DNS Traffic Analysis & Attack Detection

Logan BARRÉ, Diadié CAMARA, Ziyad BRIAND
Project for the ESILV, Machine learning class

November 29, 2025

ESILV

Contents

1 Introduction (Logan)

1.1 Business Case :

The domain of network security is constantly evolving, with attackers finding new ways to dodge the current defenses. This project focuses on the **Domain Name System (DNS)**, it translate in internet the domain names into IP addresses. Because DNS traffic is essential, it is rarely blocked by firewalls, a **blind spot for security**.

The issue is that the cybercriminals exploit this trust to perform **DNS Exfiltration** (smuggling stolen data out through DNS queries) or to launch **Denial of Service (DoS)** attacks. A DoS attack aims to surcharge a network with traffic, rendering it unavailable to legitimate users.

The Solution & Impact: By analyzing DNS traffic, we can directly reduce the probability and impact of these attacks or block them before happening for example :

- **Preventing Exfiltration:** Detecting abnormally heavy DNS queries allows us to stop data theft in progress.
- **Mitigating DoS:** Malicious DNS tunneling often consumes massive bandwidth. Identifying and filtering this "Heavy" malicious traffic early prevents the network saturation that causes a Denial of Service.

Also in some cases a benign content but to heavy can cause a Denial of Service too, then this tools can help also to not block but to reduce the number of indirect Denial of service due to benign traffic

This project addresses the critical need for an automated system that can monitor this traffic in real-time.

TO do that we have found the **CIC-Bell-DNS-EXF-2021**, the CIC is the Canadian institute of cybersecurity they have collected for days a true DNS traffic and simulated an Attack traffic too and the file is more than 270 MB from https://www.unb.ca/cic/datasets/dns-exf-2021.html

Their Goal was to do CTI (Cyber Threat Intelligence) that mean using way to detect threat and learn from it

1.2 Objective

The primary goal of this project is to develop a **Machine Learning classifier** capable of analyzing DNS query patterns to predict whether a specific traffic flow is **Malicious or Benign and it's size**.

Using the **CIC-Bell-DNS-EXF-2021** dataset, we aim to categorize traffic into four distinct levels:

Two categories each having a Heavy and Light possibility :

- **Benign:** Legitimate traffic
- **Malicious:** Attack traffic

The ultimate objective is to provide a reliable detection tool that maximizes the discovery of threats (Recall) while minimizing false alarms. We aim to achieve high accuracy and a big F1-score, ensuring that normal business operations are not disrupted by false positives.

Regarding the division of labor, Logan and Diadié initially shared the data exploration and visualization tasks. However, after encountering significant obstacles with the initial dataset, Logan made the strategic decision to restart the process. He subsequently handled the entire pipeline from the beginning up to Step 5 (the initial model implementation).

2 Data exploration & initial Obstacle (Logan)

First lets talk about our initial obstacle at first we wanted to use a dataset to classify if a DNS request was a spam https://www.unb.ca/cic/datasets/dns-2021.html and it was looking like that :

Country	ASN	TTL	IP	Domain	sld	State	typos	oc_32	len	puny_coded	shortened	dec_32	Registrant_Name	3
JP	2516	59	119.82.155.98	b'0900259.com.'	900259	Tokyo	['qq.com'	86)	('vk.com'	86)]	0	8	0	
JP	2519	3599	116.91.115.130	b'koitera.net.'	koitera	Tokyo	['php.net'	86)	('a8.net'	86)]	0	8	0	
JP	2519	3599	116.91.115.130	b'koitera.com.'	koitera	Tokyo	['qq.com'	86)	('vk.com'	86)]	0	8	0	
JP	2519	3599	116.91.115.130	b'pc.koitera.com.'	koitera	Tokyo	['qq.com'	86)	('vk.com'	86)]	0	10	0	
JP	2516	59	119.82.155.97	b'0901360.com.'	901360	Tokyo	['qq.com'	86)	('vk.com'	86)]	0	8	0	
JP	2519	3599	116.91.115.130	b'koitera.net.'	koitera	Tokyo	['php.net'	86)	('a8.net'	86)]	0	8	0	
nan	nan	3595	116.91.115.130	b'koitera.com.'	koitera	Tokyo	['qq.com'	86)	('vk.com'	86)]	0	8	0	
nan	nan	3599	116.91.115.130	b'pc.koitera.com.'	koitera	Tokyo	['qq.com'	86)	('vk.com'	86)]	0	10	0	
JP	2516	59	119.82.155.96	b'0902471.com.'	902471	Tokyo	['qq.com'	86)	('vk.com'	86)]	0	8	0	
JP	2519	3591	116.91.115.130	b'koitera.net.'	koitera	Tokyo	['php.net'	86)	('a8.net'	86)]	0	8	0	
nan	nan	3592	116.91.115.130	b'koitera.com.'	koitera	Tokyo	['qq.com'	86)	('vk.com'	86)]	0	8	0	
[redacted]	2519	3595	116.91.115.130	b'pc.koitera.com.'	koitera	Tokyo	['qq.com'	86)	('vk.com'	86)]	0	10	0	
JP	2516	59	119.82.155.95	b'0903582.com.'	903582	Tokyo	['qq.com'	86)	('vk.com'	86)]	0	8	0	
JP	2519	3592	116.91.115.130	b'koitera.net.'	koitera	Tokyo	['php.net'	86)	('a8.net'	86)]	0	8	0	
JP	2519	3589	116.91.115.130	b'koitera.com.'	koitera	Tokyo	['qq.com'	86)	('vk.com'	86)]	0	8	0	
JP	2519	3589	116.91.115.130	b'pc.koitera.com.'	koitera	Tokyo	['qq.com'	86)	('vk.com'	86)]	0	10	0	
JP	2516	59	119.82.155.104	b'0904693.com.'	904693	Tokyo	['qq.com'	86)	('vk.com'	86)]	0	8	0	
JP	2519	3584	116.91.115.130	b'koitera.net.'	koitera	Tokyo	['php.net'	86)	('a8.net'	86)]	0	8	0	
JP	2519	3599	116.91.115.130	b'koitera.com.'	koitera	Tokyo	['qq.com'	86)	('vk.com'	86)]	0	8	0	
nan	nan	3595	116.91.115.130	b'pc.koitera.com.'	koitera	Tokyo	['qq.com'	86)	('vk.com'	86)]	0	10	0	
JP	2516	59	119.82.155.103	b'0905704.com.'	905704	Tokyo	['qq.com'	86)	('vk.com'	86)]	0	8	0	
nan	nan	3585	116.91.115.130	b'koitera.net.'	koitera	Tokyo	['php.net'	86)	('a8.net'	86)]	0	8	0	
JP	2519	3595	116.91.115.130	b'koitera.com.'	koitera	Tokyo	['qq.com'	86)	('vk.com'	86)]	0	8	0	
JP	2519	3599	116.91.115.130	b'pc.koitera.com.'	koitera	Tokyo	['qq.com'	86)	('vk.com'	86)]	0	10	0	

Figure 1: Aperçu initial du dataset CIC-DNS-2021, structure non cohérente.

The CSV structure was highly inconsistent; many fields contained nested lists (e.g., {}) or irregular delimiters, which made standard parsing with Regex or Pandas ineffective. After extensive attempts to clean and concatenate the data, we realized that forcing this dataset to work would result in excessive information loss. Consequently, we pivoted to the **CIC-Bell-DNS-EXF-2021** dataset, which is cleaner and more relevant to our exfiltration objective.

Now lets start the explanation of our project on this new dataset :

We have followed the course tips then we have started with these steps :

Step 2 :

1. Descriptive analysis of your data.
2. Implementation of the necessary pre-processing.
3. Formalisation of the problem.
4. Selection of a baseline model and implementation of the model.

Given the volume of the data, we utilized Google Drive for efficient storage and retrieval within the Google Colab environment. The original data was in Parquet format, which we retained for its efficiency with Pandas.

Initially, we assumed the dataset files were separated by class (one file for "Attacks" and one for "Benign"). However, upon inspection, we discovered that the files named `heavy.parquet` and `light.parquet` **already contained both** Attack and Benign traffic mixed together. Therefore, the separate `benign.parquet` file was redundant. To avoid data duplication and unnecessary processing overhead, we excluded the standalone benign file and focused solely on the Heavy and Light files.

Code Python (Extraction) :

```
# File IDs from Google Drive
file_ids = {
#~"benign": "1dTmkbkiG60liBK1WS61SOk8p63nPtd9A", explained in the 3 part why
    we don't use benign anymore
"heavy": "1xxj7U0k90LrU8jv7gE-YHQQwa9i7Bn10",
"light": "1xqiHziNn63hcVDcp7WCzTpe6ioXCA5"
}
```

```
# Download files and load directly as Parquet
dfs = {}
for name, fid in file_ids.items():
    output_path = f"{name}.parquet"
    print(f"Downloading {name}...")
    gdown.download(f"https://drive.google.com/uc?id={fid}", output_path,
        quiet=False)

# Load parquet file
dfs[name] = pd.read_parquet(output_path, engine="pyarrow")
print(f"Loaded {name} --- shape: {dfs[name].shape}")

# Assign variables
# df_benign = dfs["benign"] # explained in the 3 part why we don't use benign
    anymore
df_heavy = dfs["heavy"]
df_light = dfs["light"]

# Quick check
print("\nBenign sample preview:")
print(df_heavy.head())
```

2.1 Statistical Analysis & Class Distribution

We concatenated the dataframes to analyze the global distribution. The dataset contains **44 columns**, divided into "Stateless" features (single packet properties) and "Stateful" features (aggregated flow properties).

Code Python (Analyse Statistique) :

```
df_all = pd.concat([df_heavy, df_light], ignore_index=True)
print("Shape of merged dataset:", df_all.shape)
print("Class distribution:\n", df_all['GlobalClass'].value_counts())
print("Column types:\n", df_all.dtypes)
print("First row preview:\n", df_all.iloc[[0]])
```

Class distribution:

```
GlobalClass
Heavy Attack    251670
Heavy Benign    181694
Light Benign     60091
Light Attack     42683
Name: count, dtype: int64
```

Observed Class Distribution:

- **Heavy Attack:** 251,670
- **Heavy Benign:** 181,694
- **Light Benign:** 60,091
- **Light Attack:** 42,683

This confirmed a significant class imbalance, particularly for the "**Light Attack**" class, which would later require specific handling .

We observed a distinct disparity in data completeness: half of the columns were not so populated (mostly null values), while the remaining columns were fully populated. The dataset distinguishes between two specific types of features:

The Stateless Category: These are features extracted from a single DNS packet in isolation, without regard to previous traffic (e.g., the length of the query or the specific record type).

The Stateful Category: These are features calculated based on the history or context of the connection over a window of time (e.g., the frequency of requests or the variance in Time-To-Live values).

After we need to know what columns are meaning so we explored first each columns with the types object to know what they mean and which will be removed or need a labeling

Code Python (Exploration Catégorielle) :

```
categorical_columns = [
'GlobalClass', 'SubClass', 'rr_type', 'distinct_ip', 'unique_country',
'unique_asn', 'distinct_domains', 'reverse_dns', 'unique_ttl',
'longest_word', 'sld'
]

for col in categorical_columns:
    print(f"-/-/- Column: {col} -/-/-")
    unique_count = df_all[col].nunique()
    print(f"Number of unique values: {unique_count}")
    if unique_count > 20:
        print(f"Column '{col}' has a large number of unique values. Showing
            top 5 most frequent:")
        print(df_all[col].value_counts().head(5))
    else:
        print("Value counts:")
        print(df_all[col].value_counts())
    print("\n")
```

2.2 Exploration by visualization :

We wanted to explore the rest of the numerical data by visualization as it's not text it's hard to understand without visualization, then we created this code :

```
numerical_columns = df_all.select_dtypes(include=['float32', 'int8']).columns
# Code for plotting Histograms and Boxplots follows
```

But in our case it was not very useful lot of useless graphics like that appears :

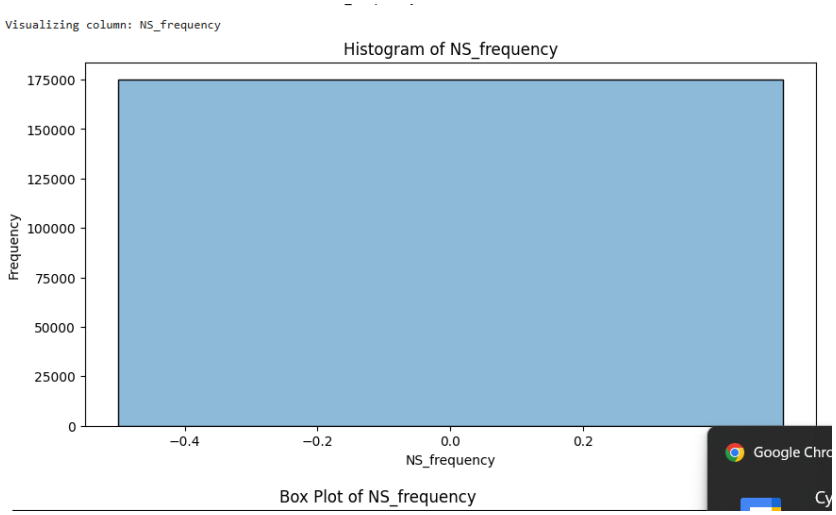


Figure 2: Exemple de graphique non pertinent (Histogramme).

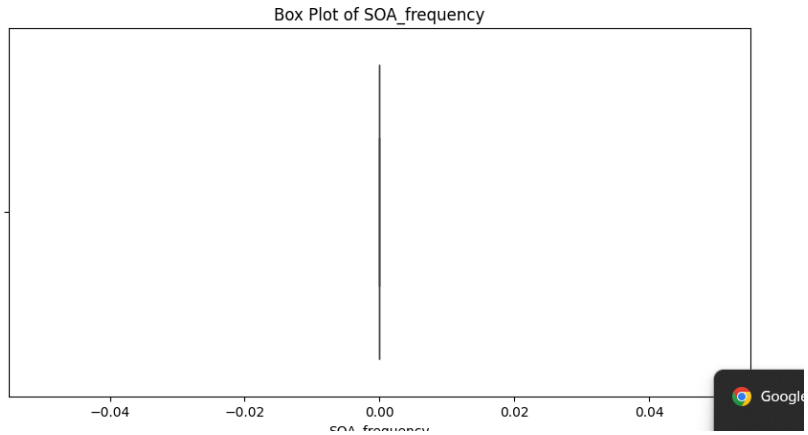


Figure 3: Exemple de graphique non pertinent (Boxplot).

We determined that a comprehensive visualization would be more effective following the pre-processing stage. To better understand the dataset's behavior and rigorously identify non-informative features, we proceeded to analyze the statistical relationships and correlations between the variables.

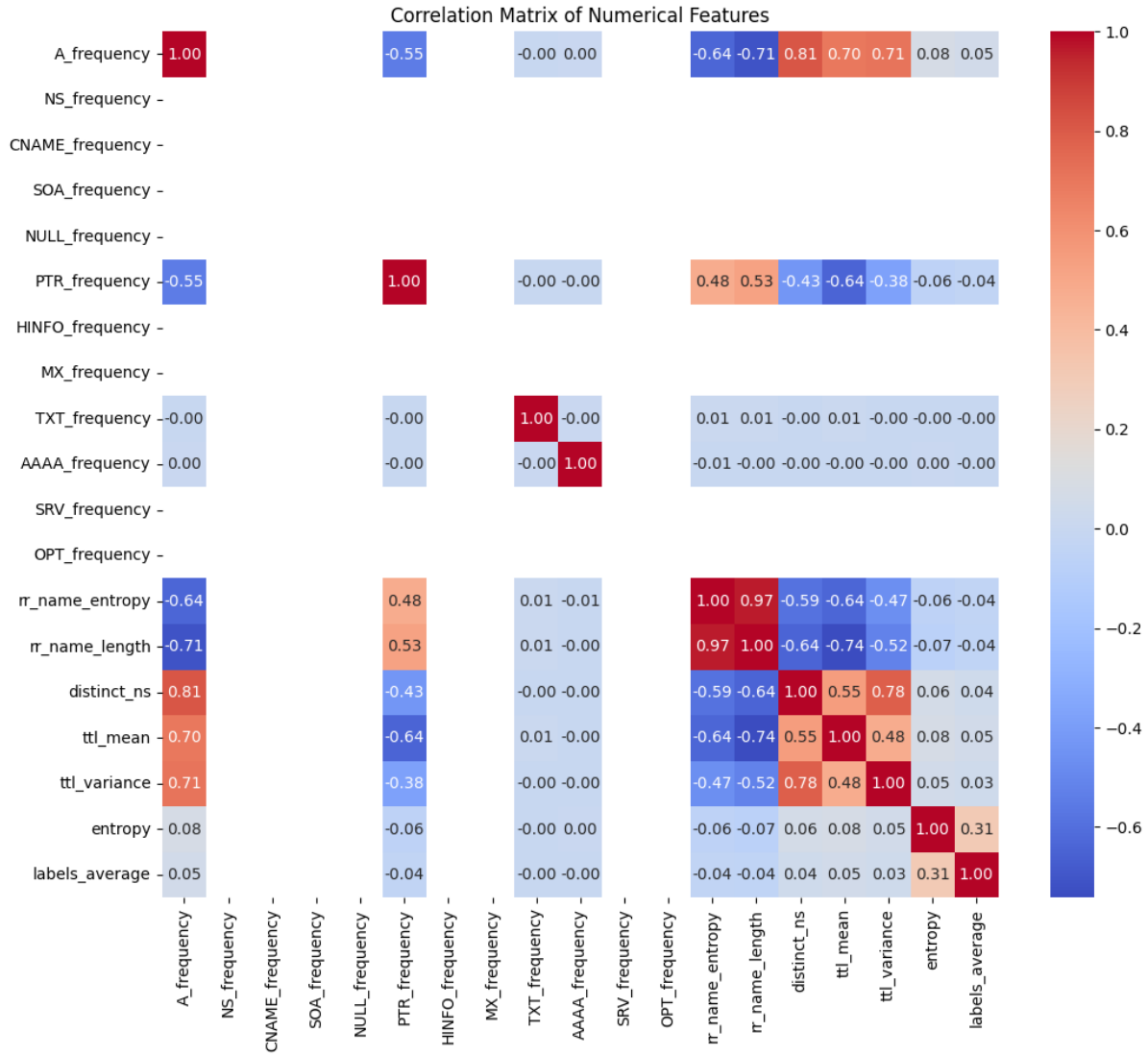


Figure 4: Matrice de corrélation initiale.

We observed that certain clusters of features (e.g., `distinct_ns`, `ttl_mean`, `ttl_variance`) were highly correlated, as were `rr_name_length` and `entropy` (logically, longer strings tend to have higher entropy).

Feature Selection Table To rigorously select features, we created the following analysis table to decide which columns to keep, drop, or engineer:

Column	Explanation	Relevance for Classification	Decision
rr	Rate of response per second.	Attack traffic might have different response rates.	Keep (handle missing values)
A_frequency	Frequency of A records.	Can indicate unusual DNS traffic patterns.	Keep
NS_frequency	Frequency of NS records.	Changes in NS record frequency may indicate attacks.	Keep
CNAME_frequency	Frequency of CNAME records.	Can indicate redirection or evasion techniques.	Keep
SOA_frequency	Frequency of SOA records.	Unusual SOA record frequency could indicate anomalies.	Keep
NULL_frequency	Frequency of NULL records.	Sometimes used in attacks.	Keep

Column	Explanation	Relevance for Classification	Decision
PTR_frequency	Frequency of PTR records.	High frequency may indicate scanning/reconnaissance.	Keep
HINFO_frequency	Frequency of HINFO records.	Unusual requests could be malicious.	Keep
MX_frequency	Frequency of MX records.	Changes in MX frequency could be unusual.	Keep
TXT_frequency	Frequency of TXT records.	Can be used in data exfiltration.	Keep
AAAA_frequency	Frequency of AAAA records (IPv6).	Can indicate unusual IPv6 traffic patterns.	Keep
SRV_frequency	Frequency of SRV records.	Unusual SRV requests may indicate attacks.	Keep
OPT_frequency	Frequency of OPT records.	Used in certain attacks or exfiltration.	Keep
rr_type	Set of unique DNS record types in a flow.	Strong indicator of traffic type; manageable number of unique values.	Keep (requires encoding)
rr_count	Total number of resource records in a flow.	Attack traffic may have different counts.	Keep (handle missing values)
rr_name_entropy	Entropy of resource record names.	High entropy often indicates DGA/malicious domains.	Keep
rr_name_length	Length of resource record names.	Malicious domains may be unusually long/short.	Keep
distinct_ns	Number of distinct name servers.	Could vary between benign and attack traffic.	Keep
distinct_ip	Set of distinct IP addresses.	High cardinality and complex type.	Drop
unique_country	Set of countries associated with IPs.	High cardinality; complex type.	Drop
unique_asn	Set of Autonomous System Numbers associated with IPs.	High cardinality; complex type.	Drop
distinct_domains	Set of distinct domains.	Extremely high cardinality; not generalizable.	Drop
reverse_dns	Result of reverse DNS lookup.	Many unknown values; high cardinality.	Drop
a_records	Number of A records.	Similar to A_frequency.	Keep (handle missing values)
unique_ttl	Set of unique TTL values.	Anomalous TTL values may indicate attacks.	Keep (requires encoding)
ttl_mean	Mean TTL value.	Helps differentiate cached vs fresh traffic.	Keep
ttl_variance	Variance of TTL values.	High variance can indicate unusual behavior.	Keep
timestamp	Timestamp of the query.	Useful for feature engineering, not directly for models.	Drop raw column, engineer time-based features if needed
FQDN_count	Number of labels in the FQDN.	Malicious domains may have unusual structures.	Keep
subdomain_length	Length of the subdomain part of FQDN.	Long/short subdomains can indicate attacks.	Keep
upper	Number of uppercase characters in FQDN.	Encoded data or unusual naming.	Keep
lower	Number of lowercase characters in FQDN.	Encoded data or unusual naming.	Keep
numeric	Number of numeric characters in FQDN.	Useful for detecting DGAs or encoding.	Keep

Column	Explanation	Relevance for Classification	Decision
entropy	Entropy of FQDN string.	High entropy indicates DGAs or exfiltration.	Keep
special	Number of special characters in FQDN.	Encoded data or unusual naming.	Keep
labels	Number of labels in FQDN.	Redundant with FQDN_count.	Drop
labels_max	Maximum length of a label in FQDN.	Malicious domains may have long labels.	Keep
labels_average	Average label length in FQDN.	Useful for feature patterns.	Keep
longest_word	Length of the longest alphanumeric sequence in FQDN.	Can help detect DGAs or encoding.	Keep (requires encoding)
sld	Second-Level Domain (SLD).	High cardinality; direct domain info.	Drop
len	Total FQDN length.	Similar to rr_name_length.	Keep
subdomain	Number of subdomain labels (excluding SLD/TLD).	Deeply nested subdomains can be suspicious.	Keep
GlobalClass	Main traffic class (Benign/Attack types).	Target variable.	Keep (as target)
SubClass	More granular class info.	Implicitly contains target label (data leakage).	Drop

The analysis detailed in the table above was instrumental in guiding our feature selection process, allowing us to systematically distinguish between relevant signals and noise. We adhered to a strict logic where features containing unique values for every instance were immediately discarded, as they function merely as identifiers rather than generalizable patterns.

Similarly, columns with zero variance, where the value remained constant across the entire dataset, were removed as they provided no predictive power. A critical decision involved the exclusion of the "**SubClass**" feature; since this column implicitly contained the target label, retaining it would have introduced **data leakage**, effectively allowing the model to "cheat" by reading the answer rather than predicting it.

We also debated the inclusion of the "**Timestamp**" feature, acknowledging its potential for identifying time-based attack vectors, but ultimately decided to remove it to prevent the model from overfitting to specific dates rather than learning traffic behaviors. Finally, we addressed issues of redundancy by removing overlapping features, such as specific IP addresses and their corresponding Country codes, to reduce dimensionality. This rigorous selection process established a clean foundation, leading directly into the necessary pre-processing steps.

3 Pre Processing (Logan)

First to avoid doing everything again each time we wanted to try new processing feature, we have copy the `df` to `df_all`

We have also removed the columns we decided on the table :

```
df_all = df_all.drop(columns=['distinct_ip', 'distinct_domains',
'reverse_dns', 'labels', 'sld', 'timestamp', 'SubClass'])
```

after we looked again at the type. So we start by pre-processing the object types it's one type that is complicated to use still in "object" type so we need to do encoding to put them into integer or float

```
[ 'rr_type', 'unique_country', 'unique_asn', 'unique_ttl', 'longest_word', 'GlobalClass'
]
```

We start with `rr_type`

```
r_type
{'PTR'} 103337
{'A'} 65982
{None} 3662
set() 2121
{'TXT'} 4
{'AAAA', 'A'} 1
```

As we have only that amount of types we have simplified the types to have the same format (The `set()` and `None`) together as they represent the same things

After for `unique country` it was like that :

```
unique_country
set()              138417
{'US'}              18672
{'JP'}               3209
{'DE'}               2629
{'RU'}               1492
                     ...
{'ZM'}                  1
{'BM'}                  1
{'DO'}                  1
{'US', 'DE'}            1
{'RU', 'KZ'}            1
```

Figure 5: Valeurs uniques pour 'unique_country'.

So we have only taken the first country to be easier for the encoding after

For `unique_asn` it was almost good but we have done a cleaning in case we have both `set()` and `{}` to fuse them

```
# 'unique_ttl'
df_all.unique_ttl.value_counts()
# In the documentation it's writent as Distinct Time-to-Live (TTL) values in window τ.
```

```
unique_ttl
[1, 1]                                    48549
[1, 1, 1, 1]                              21382
[128, 122]                                20831
[128]                                     19639
[128, 128, 122, 122]                      16930
                                          ...
[128, 128, 128, 122, 122, 128, 122, 122]      1
[128, 128, 122, 128, 122, 128, 122]           1
[128, 122, 122, 122, 122]                     1
[255, 255, 255, 255]                          1
[64, 64, 128, 64, 128, 128, 64]               1
Name: count, Length: 76, dtype: int64
```

Figure 6: Nettoyage des valeurs de 'unique_asn'.

For this variable it was a bit complicated, so I thought It was better to only keep, the unique value of each as it was time to live, it's "useless" to have 5 time the same value

So at the end of the preprocessing loop we have that :

```
unique_ttl
[]              361031
[1]             103712
[128, 122]       38569
[128]            27123
[255]             3324
[64]              1464
[122]              914
[64, 128]            1
```

Figure 7: Résultats après le nettoyage de boucle.

For the `df_all.longest_word.value_counts()`

```
longest_word
2              221043
4              141783
N                9070
C                6026
9                3847
                 ...
skins               1
backpacking         1
eland               1
vagabond            1
acquire             1
Name: count, Length: 12274
```

Figure 8: Comptage des valeurs de 'longest_word'.

Normally it was the length of the words but we got full words here too not only the number of letters so we needed to convert all the strings into their value of lengths

After this preprocessing we have that :

```
longest_word
1     389666
4      30391
5      27671
3      26510
6      18406
2      14445
7      13149
8       7297
9       4923
10      2219
11       956
12       294
13       161
14        39
15         6
16         5
```

Figure 9: Types après traitement des variables catégorielles.

After dealing with the categorical values we have managed the missings values & columns with no variance (as for me no variance mean we have the exact same values everywhere then it's useless to bother the models with columns like that

```
Columns with missing values:
 rr                 361031
A_frequency         361031
NS_frequency        361031
CNAME_frequency     361031
SOA_frequency       361031
NULL_frequency      361031
PTR_frequency       361031
HINFO_frequency     361031
MX_frequency        361031
TXT_frequency       361031
AAAA_frequency      361031
SRV_frequency       361031
OPT_frequency       361031
rr_count            361031
rr_name_entropy     361031
rr_name_length      361031
distinct_ns         361031
a_records           361031
ttl_mean            361031
ttl_variance        361031
dtype: int64
Columns with no variance:
 ['NS_frequency', 'CNAME_frequency', 'SOA_frequency', 'NULL_frequency', 'HINFO_frequency', 'MX_frequency', 'SRV_frequency',
'OPT_frequency', 'a_records']
```

Figure 10: Colonnes avec zéro variance et valeurs manquantes.

So we dropped theses columns with no variance and after we dealt with the remaining columns with missing values

```
Columns with missing values:
 rr                 361031
A_frequency         361031
PTR_frequency       361031
TXT_frequency       361031
AAAA_frequency      361031
rr_count            361031
rr_name_entropy     361031
rr_name_length      361031
distinct_ns         361031
ttl_mean            361031
ttl_variance        361031
dtype: int64
```

Figure 11: Colonnes restantes avec valeurs manquantes.

After to know if we fullfill with 0, the mean values or remove this columns we needed to analyse each one of them, to see the full tought behind check the github with the commentary but here is a resume of the preprocessing for this part as it 's a bit long :

We distinguished 2 cases one where the 0 was already a big part of value for the columns then we putted 0 in theses cases. And the other we put the median values, but while writing this rapport I just see a mistake I have made (while writing the code I put `fillna(0)` where I wanted to put `fillna(median)`...

So maybe it will explain why we have bad accuracy later or at least will make the thing a bit worst

After this part of preprocessing we needed to convert everything in the same type :

```
# Now we have correct value we need to transf
df_all.dtypes
```

```
rr                  float32
A_frequency         float32
PTR_frequency       float32
TXT_frequency       float32
AAAA_frequency      float32
rr_type              object
rr_count            float32
rr_name_entropy     float32
rr_name_length      float32
distinct_ns         float32
unique_country       object
unique_asn           object
unique_ttl           object
ttl_mean            float32
ttl_variance        float32
FQDN_count             int8
subdomain_length       int8
upper                  int8
lower                  int8
numeric                int8
entropy             float32
special                int8
labels_max             int8
labels_average      float32
longest_word          int32
len                    int8
subdomain              int8
GlobalClass          object
```

Figure 12: Types avant l'encodage final.

SO we used encoding to convert the object into integer. And for that we used the `LabelEncoder` function from `skleanr-preprocessing`

So after the Encoding we have that :

```
rr                          float32
A_frequency                 float32
PTR_frequency               float32
TXT_frequency               float32
AAAA_frequency              float32
rr_type                       int64
rr_count                    float32
rr_name_entropy             float32
rr_name_length              float32
distinct_ns                 float32
ttl_mean                    float32
ttl_variance                float32
FQDN_count                     int8
subdomain_length               int8
upper                          int8
lower                          int8
numeric                        int8
entropy                     float32
special                        int8
labels_max                     int8
labels_average              float32
longest_word                  int32
len                            int8
subdomain                      int8
GlobalClass                  object
unique_country_encoded        int64
unique_asn_encoded            int64
unique_ttl_mean             float32
dtype: object
```

Figure 13: Types après l'encodage final.

The `GlobalClass` is the only "object" left that is normal as it will be our classification label variable. We converted everything into float 32 too to be easier to manage. And transformed the `GloalClass` as strings.

After this step we have redone a visual analysis and it was much better as for exemple the correlation matrix is full.

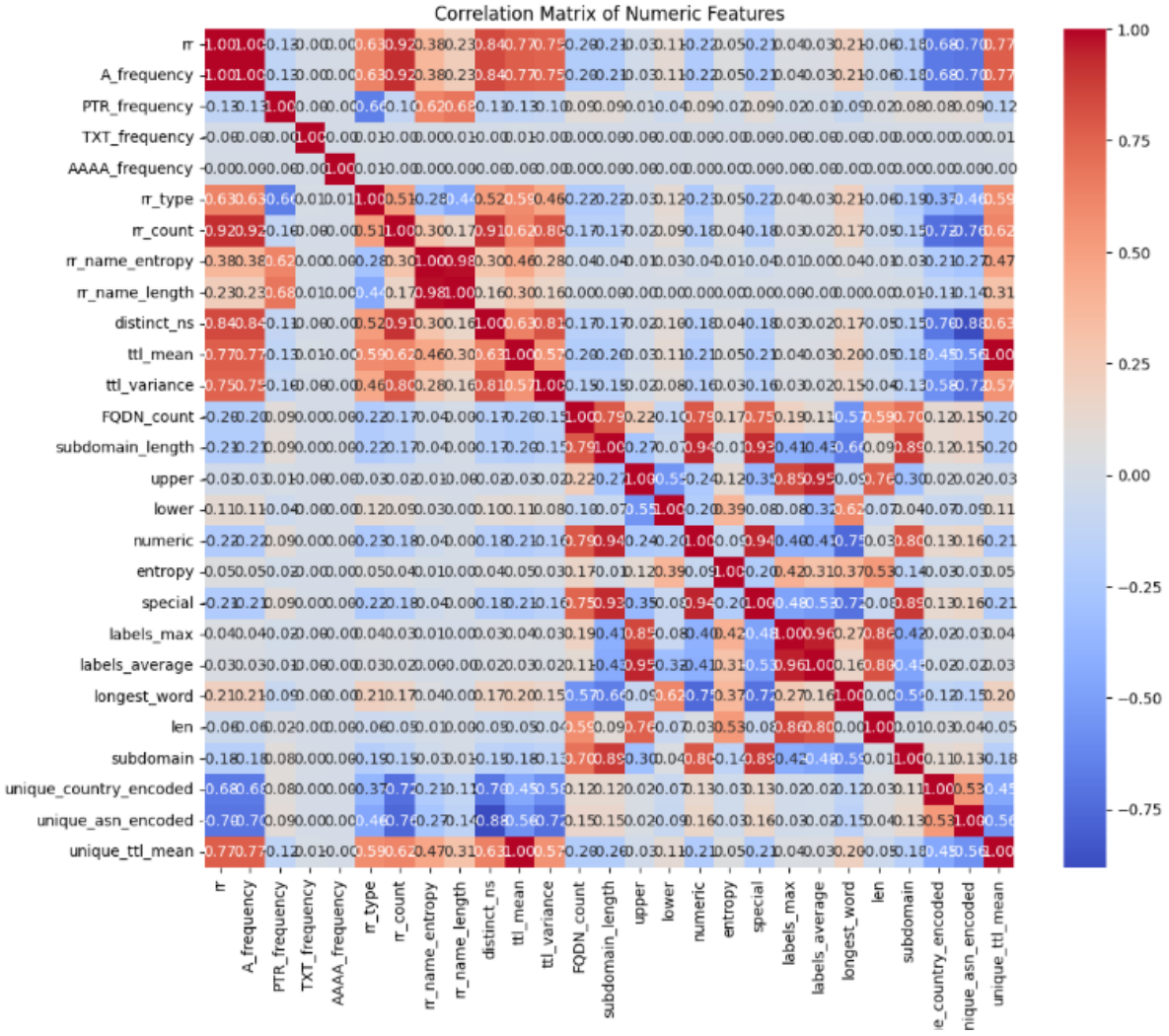
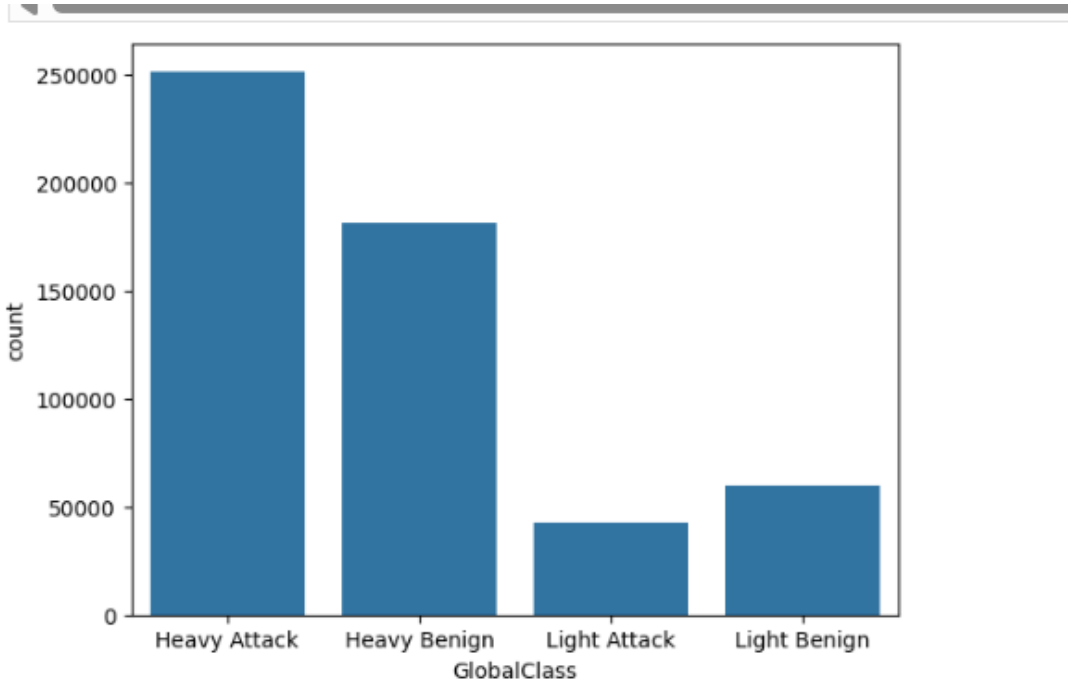


Figure 14: Matrice de corrélation finale.

Here is the repartition of the classes too :




```
GlobalClass
Heavy Attack    251670
Heavy Benign    181694
Light Benign     60091
Light Attack     42683
Name: count, dtype: int64
GlobalClass
Heavy Attack    46.941273
Heavy Benign    33.889409
Light Benign    11.208122
Light Attack     7.961197
Name: count, dtype: float64
```

we have a bit of an imbalance, the Heavy Attack and the Benign are representing a big part

Figure 15: Répartition des classes finales.

4 Formalisation of the probleme (Logan)

For the formalisation of the problem, first we have tought if we needed to fuse all the class or if we keep them split between heavy and light, we think it's good to keep it like that as the light attack can in some cases be ignored as it doesn't impact much but if we fuse everything and we try to do some classification regression and we find it's an "attack" and don't have any precision on the type for decision maker, they can't be able to know what is the amount of thing to do,Because in lot of case begign haeavy DNS request can cause a DOS attack indirect

So our goal here will be based on the dataset provided we will try to predict if it's a **Benign** or an **attack** and the **weight** of the DNS request ("Heavy, Light").

By training an learning machine algorithm on this subject we will be able to test with new DNS data gathered in realt time and predict the best we can, and be able to guide decision maker about the choice or doing a dynamic firewall / cache system who can filter the attack and the Benign request

5 Models Development : Logan Part

I firstly wanted to start with **NaiveBayes classification** for this project as we have a lot of feature I thought it could be a good project for that

First we splited the dataset between the Non GlobalClass and globalClass to have the label, and as normally I split between 20 % test and 80% train

By using the confusion matrix to see the result of this first default baseline model (without any particular change or parameter) we have this result :

```
Test class distribution:
 GlobalClass
Heavy Attack    50334
Heavy Benign    36339
Light Benign    12018
Light Attack     8537
Name: count, dtype: int64
Confusion Matrix:
 [[  255  1861 48214     4]
 [  149 19076 11400  5714]
 [   72   267  8198     0]
 [   62  6332  3703  1921]]

Classification Report:
               precision    recall  f1-score   support

Heavy Attack       0.47      0.01      0.01     50334
Heavy Benign       0.69      0.52      0.60     36339
Light Attack       0.11      0.96      0.20      8537
Light Benign       0.25      0.16      0.20     12018

    accuracy                           0.27    107228
   macro avg       0.38      0.41      0.25    107228
weighted avg       0.49      0.27      0.25    107228
```

Figure 16: Matrice de confusion, NaiveBayes (baseline).

We remember directly the unbalancing of the dataset then it's necessary to use **SMOTE** So it's what I have done after :

```
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
print("Resampled training set shape:", X_train_resampled.shape)
print("Resampled training class distribution:\n", y_train_resampled.value_counts())
```

Figure 17: Résultats après application de SMOTE (texte).

```
Confusion Matrix:
 [[ 2776  1901 45653     4]
 [  962 18964 10565  5848]
 [  441   274  7822     0]
 [  326  6306  3427  1959]]

Classification Report:
              precision    recall  f1-score   support

Heavy Attack       0.62      0.06      0.10     50334
Heavy Benign       0.69      0.52      0.59     36339
Light Attack       0.12      0.92      0.21      8537
Light Benign       0.25      0.16      0.20     12018

    accuracy                           0.29    107228
   macro avg       0.42      0.41      0.27    107228
weighted avg       0.56      0.29      0.29    107228
```

Figure 18: Matrice de confusion après SMOTE.

The accuracy is a little bit beter only but the F1 score are lot better than before, more close to each other but still a big issue of accuracy only a bit better than true randomness (25%) as we have 4 classes

So I tried after to use an other model especially the **randomForest** tree so i with basic parameters these results

```
Classification Report:
              precision    recall  f1-score   support

Heavy Attack       0.69      1.00      0.81     50334
Heavy Benign       0.75      0.63      0.69     36339
Light Attack       0.23      0.00      0.00      8537
Light Benign       0.26      0.07      0.11     12018

    accuracy                           0.69    107228
   macro avg       0.48      0.43      0.40    107228
weighted avg       0.62      0.69      0.63    107228
```

Figure 19: Résultats de RandomForest (baseline).

An accuracy lot better and a good recision for the 2 heavy feature but we still struggle for the F1 score and here it's worst than all because we have **0% for light attack**

So a biggest variance between each

So to correct that without taking to much time I used **undersampler** this time

```
Confusion Matrix:
 [[10636    71 39563    64]
 [ 1201 13386  9578 12174]
 [ 1508     8  7017     4]
 [  382  3844  3073  4719]]

Classification Report:
              precision    recall  f1-score   support

Heavy Attack       0.77      0.21      0.33     50334
Heavy Benign       0.77      0.37      0.50     36339
Light Attack       0.12      0.82      0.21      8537
Light Benign       0.28      0.39      0.33     12018

    accuracy                           0.33    107228
   macro avg       0.49      0.45      0.34    107228
weighted avg       0.67      0.33      0.38    107228
```

Figure 20: Résultats de RandomForest avec Undersampler.

We got again a bad accuracy but a better F1 score...

SO to conclude the use of baseline models, and basics parameters I concluded that it was not that relevant for our project maybe due to the amount of feature and the amount of lines (cf the latex for more commentary)

After during the Lab6 I implemented **grid search** and more complexe models

After a very long use of grid search for the Random Forest, I found that the best parameters were the defaults ones so at the end it didn't upgrade much the precision of the model

But for the naïve Bayes it increased a lot the results from 27 % to **63%** (non resempled)

```
GaussianNB Classification Report:
              precision    recall  f1-score   support

Heavy Attack       0.68      0.90      0.78     50334
Heavy Benign       0.69      0.54      0.61     36339
Light Attack       0.12      0.06      0.08      8537
Light Benign       0.25      0.16      0.20     12018

    accuracy                           0.63    107228
   macro avg       0.44      0.42      0.41    107228
weighted avg       0.59      0.63      0.60    107228
```

Figure 21: Résultats de NaiveBayes après GridSearch.

And it was very fast to compare so the grid search is better to use with naïve bayes

After I tried to use **soft voting** to see if it can help between the two models and the result are not that great :

Voting Classifier Classification Report:

	precision	recall	f1-score	support
Heavy Attack	0.69	0.93	0.79	50334
Heavy Benign	0.71	0.65	0.68	36339
Light Attack	0.12	0.03	0.05	8537
Light Benign	0.28	0.07	0.11	12018
accuracy			0.67	107228
macro avg	0.45	0.42	0.41	107228
weighted avg	0.60	0.67	0.62	107228

Figure 22: Résultats de Soft Voting.

Still have same issue as before. And with **bagging classifier** it's not better (worst for the un represented class) because I wanted to see the result without rebalancing for both of the tools

So at the end I wanted to do a bit of **feature engeniering** to maybe found a way to improve the accuracy

For exemple the feature importance was

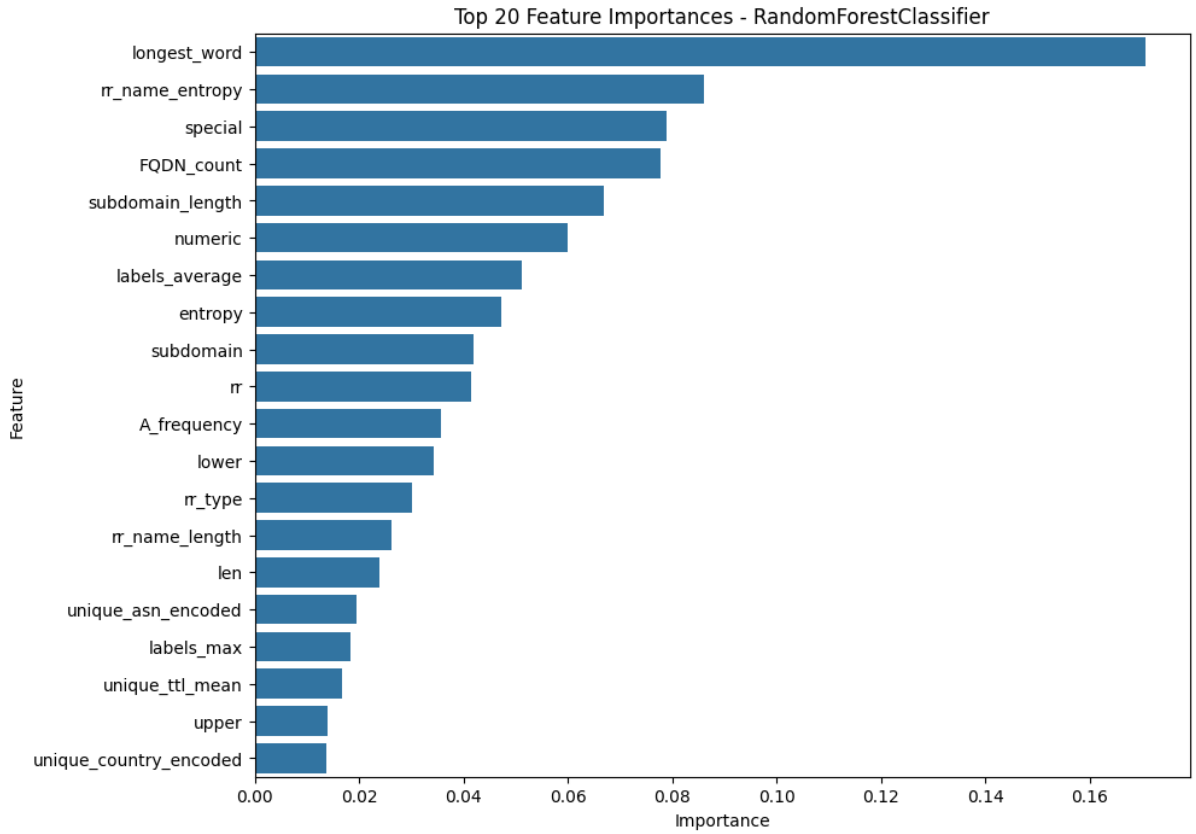


Figure 23: Importance des caractéristiques (avant rééchantillonnage).

And when we use the ressempling the feature are even more impacted lets see :

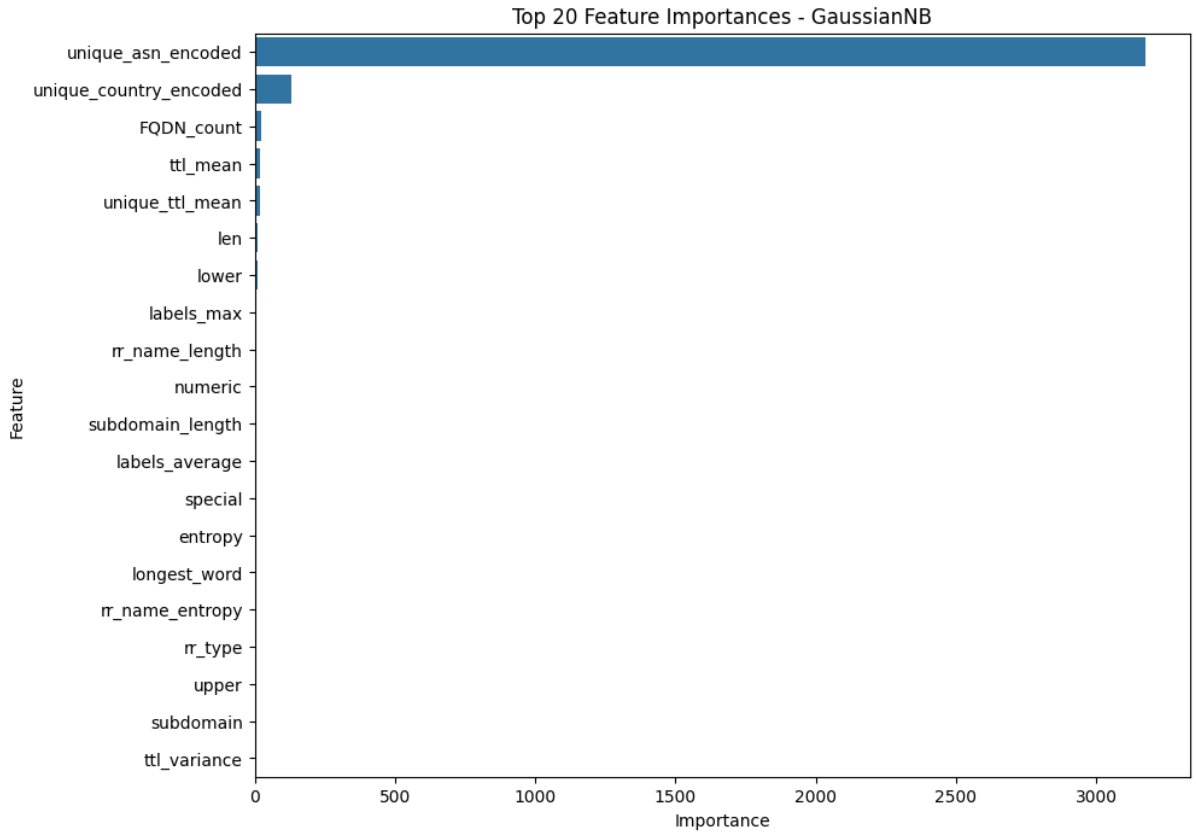


Figure 24: Importance des caractéristiques (après rééchantillonnage).

The `unique Asn_ encoded` and `unique _country encoded` are very very important but are not that useful in meaning so I tried to thing okay what can it to to the accuracy if we remove theses feature too much relevant

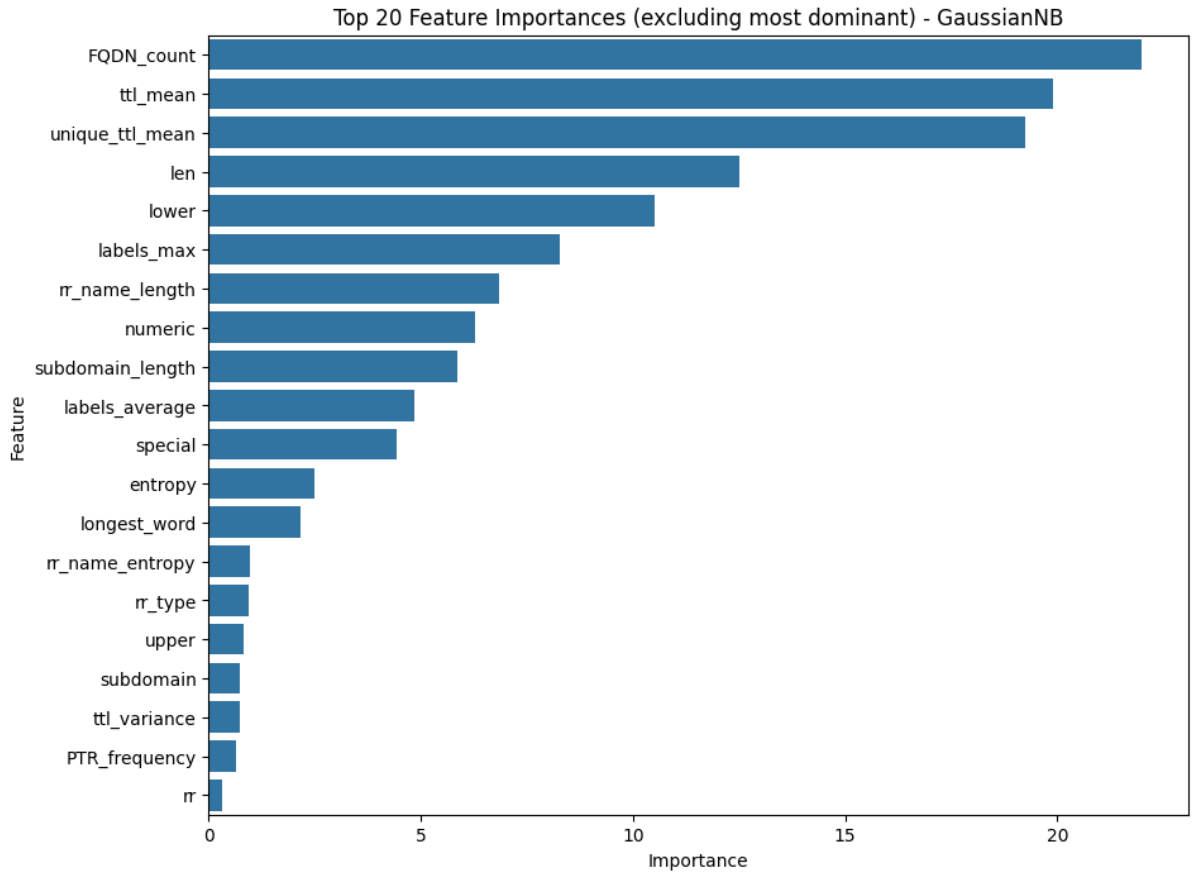


Figure 25: Résultats après suppression des caractéristiques trop pertinentes (Matrice de confusion 1).

[90]:
```
# Lets retrain the model with only these feature
X_train_top20 = X_train_resampled[top_20_features]
X_test_top20 = X_test[top_20_features]
best_gnb_top20 = GaussianNB(var_smoothing=1e-07)
best_gnb_top20.fit(X_train_top20, y_train_resampled)
# Let see the results
y_pred_top20 = best_gnb_top20.predict(X_test_top20)
print("Confusion Matrix with Top 20 Features:\n", confusion_matrix(y_test, y_pred_top20))
print("\nClassification Report with Top 20 Features:\n", classification_report(y_test, y_pred_top20))
```

```
Confusion Matrix with Top 20 Features:
 [[10380  2068 37881     5]
 [ 1090 18239  9314  7696]
 [ 1572   300  6665     0]
 [  357  6020  2990  2651]]

Classification Report with Top 20 Features:
                precision    recall  f1-score   support

Heavy Attack       0.77      0.21      0.33     50334
Heavy Benign       0.68      0.50      0.58     36339
Light Attack       0.12      0.78      0.20      8537
Light Benign       0.26      0.22      0.24     12018

    accuracy                           0.35    107228
   macro avg       0.46      0.43      0.34    107228
weighted avg       0.63      0.35      0.39    107228
```

Figure 26: Résultats après suppression des caractéristiques trop pertinentes (Matrice de confusion 2).

We see that we increase the F1 score so overall the accuracy than before :

```
Classification Report:
              precision    recall  f1-score   support

Heavy Attack       0.72      0.04      0.07     50334
Heavy Benign       0.69      0.54      0.61     36339
Light Attack       0.12      0.93      0.21      8537
Light Benign       0.25      0.16      0.20     12018

    accuracy                           0.29    107228
   macro avg       0.44      0.42      0.27    107228
```

Figure 27: Rapport de classification après suppression des caractéristiques.

The f1 score is lot more balanced than before as we see
And by using the **Random forest** the result is even better

```
Confusion Matrix with Top 20 Features - RandomForest:
 [[12126    44 38139    25]
 [ 1387 17749  9386  7817]
 [ 1771     8  6756     2]
 [  448  5527  3008  3035]]

Classification Report with Top 20 Features - RandomForest:
              precision    recall  f1-score   support

Heavy Attack       0.77      0.24      0.37     50334
Heavy Benign       0.76      0.49      0.59     36339
Light Attack       0.12      0.79      0.21      8537
Light Benign       0.28      0.25      0.27     12018

    accuracy                           0.37    107228
   macro avg       0.48      0.44      0.36    107228
weighted avg       0.66      0.37      0.42    107228
```

Figure 28: Résultats de RandomForest après suppression des caractéristiques.

We arrive to get **37%** and at least 20% f1score with 60% at max, so we fighted the 25% accuracy for randomness and it's the best result so far between the accuracy and the f1 score

Then for the next part I used a much complex model, a **XGB classifier** with undersample to be faster

```
Resampled training set shape: (136584, 27)
Resampled training class distribution:
 GlobalClass
Heavy Attack    34146
Heavy Benign    34146
Light Attack    34146
Light Benign    34146
Name: count, dtype: int64
XGBoost Classification Report:
              precision    recall  f1-score   support

Heavy Attack       0.76      0.26      0.39     50334
Heavy Benign       0.77      0.41      0.54     36339
Light Attack       0.12      0.77      0.21      8537
Light Benign       0.28      0.34      0.31     12018

    accuracy                           0.36    107228
   macro avg       0.48      0.45      0.36    107228
weighted avg       0.66      0.36      0.42    107228
```

Figure 29: Résultats de XGBoost avec undersample.

The first gave us a pretty good thing one of the best for now

After I tried to use a **deep learning algorithm** provided by tansor flow with classic layers and classic parameter but didn't get much for the f1 score

And after I tried to use **PCA** with the random forest and didn't get much better so I finished on this try

6 Models Development : Diadie Part

In the model development phase, my goal was to surpass the 69% accuracy of our baseline model Random Forest by applying optimization and **Ensemble Learning** techniques.

The first challenge was to optimize without exhausting the memory. I started by searching for the best hyperparameters for our models (Random Forest and Decision Tree) via a **GridSearch**. However, with over 400,000 lines of data, the process was too computationally intensive and caused the environment's RAM to crash. Therefore, I isolated 20% of the data... Once the best settings were found, I applied them to the entire dataset for final training.

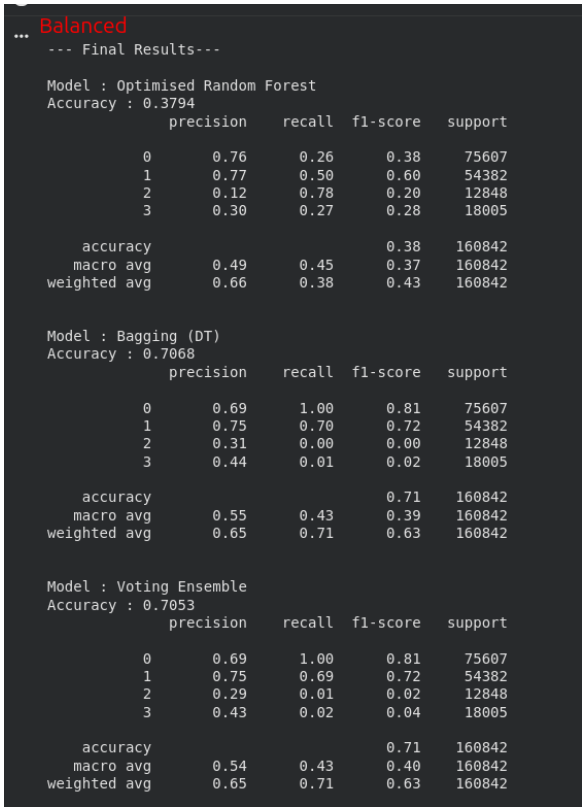
I was then quickly exposed to a dilemma regarding **data balancing**. Indeed, thanks to the parameters obtained by the GridSearch, I tested the `class_weight='balanced'` option to attempt to better detect "Light Attacks" and "Light Benign". Against all expectations, the global Accuracy collapsed to **38%**, whereas I expected to exceed 69%. The model actually became "paranoid": in order not to miss any small attack, it began classifying a large amount of normal traffic as malicious (False Positives).

Faced with this observation, I decided to retry by setting `class_weight=None`. I therefore chose to remove this weighting to return to a more stable model (**71% accuracy**), prioritizing good global performance over hypersensitivity. The downside of this choice is that the **Recall for "Light" attacks drops back to 0.00 and 0.01**. This means the model completely ignores small attacks to optimize its global score. The problem is that, in terms of cybersecurity, this represents a major vulnerability because subtle attacks will go completely unnoticed.

In other words, the dilemma is as follows:

- The unbalanced approach prioritizes quality of service (not blocking the user) but leaves a door open to small attacks. This is the reasonable choice for a main production system.
- The balanced approach prioritizes absolute security against rare threats, but at the cost of an unacceptable rate of false alarms for automatic filtering.

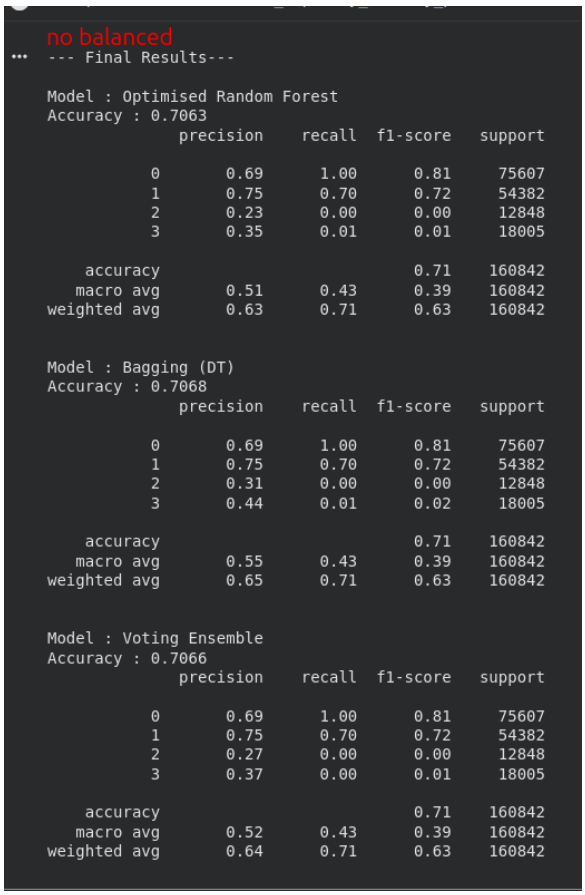
Final Decision: For this reason, the "Unbalanced" **Voting Ensemble model** (71%) is retained as the final solution, because a security system cannot afford to have only 38% global precision.



Balanced

```
--- Final Results---

Model : Optimised Random Forest
Accuracy : 0.3794
              precision    recall  f1-score   support

           0       0.76      0.26      0.38     75607
           1       0.77      0.50      0.60     54382
           2       0.12      0.78      0.20     12848
           3       0.30      0.27      0.28     18005

    accuracy                           0.38    160842
   macro avg       0.49      0.45      0.37    160842
weighted avg       0.66      0.38      0.43    160842


Model : Bagging (DT)
Accuracy : 0.7068
              precision    recall  f1-score   support

           0       0.69      1.00      0.81     75607
           1       0.75      0.70      0.72     54382
           2       0.31      0.00      0.00     12848
           3       0.44      0.01      0.02     18005

    accuracy                           0.71    160842
   macro avg       0.55      0.43      0.39    160842
weighted avg       0.65      0.71      0.63    160842


Model : Voting Ensemble
Accuracy : 0.7053
              precision    recall  f1-score   support

           0       0.69      1.00      0.81     75607
           1       0.75      0.69      0.72     54382
           2       0.29      0.01      0.02     12848
           3       0.43      0.02      0.04     18005

    accuracy                           0.71    160842
   macro avg       0.54      0.43      0.40    160842
weighted avg       0.65      0.71      0.63    160842
```

(a) Unbalanced Voting Ensemble (71% Acc)



no balanced

```
--- Final Results---

Model : Optimised Random Forest
Accuracy : 0.7063
              precision    recall  f1-score   support

           0       0.69      1.00      0.81     75607
           1       0.75      0.70      0.72     54382
           2       0.23      0.00      0.00     12848
           3       0.35      0.01      0.01     18005

    accuracy                           0.71    160842
   macro avg       0.51      0.43      0.39    160842
weighted avg       0.63      0.71      0.63    160842


Model : Bagging (DT)
Accuracy : 0.7068
              precision    recall  f1-score   support

           0       0.69      1.00      0.81     75607
           1       0.75      0.70      0.72     54382
           2       0.31      0.00      0.00     12848
           3       0.44      0.01      0.02     18005

    accuracy                           0.71    160842
   macro avg       0.55      0.43      0.39    160842
weighted avg       0.65      0.71      0.63    160842


Model : Voting Ensemble
Accuracy : 0.7066
              precision    recall  f1-score   support

           0       0.69      1.00      0.81     75607
           1       0.75      0.70      0.72     54382
           2       0.27      0.00      0.00     12848
           3       0.37      0.00      0.01     18005

    accuracy                           0.71    160842
   macro avg       0.52      0.43      0.39    160842
weighted avg       0.64      0.71      0.63    160842
```

(b) Balanced Voting Ensemble (38% Acc)

Figure 30: Comparaison des matrices de confusion pour l'approche équilibrée vs non équilibrée.

7 Models Development : Ziyad Part

For my model, I chose to use **Logistic Regression**. The main objective was to test a lighter and faster algorithm than ensemble models, to see if we could still achieve competitive performance on DNS traffic attack detection.

I used the cleaned features prepared earlier and kept the 4-class prediction problem (Heavy Attack, Heavy Benign, Light Attack, Light Benign). Logistic Regression requires normalized values, so I applied scaling before training.

After training the classifier, I evaluated it on the test set. The best hyperparameters found were:

- C = 0.1
- class_weight = None

These settings gave the highest performance before RAM limitations and training time became too high.

```
Fitting 3 folds for each of 8 candidates, totalling 24 fits
Meilleurs paramètres LogReg : {'C': 0.1, 'class_weight': None}

=== Logistic Regression (après GridSearch) ===
Accuracy : 0.7044055657104488
F1-weighted : 0.6258437728174796

Classification report :
              precision    recall  f1-score   support

Heavy Attack       0.69      0.99      0.81     75501
Heavy Benign       0.74      0.70      0.72     54509
Light Attack       0.00      0.00      0.00     12805
Light Benign       0.33      0.00      0.00     18027

    accuracy                           0.70    160842
   macro avg       0.44      0.42      0.38    160842
weighted avg       0.61      0.70      0.63    160842
```

Figure 31: Matrice de confusion, Régression Logistique (70% Acc).

The confusion matrix confirms that the model almost never detects **Light traffic**, especially Light Attacks. Light samples are often classified as Heavy, meaning the model struggles with subtle DNS flows.

The Logistic Regression model shows that it mainly focuses on the biggest traffic categories. It performs well on Heavy Attack and Heavy Benign flows, which means it is able to detect clear and obvious malicious behavior. However, it fails almost completely on Light traffic, especially Light Attacks. This is a critical issue, because these small attacks are often the most dangerous ones in real scenarios, where attackers try to stay hidden. So even if the global accuracy looks good, the model does not actually provide strong security against stealthy threats.

Logistic Regression is a good reference model because it is simple and fast, but it cannot correctly identify the most dangerous attacks the ones designed to stay small and hidden. Because of this, more advanced models are needed to reliably secure DNS traffic.

8 Conclusion and future evolution (Logan)

The three of us didn't succeed to have a very good model from the basics models so I can conclude that the issue is either in the data, in the processing or in the choice of the model. The results show that detecting "**Light**" attack is very hard with standard classification.

Maybe for future evolution we can **split the problem into two** first Predict if it's an attack or Benign and after to see if it's Heavy or Light this way the big amount of heavy data will not overdie the small amount but even with undersampling tool it didn't helped much so maybe spend more time on the preprocessing or remove more columns.

Since "Light Attack" tries to hide, maybe supervised learning is not the good method. We could try machinelearning like **unsupervised learning** (like Isolation Forest or Auto-Encoders). We train the model ONLY on Benign traffic to learn what is "Normal". Then, anything that is a bit different (even if it is light) will be detected as an anomaly.

We can also use the time to use **RNN or LSTM** based on the `timestamp` we removed to analyse with the time, there are lot of other thing we can do to increase the accuracy.