Indian Institute of Technology Guwahati

N-Gram based Vector Representation for ADFA-LD Dataset

Anup Agarwal: 150101009, Mohit Kumar: 150101035, Satti Sai Chandan Reddy: 150101059

Abstract

This paper describes the preprocessing stage involved in building intrusion detection systems for predicting anomalous behaviours using system call signature. For the purpose of this paper we used the Australian Defence Force Academy Linux Dataset (ADFA-LD) data set.

Keywords: python, Intrusion Detection, ngrams

1. Introduction

Intrusion detection system based on predicting anomalous behaviour of a process using it's system calls signature is a wide known technique.[1] This problem can be dealt with various machine learning algorithms. The Australian Defence Force Academy Linux Dataset (ADFA-LD) and Australian Defence Force Academy Windows Dataset (ADFA-WD) are some generation system calls datasets that contain labelled system call traces for modern exploits and attacks on various applications.[2]

This project shows a method to process these datasets such that the output can be feeded into a standard machine learning algorithm for anomaly prediction.

1.1. Defining the Stages of the Project

1.1.1. The Data

Table 1 describes the diversity in data set.

Name	Count
Normal Training	833
Normal Validation	4372
Attack	746

Table 1: Trace Segregation

Table 2 shows the distribution of the traces accross various attack types.

Name	Count
Java Meterpreter	124
Meterpreter	75
Web Shell	118
Adduser	91
Hydra FTP	162
Hydra SSH	176

Table 2: Attack Trace Segregation

1.1.2. Divide

In the first phase we divide the attack dataset into 2 parts viz. Training and Test datasets with 70% signatures in Training set and remaining 30% in Test set.

After this procedure here are roughly 1329 training data set traces (496 Attack + 833 Normal traces) and 4622 test data set traces (250 Attack + 4372 Normal traces)

1.1.3. Conquer

In the second phase we compute the n-grams for each of the file in the particular dataset and store them. Next, for each particular attack data type and the normal set we computed the top 30% n-grams, took the union of these and stored them, this forms the unique top 30% n-grams. Finally for each trace we computed the frequency of the unique top 30% n-grams and formed a feature vector where the ith entry denotes the frequency of the ith n-gram in the ordered set of unique top 30% n-grams.

2. The Divide Stage

There are 6 attack categories viz. Java Meterpreter, Meterpreter, Web Shell, Adduser, Hydra FTP, Hydra SSH. Each has 10 directories out of which we randomly selected 7 for the training directory and remaining for validation.

To find get a list of all the attack types we used the following bash script: (in this case although there are only 6 types and this could be done by hand but in a general setup the script could be quite useful)

```
find <path> -type d | grep ".*_2" | sed s_.*/__g | sed s_2__g
```

By this we are finding all directories which end with "_2" and then we remove the prefix and suffix of the path to all directories which satisfy the above conditions.The final output being:

```
Java_Meterpreter_
Meterpreter_
Web_Shell_
Adduser_
```

```
Hydra_FTP_
Hydra_SSH_
```

For randomly selecting 7 directories we used the function `random.shuffle(x[, random])`[3] which shuffles the sequence `x` in place. The input `x` was `range(1,11)` i.e. list of integers from 1 to 10 inclusive and we spliced it with `x[:7]` to seelct the first 7 items from the shuffled list, and put the correspoinding directories into training directory and rest into test set using the cp[4] command.

3. The Conquer Stage

3.1. Computing ngrams

We traverse through the directory tree using the `os.path.walk` library and for each file we compute its ngrams and save them in a dictionary referenced by their attack type. To do this the following segment is used: (Here `current_ngrams` is the list of ngrams obtained from the current file being traversed and `all_ngrams` is the list of all ngrams seen till now)

```
for tt in training_types:
  if tt in dir_name:
    all_ngrams[tt].extend(current_ngrams)
```

To find ngrams for a given list of numbers, we use the following algorithm:

```
def ngrams(inlist, n):
  out = []
  for i in range(len(inlist)-n+1):
    out.append(tuple(inlist[i:i+n]))
  return out
```

3.1.1. Complexity Analysis

This algorithm takes $\mathcal{O}(|inlist|)$ where $|inlist|$ is the number of elements in the list $inlist$[5]

The overall routine that traverses through all the directories and files takes $\mathcal{O}(\sum_{trace} |trace|)$ where $|trace|$ is the number of characters in the trace file and the summation is over all the trace files in the training data set.

3.2. Finding top 30% ngrams

For the frequency part we first sorted the ngrams, then we used the Counter collection in python library. A Counter is a dictionary subclass for counting hashable objects. It is an unordered collection where elements are stored as dictionary keys and their counts are stored as dictionary values. The Counts are allowed to be any integer value including zero or negative counts.[6] This library was 2 times faster than the brute force dictionary method to find frequency of the ngrams.

Then we sorted the ngrams obtained by their frequency and spliced the top 30%.

```
from collections import Counter

#Sort
#Find Frequency of ngram
#Sort by Frequency of ngram
for tt in training_types:
  curr_ngrams = sorted(all_ngrams[tt])

  freq = Counter(curr_ngrams)
  sorted_ngrams_by_freq =
  sorted(freq.items(),key=lambda x: x[1], reverse = True)

  freq_ngrams[tt].extend(sorted_ngrams_by_freq)

import math

#Compute top30% ngrams by frequncy for each attack type
for tt in training_types:
  curr_ngrams = freq_ngrams[tt]
  selected_indices = int(math.ceil(len(curr_ngrams)*0.3))
  top30 = curr_ngrams[:selected_indices]

  top30_ngrams.extend(map((lambda x: x[0]),top30))
```

3.2.1. Time Complexity Analysis

For sorting we take $\mathcal{O}(nlog(n))$ time[5] where n is the length of the list. The Counter object is constructed in $\mathcal{O}(n)$ time and returns the unique items with their frequencies in $\mathcal{O}(n)$ time. Splicing the list for the top 30% ngrams takes $\mathcal{O}(k)$ time where k is the length of list to be retained in the splice. Therefore the time is dominated by sorting. This procedure is done for all training types (Attack Trace types + Normal). This yields a time of $\mathcal{O}(\sum_{tt \in training\_types} n_{tt} * log(n_{tt}))$ where $n_{tt} = |all\_ngrams[tt]|$ i.e. number of ngrams of that training type.

3.3. Feature Vector Formation

We took the union of top 30% ngrams of each type and then sorted them to obtain the feature list. Next we traversed through all the traces and computed their ngrams and then calculated the frequency of ngrams present in the feature list. The top 30% ngrams along with its frequency in a given trace formed the feature vector for a particular trace. We did this for each trace using the following code segment:

```
#unique_top_30% is the sorted feature list

dic = {}
#Initialise dictionary with 0 frequencies
for ngram in unique_top30:
  dic[ngram] = 0

#Iterate through the ngrams appropraitely increase frequency count
for ngram in l: #l is the list of ngrams in a trace
  if ngram in dic:
    dic[ngram] += 1

#Append the vector formed to the list of feature vectors.
feature_vec = []
for ngram in dic:
  feature_vec.append(dic[ngram])
```

3.3.1. Time Complexity Analysis

For this particular segment for calculating the frequencies we did not use the counter as it was consuming more time. The initialization takes $\mathcal{O}(|unique\_top30|)$ time. Computation of frequencies takes $\mathcal{O}(|trace|)$ time and appending is $\mathcal{O}(|unique\_top30|)$ time[5]. This procedure is carried for each of the trace. Therefore the total time is: $\mathcal{O}(\sum_{trace} |unique\_top30| + |trace|)$

Thus the overall time complexity of the Conquer Stage is:
$\mathcal{O}(\sum_{trace} |unique\_top30| + |trace| + \sum_{tt \in training\_types} n_{tt} * log(n_{tt}))$
where $n_{tt} = |all\_ngrams[tt]|$ i.e. number of ngrams of that training type.

Appendix A. Code Attached

1. Divide Stage Jupyter Notebook
2. Conquer Stage Jupyter Notebook

References

[1] N. Hubballi, S. Biswas, S. Nandi, Sequencegram: n-gram modeling of system calls for program based anomaly detection, in: 2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011), 2011, pp. 1–10. doi:10.1109/COMSNETS.2011.5716416.

[2] B. Borisaniya, D. Patel, Evaluation of modified vector space representation using adfa-ld and adfa-wd datasets, Journal of Information Security 6 (2015) 250–264. doi:10.4236/jis.2015.63025.

[3] Python, The python standard library, `https://docs.python.org/2/library/` (2017).

[4] GNU, Bash reference manual, `https://www.gnu.org/software/bash/manual/bash.html` (2016).

[5] Python, Python time complexity, `https://wiki.python.org/moin/TimeComplexity`.

[6] Python, High-performance container datatypes, `https://docs.python.org/2/library/collections.html`.