Indian Institute of Technology
Guwahati

CS-244 Python Assignment

Data Processing on ADFA Dataset

Submitted To:
Santosh Biswas
Associate Professor
CSE Department

Submitted By :
Neel Mittal
150101042
Prashansi Kamdar
150101047
Suhas Kantekar
150101077

Contents

1 Introduction

Computer security, also known as cybersecurity or IT security, is the protection of computer systems from the theft or damage to the hardware, software or the information on them, as well as from disruption or misdirection of the services they provide.

In computer security a countermeasure is an action, device, procedure, or technique that reduces a threat, a vulnerability, or an attack by eliminating or preventing it, by minimizing the harm it can cause, or by discovering and reporting it so that corrective action can be taken.

Attack detection can be posed as a statistical learning problem for different attack scenarios in which the measurements are observed in the log files generated by an attacked system. Machine Learning Algorithms can be used to classify measurements as being either secure or attacked.

The Australian Defence Force Academy - Linux Dataset was generated on Linux local server running on Ubuntu11.04, offering a variety of functions such as file sharing, database, remote access and web server.

Six types of attacks occur in ADFA-LD including two brute force password guessing attempts on the open ports enabled by FTP and SSH respectively, an unauthorised attempt to create a new user with root privileges through encoding a malicious payload into a normal executable, the uploads of Java and Linux executable Meterpreter payloads for the remote compromise of a target host, and the compromise and privilege escalation using C100 webshell. These types are termed as Hydra-FTP, Hydra-SSH, Adduser, Java-Meterpreter, Meterpreter and Webshell respectively.

The Dataset provided has three folders :

- Attack_Data_Master
  Contains the log files generated during the period the attacks were happening, separated into folders for each type of attack. 70% of the files are to be used to generate the training dataset for malicious behaviour and the remaining 30% to be used to generate the testing dataset.
- Training_Data_Master
  Contains the log files generated during the period no attacks were happening. These log files are to be used for generating the training dataset for normal behaviour.
- Validation_Data_Master

Contains the log files generated during the period no attacks were happening. These log files are to be used for generating the test dataset for normal behaviour.

The first step to solve the problem of attack detection is to generate the training and testing dataset for the Machine Learning Algorithm. In this document is the description for an algorithm to process the "Australian Defence Force Academy - Linux Dataset" to generate the required training and testing datasets.

2 The Algorithm

The Algorithm can be broken down into two subproblems to be solved sequentially.

Each of the attacks has a signature which can be found out from the log files. Since the log files just have a lot of numbers, we must analyse these numbers to obtain a pattern in the log files. One of the ways to obtain some pattern from the files is to take 3-grams, 5-grams or 7-grams from the log files and look at their frequencies. If there are some n-grams which have a high frequency in most of the log files corresponding to one type of an attack, it can be interpreted that these n-grams are generated most frequently when such an attack is occuring. These n-grams will thus, give us a signature for each attack.

The first problem is selecting the features which indicate malicious activity. To get these n-grams:
1. We concatenate the log files corresponding to one attack.
2. We count the occurences of the various n-grams in the concatenated file.
3. After getting the frequencies for each n-gram, we select the top 30% and add them to a set.
4. We repeat this for all the attack types and thus obtain the set of n-grams that indicate malicious activity.

The second problem is to obtain the signature of each attack, and normal behaviour after getting the set of features which indicate malicious activity. To get this signature:
1. We go through the log files and generate a mapping between the set of malicious n-grams and their frequencies in each log file.
2. We label these with the attack type they are associated with.
3. We append the frequencies and labels in the next row of a file.
Applying this procedure to the files kept for generating training dataset will generate the table required to train the Machine Learning Algorithm. Applying it to the files kept for generating testing dataset will generate the table required to test the Machine Learning Algorith.

3 Implementing Concatenation

The following Algorithm is implemented :

1. Go through the Attack_Data_Master folder and get the labels of all attack types. This takes $O(n)$ where n is the number of folders in Attack_Data_Master.
2. Open each log file in the subfolder in Attack_Data_Master and append the contents of each file into the concatenated output file corresponding to the attack. This will take total $O(m)$ where m is the total number of log files.
3. Open each log file in the Training_Data_Master and append the contents to the concatenated output file corresponding to normal. This will take total $O(m)$ since m is the total number of log files.

Therefore, this will take $O(m+n)$ time. Since,

$$m \geq n$$

It is an $O(m)$ procedure.

```
#Get the attack folders present in attack data master folder
root, dirs, files = os.walk("./Attack_Data_Master").__next__()
#attack types
temp_l = list()

#folder names in reverse
hello = list()

#cataloguing the types of attacks given
for x in dirs :
        hello.append(str(x)[::-1])
#getting names of directories uptil underscore and adding it to
 ↪  temp_l
for x in hello:
    i=0
    for y in x:
        if x[i] == '_':
            temp = x[i:]
            temp = temp[::-1]
```

```
            if temp in temp_l:a=1
            else :temp_l.append(temp)
            break

        else : i=i+1

#initialising sets to hold the features generated
set_3 = set()
set_5 = set()
set_7 = set()

#traversing the folders
for xx in temp_l:
    jj=1

    #truncating the files if they exist
    open("training_"+xx+".txt",'w').close()
    open("testing_"+xx+".txt",'w').close()


    #option to include attack type or not
    choice=""
    while choice!="y" and choice!="n" :
        choice = input("Include attack \""+xx[:-1]+"\" for feature
        ↪  selection? (y/n) : ")


    #if input is no move to next attack type
    if(choice=="n") :
        continue


    #option to choose which folders to include in generation of
    ↪  training data set
    choice=""
    while choice!="y" and choice!="n" :
        choice = input("Include top 70% folders of the attack
        ↪  \""+xx[:-1]+"\"? (y/n) : ")
```

```
#setting a flag according to user input to include top 70%
↪  folders or not
flag=0
if(choice=="y"):
    flag=1



#concatenating all the files of an attack type


#loop to count total number of folders
rr=1
while xx +str(rr) in dirs :
    rr+=1


#getting 70% of total number of folders
temp_count = int(math.ceil(0.7*(rr-1)))


#getting 30% of total number of folders
no_temp_count = rr-1-temp_count


while xx+str(jj) in dirs :

    if temp_count==0 :

        #concatenate to testing file
        read_files = glob.glob("./Attack_Data_Master/" + xx +
        ↪  str(jj) + "/*.txt")
        with open("testing_" + xx + ".txt", "ab") as outfile:
            for f in read_files:
                with open(f, "rb") as infile:
                    outfile.write(infile.read())
                outfile.write(str.encode("% "))
        jj+=1
        continue
```

```
if no_temp_count==0:

    #concatenate to training file
    read_files = glob.glob("./Attack_Data_Master/" + xx +
    ↪  str(jj) + "/*.txt")
    with open("training_" + xx + ".txt", "ab") as outfile:
        for f in read_files:
            with open(f, "rb") as infile:
                outfile.write(infile.read())
            outfile.write(str.encode("% "))
    jj+=1
    continue


if flag==0 :
    choice=""
    while choice!="y" and choice!="n" :
        choice = input("Include folder \""+xx+str(jj)+"\" for
        ↪  feature selection?(y/n) : ")


if choice=="n":

    #concatenate to testing file
    read_files = glob.glob("./Attack_Data_Master/" + xx +
    ↪  str(jj) + "/*.txt")
    with open("testing_"+xx+".txt","ab") as outfile:
        for f in read_files:
            with open(f, "rb") as infile:
                outfile.write(infile.read())
            outfile.write(str.encode("% "))

    no_temp_count -=1
    jj += 1
    continue

#concatenate to training file
read_files =
↪  glob.glob("./Attack_Data_Master/"+xx+str(jj)+"/*.txt")
with open("training_"+xx+".txt", "ab") as outfile:
```

```
            for f in read_files:
                with open(f, "rb") as infile:
                    outfile.write(infile.read())
                outfile.write(str.encode("% "))
        jj += 1
        temp_count -=1


#concatenation for normal log files


open("training_Normal_.txt",'w').close()

read_files = glob.glob("./Training_Data_Master/*.txt")

with open("training_Normal_.txt", "ab") as outfile:
    for f in read_files:
        with open(f, "rb") as infile:
            outfile.write(infile.read())
        outfile.write(str.encode("% "))

open("testing_Normal_.txt",'w').close()

read_files = glob.glob("./Validation_Data_Master/*.txt")

with open("testing_Normal_.txt",'ab') as outfile:
    for f in read_files:
        with open(f, "rb") as infile:
            outfile.write(infile.read())
        outfile.write(str.encode("% "))
```

4 Feature Selection

The following algorithm is implemented :

1. Open each of the concatenated file created during the last step. This takes $O(n \times k)$ time since the number of attacks is n and so there will be only n concatenated files. k is the size of the biggest file.
2. Load the various 7-grams, 5-grams, 3-grams from the concatenated files. This takes $O(n)$ since the ngrams method is $O(1)$.
3. Count the frequencies of the ngrams. This takes $O(n \times l)$ time if l is the highest number of different ngrams obtained.
4. Add the top 30% to feature set. This takes $O(n \times l)$ time since union is $O(l)$.
5. Write the feature set to a file. This takes $O(l)$ time.

Therefore, this entire procedure takes $O(n \times (l + k))$ time where l is the highest number of different ngrams obtained. Since

$$(n \times k) \geq (n \times l)$$

We can say its an $O(n \times k)$ procedure.

```
read_files = glob.glob("./training_*.txt")
for f in read_files :
    fl = open(f,"r")  # open the concatenated file

    dict_7 = OrderedDict()  # dictionary to store 7 grams
    dict_5 = OrderedDict()  # dictionary to store 5 grams
    dict_3 = OrderedDict()  # dictionary to store 3 grams

    file = fl.read().split()    #splitting the file at every space

    sevengrams = ngrams(file,7) #getting the 7-grams
    fivegrams = ngrams(file,5)  #getting the 5-grams
    threegrams = ngrams(file,3) #getting the 3-grams

    dict_5 = dict(Counter(fivegrams))   #getting the frequencies of
     ↪  the various 7-grams
    dict_3 = dict(Counter(threegrams))  #getting the frequencies of
     ↪  the various 3-grams
```

```
dict_7 = dict(Counter(sevengrams))  #getting the frequencies of
↪  the various 5-grams

for i in list(dict_7.keys()):   #removing the invalid 7-grams
    if '%' in str(i):
        del dict_7[i]

for i in list(dict_3.keys()):   #removing the invalid 3-grams
    if '%' in str(i):
        del dict_3[i]

for i in list(dict_5.keys()):   #removing the invalid 5-grams
    if '%' in str(i):
        del dict_5[i]

#sorting the n-grams according to their counts
dict_5 = (sorted(dict_5.items(), key=operator.itemgetter(1)))  #
↪  sort according to value
dict_7 = (sorted(dict_7.items(), key=operator.itemgetter(1)))  #
↪  sort according to value
dict_3 = (sorted(dict_3.items(), key=operator.itemgetter(1)))  #
↪  sort according to value

# reversing the dictionaries to get them in descending order
dict_3 = OrderedDict(dict_3[::-1])
dict_5 = OrderedDict(dict_5[::-1])
dict_7 = OrderedDict(dict_7[::-1])

top_30_3 = int(math.ceil((len(dict_3) + 0.0) * 0.3))  # get top
↪  30 % tuples and convert it to a int
top_30_5 = int(math.ceil((len(dict_5) + 0.0) * 0.3))  # get top
↪  30 % tuples and convert it to a int
top_30_7 = int(math.ceil((len(dict_7) + 0.0) * 0.3))  # get top
↪  30 % tuples and convert it to a int

#adding the top 30% frequent n-grams to the feature set
set_3 = set_3.union(set(list(dict_3.keys())[0:top_30_3]))
set_5 = set_5.union(set(list(dict_5.keys())[0:top_30_5]))
set_7 = set_7.union(set(list(dict_7.keys())[0:top_30_7]))
```

```
#writing the sets into the final output files
c_file = open("3_features.txt","w")
for key in set_3 :
    c_file.write(str(key)+"\n")
c_file.close()

c_file = open("5_features.txt","w")
for key in set_5 :
    c_file.write(str(key)+"\n")
c_file.close()

c_file = open("7_features.txt","w")
for key in set_7 :
    c_file.write(str(key)+"\n")
c_file.close()

input("Feature Selection completed.\n\nPress enter to continue with
↪  he training table generation...")
```

5 Generating Training Table

The following algorithm is implemented :

1. Load the feature n-grams into an OrderedDict with values 0. This will $O(l)$ time.
2. Open each concatenated training file and split it back into the individual log files. This will take $O(n \times k)$ time as split() method is $O(k)$.
3. For each of log file, count the n-grams. This will take $O(m \times l)$ time since there are m log files and l is the upperbound on the number of n-grams.
4. For each n-gram, add the frequency to the OrderedDict. This will throw an error if they key doesn't exist, but using try except will get around that. This will take $O(m \times l)$ time since l is the upperbound on the number of ngrams and this will repeat for m log files.

Thus, this procedure is $O((n \times k) + (m \times l))$. Since k is the largest size of the concatenated file,

$$l \leq k$$

Thus, this procedure becomes $O((n+m) \times k)$, where n is the number of attack types and m is the total number of log files. But,

$$m \leq n$$

Thus this becomes an $O(m \times k)$ procedure.

```
#Training Table Generation begins here
print("\nGenerating the training table...")

#Ordered dictionaries to hold count of the features selected
dict_3 =
 ↪  OrderedDict.fromkeys(open('3_features.txt','r').read().splitlines(),0)
dict_5 =
 ↪  OrderedDict.fromkeys(open('5_features.txt','r').read().splitlines(),0)
dict_7 =
 ↪  OrderedDict.fromkeys(open('7_features.txt','r').read().splitlines(),0)

#opening the files which store the final training and testing
 ↪  datasets
res_3 = open('3-gram-train.txt','w')
res_5 = open('5-gram-train.txt','w')
```

```
res_7 = open('7-gram-train.txt','w')
val_3 = open('3-gram-test.txt','w')
val_5 = open('5-gram-test.txt','w')
val_7 = open('7-gram-test.txt','w')

#timing the algorithm
start_time_train = time.time()

#looping over the concatenated files
read_files = glob.glob("./training_*.txt")
for f in read_files:

    infile = open(f,"r").read()
    if infile == "":
        continue

    print("Processing : " + f[11:len(f)-5])

    #strings to hold the data to be written into the final files
    str_3 = ""
    str_5 = ""
    str_7 = ""

    for infiles in infile.split('% '): #splitting the concatenated
     ↪  file at each % to get the original log files

        grams_7 = dict(Counter(ngrams(infiles.split(),7)))  #getting
         ↪  the count of each unique 7-gram in the log file

        #making temporary dictionaries with counts of each n-gram 0
        dict_3_temp = dict_3.copy()
        dict_5_temp = dict_5.copy()
        dict_7_temp = dict_7.copy()

        #looping over the various 7-grams found
        for i in grams_7.keys():

            #adding the count if the 7-gram is in the feature set
            try:
                dict_7_temp[str(i)] += grams_7[i]
```

```
            except KeyError:
                pass

            #adding the count if the 5-gram is in the feature set
            try:
                dict_5_temp[str(i[0:5])] += grams_7[i]
            except KeyError:
                pass

            #adding the count if the 3-gram is in the feature set
            try:
                dict_3_temp[str(i[0:3])] += grams_7[i]
            except KeyError:
                pass

        str_3 += (str(list(dict_3_temp.values())) + " " +
        ↪  f[11:len(f)-5] + "\n")
        str_5 += (str(list(dict_5_temp.values())) + " " +
        ↪  f[11:len(f)-5] + "\n")
        str_7 += (str(list(dict_7_temp.values())) + " " +
        ↪  f[11:len(f)-5] + "\n")

    #adding to the final output file
    res_3.write(str_3)
    res_5.write(str_5)
    res_7.write(str_7)

#printing the time taken to generate the training table
print("Time Taken to Generate Training Table : " + str(time.time() -
↪  start_time_train))
```

6 Generating Testing Table

This procedure is just the same as the previous one. It will thus be $O(m \times k)$.

```
#timing the algorithm
start_time_test = time.time()

#opening the concatenated files
read_files = glob.glob("./testing_*.txt")
for f in read_files:

    infile = open(f,"r").read()
    if infile == "":
        continue

    print("Processing : " + f[11:len(f)-5])

    #strings to hold the data to be written into the final files
    str_3 = ""
    str_5 = ""
    str_7 = ""

    for infiles in infile.split('% '): #splitting the concatenated
     ↪  file at each % to get the original log files

        grams_7 = dict(Counter(ngrams(infiles.split(),7)))  #getting
         ↪  the count of each unique 7-gram in the log file

        #making temporary dictionaries with counts of each n-gram 0
        dict_3_temp = dict_3.copy()
        dict_5_temp = dict_5.copy()
        dict_7_temp = dict_7.copy()

        #looping over the various 7-grams found
        for i in grams_7.keys():

            #adding the count if the 7-gram is in the feature set
            try:
                dict_7_temp[str(i)] += grams_7[i]
            except KeyError:
```

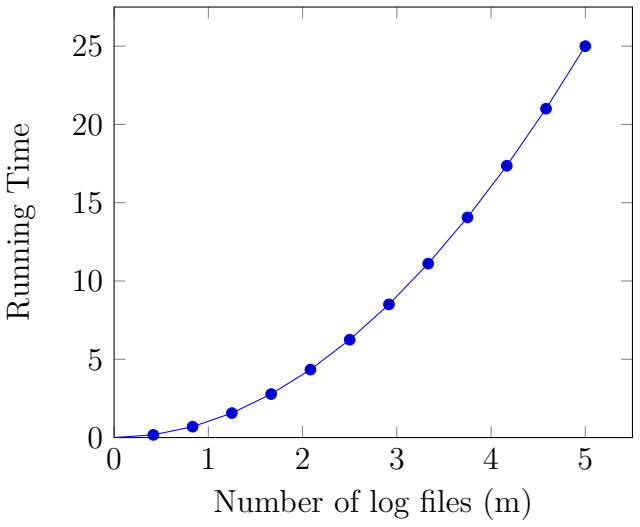
```
                pass

            #adding the count if the 5-gram is in the feature set
            try:
                dict_5_temp[str(i[0:5])] += grams_7[i]
            except KeyError:
                pass

            #adding the count if the 3-gram is in the feature set
            try:
                dict_3_temp[str(i[0:3])] += grams_7[i]
            except KeyError:
                pass

        str_3 += (str(list(dict_3_temp.values())) + " " +
        ↪  f[11:len(f)-5] + "\n")
        str_5 += (str(list(dict_5_temp.values())) + " " +
        ↪  f[11:len(f)-5] + "\n")
        str_7 += (str(list(dict_7_temp.values())) + " " +
        ↪  f[11:len(f)-5] + "\n")

    #adding to the final output file
    val_3.write(str_3)
    val_5.write(str_5)
    val_7.write(str_7)

#printing the time taken to generate the testing table
print("Time Taken to Generate Testing Table : " + str(time.time() -
↪  start_time_test))
```

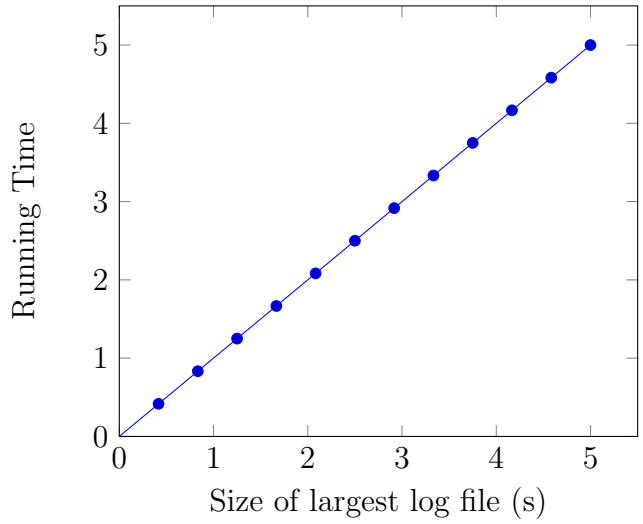
7 Time Complexity

We saw how the algorithm taking $O(m)$ time for the first step. The second step is $O(n \times k)$. The third and fourth steps are $O(m \times k)$. Therefore the entire algorithm is $O(m \times k)$ time, where m is the total number of log files and k is the largest concatenated log file. Thus, if the largest log file is of size s.

$$k \leq m \times s$$

Thus, the order of the algorithm is $O(s \times m^2)$ when written strictly in terms of input parameters, where s is the size of the largest log file and m is the number of log files.





8 Test Run

Running the script on an Ubuntu 16.04 LTS system :

```
neel@OmegaPC:~/Downloads/ADFA-LD-and-Weka-KDD(2)/ADFA-LD-and-Weka-KDD/ADFA-LD/ADFA-LD$ python script4.py
Include attack "Web_Shell" for feature selection? (y/n) : y
Include top 70% folders of the attack "Web_Shell"? (y/n) : y
Include attack "Hydra_SSH" for feature selection? (y/n) : y
Include top 70% folders of the attack "Hydra_SSH"? (y/n) : y
Include attack "Adduser" for feature selection? (y/n) : y
Include top 70% folders of the attack "Adduser"? (y/n) : y
Include attack "Meterpreter" for feature selection? (y/n) : y
Include top 70% folders of the attack "Meterpreter"? (y/n) : y
Include attack "Java_Meterpreter" for feature selection? (y/n) : y
Include top 70% folders of the attack "Java_Meterpreter"? (y/n) : y
Include attack "Hydra_FTP" for feature selection? (y/n) : y
Include top 70% folders of the attack "Hydra_FTP"? (y/n) : y
Feature Selection completed.

Press enter to continue with he training table generation...

Generating the training table...
Processing : Adduser
Processing : Java_Meterpreter
Processing : Hydra_SSH
Processing : Web_Shell
Processing : Meterpreter
Processing : Normal
Processing : Hydra_FTP
Time Taken to Generate Training Table : 38.28456115722656
Processing : Normal
Processing : Hydra_SSH
Processing : Web_Shell
Processing : Java_Meterpreter
Processing : Meterpreter
Processing : Hydra_FTP
Processing : Adduser
Time Taken to Generate Testing Table : 133.11507487297058
```

We can see the running time is just under 3 minutes to generate the training and testing tables. The feature set is stored in 3_features.txt, 5_features.txt, 7_features.txt for 3-gram, 5-gram, and 7-gram respectively. The training tables are 3-gram-train.txt, 5-gram-train.txt, and 7-gram-train.txt. The testing tables are 3-gram-test.txt, 5-gram-test.txt, and 7-gram-test.txt. The tables have the n^{th} integer entry in a row correspond to the n^{th} n-gram in their respective feature file. The last entry in the row is the label of the attack type.