Group33 - Lab3 Report

June 12, 2020

0.1 CS4035 - Cyber Data Analytics (Lab 3)

0.1.1 Group Number : 33

0.1.2 Student 1

Name : Shipra Sharma

ID : 5093406

0.1.3 Student 2

Name : Sudharshan Swaminathan

ID : 5148340

0.1.4 Readme (setup instructions)

We installed two explicit modules named `yellowbricks`, `tslearn` and `mmh3`. This could be done using `pip install` command in your python environment. Apart from these, all the functions and objects needed are included in the import statements. The other genral modules that are being imported are `sklearn`, `statsmodel`, `numpy`, `matplotlib`, and `pandas`. These should have already been installed on your system while doing the assignment (in case they are not, kindly install the same to execute the code smoothly).

Note: installation of mmh3 might throw error. Inorder to solve the same, please install Microsfot Visual Studio C++ build tools.

0.1.5 Familiarization and discretization task

We pick one of the infected hosts from the dataset by filtering them with the help of Labels. We then analyse which protocols do these hosts use the most to communicate. We pick one host `147.32.84.165` to visualize the botnet traffic.

As it can be seen from the above plot, the selected infected host seems to be using ICMP the most to communicate. We now see how the communication from non-malicious netflows looks like. To do this we remove the botnet and the background related netflows.

The non-malicious traffic contains a very less proportion of ICMP packets compared to the one observed for an infected host. Therefore, it is safe to make an assumption that if abnormally large number of ICMP packets are seen, then it probably could be an infected host acting in the network.

It can also be seen from below given boxplots on the basis of total packets sent, that an infected host is specific in its behaviour in terms of the type of traffic generated. On the contrary, the traffic generated by other hosts on the network is more spread out in terms of the total packets in the netflow. Such features can be successfully used to generate signatures for infected hosts in the network.

b. Discretize selected features After removing the background data, we label encode `Proto` and `Dir`. Since `Proto`, `TotPkts` and `TotBytes` have the capability identify botnet traffic from normal benign ones, we can choose these features to be discretized into quantiles. We use KBins-Discretizer to discretize `TotPkts` and `TotBytes` and choose the optimum number through the Elbow method.

We use the discretized data to combine the features and encode each netflow. This is done using the Algorithm 1 found in the paper "Learning Behavioral Fingerprints From Netflows Using Timed Automata", by Pellegrino, G. et al. The encoded value can be found in the 'code' attribute seen in the displayed dataframe. As shown above in the visualization section, the Protocol, and the total packets and bytes in every flow helps in characterizing a malicious netflow. Therefore, we use `Proto`, `TotPkts` and `TotBytes` attributes to encode the netflows as well.

0.1.6 Frequent Task - Shipra Sharma

For the frequent task we implement the following steps: 1. Derive the 3-grams based on the code value (the one that we got from discretization task) and find 3-grams' original frequency. 2. Select 10 random 3-grams which serve as the algorithm's counter (moderator dictionary) 3. Implement the space saving algorithm to find the top 10 most frequent 3-grams

The ten most frquent 3-grams as per the space saving algorithm can be observed as follows:

3-gram	Computed Frequency	Original Frequency
664	1678	1678
666	3393	3389
333	105745	105391
222	1691	272
422	1615	46
226	1617	113
266	1616	174
642	1615	72
262	1613	42
622	1614	54

From the above results we can easily conclude that the 3-grams which have larger frequeries in the orginal data like 333, 666, 664 are estimated somewhat the same with minimal errors but the ones that have smaller frequencies in the orginal data are estimated with larger values leading to the limitations in algorithm's calculation. It might be because of the key replacement logic that the algorithm uses, i.e. the element with minimum occurance would be replaced first if a new element is found in the data stream. Also, only three 3-grams collide in the top-10 most frequent 3-grams original vs space saving algorithm list. These elements are the most frequent ones in both of these lists (333, 666 and 664). Hence, from our implementation and analysis, we can deduce that Space

saving algorithm works almost correctly for the elements with larger occurences but fails to justify the estimation for less frequent entries in a given stream.

0.1.7 Sketching Task - Shipra Sharma

To implement the sketching task we have followed the following steps: 1. Implement a class to calculate the hash values 2. Implement a method to calculate width and height of the hash table, create the sketch and count the frequencies of the 3-grams 3. Prepare the input data for the min count sketch implementation 4. Call the function using different values of epsilon and delta and find the top 10 most frequent 3-grams

In this task, we computed the top-10 3-grams using the Count-min Sketch Algorithm. The algorithm works on a hash matrix that is obtained by multiplying height(depth) and widht which are computed using two parameters, namely delta (error probability) and epsilon (error factor). The width is e/epsilon and the depth is calculated by the formula log(1/delta). Using the algorithm we got the following results:

Top- 10 Width = 27183 Height = 9	Computed Frequency	Original Frequency	Top - 10 Width = 27 Height = 2	Computed Frequency	Original Frequency
333	105391	105391	333	105422	105391
666	3389	3389	666	3594	3389
664	1678	1678	425	1788	3
466	1432	1432	646	1761	1397
646	1397	1397	355	1761	1397
566	1358	1358	664	1755	1678
665	1268	1268	466	1564	1432
555	1206	1206	566	1515	1358
656	1039	1039	265	1515	5
556	332	332	606	1515	2

From the above result, it can be concluded that with larger width, the algorithm estimates the exact results as compared to the original data. The top-10 3-grams of the original stream matches with the estimated top-10 3-grams with the exact frequencies, hence, producing no errors. But this increases the space required by the sketch. However, if the width is decreased, like in the second result, not only the top-10 3-gram list's elements vary from the original data but the frequencies are also estimated incorrectly. The main reason behind this is that larger width causes the matrix to cause less errors. Thus, from all this analysis one can deduce that if correct tuning of the parameters is done in count-min sketch algrotihm, it produces almost 100% accuracy with some compromises in the memory requirements.

It can also be observed from the above implementations that the run-time and frequency estimation of count-min sketch algorithm is better than the space saving algorithm.

0.1.8 Min-wise locality sensitive hashing task - Sudharshan Swaminathan

We find all the unique IP connection pairs, i.e if all netflows having unique A-B, B-A pairs are found and put together. For each connection pair found, we then take the values of the encoded attribute and form 3-grams from that column. This 3-gram formation is done for each unique pair

and therefore, is right padded in case there are less than 3-grams left in the end. `ip_pairs` is a dictionary containing all unique IP pairs and their corresponding list of encoded values. `n_grams` contains all the possible 3-grams computed for all connection pairs. `ip_pair_grams` consists of all unique connection pairs and their corresponding 3-grams.

We use the above formed 3-grams as the rows and the connection pairs as the columns for creating profiles. For each possible 3-gram formed, we check whether the 3-gram exists for the particular connection pair or not. If it exists, we place a `1` and a `0` otherwise. Doing this gives us a binary matrix that will let us calculate the jaccard similarity between 2 connection pairs. This matrix is stored in `connection_profile`.

We use the above matrix to calculate the pair-wise similarity between connections using jaccard similarity. We have 557 unique connection pairs. Here we calculate create a 2 dimensional matrix of dimensions of 557 * 557, and if the similarity between any columns is > 0.8, then we record a `1` in the cell corresponding to these 2 columns. For example, if the jaccard similarity, sim(C1,c2) $>$ 0.8, then we record a `1` in (C1,C2) in this new matrix represented by `similarity_matrix1` below. We also measure the average time it takes to find the pairwise jaccard similarities between all the connection pairs over 10 runs, which is approximately 160 seconds.

The hash functions are then created randomly and are used as was defined in the lectures. That is, every row's number is hashed and for each connection pair having a `1` in its connection profile, the minimum between the hash value and the already present value is taken. The subsequently formed signature matrix is present in `signature_matrix`.

We then use the above signature matrix to find the connection pairs that are similar to each other. We do this by splitting all the rows in the signature matrix is bands(`b`) with each band consisting of a specific number of rows(`r`). So if there are 30 hashes, we can have 3 bands of 10 rows each. So for 2 columns of the signature matrix S1 and S2, we check for how many rows in each band are equal. For even 1 band, if the number of rows equal is more than the threshold, then those 2 signatures are considered equal and the signature similarity matrix will have a `1` corresponding to cell (S1,S2). In this case, we have set the threshold to 1.0 which means we consider 2 signatures equal only when the entire columns are equal.

We experiment with different number of `b` and `r` combinations and choose the one with the least number of false positives/ false negatives. So we choose `b` $= 3$ and `r` $= 10$ as given in the results below (note that we get different results with different hashes with every run, and (3,10) gave the best results in most of these runs). The similarity of such a signature matrix is the closest to the jaccard similarity. On measuring the average time it takes to compute the similarity matrix using signatures over 10 runs, we observe that it takes approximately 70 seconds. This is the less than half the running time of pairwise jaccard similarities, and thus for enormous datasets will give us a significant advantage in terms of runtime.

Now we can use this matrix to calculate the number of bins into which the connection pairs have been clustered, and this is done by recording the cells in which we have a `1`. That is, if the cell (S1, S2) and (S1, S6) consists of a `1` , we can cluster (S1, S2, S6) in one bin. We count the number of such bins created from the signature's similarity matrix which is represented by `bin_count`. For this run of the program, we have approximately 60 bins as seen below (again changes per run).

0.1.9 Random hyperplane locality sensitive hashing task - Sudharshan Swaminathan

For random hyperplane LSH, we first compute the count of each n-gram per connection pair. This is given in the dataframe `connection_profile_counts`. We then create `n` number of random hyperplanes using the `np.random.randn` function which gives us a list of unsigned floats that are used as random vectors. Then we take a dot product of each random vector with each connection's n-gram counts. If the resultant dot product is >=0, we assign it a `1` and `0` otherwise. Therefore, for each connection we get a set of `n` values of 1s and 0s which serves as its "hash". We then cluster all the connection pairs having the same "hash" together and store it in the variable `hash_clusters`.

We then compute 2 types of pairwise euclidean distance matrices. The first type is derived by computing the euclidean distances between all the pair of connections. This takes approximately 30 seconds of run time on an average over 10 runs of the program. The second type is when we use the random hyperplane LSH to create clusters and compute the pairwise euclidean distances only for the connections belonging to the same cluster. Since we compute less number of euclidean distances in each case now, it takes only 15 seconds on an average which is approximateley half of run time of the pairwise euclidean matrix.

We also see that the number of bins obtained here is nothing but the number of clusters obtained after hashing. We get 20 bins in the case of chosing 5 random hyperplanes.

For pairwise euclidean matrices, we only take 2 columns to be similar when the distance between them is <=1. We compute the second similarity matrix depending on the hash clusters formed. On calculating the confusion matrix with the pairwise distances as the actual similarities and the similarities computed through LSH as the predicted ones, we get the result for different number of random hyperplanes used with different distance thresholds for euclidean distances. We see that using threshold of 1 and 5 hyperplanes gives a relative better balance for false positives and false negatives. Therefore, we use these parameters when showing the final similarity matrices in the end.

However, we can also conclude here that using Euclidean distances does not perform well in high dimensional feature space as it does not remain straightforward to set a distance threshold due to many constant distances. Therefore computing the similarity on the basis of euclidean distances might not prove to be a good measure for the same.

0.1.10 Botnet profiling task

We performed the botnet profiling task on all scenarios based on n-gram model. We first combined all the scenarios data into 1 dataframe and found the discretised "code" values (as calculated in the task 1) for the entire dataset. We then defined a funtion to return a code value list (state list) of a host, in a given time frame based on the StartTime. Considering the state list, we perform functions to calculate the 3-grams and sort them. The next we define a function to calculate distance between any two given profiles. Then we present a profile matching function which compares the profiles (3-grams) of a test host with predefined normal and infected profiles and calculate the respective distance of a test host from these two profiles. Further we define training profiles of a normal host and an infected host (both belong to scenario 9). Then we prepare the test data list with all the hosts from scenario 9 and track their test labels to evaluate the performance (to keep the computation at minimum level we took 50-50 cases of both 0 and 1 label). At last we compare the test profiles with our two training profiles and represent its distance from both of them, and based on this distance, we judge whether a test host is infected or normal. The results show 50

true positives, 30 false positives and 20 true negatives with an accuracy of 62.5%.

0.1.11 Botnet fingerprinting task

After fetching the profiles from the previous task using one infected host, we now make the n-grams for all the hosts and netflows. (we only use scenario 11 keeping in mind the processing overhead for taking all the scenarios).

Using the above ngrams for all the netflows in scenario 11, we check the occurence of an ngram from the infected profile and if one occurs we raise an alarm and predict it to be botnet traffic. Since we also keep the Label column in an encoded form (Botnet = 1, and 0 otherwise), we use this to compare our prediction simultaneosuly to record false positives and true positives, and true negatives. Since we have a huge class imbalance, it can be seen from the below results that we have a huge number of true negatives showing representing the benign netflows.

We see that fingerprinting is more effective than profiling. While fingerprinting gives 100% accuracy, Profiling manages only 60% accuracy, making fingerprinting more efficient with regards to eliminating false positives.