Final Paper for the Anomaly Detection Course

Tănasă Florin Petrișor

Computer Science Department, University of Bucharest, Romania

Abstract

The **DDoS** detection field has been struggling to keep up with new innovations from the offensive part. Implementing a solution is subjective, because some arhitectures require complex nodes to be setup and redesign the network, offering extremely good results, while others only require a processing node to detect anomalous behaviour and report it.

The good thing about this is that by trying all sorts of stuff from deep learning to graph theory, we are bound to find the right solution.

1 Introduction

DDoS (Distributed Denial of Service) is a type of attack that aims to do exactly what it says, deny access. While not getting out secrets, or leaking data, its job its to keep busy the servers in order to block legitimate requests, causing companies to lose traffic which translates into business lost, harming the site's owner.

DDoS is the improvement of **DoS** (Denial of Service). The innovation with it was that instead of one location overloading the server, we will now have a bunch of machines that run at the same time, leading to a distributed **DoS** (**DDoS**).

The basis of this paper is firstly based on a survey on a range of solutions and datasets for both supervised and unsupervised methods [2]. After choosing the unsupervised methods, one of the papers caught my eye and I decided to implement its version of a modified **DBSCAN** [1] (Algorithm 1).

As seen from the pseudocode algorithm, the modified **DBSCAN** provided by the authors also uses an expand function (Algorithm 2) used for adding points to different clusters.

Algorithm 1: [1] Modified DBSCAN $(D, \epsilon, MinPts)$

Input: Training dataset D, neighborhood radius ϵ, density threshold $MinPts$

Output: Cluster labels and centroid points μ_k

Label all data $x \in D$ as `UNCLASSIFIED`
Initialize cluster counter $cid \leftarrow 0$
foreach $x \in D$ **do**
    if x *is* `UNCLASSIFIED` **then**
        if *Expand*$(D, x, cid, \epsilon, MinPts)$ **then**
            $cid \leftarrow cid + 1$
        end
    end
end
foreach *cluster* k **do**
    Compute centroid $\mu_k = \frac{1}{n_k} \sum_{i=1}^{n_k} x_i$
    // n_k is the number of points in cluster C_k
end
return Set of μ_k

2 DBSCAN [3]

Data Mining is an important aspect of data analysis techniques. It's role is to find hidden and important patterns from large datasets. Clustering techniques are important when it comes to extracting features from data collected from all over the place.

"**DBSCAN** is the first density-based clustering algorithm. It was proposed by Ester et al. in 1996, and it was designed to cluster data of arbitrary shapes in the presence of noise in spatial and non-spatial high-dimensional databases. The key idea of **DBSCAN** is that for each object of a cluster, the neighborhood of a given radius (Eps) has to contain at least a minimum number of objects ($MinPts$), which means that the cardinality of the neighborhood has to exceed some threshold" [3].

The ϵ-neighborhood of an arbitrary point p is defined as:

$$N_{Eps}(p) = \{q \in D \mid \text{dist}(p, q) \le Eps\} \text{ [3]}$$

- **D**: the objects
- ϵ: threshold defining the minimum amount of points to create a centroid, which is defined as:

$$|N_{Eps}(p)| \ge MinPts \text{ [3]}$$

- **Eps**: radius of the neighbourhood
- **MinPts**: minimum number of points for a centroid

```
Algorithm: DBSCAN (D, Eps, MinPts)
// All objects in D are unclassified.
Begin
FOR ALL objects o in D DO:
If o is unclassified
        Call function expand_ cluster to construct a
cluster wrt. Eps and MinPts containing o.
End

FUNCTION expand_ cluster (o, D, Eps, MinPts)
Begin
      Retrieve the Eps-neighborhood (o) of o;
IF | N_Eps (o) | < MinPts      //i.e. o is not a core object
        Mark o as noise point and RETURN;
ELSE            // i.e. o is a core object
        Select a new cluster- id and mark all objects
in N_Eps (o) with
This current cluster-id
    Push all objects from N_Eps (o)| (o) onto the
 Stack seeds;
    WHILE NOT seeds.empty () DO
CurrentObject: = seeds.top ();
Retrieve the Eps-neighborhood
N_Eps (CurrentObject)  of CurrentObject;
   IF| N_Eps (CurrentObject) | ≥ MinPts.
        Select all objects in N_Eps (currentObject) not
yet classified or are marked as noise,
        Push the unclassified objects onto seeds and
mark all of these objects with current
 Current-id;
Seeds. Pop ();
RETURN
End
```

Figure 1: Pseudocode of DBSCAN [3]

3 Arhitecture of the proposed solution

The original image describing the arhitecture is Figure 2.

3.1 Preprocessing and Feature Extraction

As the first step, data is splited in **sampling windows** (SW) at one second interval, in order to create a new record **R** that holds the entropy computed for each SW of the data. With this new data set **D** made from **R**'s, we will normalize it and split into 2/3 for training and 1/3 for testing.

3.2 Training

With the subset of **D** with the length 2/3 of the initial, we start training the modified **DBSCAN** mentioned earlier.

3.3 Testing

The remaining 1/3 is now used for testing and a final model evaluation.

4 Implemented solution

The motivation behind implementing this paper was finding a lightweight and reliable solution that could trigger an alert to then be analyzed by someone. I'm currently working in Adobe Managed Services, that is the team of Adobe responsible with managing different client's machines that hosted in the cloud. Currently there has been an interest shown in developing a **DDoS** detection mechanism that will alert Customer Succes Engineers (CSE) tasked with the company that there might be an active attack.

My idea is deploying the machine with this fast algorithm that will cluster logs and determine if the last hour presented anomalous behaviour, to then ping the CSE in order to investigate and take the appropiate actions to remediate.

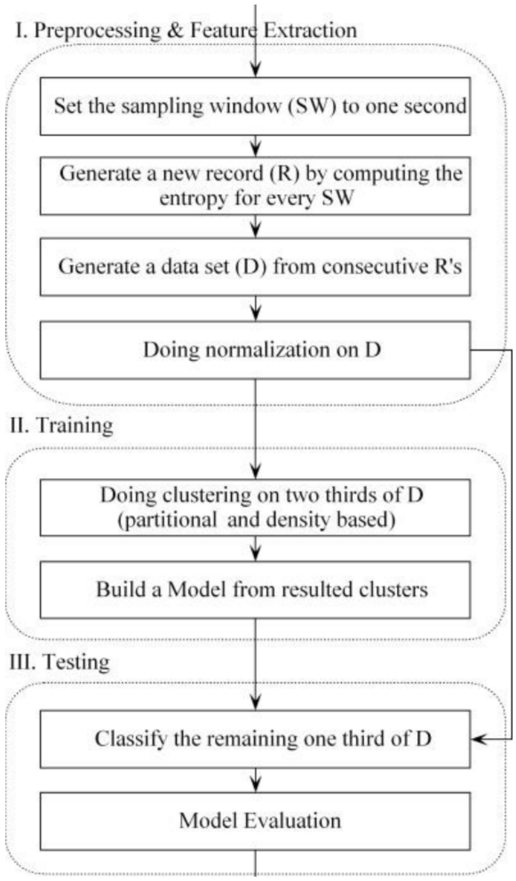


Figure 2: The phases of the proposed system [1]

4.1 Dataset

It consists of normal access logs, that we're captured during a **DDoS** attack a client had on 10th of January starting 8 am UTC. Them being access logs, they have available the following fields: the client's ip address, identity, user, timestamp, http request, status code, response size, referer, user-agent.

4.2 Data processing and feature extraction

After splitting the data and storing it in an array accordingly, the next step it's to exact the features from it. The following data is being extracted:

- **Ip**
- **Timestamp**: used in order to use the sampling window mentioned of one second
- **Method**
- **Path**
- **Path length**: the length of Path
- **Path depth**: the amount of "/" found inside the path, to determine how in-depth the request is
- **Encoded characters**: counts the number of conversions that we're made inside the url by counting "%"
- **Status**
- **Size**
- **Response size category**: either small, medium or large

As the next step of the arthitecture says, its now time to calculate the entropy for each sampling window to generate de final data set **D**. In preparation for this, we **one hot encode** string type parameters (ip, method, path, response size category) and normalize numerical data (status, size, path length, path depth, encoded characters) to feed into a **Principal Component Algorithm** (PCA) that returns only 3 values, reducing the dimensionality of the data. We then go window by window and call the function that will calculate the entropy.

4.3 Training the modified DBSCAN

Implementing the algorithm inside the paper was quite challenging, and in the end involved the creation of a **ModifiedDBSCAN** class that was able to train based on the proposed algorithm and a predict function that gives the label based on the closest centroid to the point given.

4.4 Testing phase

Unfortunately, the dataset being straight from the machine, there are no true labels that we can rely on in order to calculate accurate metrics. We can tell if there is some attack going on by seeing the proportions each cluster has. If during a normal period of time the requests are split more evenly throughout the clusters, an attack will most likely begin clustering in the same.

We can observe on the various images presented a comparison between 10th of January and 11th of January. We can see that on the 11 th most of the traffic was clustered in 1, while on the 10th besides the increase in its sheer number, we can see that its a lot more split up.

We know that there was an **DDoS** attack due to it being confirmed manually and a spike in the **Splunk** analysis (Figure 3).

4.5 Trial and error

Rereading the paper, I saw that for better results it was also added a **k-Means** algorithm before the training of **DBSCAN**. The scope was that **k-Means** will also add a new feature on the data by splitting it evenly throughout the clusters, and then feeding those labels to the **DBSCAN** in order for it to work on them directly. Also, specified in the paper we're the best parameters that the algorithm had, which are obviously in the code.

For better results, I also removed the server's internal requests. This lead to a significant change in the way things are clustered and eliminated unnecesarry logs, because they we're not helping in any way, they would have run even in the absence of a **DDoS**. Eliminated requests will be those initiated by an ip starting with 10.68, as this seems to be the server's local ip. To exemplify further, the type of request eliminated are:

- 10.68.X.X - replication-receiver 10/Jan/2025:00:00:05 -0500 "GET /bin/receive?sling:authRequestLogin=1 HTTP/1.1" 200 32 "-" "Jakarta Commons-HttpClient/3.1"
- 10.68.X.X - - 10/Jan/2025:00:00:11 -0500 "GET /content/ams/healthcheck/regent.html HTTP/1.1" 200 704 "-" "Ruby"
- 10.68.X.X - - 10/Jan/2025:00:00:12 -0500 "HEAD / HTTP/1.1" 200 - "-" "Apache-HttpClient/4.5.13 (Java/1.8.0_371)"

4.6 Results

This subsection will be dedicated to comparing results. As stated previously, there are no labels on the data, so we will have to just rely on comparing graphs and drawing conclusion. What we do know about the dataset, is that a spike in the requests happened at around 8 am.

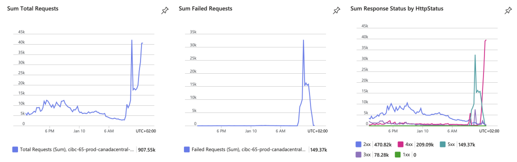


Figure 3: The spike analyzed by Splunk

The followings images will display the differences in the 10th January (Figure 4) traffic and 11th January (Figure 5).

As seen from the images, there is a clear difference. Besides the increased number of requests, they are also split between more clusters. The 10th of January graphs are full of dots all over the place, while on the 11th of January we only have one or two principal clusters where most of the requests fall under.

5 Comparison with other algorithms [1]

Having the work started by checking [2], I already had a comparison of the algorithm with other unsupervised methods, from which derived that using this was the best fit on the problem. Also, [1] also had a section specific for comparing their results (Table 1).

6 Conclusion

DDoS is a current and big threat to everyone having an online presence through a site. While many solutions have been found, none have achieved 100% and not all are applicable on any type of arhitecture.

By choosing a lightweight algorithm with consistent results, you can add it to a client's machine and get information about the current traffic, to then verify manually if an anomaly happened, respectively an **DDoS** attack is on-going.

It's a topic of cybersecurity that is constantly evolving and is a race between tools that are becoming so effective and detection mechanism that aim to block their attempts at denying the acces to the site to legitimate users

References

[1] Safaa O Al-mamory and Zahraa M Algelal. A modified dbscan clustering algorithm for proactive detection of ddos attacks. In *2017 Annual Conference on New Trends in Information & Communications Technology Applications (NTICT)*, pages 304–309. IEEE, 2017.

[2] Mohammad Najafimehr, Sajjad Zarifzadeh, and Seyedakbar Mostafavi. Ddos attacks and machine-learning-based detection methods: A survey and taxonomy. *Engineering Reports*, 5(12):e12697, 2023.

[3] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. Dbscan revisited, revisited: why and how you should (still) use dbscan. *ACM Transactions on Database Systems (TODS)*, 42(3):1–21, 2017.

Algorithm 2: [1] Expand $(D, x, cid, \epsilon, MinPts)$

Input: Dataset D, point $x \in D$, cluster id cid, neighborhood radius ϵ, density threshold $MinPts$

Output: `true` if a new cluster is found, otherwise `false`

$S \leftarrow \{y \in D \mid \|x - y\| \leq \epsilon\}$ `// Find neighborhood`
if $|S| < MinPts$ **then**
    Label x as `NOISE`
    return `false`
end
foreach $x' \in S$ **do**
    Label x' with cluster id cid
    Remove x' from S
end
foreach $x' \in S$ **do**
    $T \leftarrow \{y \in D \mid \|x' - y\| \leq \epsilon\}$ `// Expand cluster`
    if $|T| \geq MinPts$ **then**
        foreach $y \in T$ **do**
            if *y is UNCLASSIFIED or NOISE* **then**
                Label y with cluster id cid
                if *y is UNCLASSIFIED* **then**
                    Insert y into S
                end
            end
        end
    end
    Remove x' from S
end
return `true`

Algorithm 3: Extract Features from Parsed Logs

Input: parsed_logs: List of log dictionaries
Output: DataFrame containing extracted features

```
features ← empty list;
for log in parsed_logs do
    timestamp ← Convert log['timestamp'] using datetime format "%d/%b/%Y:%H:%M:%S";
    ip ← log['ip'];
    method ← log['method'];
    path ← log['path'];
    status ← Integer value of log['status'];
    size ← Integer value of log['size'];
    size ← 0;
    path_length ← length of path;
    path_depth ← count of "/" in path;
    encoded_characters ← count of "%" in path;
    if size < 1000 then
        response_size_category ← "small";
    else
        if size < 10000 then
            response_size_category ← "medium";
        else
            response_size_category ← "large";
    Append dictionary {ip, timestamp, method, path, path_length, path_depth, encoded_characters, status, size, response_size_category} to features;
return DataFrame from features;
```

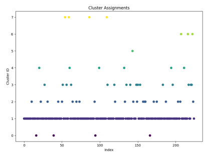
Algorithm 4: Compute Entropy Feature

Input: features_df: DataFrame containing extracted features
Output: Array of entropy values per time window

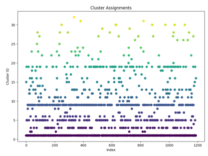
```
Convert features_df['timestamp'] to datetime format (unit = seconds);
Extract categorical features: {'ip', 'method', 'path', 'response_size_category'};
Initialize OneHotEncoder;
encoded_categorical ← Encode categorical features;
Extract numerical features: {'status', 'size', 'path_length', 'path_depth', 'encoded_characters'};
Initialize StandardScaler;
scaled_numerical ← Normalize numerical features;
Combine encoded_categorical and scaled_numerical using horizontal stacking;
Initialize PCA with 3 components;
pca_features ← Apply PCA transformation;
Create pca_df with columns {'pca1', 'pca2', 'pca3'} and include timestamp;
Resample pca_df at 1-second intervals using timestamp and apply compute_entropy_for_window function;
entropy_per_window ← Reset index and rename column to 'entropy';
return entropy_per_window as a NumPy array;
```

Table 1: Comparison of Algorithms [1]

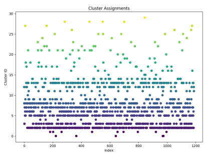
Algorithm	Accuracy (%)	DR (%)	FA (%)
Hierarch. clust. [13]	-	-	-
MGKM [12]	45.83	1.16	54.36
K-NN [24]	91.89	-	-
TCM-KNN [11]	99.7	-	0.2
CRF [8]	95.0	- -	
Proposed system	98.89	52.17	0.68



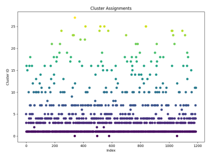
(a) 8 am



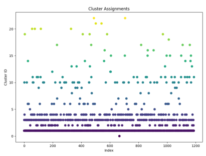
(b) 9 am



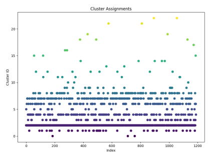
(c) 10 am



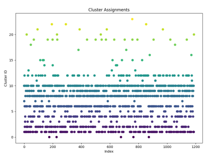
(d) 11 am



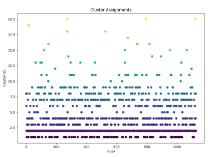
(e) 12 am



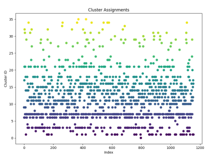
(f) 1 pm



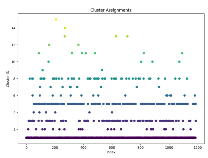
(g) 2 pm



(h) 3 pm



(i) 4 pm

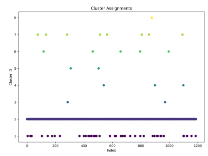


(j) 5 pm

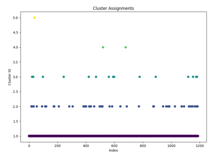
Figure 4: Traffic on 10th of January 8 am to 6 pm.

(a) 8 am

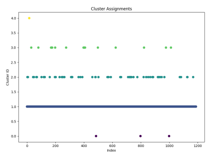
(b) 9 am



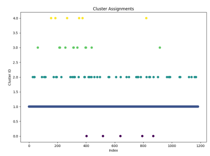
(c) 10 am



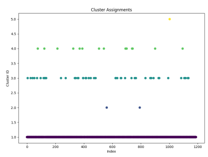
(d) 11 am



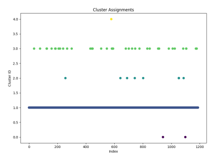
(e) 12 am



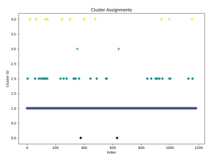
(f) 1 pm



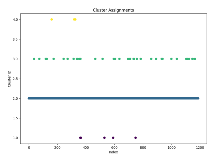
(g) 2 pm



(h) 3 pm



(i) 4 pm



(j) 5 pm

Figure 5: Traffic on 11th of January 8 am to 6 pm.