((len(src_ip),len(src_ip)))
    for i in range(src_ip.keys()[0],src_ip.keys()[-1]+1):
        if i in src_ip.keys():
            y.append(Malicious(src_ip[i]))
            G1=Profile(src_ip[i])
            for j in range(src_ip.keys()[0],src_ip.keys()[-1]+1):
                if j in src_ip.keys():
                    G2=Profile(src_ip[j])
                    M[c1][c2]=Kernel(G1,G2,landa)
                    c1+=1
            c2+=1
            c1=0
    return (M,y)
```

Now I will call this function to run later the SVM function :

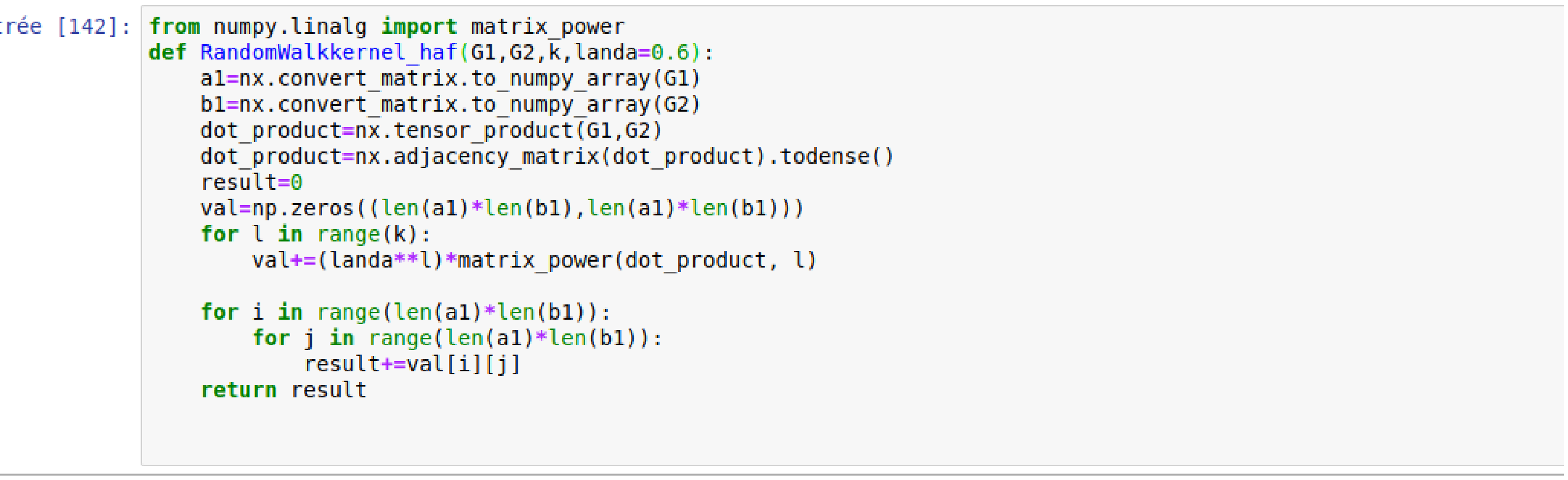


```
Entrée [14]: c,m=kernelTrick(0.5)
             print(c)
             print(m)
```

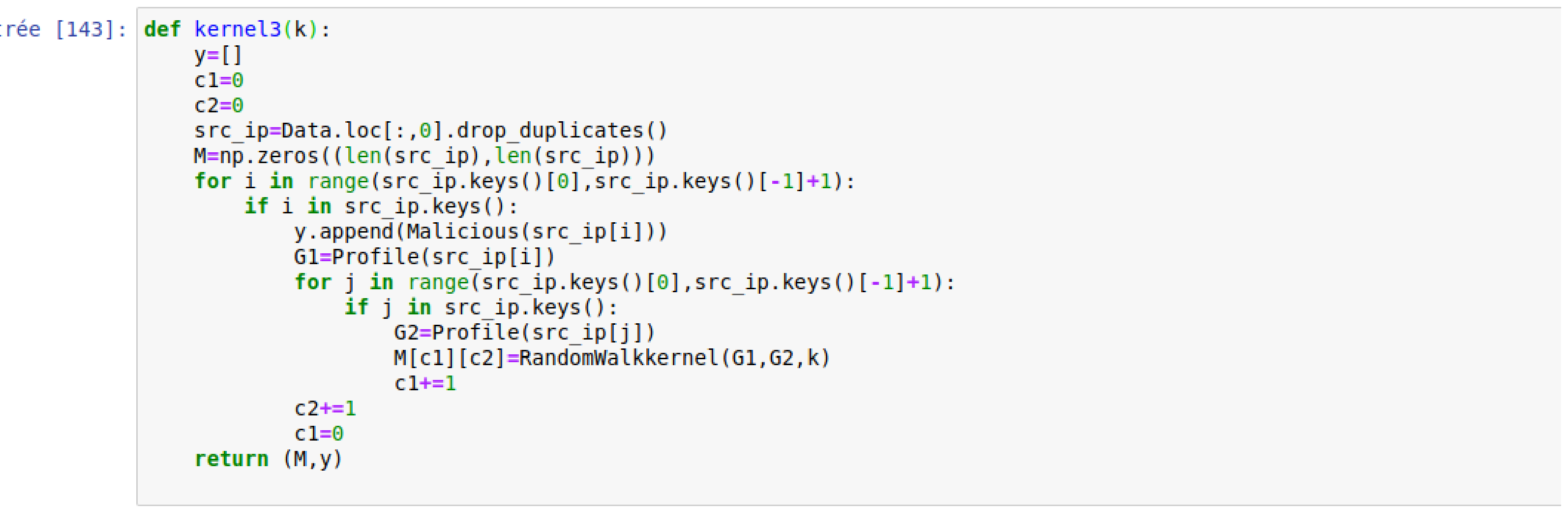


```
Entrée [141]: from sklearn import svm
              kernel_trick_Svm=svm.SVC(kernel='precomputed')
              kernel_trick_Svm.fit(c,m)

    Out[141]: SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
                  decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
                  kernel='precomputed', max_iter=-1, probability=False, random_state=None,
                  shrinking=True, tol=0.001, verbose=False)
```

The first codes below gives one function that calculate the Random walk Kernel that I will Use later in the kernel3 Function to run at the end the SVM classifier:



```
rée [142]:
from numpy.linalg import matrix_power
def RandomWalkkernel_haf(G1,G2,k,landa=0.6):
    a1=nx.convert_matrix.to_numpy_array(G1)
    b1=nx.convert_matrix.to_numpy_array(G2)
    dot_product=nx.tensor_product(G1,G2)
    dot_product=nx.adjacency_matrix(dot_product).todense()
    result=0
    val=np.zeros((len(a1)*len(b1),len(a1)*len(b1)))
    for l in range(k):
        val+=(landa**l)*matrix_power(dot_product, l)

    for i in range(len(a1)*len(b1)):
        for j in range(len(a1)*len(b1)):
            result+=val[i][j]
    return result
```



```
rée [143]:
def kernel3(k):
    y=[]
    c1=0
    c2=0
    src_ip=Data.loc[:,0].drop_duplicates()
    M=np.zeros((len(src_ip),len(src_ip)))
    for i in range(src_ip.keys()[0],src_ip.keys()[-1]+1):
        if i in src_ip.keys():
            y.append(Malicious(src_ip[i]))
            G1=Profile(src_ip[i])
            for j in range(src_ip.keys()[0],src_ip.keys()[-1]+1):
                if j in src_ip.keys():
                    G2=Profile(src_ip[j])
                    M[c1][c2]=RandomWalkkernel(G1,G2,k)
                    c1+=1
            c2+=1
            c1=0
    return (M,y)
```

```
ntrée [144]: c,m=kernel3(4)
             print(c)
             print(m)
```

```
[[1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]
 ...
 [1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
Entrée [ ]: c,m=kernel3(20)
            print(c)
            print(m)
```

```
Entrée [ ]: kernel20=svm.SVC(kernel='precomputed')
            kernel20.fit(c,m)
```

```
Entrée [ ]: c,m=kernel3(50)
            print(c)
            print(m)
```

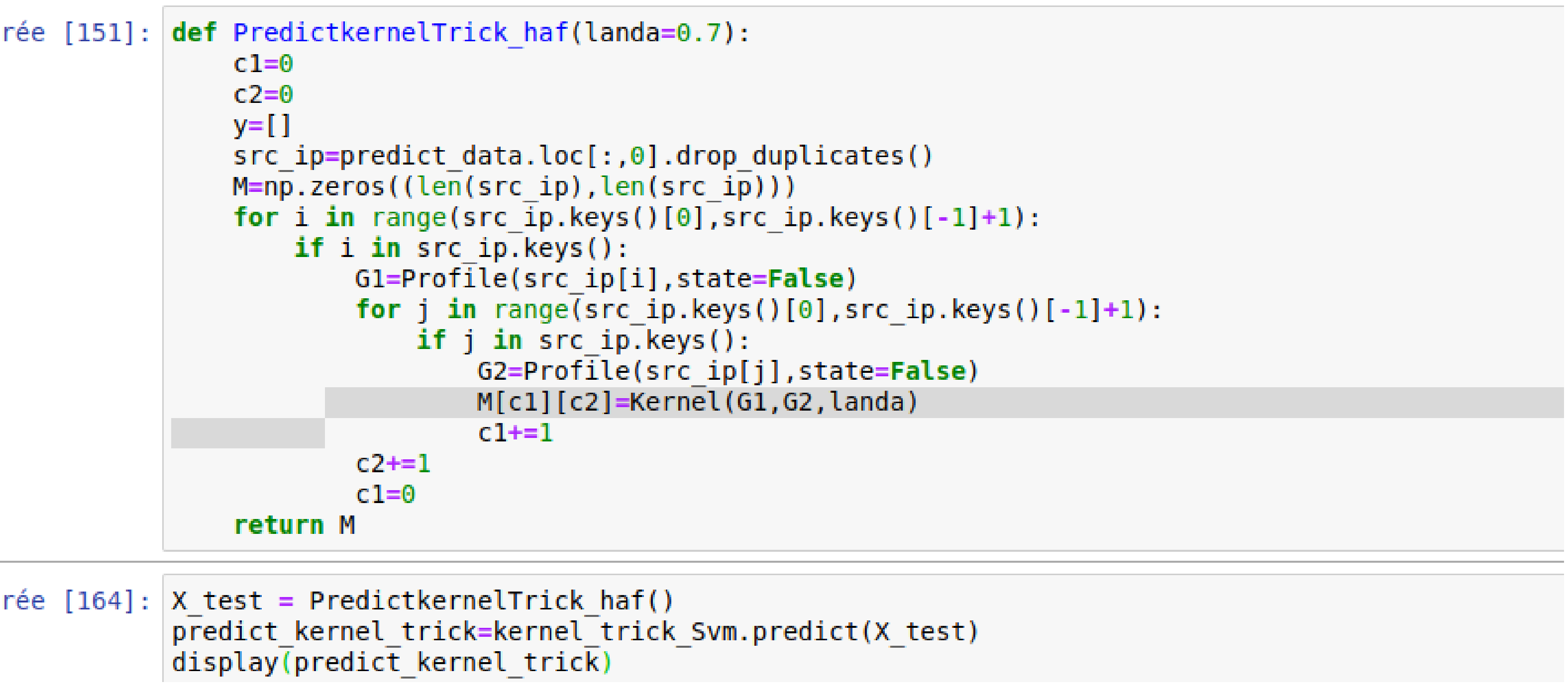
```
Entrée [ ]: kernel50=svm.SVC(kernel='precomputed')
            kernel50.fit(c,m)
```

```
Entrée [ ]: c,m=kernel3(100)
            print(c)
            print(m)
```

```
Entrée [ ]: kernel100=svm.SVC(kernel='precomputed')
            kerel100.fit(c,m)
```

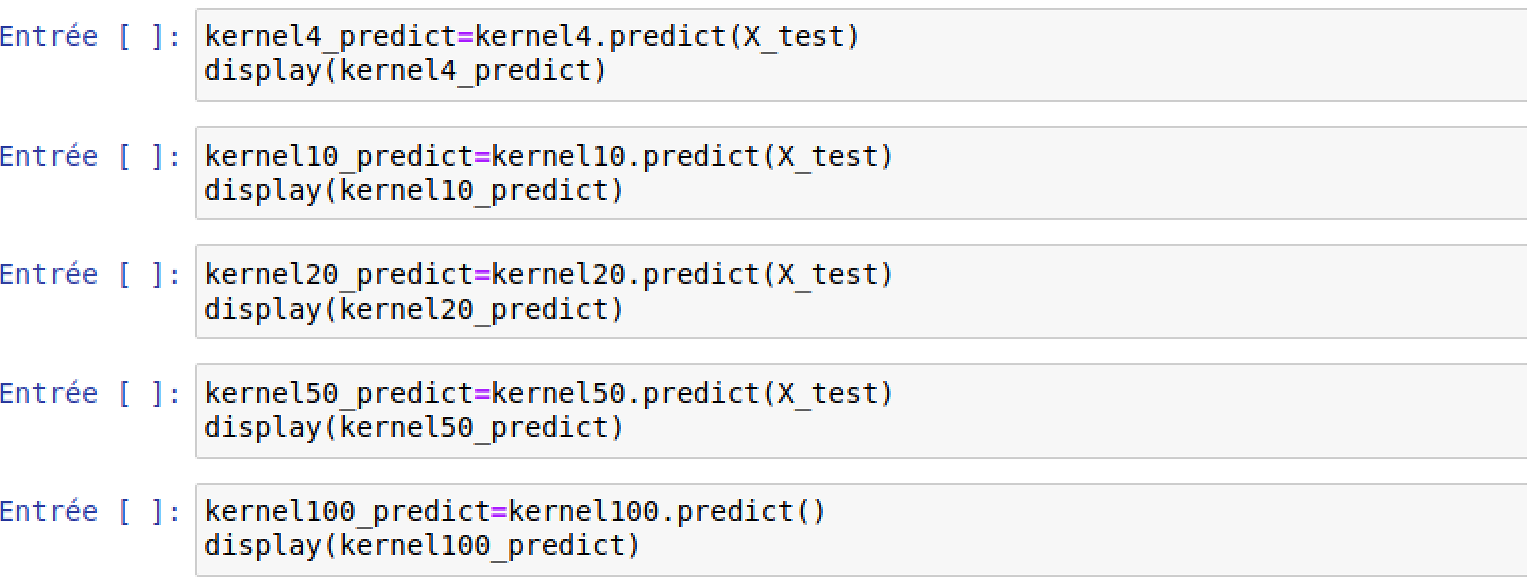
09. Apply the model to the not-annotated-trace.txt

The function Predictkerneltrick_haf below take as input the landa and give us as output X_test
That I will predict later by the kernel4 , kernel10 , kernel50 and kernel100



```
rée [151]: def PredictkernelTrick_haf(landa=0.7):
               c1=0
               c2=0
               y=[]
               src_ip=predict_data.loc[:,0].drop_duplicates()
               M=np.zeros((len(src_ip),len(src_ip)))
               for i in range(src_ip.keys()[0],src_ip.keys()[-1]+1):
                   if i in src_ip.keys():
                       G1=Profile(src_ip[i],state=False)
                       for j in range(src_ip.keys()[0],src_ip.keys()[-1]+1):
                           if j in src_ip.keys():
                               G2=Profile(src_ip[j],state=False)
                               M[c1][c2]=Kernel(G1,G2,landa)
                               c1+=1
                       c2+=1
                       c1=0
               return M
```

```
rée [164]: X_test = PredictkernelTrick_haf()
           predict_kernel_trick=kernel_trick_Svm.predict(X_test)
           display(predict_kernel_trick)
```



```
Entrée [ ]: kernel4_predict=kernel4.predict(X_test)
            display(kernel4_predict)
```

```
Entrée [ ]: kernel10_predict=kernel10.predict(X_test)
            display(kernel10_predict)
```

```
Entrée [ ]: kernel20_predict=kernel20.predict(X_test)
            display(kernel20_predict)
```

```
Entrée [ ]: kernel50_predict=kernel50.predict(X_test)
            display(kernel50_predict)
```

```
Entrée [ ]: kernel100_predict=kernel100.predict()
            display(kernel100_predict)
```

Finally the function Malicious_predict_haf() return one vector Y_test that separate the malicious addresses with
The value 1 from the normal addresses with value 0 and if it finds an empty case it will give it 0.5. So Now we
Can compare the Y_test and the Y_Pred given by SVM classifier to show the false positive and the false
Negative cases.

```
Entrée [153]: def Malicious_predict_haf():
                  def predict(ip):
                      for i in range(len(predict_data)):
                          if (predict_data.isnull().iloc[i,5]):
                              return 0.5
                          if(predict_data.iloc[i,0]==ip and predict_data.iloc[i,5]=="anomaly"):
                              return 1
                      return 0
                  Y_test=[]
                  src_ip=predict_data.loc[:,0].drop_duplicates()
                  for i in range(src_ip.keys()[0],src_ip.keys()[-1]+1):
                      Y_test.append(predict(src_ip[i]))
                  return Y_test
```