



Rapport de projet long

Réseau convolutif graphique : étude et application au trafic internet

Benoit BAUDE
Alexandre Rivière
Avy Nizard
Ignacio Oros Campo
Cédric Martin

Département Sciences du Numérique - 3 SN

Contents

1	Introduction	3
2	Traitement des données CAIDA	5
2.1	Analyse générale des données brutes	5
2.2	Caractéristiques des graphes et datasets obtenus	7
2.3	Sélection et création de node-feature pertinents	8
2.4	Mise à l'échelle des node-features	9
3	Traitement des données PerringDB	11
3.1	Caractéristiques des graphes et datasets obtenus	11
3.2	Sélection des node-feature pertinents	14
3.3	Mise à l'échelle des node-features	16
4	Mise en place du GCN	18
4.1	Récupération des données	18
4.2	Établissement d'une baseline	18
4.2.1	Caida	18
4.2.2	PeeringDB	24
4.3	Implémentation de GraphSAGE et d'autres méthodes	26
4.4	Pistes de recherche	29
5	Conclusion	30

1 Introduction

Internet peut-être défini comme un ensemble d'hôtes interconnectés par des réseaux de liens et de routeurs en constante évolution. Un système autonome (Autonomous System - AS) est un ensemble de préfixes IP routables sur Internet, appartenant à un réseau ou à un ensemble de réseaux qui sont tous gérés, contrôlés et supervisés par une seule entité ou organisation. Cela peut-être par exemple un fournisseur d'accès internet - ou ISP (Internet Service Provider), une université, un réseau social, un fournisseur de contenu (Netflix) etc... Chaque AS possède un numéro unique qui lui sert d'identifiant.


AS number	15169				
AS name	GOOGLE				
organization	Google LLC				
country	United States 				
AS rank	2036				
customer cone	12 asn	2275 prefix	14113760 address		
AS degree	383 global	343 transit	4 provider	368 peer	11 customer

Figure 1: Première exemple d'AS

Par exemple, l'AS de GOOGLE porte le numéro 15169. Internet est également composé d'IXP (Internet Exchange Point). Un IXP est une infrastructure physique utilisée par les fournisseurs de services Internet (ISP) et les réseaux de diffusion de contenu (Content Delivery Network - CDN) pour échanger du trafic Internet entre leurs réseaux. Chaque IXP possède un préfixe, ou une collection de préfixes, qui sont utilisés par les AS pour adresser les machines au sein de l'infrastructure IXP.

Une caractéristique notable d'internet est sa structure hiérarchique (que nous retrouverons lors de l'analyse des données CAIDA ou PeeringDB).

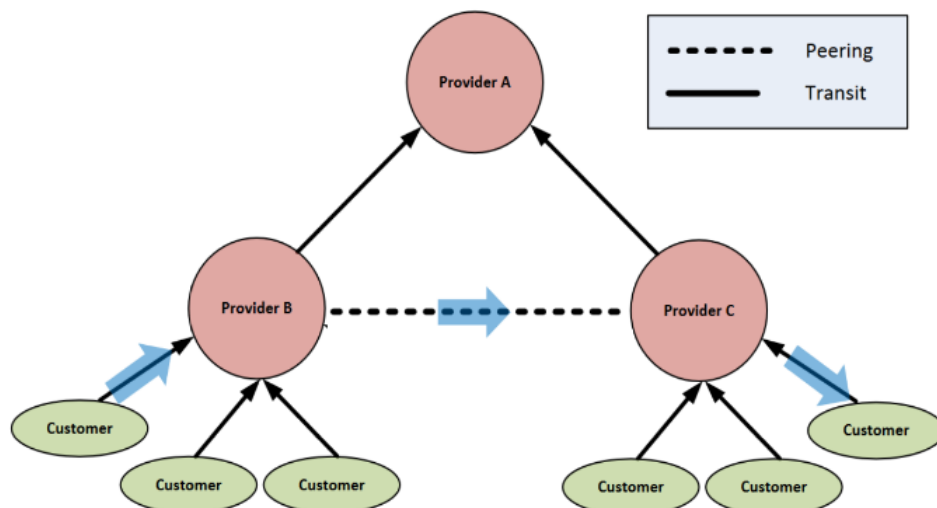


Figure 2: Structure hiérarchique d'internet

Cette figure illustre la structure hiérarchique d'internet. Le transit signifie simplement l'obtention de services Internet auprès d'un fournisseur de services Internet plus élevé que soi dans la hiérarchie.

Par exemple un customer va profiter des services internet du Provider B qui lui même va profiter des services internet du Provider A. Au contraire, un lien de peering est établi lorsque deux administrateurs Internet partagent les responsabilités de routage des données sur plusieurs réseaux. Il s'agit alors d'une interconnexion entre deux acteurs, plus ou moins, de même rang dans la hiérarchie.

Le but de ce projet est de modéliser les échanges au sein d'internet par un graphe, l'appliquer à un réseau convolutif graphique (GCN) afin d'en extraire des caractéristiques et de classer les AS. Les métadonnées de l'Internet sont collectées par des organisations qui exploitent les protocoles réseaux classiques comme TCP, DNS et BGP. Nous allons utiliser deux jeux de données : CAIDA et PeeringDB.

2 Traitement des données CAIDA

2.1 Analyse générale des données brutes

Se référer au notebook `metadata_overview.ipynb` pour cette sous-partie.

Avant de se focaliser sur un jeu de données CAIDA particulier, nous avons analysé l'ensemble des jeux de données disponibles pour en extraire des caractéristiques apparentes. Cela permet de raisonner rétrospectivement après les résultats, et expliquer si ces derniers ont été bon ou mauvais, puisque les métadonnées sont la source de l'étude d'Internet. Des métadonnées biaisées ou incomplètes peuvent être un frein aux résultats, ou inversement, peuvent mener vers de très résultats avec une étude approfondie du GCN.

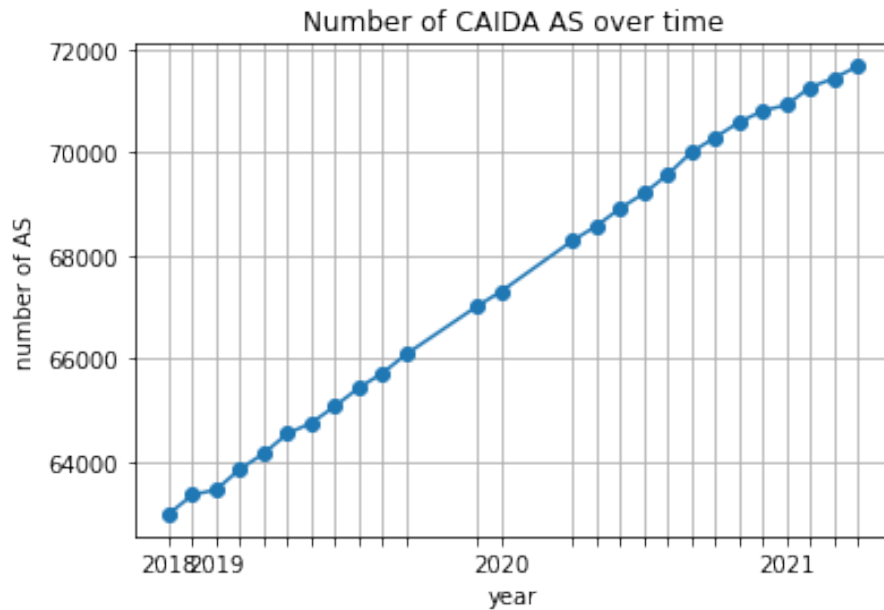


Figure 3: Évolution du nombre d'AS présents dans les jeux de données

Afficher le nombre d'AS présent par dataset en fonction du temps nous permet de nous rendre compte d'une chose : Internet se complexifie, et de nouveaux acteurs entre en jeu chaque jour. En effet, le nombre d'AS évolue linéairement à la hausse en fonction du temps, avec environ 500 nouveaux AS chaque mois. Ces informations sont très pertinentes.

Étant donné l'augmentation du nombre d'AS année par année, nous avons travaillé avec les datasets les plus récents, pour profiter d'un graphe plus grand donc plus diversifié. Nous utilisons le dataset `20210301.as-re12.txt` pour la suite.

Nous avons ensuite affiché l'évolution des types de lien du graphe. En effet, les datasets contiennent 2 types de lien : les liens peering (p2) et provider to customer (p2c).

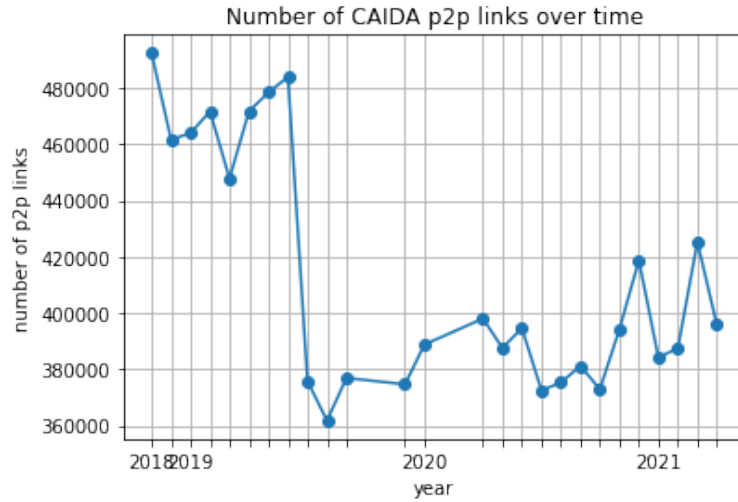


Figure 4: Évolution du nombre de liens p2p en fonction du temps

Concernant les liens p2p, la chute du nombre de Peering links entre les mois de juin et de juillet 2019 est assez problématique. En effet, on chute de 20% en à peine un mois. Les datasets d'avant mi-2019 ont un nombre de lien p2p 'anormalement' élevé.

Concernant le nombre de liens p2c, il suit en moyenne une augmentation linéaire.

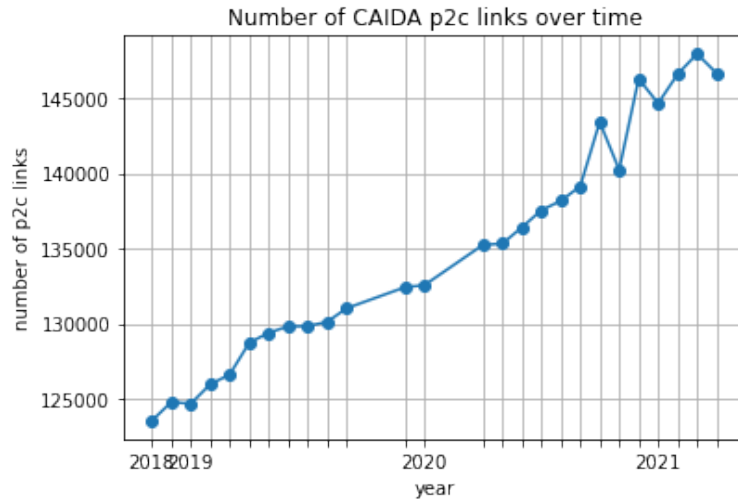


Figure 5: Évolution du nombre de liens p2c en fonction du temps

2.2 Caractéristiques des graphes et datasets obtenus

Se référer au notebook `creation_datasetgraph.ipynb` pour cette sous-partie.

L'étape suivante consistait à afficher et analyser les caractéristiques du graphe et du dataset CAIDA obtenu. Nous avons vu que CAIDA ne classifiait que selon 3 classes, ce qui est assez pauvre. De plus, les classes d'AS sont très déséquilibrées puisque la classe **Content** ne représente que 5% des AS.

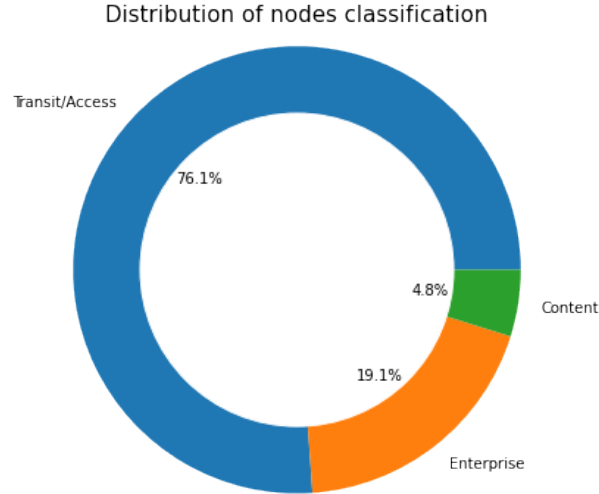


Figure 6: Distribution de la classification des noeuds du graphe

Ce **déséquilibre de classification des noeuds** a eu pour conséquence de mauvais résultats du GCN, lors de la sélection de noeuds d'apprentissage (Cf partie 4).

Le graphe CAIDA obtenu a une distribution de degré particulière, comme le montre la figure ci-dessous.

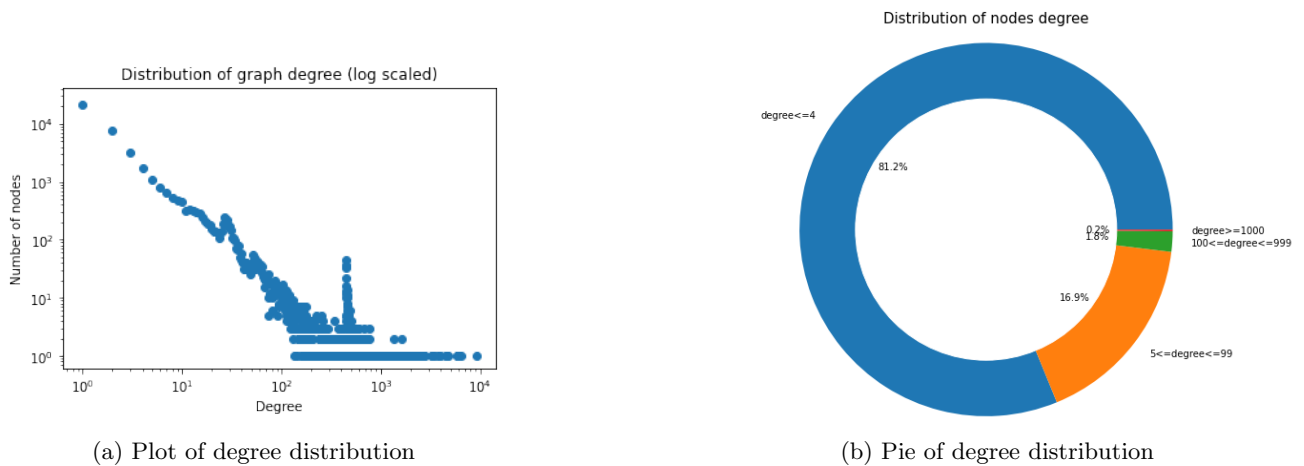


Figure 7: Affichage complet de la distribution des degrés du graphe CAIDA

On retrouve une droite assez linéaire alors que l'affichage est en échelle logarithmique. Cela signifie que le nombre de noeuds de même degré d est inversement et exponentiellement proportionnel à la valeur de d . Comme on le retrouve sur le pie, on a un très grand nombre de noeuds à faible degré et un très faible nombre de noeuds extrêmement connectés.

2.3 Sélection et création de node-feature pertinents

Les métadonnées de CAIDA offrant très peu de node-features, nous avons utilisé des métriques de NetworkX pour en créer de nouveaux. Notamment `page_rank` qui a été utilisé 3 fois différemment, en utilisant les 3 types de lien : p2p, c2p et p2c. Nous avons aussi utilisé la fonction `degree_centrality` de NetworkX pour définir un nouveau node-feature.

Le node-feature `page_rank_not_directed` a été calculé sur un graphe non orienté. Cela signifie que les liens p2p et p2c étaient les mêmes. Ceci met en évidence les noeuds fortement connectés du graphe.

Le node-feature `page_rank_directed` a été calculé sur un graphe orienté, où les liens p2p entre AS `as1` et `as2` se traduisaient par 2 arêtes de sens différent `as1 → as2` et `as2 → as1` ; avec les liens p2c dirigés du provider `p` vers le customer `c` (`as_c → as_p`). Ceci met en évidence les noeuds qui possèdent beaucoup de providers.

Le node-feature `page_rank_directed_inverse` a été calculé sur un graphe orienté, où les liens p2p entre AS `as1` et `as2` se traduisaient par 2 arêtes de sens différent : liens p2c dirigés du customer vers le provider (`as_p → as_c`). Ceci met en évidence les noeuds qui possèdent beaucoup de customers.

On extrait par exemple le nombre maximal de chaque `page_rank`, avec l'ASN du noeud correspondant :

```
Max page_rank value : 0.010812990990635204, ASN related : 174 (neighbors converging)
Max page_rank_directed value : 0.009920743675492893, ASN related : 6939 (provider converging)
Max page_rank_directed_inverse value : 0.010811150120130737, ASN related : 6939 (customer converging)
```

Figure 8: Exemple de valeurs maximale de `page_rank`

Ensuite, nous avons défini les nouveaux attributs *peering_ratio*, *customer_ratio* et *provider_ratio* désignant pour chaque noeud le nombre de liens de type correspondant ; par exemple *peering_ratio* désigne le nombre de liens p2p d'un noeud (et non pas son ratio $\frac{nb_voisins_p2p}{nb_voisins}$). Enfin, *degree_normalized* désigne le degré de chaque noeud.

À la fin du traitement, nous obtenons un dataset de **8 node-features**.

	ASN	source_label	label	page_rank_not_directed	page_rank_directed	page_rank_directed_inverse	degree_centrality	degree_normalized	ratio_peering	ratio_customer	ratio_provider
2192	6318	CAIDA_class	Content	0.000008	0.000009	0.000002	0.000042	3.0	0.0	0.0	3.0
2193	6319	CAIDA_class	Enterprise	0.000006	0.000008	0.000002	0.000028	2.0	0.0	0.0	2.0
2194	6325	peerDB_class	Transit/Access	0.000111	0.000016	0.000052	0.000700	50.0	9.0	35.0	6.0
2195	6327	peerDB_class	Transit/Access	0.000457	0.000030	0.000406	0.003444	246.0	49.0	190.0	7.0
2196	6332	CAIDA_class	Transit/Access	0.000006	0.000009	0.000002	0.000028	2.0	0.0	1.0	1.0
2197	6334	CAIDA_class	Enterprise	0.000007	0.000009	0.000002	0.000042	3.0	0.0	0.0	3.0

Figure 9: Aperçu du dataset après sa création

2.4 Mise à l'échelle des node-features

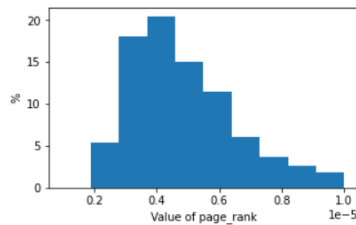
Se référer au notebook `data_scaling.ipynb` pour cette sous-partie.

Les nodes-features obtenus n'étaient pas prêt à être utilisé directement par le GCN. En effet, leurs valeurs étaient trop grandes ou trop faibles et leur dispersion mauvaise ce qui ne mettait pas en évidence les écarts de valeurs observés. Il a alors fallu décider de comment gérer les valeurs nulles et de la fonction à appliquer pour disperser au mieux les valeurs des node-features (pour que toutes les données n'aient pas la même valeur de node-feature).

Pour les divers *page_rank*, leurs valeurs sont souvent très faibles : de l'ordre de 10^{-4} et très peu variables. Une variation de 10^{-3} n'est pas détectée par le GCN, elle est infime. Nous avons donc multiplié les valeurs pour **les recentrer autour de -1 et 1** (avec 1 comme borne max pour les valeurs au dessus, après multiplication). Chaque node-feature a son histogramme post, puis pré traité associé.

Distribution of *page_rank_not_directed* node-feature :

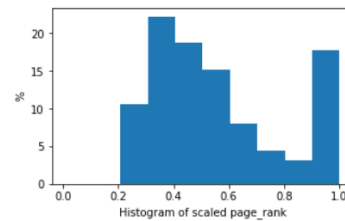
```
*****
count      71434.000000
mean        0.000014
std         0.000106
min         0.000002
25%         0.000004
50%         0.000005
75%         0.000007
max         0.010813
Name: page_rank_not_directed, dtype: float64
*****
```



(a) page rank distribution when unscaled

Distribution of **scaled** *page_rank_not_directed* node-feature :

```
*****
count      71434.000000
mean        0.565444
std         0.241283
min         0.234351
25%         0.378069
50%         0.492291
75%         0.705260
max         1.000000
Name: page_rank_not_directed, dtype: float64
*****
```



(b) page rank distribution when scaled

Figure 10: Mise à l'échelle des valeurs de l'attribut *page_rank*

En multipliant par 100000 les valeurs et en bornant par 1, les valeurs de *page_rank* deviennent très bien dispersées. L'historgramme de droite présente une donnée de dispersion intéressante pour être utilisée dans un GCN.

L'exemple du node-feature *degree_normalized* est intéressant puisque la composition par la fonction **log₁₀** permet de mieux disperser les données. Comme vu dans le notebook `creation_datasetgraph.ipynb`, le nombre de nœuds de même degré étant exponentiellement inversement proportionnel à la valeur du degré.

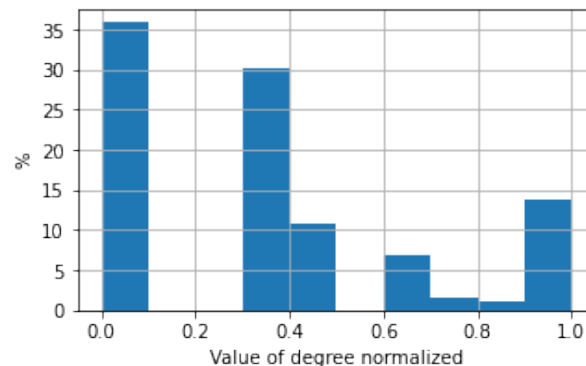


Figure 11: Histogramme de l'attribut *degree_normalized* mis à l'échelle

On a réutilisé cette information précieuse pour effectuer une mise à l'échelle pertinente.

Les attributs *peering_ratio*, *customer_ratio* et *provider_ratio* ont été mis à l'échelle à peu près de la même manière, en composant par les fonctions **log** ou **log₁₀** en fonction des meilleurs résultats obtenus.

On obtient à la fin un dataset mis à l'échelle, prêt à être utilisé par le GCN.

	ASN	source_label	label	page_rank_not_directed	page_rank_directed	page_rank_directed_inverse	degree_centrality	degree_normalized	ratio_peering	ratio_customer	ratio_provider
2192	6318	CAIDA_class	0	0.768686	0.386338	0.083994	0.419974	0.477121	0.0	0.000000	0.602060
2193	6319	CAIDA_class	2	0.557161	0.379842	0.083994	0.279983	0.301030	0.0	0.000000	0.477121
2194	6325	peerDB_class	1	1.000000	0.726593	1.000000	1.000000	1.000000	1.0	1.000000	0.845098
2195	6327	peerDB_class	1	1.000000	1.000000	1.000000	1.000000	1.000000	1.0	1.000000	0.903090
2196	6332	CAIDA_class	1	0.599182	0.388294	0.096947	0.279983	0.301030	0.0	0.693147	0.301030
2197	6334	CAIDA_class	2	0.650865	0.386042	0.083994	0.419974	0.477121	0.0	0.000000	0.602060

Figure 12: Aperçu du dataset après mise à l'échelle

3 Traitement des données PerringDB

3.1 Caractéristiques des graphes et datasets obtenus

Se référer au notebook `creation_datasetgraph.ipynb` pour cette sous-partie.

Les données PeeringDB sont obtenus grâce à l'API PeeringDB et sont bien plus complètes que celles de CAIDA. Premièrement, le nombre de classes pour les AS est de 10 (contre 3 pour CAIDA) ; ce qui nous permettra d'avoir une classification plus précise des AS. Dans le jeu de données à notre disposition nous avons la distribution suivante :

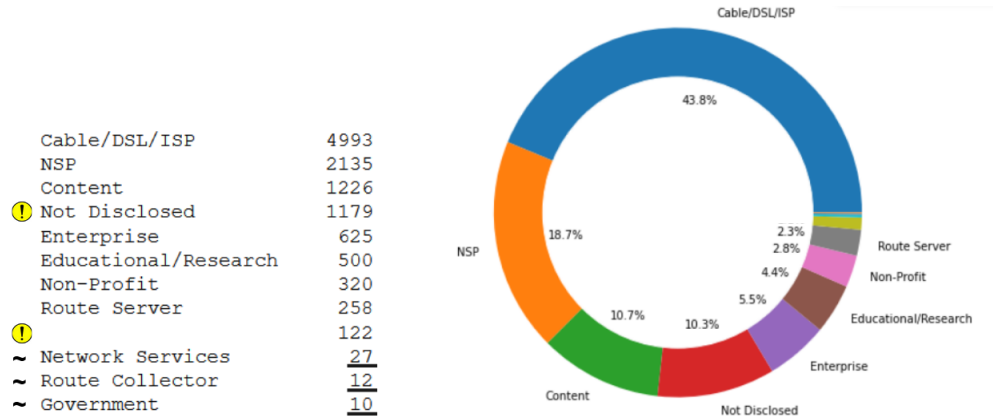


Figure 13: Distrubution de la classification des AS

Nous pouvons voir que certains AS n'ont pas été étiquetés. En effet, 1179 d'entre eux ont la valeur de label Not Disclosed et 122 n'ont pas de label. Nous devrons traiter ce problème à l'avenir pour la classification des AS ainsi que pour les métadonnées vides. Le dataset PeeringDB est composé d'AS et IXP et forme un graphe biparti : il y a des liens entre AS et IXP mais il n'y a pas de liens inter AS et inter IXP. Nous avons donc à notre disposition un jeu de données PeeringDB formant un graphe biparti dirigé de 12282 noeuds (11472 AS et 810 IXP). Le graphe a été construit grâce au métadonnée de l'API de la façon suivante :

Toutes les entrées sont définies de manière unique par un index.

ASes : l'index est le numéro d'AS (asn)

IXPs : un nombre négatif attribué

Le graphe est d'abord construit à partir des infos présentes dans `netixlan_set` de l'API. Cela donne un graphe biparti (AS-IXP) avec des liens pondérés par la taille du port du routeur (**speed** dans l'API). Les liens (arêtes) sont définis de la façon suivante :

- *Inbound* : un lien est crée d'IXP à AS avec un poids égal à la taille du port et un autre lien de poids $(1-\beta)$ est crée dans l'autre direction.
- *Outbound* : un lien est crée d'AS à IXP avec un poids égal à la taille du port et un autre lien de poids $(1-\beta)$ est crée dans l'autre direction.
- *Balanced or Not Disclosed* : Un lien bidirectionel avec un poids égal à la taille du port.

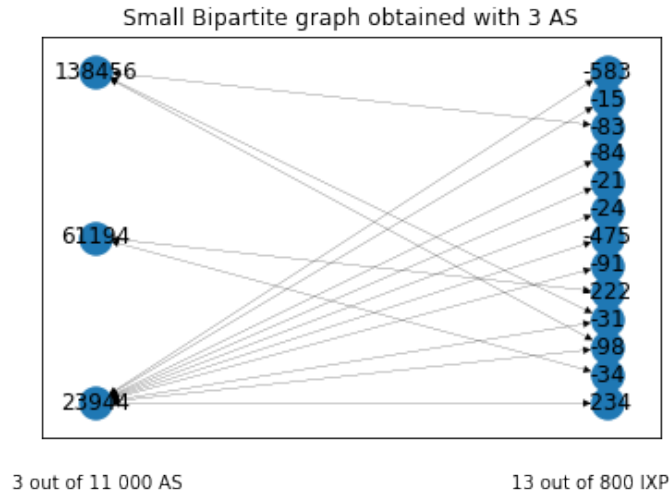


Figure 14: Exemple de graphe biparti extrait de PeeringDB

Nous avons ci-dessus un graphe de petite taille extrait de PeeringDB (le graphe complet n'étant pas affichable en raison de sa taille). A droite nous avons les IXP ($asn < 0$) et à gauche nous avons les AS ($asn > 0$). La première étape est de se familiariser avec ce nouveau jeu de données. Nous avons alors affiché des caractéristiques du graphe tel que la distribution du degré des noeuds.

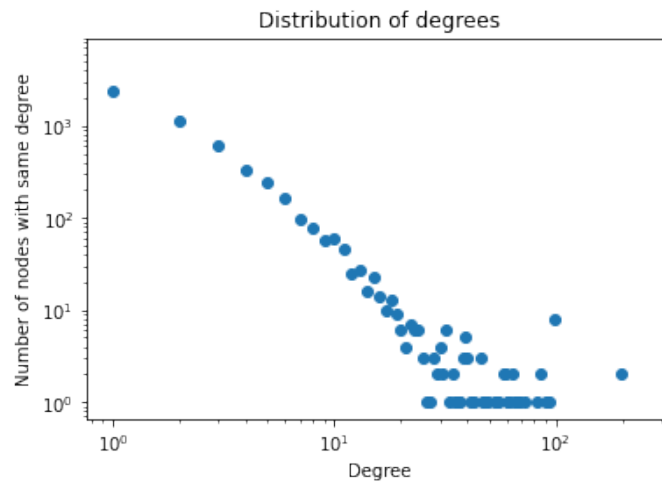


Figure 15: Distribution du degré des noeuds

Comme pour Caida, nous voyons que la distribution des degrés est plutôt linéaire. Cela signifie que le nombre de noeuds de même degré d est inversement et exponentiellement proportionnel à la valeur de d . La plupart des noeuds ont 1 ou 2 voisins, mais peu d'entre eux ont plus de 100 voisins. Ce résultat est intéressant et montre la structure hiérarchique d'internet.

Un autre comportement des données intéressant à regarder est la corrélation entre le *port_capacity* et le degré d'un AS.

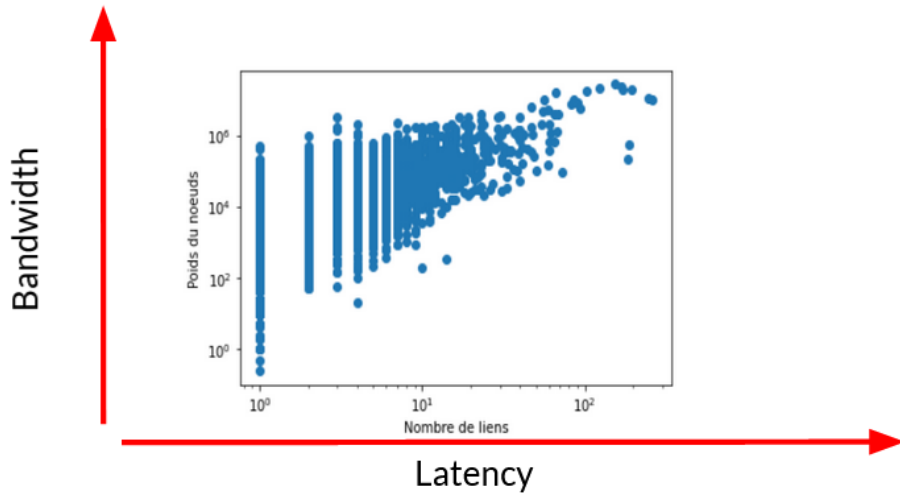


Figure 16: Corrélation entre le degré et la taille du port

Nous voyons sur cette figure (échelle log) que le *port_capacity* est particulièrement lié au degré. En effet, selon leur type les AS vont avoir des comportements différents. Par exemple les deux points décentrés à droite représentent les serveurs DNS qui essaient d'avoir une bonne latence. Comme ils sont à droite et pas beaucoup en haut, cela signifie qu'ils ont beaucoup plus de liens que les autres, sans pour autant avoir un *port_capacity* plus élevé.

Le dataset nous était également fourni avec 34 nodes features.

```
net table summary
<class 'pandas.core.frame.DataFrame'>
Int64Index: 11472 entries, 20940 to 61437
Data columns (total 35 columns):
#   Column                                Non-Null Count  Dtype
---  ---                                ---
0   status                                11472 non-null  object
1   looking_glass                        11472 non-null  object
2   route_server                        11472 non-null  object
3   netixlan_updated                    11472 non-null  object
4   info_ratio                          11472 non-null  object
5   id                                   11472 non-null  int64
6   policy_ratio                        11472 non-null  bool
7   info_unicast                       11472 non-null  bool
8   policy_general                      11472 non-null  object
9   website                            11472 non-null  object
10  allow_ixp_update                    11472 non-null  bool
11  updated                             11472 non-null  object
12  netfac_updated                      7121 non-null  object
13  info_traffic                        11472 non-null  object
14  info_multicast                      11472 non-null  bool
15  policy_locations                    11472 non-null  object
16  name                                11472 non-null  object
17  info_scope                          11472 non-null  object
18  notes                               11472 non-null  object
19  created                             11472 non-null  object
20  org_id                              11472 non-null  int64
21  policy_url                          11472 non-null  object
22  info_never_via_route_servers        11472 non-null  bool
23  poc_updated                         10524 non-null  object
24  info_type                           11472 non-null  object
25  policy_contracts                    11472 non-null  object
26  info_prefixes6                     11472 non-null  int64
27  aka                                 11472 non-null  object
28  info_prefixes4                     11472 non-null  int64
29  info_ipv6                           11472 non-null  bool
30  irr_as_set                          11472 non-null  object
31  netixlan_set                        11472 non-null  object
32  ix_count                            11472 non-null  int64
33  port_capacity                       11472 non-null  float64
34  asn                                 11472 non-null  int64
dtypes: bool(6), float64(1), int64(6), object(22)
memory usage: 2.7+ MB
```

Figure 17: Nodes features PeeringDB

La plupart de ces features étaient inintéressants pour notre étude, soit car ils apportent une information déjà donnée par une autre feature (par exemple si deux features sont linéairement dépendantes) soit car elles n'apportent pas d'information pertinente pour le classification des AS (*info_ipv6* par exemple). Il a donc fallu les analyser et les comprendre afin de choisir les plus pertinentes pour la classification d'AS.

3.2 Sélection des node-feature pertinents

Se référer au notebook `creation_datasetgraph.ipynb` pour cette sous-partie.

Dans cette sous partie nous allons déterminer quelles sont les features que l'on va garder pour la classification. Une première feature intéressante est le `port_capacity` qui, comme nous avons vu dans la partie précédente nous donne une information essentiel sur le comportement d'un AS. Intéressons nous maintenant à la pertinence du feature **policy_locations**. Nous pouvons déjà voir que la distribution de `policy_locations` n'est déjà pas très intéressante à cause de sa distribution. Près de 90% des valeurs sont concentrées dans le champ 'Not Required'.

La première idée était d'afficher la corrélation entre ces données et la capacité du port. Par exemple, montrer que Required - US a un `port_capacity` élevée afin de créer une classe ordinale du champ `policy_locations`.

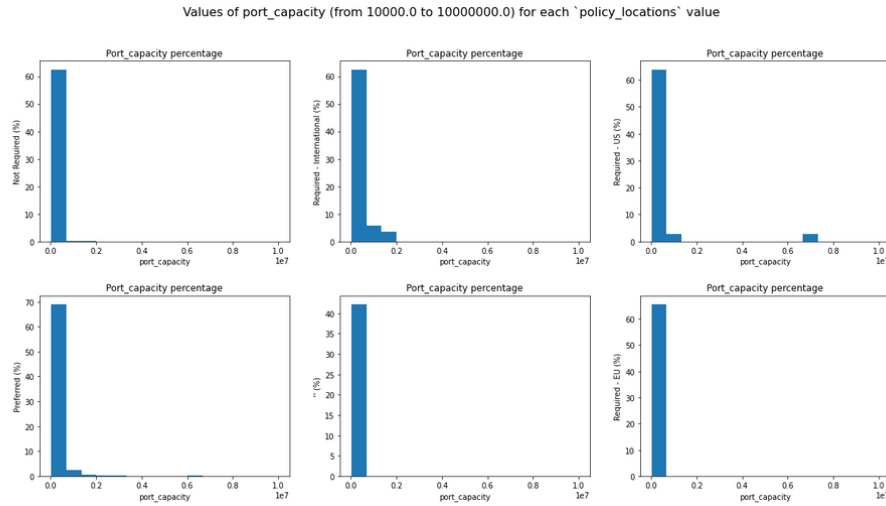


Figure 18: Corrélation entre `policy_locations` et `port_capacity`

Nous pouvons voir que les valeurs sont tout à fait les mêmes. Il n'y a aucune information ajoutée par `policy_locations` et sa distribution n'est pas sélective ou discriminante.

Par conséquent, nous n'utiliserons pas cette métadonnée.

Intéressons nous maintenant à la pertinence de la feature **info_traffic**.

Malgré les 27% de valeurs vides, la distribution est assez intéressante. Nous avons 18 champs qui ont en moyenne une bonne contribution (plus de 1%).

La seule chose à faire est d'attribuer les 27% de valeurs vides de la manière la plus pertinente possible.

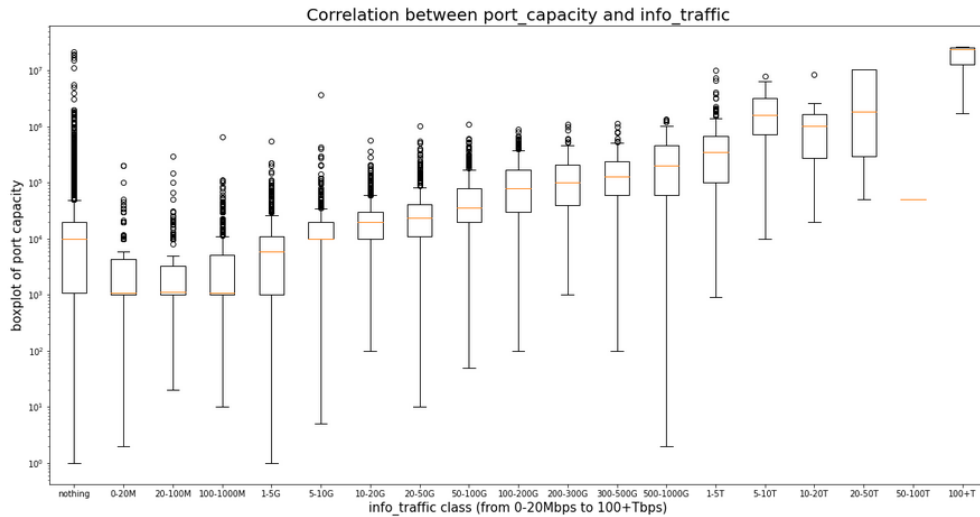


Figure 19: Corrélation entre `info_traffic` et `port_capacity`

La box à gauche est la valeur `port_capacity` des ASes avec des données `info_traffic` vides. Nous pouvons voir

que la médiane se situe entre les valeurs **5-10Gbps** et **10-20Gbps**. Par conséquent, nous pouvons étiqueter ces AS vides entre les valeurs de ces dernières.

En effet, info_traffic sera transformé en une classe ordinale avec 19 valeurs possibles allant de 1/19, 2/19 ... à 19/19 = 1 (**cf partie 3.3**).

Nous avons également gardé les features **net_count** et **ix_count** correspondant respectivement au degré des IXP et des AS. Ces features sont essentielles car ils donnent une information importantes sur le type d'AS. En effet, par exemple les **NSP** (Network Service Provider) vont avoir tendance à avoir un degré élevé alors que les **Enterprise** vont avoir un degré plus faible. Au final les features que nous avons gardé sont rassemblés dans le tableau suivant :

Attribute name	Meaning	Possible values
info_scope (AS)	origine géographique de la mesure	1/3 2/3 1
info_traffic	ordre de grandeur du trafic	1/18 2/18 3/18 4/18 5/18 6/18 7/18 8/18 9/18 10/18 11/18 12/18 13/18 14/18 15/18 16/18 17/18 1
ix_count (AS)	nombre d'IXP auquel l'AS se connecte	int scaled (0 to 1)
port_capacity (both)	bande passante d'un port	std value of port_capacity (from 0 to 1)
info_ratio (AS)	politique de communication de l'AS	0.2 0.4 0.6 0.8 1
policy_general	politique de trafic de l'AS	0.25 0.5 0.75 1
info_type (AS)	classification de l'AS (NaN pour un IXP)	0 1 2 3 4 5 6 7 8 9
net_count (IXP)	nombre d'AS auquel l'AS se connecte	int scaled (0 to 1)

Figure 20: Tableau récapitulatif des features gardés

3.3 Mise à l'échelle des node-features

Se référer au notebook `data_scaled.ipynb` pour cette sous-partie.

Les nodes-features choisis ne sont pas prêt à être utilisé directement par le GCN. En effet, nous devons fournir aux algorithmes des objets mathématiques (réels, vecteurs ...) et faire en sorte de mettre en évidence les écarts de valeurs observés. Pour la feature **info_scope** nous avons scale de façon ordinal : L'étiquette **Regional** prend la valeur 1/3, les étiquettes représentant des continents prennent la valeur 2/3 et l'étiquettes **global** prend la valeur 3/3. En ce qui concerne l'étiquette **Not Disclosed** nous l'avons placé à 2/3 également.

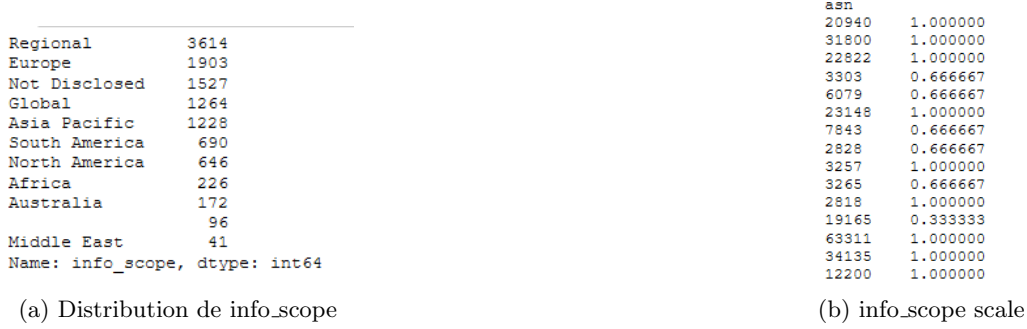


Figure 21: info_scope ditribution et valeurs scale

Pour la feature **info_traffic** nous avons également procédé de façon ordinal pour des valeurs allant de 1/19 à 19/19 (avec une valeur de 6/19 pour les noeuds n'ayant pas d'étiquette).

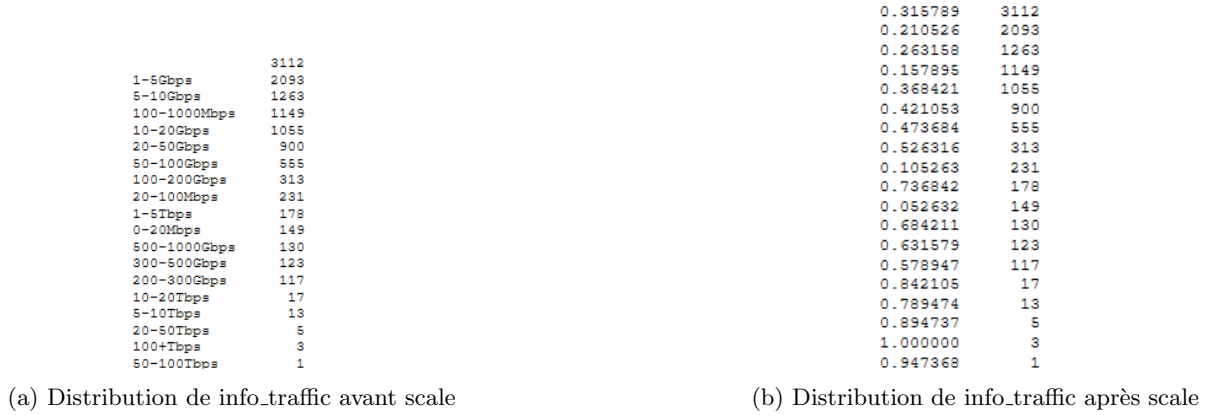


Figure 22: Distribution de info_traffic avant et après scale

Pour les features **net_count** et **ix_count**, on divise respectivement toutes les valeurs par $\max(\text{net_count})$ et $\max(\text{ix_count})$.



Figure 23: net_count et ix_count après scale

Pour le **port_capacity** : On divise par toutes les valeurs par le port_capacity le plus élevé indépendamment entre IXP et AS afin d'obtenir des valeurs entre 0 et 1. Comme pour CAIDA nous pouvons Nous appliquons également des méthodes ordinal pour les features **info_ratio** et **policy_general**

asn		asn	
20940	23992000.0	20940	0.893224
31800	6800.0	31800	0.000253
22822	7331000.0	22822	0.272934
3303	407700.0	3303	0.015179
6079	370000.0	6079	0.013775

39928	420000.0	39928	0.015637
204923	20000.0	204923	0.000745
133279	1000.0	133279	0.000037
34959	10000.0	34959	0.000372
61437	12500.0	61437	0.000465

(a) port_capacity avant scale

(b) port_capacity après scale

Figure 24: port_capacity avant et après scale

Finalement nous retrouvons dans le tableau suivant la façon dont nous avons scale toutes les features ; prêt à l'emploi par le GCN.

Attribute name	Meaning	Possible values
info_type (AS)	classification de l'AS (NaN pour un IXP)	0 1 2 3 4 5 6 7 8 9
info_ratio (AS)	politique de communication de l'AS	0.2 0.4 0.6 0.8 1
info_scope (AS)	origine géographique de la mesure	1/3 2/3 1
policy_general	politique de trafic de l'AS	0.25 0.5 0.75 1
info_traffic	ordre de grandeur du trafic	1/18 2/18 3/18 4/18 5/18 6/18 7/18 8/18 9/18 10/18 11/18 12/18 13/18 14/18 15/18 16/18 17/18 1
port_capacity (both)	bande passante d'un port	std value of port_capacity (from 0 to 1)
ix_count (AS)	nombre d'IXP auquel l'AS se connecte	int scaled (0 to 1)
net_count (IXP)	nombre d'AS auquel l'AS se connecte	int scaled (0 to 1)

Figure 25: Tableau récapitulatif des features PeeringDB scale

4 Mise en place du GCN

4.1 Récupération des données

On utilise github et on récupère les données sur le workspace google colab.

```
[3] #Initialize workspace with github and gg drive

from google.colab import drive
drive.mount('/content/drive')
%cd drive/MyDrive/ProjetLong/
! if cd projet_long_GCN_internet; then git pull; else git clone https://github.com/Viperine2022/projet_long_GCN_internet; fi
```

Figure 26: Récupération des données à partir de Github

Nous avons décidé d'utiliser Google Colab (un framework accessible en ligne) et pas nos propres machines en local parce que:

- Cet outil offre accès à une machine ayant 16 GB de mémoire RAM et assez Gb de mémoire stable de façon gratuite. Il s'agit de spécifications souvent plus performantes que celles dont nous disposons sur nos machines personnelles. En effet, on peut notamment utiliser l'ordinateur normalement pour coder ou autre pendant que le code s'exécute ce qui serait difficile si on lançait le code en local parce que l'exécution consommerait toutes les ressources disponibles
- La machine virtuelle à laquelle nous connecte google colab comporte déjà un environnement *Python 3* avec les librairies classiques pour des projets de machine learning comme **Tensorflow**. Ceci nous évite de faire des installations dans le cas des baselines cependant pour la version qui prend en compte des graphes et utilise la librairie **DGL** il faut installer certains modules comme **PyTorch**, **DGL** ou **Torchmetrics**.
- Finalement, cet outil permet de partager facilement le code et l'éditer de façon synchrone avec le reste de membres du groupe. Les solutions comme **GitHub** ou **SVN** sont aussi envisageables pour développer du code en groupe, d'ailleurs **GitHub** a été utilisé comme support pour la partie de traitement des données(notamment leur transmission d'un groupe à l'autre). Cependant, pour l'implémentation et exécution des GCN(Graph Convolutional Networks) ou NN (Neural Networks) en général un nombre réduit de lignes de code suffit pour la définition et c'est le choix des hyperparamètres et l'exécution qui est plus important, d'où qu'il soit envisageable de tout intégrer dans un même fichier dynamiquement modifiable.

4.2 Établissement d'une baseline

Consulter le notebook `baseline_caيدا.ipynb`

4.2.1 Caيدا

Chargement des données: Initialement les données étaient chargées en définissant autant de variables que de fichiers de données différents et utilisant la variable correspondant au path désiré comme le montre la photo suivante.

```
1 #Définition des paths
2 supervisedPath = "data/supervised/"
3 groupRepoPath = "groupe/"
4 dataPath1 = os.path.join(groupRepoPath, "IMPLANTATION/CAIDA/data_GCN/dataset_202001.csv")
5 dataPath2 = os.path.join("groupe/", "IMPLANTATION/CAIDA/data_GCN/dataset_20210301.csv")
6 dataPath3 = os.path.join("groupe/", "IMPLANTATION/CAIDA/data_GCN/dataset_v2_20210301.csv")
7 labelsPath = os.path.join(groupRepoPath, "IMPLANTATION/CAIDA/data_GCN/labels_202001.csv")
8 featuresPath = os.path.join(groupRepoPath, "IMPLANTATION/CAIDA/data_GCN/node_features_202001.csv")
9 supervised = pd.read_csv(dataPath3)
```

Figure 27: Chargement de données en utilisant plusieurs variables

Néanmoins on s'est vite rendu compte que cette méthode n'était pas scalable quand le nombre de base de données croît (ce qui arrive assez souvent puisque des corrections sont fréquemment nécessaires ce qui produit des nouvelles versions).

Ainsi en s'inspirant de des codes exemple disponibles sur l'apprentissage profond nous avons créé une fonction où nous passerions en argument le nom du dataset à charger.

Pour garder la trace de quel est le contenu de chaque dataset l'idéal serait de définir un fichier texte dans le répertoire GitHub contenant les dataset où pour chaque ligne on aurait un nom de fichier csv et une explication du contenu du dataset.

Ceci n'a pas été implémenté parce que le nombre de datasets manipulés n'était pas finalement si grand que ça donc ce n'était pas nécessaire. Dans la figure suivant vous pouvez voir la fonction python qui charge les données en fonction du nom de Dataset.

```

3 #le dataset doit se trouver dans le répertoire IMPLANTATION/CAIDA/data_GCN/ du répertoire github
4 def load_dataset(nom_dataset):
5     groupRepoPath = "groupe/"
6     pathData = os.path.join(groupRepoPath, "IMPLANTATION/CAIDA/data_GCN/")
7     #path vers le fichier qui contient les données
8     pathDataset = os.path.join(pathData, nom_dataset)
9     #on prend la dernière version du repository
10    if os.path.exists(groupRepoPath):
11        shutil.rmtree(groupRepoPath)
12    Repo.clone_from("https://github.com/Viperine2022/projet_long_GCN_internet.git", groupRepoPath)
13    #on charge les données dans un dataframe depuis un fichier en format CSV
14    dataset = pd.read_csv(pathDataset)
15    return dataset

```

Figure 28: Chargement de données en utilisant plusieurs variables

Verification visuelle: Ensuite il est intéressant de visualiser sous forme d'histogramme les données reçues de l'autre équipe pour valider leur cohérence et confirmer qu'il n'y a pas eu des erreurs grossières. Cette vue permet par exemple d'observer la distribution des valeurs pour chaque attribut. Ceci peut être fait par la fonction *hist()* de la classe DataFrame dans le module Pandas.

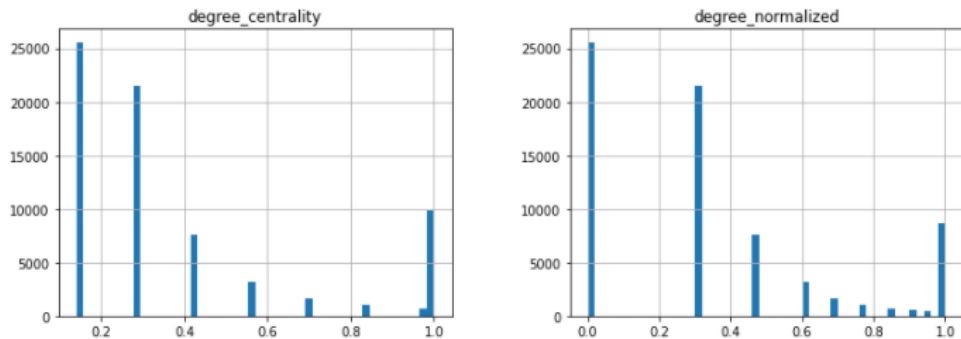


Figure 29: Affichage des valeurs de deux attributs pour le dataset

Une autre façon de visualiser rapidement quelques valeurs de la base de données est d'utiliser la fonction *head()* de la même classe qui affiche la première dizaine de datapoints. Cette deuxième méthode étant pourtant moins informative pourtant puisque les premières données risquent d'appartenir toutes à la même classe et avoir des valeurs d'attribut similaires ce qui n'est pas forcément représentatif sur tout le dataset ou ne donne pas beaucoup d'informations sur l'ensemble de données.

	ASN	source_label	label	peering_link	customer_link	provider_link	page_rank_not_directed	page_rank_directed	page_rank_directed_inverse	degree centrality	degree normalized
0	1	CAIDA_class	1	2.0	2.0	10.0	0.500388	1.0	0.675913	1.000000	1.000000
1	2	CAIDA_class	1	1.0	2.0	18.0	0.687322	1.0	0.169510	1.000000	1.000000
2	3	CAIDA_class	1	5.0	1.0	14.0	0.623851	1.0	1.000000	1.000000	1.000000
3	4	CAIDA_class	1	2.0	1.0	12.0	0.569730	1.0	0.155139	1.000000	1.000000
4	5	CAIDA_class	2	2.0	0.0	5.0	0.319572	1.0	0.142545	0.979939	0.845098

Figure 30: Affichage des premières valeurs du dataframe

Il est aussi intéressant de visualiser quels sont les attributs comportant la base de données chargé. Sachant que dans l'architecture choisie chaque attribut correspond à une colonne du dataframe nous pouvons obtenir l'ensemble de ces colonnes en appliquant la méthode *keys()* au dataframe pandas chargé du fichier CSV. Ce qui donne le résultat suivant :

```
Index(['ASN', 'source_label', 'label', 'page_rank_not_directed',
      'page_rank_directed', 'page_rank_directed_inverse', 'degree_centrality',
      'degree_normalized', 'ratio_peering', 'ratio_customer',
      'ratio_provider'],
      dtype='object')
```

Figure 31: Affichage des attributs présents dans le dataframe

Division des données: Ensuite, il est important de diviser notre dataset en un ensemble d'entraînement et un ensemble de test. En effet, si un réseau de neurones s'entraîne beaucoup sur un jeu de données, il pourra prédire exactement la classe de chaque élément dans le jeu de données.

Cependant ce même réseau aura un potentiel de generalisation assez faible : quand on lui présentera des nouvelles données son taux de réussite de prédiction diminuera drastiquement parce qu'il a fait ce qu'on appelle du **overfitting**. Ainsi nous conservons des un sous-échantillon de test pour évaluer la performance sur ces nouvelles valeurs lors de l'entraînement (ensemble de validation) ainsi que après l'entraînement (ensemble de test).

Cette séparation peut être réalisé par plusieurs méthodes, mais nous avons choisi d'utiliser deux fois la méthode `train_test_split` du module **sklearn**. Un autre choix possible est de modifier le paramètre `validation_split` dans la méthode `fit()` de **Keras** où ce paramètre représente le pourcentage de données qui seront utilisés pour la validation pendant le processus d'entraînement. Cependant, en utilisant cette option dans la pratique nous avons obtenu des résultats incohérents donc nous avons choisi d'appliquer la première option.

Selection des attributs: même si nous disposons d'un large choix d'attributs dans la base de données sur lesquels réaliser nos itérations il est peut être intéressant de choisir quelques sous-ensembles d'attributs pour contempler l'impact du choix sur les résultats. En particulier nous pouvions utiliser l'analyse de corrélation réalisé par l'autre équipe pour garder un ensemble d'attributs qui seraient des variables aléatoires les plus indépendantes possibles.

Dans la pratique nous pouvons simplement définir une liste de strings contenant les attributs que nous voulons conserver et passer cette liste comme index au dataframe comme le montre la figure suivante. Ceci restreindra les colonnes du dataframe pandas aux colonnes dont le nom se trouve dans la liste passée en index.

```
1 #Configurations d'attributs pour l'entraînement
2 attributsLred = ['page_rank_not_directed', 'degree_centrality', 'peering_links', 'customer_links', 'provider_links']
3 attributsL = ['page_rank_not_directed', 'page_rank_directed', 'page_rank_directed_inverse', 'degree_centrality', 'peering_links',
4              'customer_links', 'provider_links']
5 attributsPage = ['page_rank_not_directed', 'page_rank_directed', 'page_rank_directed_inverse', 'degree_centrality']
6 attributsRatio = ['page_rank_not_directed', 'page_rank_directed', 'page_rank_directed_inverse',
7                  'degree_normalized', 'ratio_peering', 'ratio_customer', 'ratio_provider']
8 attributsJusteRatio = ['degree_normalized', 'ratio_peering', 'ratio_customer', 'ratio_provider']
9 uniqueatr = ['degree_centrality']
10
11 #Ligne pour choisir les attributs
12 attributs = attributsRatio
13 x_train = train set final[attributs]
```

Figure 32: Selection d'un sous-ensemble d'attributs pour l'entraînement

Définition du modèle: Maintenant nous définirons notre modèle à l'aide de la classe **Sequential** dans le module **Keras.models** qui nous permet de définir des réseaux neuronaux simples et séquentiels.

Quant aux **couches** nous utiliserons que des couches *Denses* et des couches de *Dropout*, ces premières constituant une implementation directe de la forme la plus simple de **MultiLayerPerceptron** et ces deuxièmes permettant de diminuer l'impact de l'échantillon de données sur les poids du NN en oubliant un certain pourcentage de poids à chaque passage par la couche.

Quelques contraintes existent sur la première et la dernière couche. D'une part il faut que la première couche aie la même dimension que le nombre d'attributs disponibles sur chaque échantillon pour pouvoir donner ces échantillons en input au modèle.

D'autre part comme il s'agit d'une tâche de classification **multiclasse** et pas binaire il faut que l'output à la dernière couche soit une distribution de probabilité avec autant de valeurs possibles que de classes disponibles (donc un vecteur de taille *Nombre de classes*) ce qui impose la taille de cette dernière couche. De plus pour assurer que tous les valeurs du vecteur somment 1 (distribution de probabilité) nous devons appliquer une fonction d'activation *softmax* à cette dernière couche. Pour le reste de couches nous avons choisi de prendre des fonctions d'activation *tanh* parce que les résultats étaient meilleurs que avec la fonction *ReLU* ou *sigmoïde*

```

1 #definition du modèle
2 model = keras.Sequential(
3     [Dense(input_dim=len(attributs),units =50,activation='tanh'),
4       Dropout(0.2),
5       Dense(50,activation='tanh'),
6       Dropout(0.2),
7       Dense(50,activation='tanh'),
8       Dense(3,activation='softmax')
9 ])

```

Figure 33: Modèle sequential utilisé pour l'entraînement

Pour choisir le nombre de couches et le nombre d'unités par couche nous avons décidé d'utiliser une methode souvent utilisé en machine learning comportant deux étapes:

- Pendant la première étape nous forçons l'overfitting du réseau en ajoutant un grand nombre de couches et beaucoup d'unités par couche pour donner au modèle la capacité d'apprendre le jeu de données. Il s'agit d'assurer que le modèle est assez expressif pour pouvoir rendre compte des patrons qui découlent des données (en pratique ici nous avons expérimenté avec entre 3 et 5 couches et 100 - 200 unités par couche).
- Pendant la deuxième étape nous réduisons au maximum le modèle tout en garrant la propriété de convergence. Nous voulons obtenir le plus petit modèle qui soit capable de rendre compte des données. Finalement, nous voulons que notre modèle soit aussi adapté à l'ensemble de validation que à l'ensemble d'entraînement. Pour avoir cet effet nous allons ajouter des couches de Dropout et de la regularization pour que le modèle soit un peu moins sensible aux données d'entraînement et il puisse apprendre d'une façon plus généraliste.

Autres hyperparamètres: Nous avons agi sur d'autres hyperparamètres dans le processus d'apprentissage comme le choix de l'optimisateur, **Adam** étant celui par défaut et **RMSprop** étant celui conseillé pour des problèmes de classification multiclass. Définir notre propre objet d'optimisateur à partir de **Keras** nous permet d'autre part de changer la valeur du **Learning Rate**. Pour cette valeur $1e-4$ est le paramètre le plus utilisé mais nous pouvons également prendre jusqu'à $1e-3$ pour avoir des oscillations plus importantes dans la précision au fil des epochs pendant l'entraînement. La fonction de perte utilisé pour l'entraînement est celle de *Categorical Crossentropy* (si les labels sont passés sous la forme d'un codage One-hot) ou dans ce cas la *Sparse Categorical Crossentropy* (si les labels sont passés sous la forme d'un entier entre 0 et le nombre de classes disponibles)

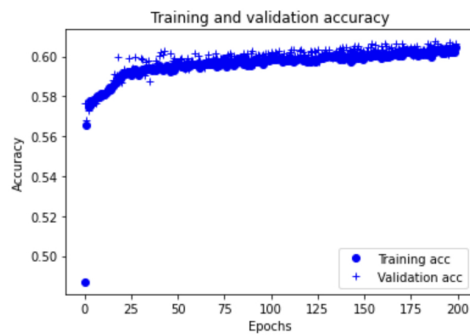
```

1 opt = tf.keras.optimizers.Adam()
2 opt.learning_rate = 1e-3
3 model.compile(loss='sparse_categorical_crossentropy',metrics=['accuracy'],optimizer=opt)

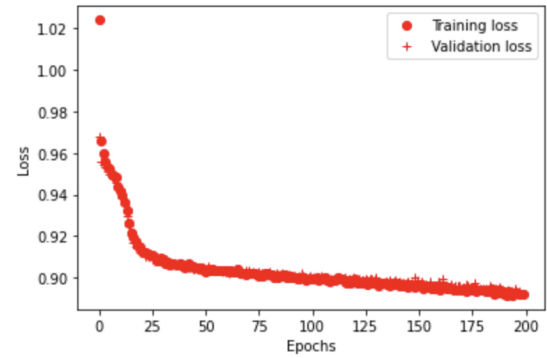
```

Figure 34: Définition de l'optimisateur et compilation du modèle

Lancement de l'entraînement: L'entraînemet se lance avec la fonction *fit()* et itere un certain nombre de fois. Ce nombre de fois est appelé **epoch** et nous pouvons le choisir. Néanmoins dans la pratique il s'avère que l'entraînement converge au bout d'entre 50 et 100 epoch et il est inutile d'en ajouter plus. Une autre option est de lancer l'entraînement sur un grand nombre d'epoch et configurer un early stop pour que l'algorithme s'arrête une fois un objectif d'accuracy ou de perte est atteint. D'autre part, dans une itération de l'entraînement nous n'appliquons pas la backpropagation(mise à jour des paramètres) après calcul de prediction sur toutes les données, nous la calculons après passage sur un sous-ensemble des données qu'on appelle **batch**. La taille de cet sous-ensemble est aussi réglable et l'argument s'appelle **batch size** dans les modules utilisés. Plus le batch size est petit plus l'entraînement produira un modèle capable de generaliser, et plus le batch size est grand plus le modèle resultant sera adapté à l'ensemble d'entraînement. Comme l'objectif final de cet analyse est de déterminer dans quelle mesure la topologie du réseau internet nous permet de conclure sur la nature des AS qui le composent; nous avons choisi de prendre des batch-size plutôt petits pour avoir une forme d'inférence sur notre modèle final. L'entraînement produit des graphes qui ont l'allure suivante:



(a) Plot de l'accuracy sur l'ensemble d'entraînement et de validation



(b) Plot de la valeur de la fonction de perte sur les ensembles d'entraînement et de validation

Figure 35: Affichage de la précision du modèle et la valeur de la fonction de perte après chaque epoch

Métrique d'évaluation des résultats: Pour évaluer les résultats la précision du modèle sur l'ensemble de test est une mesure très peu expressive de la validité réelle de ce dernier. Pour connaître la qualité réelle du réseau neuronaux après apprentissage nous avons choisi d'utiliser

- **Matrice de confusion:** Après avoir produit des prédictions par le modèle nous pouvons facilement obtenir les classes prédites pour chaque donnée à l'aide de la fonction `np.argmax(predictions,axis=1)` qui donne l'indice de la probabilité maximale dans l'output pour chaque donnée, c'est à dire sa classe prédite. Ensuite il suffit de donner ce vecteur ainsi que le vecteur comportant les vrais labels présents dans la base de données (notre vérité terrain) à la fonction `confusion_matrix` dans le module `sklearn.metrics`. Pour comprendre le fonctionnement d'une matrice de confusion vous pouvez regarder l'image suivante dans laquelle "Negative" et "Positive" seraient deux classes différentes (alors que dans notre dataset Caida nous pouvons classifier dans 3 classes)

		True Class	
		Positive	Negative
Predicted Class	Positive	TP	FP
	Negative	FN	TN

Figure 36: Description du principe d'une matrice de confusion

- **Classification Report:** La matrice de confusion est un bon premier indicateur mais elle est souvent exprimé en termes d'effectif global et il est plus intéressant de visualiser des pourcentages. Ceci peut être fait à l'aide de la fonction `classification_report` du même module `sklearn.metrics`. Sur le vocabulaire présent dans l'exemple de classification report suivant il faut savoir que (avec le meme vocabulaire que pour l'image de la matrice de confusion) :

- *Precision* : Mesure la capacité du classifieur à ne pas classer comme positif un label qui est négatif
- *Recall* : Mesure la capacité du classifieur à trouver tous les samples positifs (c'est la mesure la plus intéressante dans notre analyse parce qu'une mauvaise précision n'est pas si grave)
- *f1-score*: Moyenne harmonique de la precision et du recall

	precision	recall	f1-score	support
class 0	0.50	1.00	0.67	1
class 1	0.00	0.00	0.00	1
class 2	1.00	0.67	0.80	3
accuracy			0.60	5
macro avg	0.50	0.56	0.49	5
weighted avg	0.70	0.60	0.61	5

Figure 37: Exemple d'output du classification report

Premiers résultats: En appliquant le modèle tel qu'il est défini sur la figure préalable (meilleure configuration d'hyperparamètres que nous avons trouvé par "force brute") et sur les données sans prétraitements autres que ceux réalisés dans la première section du projet nous obtenons la matrice de confusion et classification report suivants:



Figure 38: Affichage des metriques pour le modèle entraîné sur les données brutes

Nous observons une assez bonne accuracy générale (0.76) cependant en regardant les classes on découvre qu'une seule classe est prédite systématiquement, **Transit/Access** c'est à dire la classe majoritaire dans le dataset. Ce problème découle des proportions présentes dans les données et elle peut être résolue en appliquant la technique dite d'oversampling (generer plus de données dans les classes avec moins d'éléments pour ainsi avoir le même nombre d'elements dans toutes les classes) ou celle de downsampling (nous pouvons également restreindre l'ensemble d'éléments dans chaque classe à la classe la plus minoritaire)

Downsampling les données: C'est la première technique utilisé en l'appliquant avec la fonction `sample()` de la classe **DataFrame** dans le module **Pandas**. Comme nous l'observons sur l'image, les prédictions sont maintenant plus équilibrés avec cependant une claire tendance à prédire dans la classe entreprise (nous n'avons pas identifié la raison). Mais l'accuracy en general est de *0.28* ce qui est pire qu'un prédicteur aléatoire. C'est normal car seulement 10% des données dans la classe majoritaire sont bien prédites (les proportions dans l'ensemble de test conservent les mêmes proportions que les données originelles)

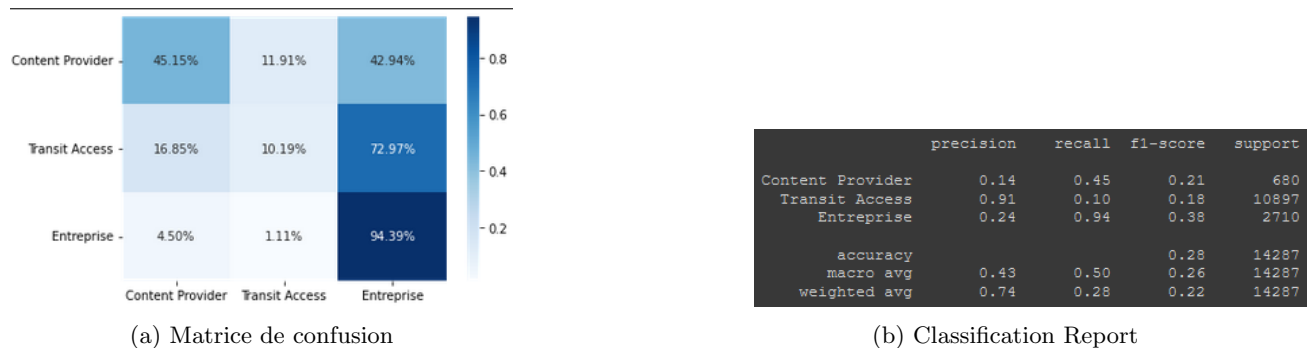


Figure 39: Affichage des metriques pour le modèle entraîné sur les données downsampled

Oversampling les données: L'opération a été réalisé avec la fonction `resample()` du module **sklearn.utils**.

Nous observons qu'en réalisant cette opération les recalls sont encore plus équilibrés qu'en réalisant le down-sampling avec une meilleure performance pour la classe "Entreprises" mais moins marquée. Comme la classe majoritaire a un meilleur recall ceci se traduit par une accuracy de 0.57 qui a presque doublé par rapport à la technique précédente. Ce résultat est le meilleur obtenu en prenant compte des recalls sur les différentes classes. Malheureusement comme nous le verrons plus tard cette technique est difficilement applicable aux graphes (par souci de conservation de topologie)

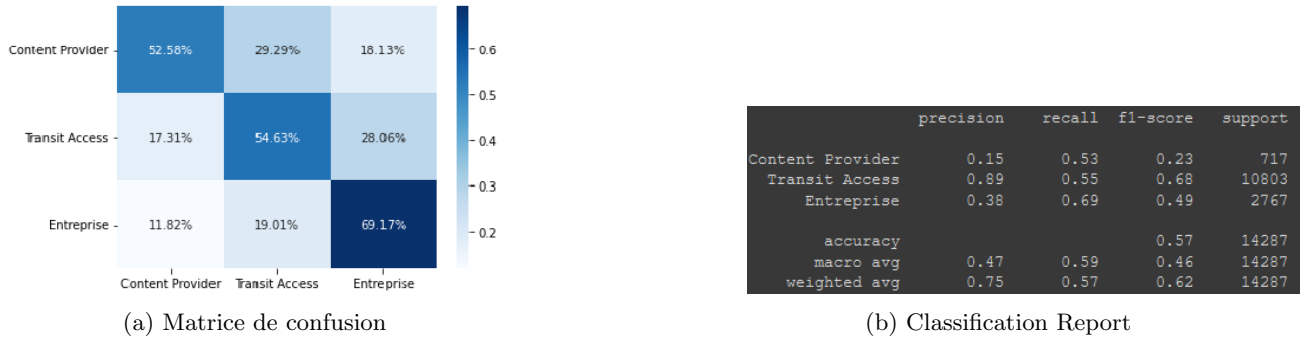


Figure 40: Affichage des metriques pour le modèle entraîné sur les données oversampled

Utilisation d'un forêt aléatoire: Finalement, nous signalons que pour chaque jeu de données considéré nous avons utilisé un forêt aléatoire (*RandomForestClassifier*) extrait du module `sklearn.ensemble` pour savoir si les résultats obtenus avec le MLP étaient pertinents par rapport à ce classifieur classique. Néanmoins nous n'afficherons pas les résultats pour avec ce classifieur parce qu'ils sont assez proches de ceux du réseau neuronaux

4.2.2 PeeringDB

Consulter le notebook `baseline_peeringdb.ipynb`

Dans le dataset peering DB nous avons un problème similaire de déséquilibre de classes comme nous le pouvons voir sur la colonne support du classification report (le support est l'ensemble d'effectifs appartenant à la classe dans l'ensemble de test). Ceci produit que l'accuracy 31% soit assez mauvaise en prenant les données telles qu'elles sont passées.

	precision	recall	f1-score	support
Cable/DSL/ISP	0.79	0.29	0.43	1302
NSP	0.35	0.31	0.33	422
Content	0.46	0.59	0.52	230
Enterprise	0.08	0.19	0.11	112
Educational/Research	0.12	0.24	0.16	100
Non-Profit	0.12	0.08	0.09	64
Route Server	0.09	0.36	0.14	44
Network Services	0.01	0.25	0.02	4
Route Collector	0.00	0.00	0.00	3
Government	0.00	0.00	0.00	1
accuracy			0.31	2282
macro avg	0.20	0.23	0.18	2282
weighted avg	0.58	0.31	0.37	2282

Figure 41: Classification Report avec toutes les classes

Cette fois réaliser de l'oversampling n'a pas un effet considerable sur les résultats parce que les classes avec

peu d'échantillons sont très petites donc le surechantillonnage est trop "artificiel" et l'accuracy n'améliore pas beaucoup (la distribution des recalls reste aussi à peu près pareille)

	precision	recall	f1-score	support
Cable/DSL/ISP	0.79	0.31	0.45	1302
NSP	0.39	0.29	0.33	422
Content	0.48	0.57	0.52	230
Enterprise	0.07	0.19	0.10	112
Educational/Research	0.16	0.33	0.22	100
Non-Profit	0.07	0.06	0.07	64
Route Server	0.10	0.50	0.16	44
Network Services	0.01	0.25	0.01	4
Route Collector	0.00	0.00	0.00	3
Government	0.00	0.00	0.00	1
accuracy			0.33	2282
macro avg	0.21	0.25	0.19	2282
weighted avg	0.59	0.33	0.39	2282

Figure 42: Classification Report avec toutes les classes et oversampling

La seule option qui a considérablement amélioré les performances est de prendre seulement les 5 premières classes les plus représentées et soustraire celles très minoritaires de l'analyse. Comme vous pouvez le voir sur le report suivant en utilisant cette méthode nous obtenons des accuracies de l'ordre de 50% avec des valeurs de recall acceptables.

	precision	recall	f1-score	support
Cable/DSL/ISP	0.76	0.55	0.63	1279
NSP	0.35	0.30	0.32	418
Content	0.49	0.56	0.52	228
Enterprise	0.11	0.28	0.16	130
Educational/Research	0.13	0.39	0.20	101
accuracy			0.48	2156
macro avg	0.37	0.41	0.37	2156
weighted avg	0.58	0.48	0.51	2156

Figure 43: Classification Report avec 5 classes

4.3 Implémentation de GraphSAGE et d'autres méthodes

Se référer au notebook `GCN_implementation_CAIDA.ipynb` pour cette sous-partie.

Chargement des données: Pour l'utilisation de GCN, nous récupérons les données de la première équipe sous 2 formes.

Des fichiers au format **pickle** qui représentent les graphes (ensemble de noeuds, ensemble d'arêtes, label de chaque noeud). Ainsi que des fichiers au format **csv** qui représentent les attributs des noeuds sous la forme de dictionnaires.

Division des données : Pour rendre les données utilisables, nous allons utiliser la même méthode que pour la Baseline. Nous divisons l'ensemble de données en 3 groupes l'ensemble de test, l'ensemble de validation et l'ensemble d'entraînement. Nous représentons ces ensembles à l'aide de masques que nous ajoutons au graphe sous la forme d'attributs de noeuds.

```
# Loading Graph with pickle5
path_to_protocol5 = 'IMPLANTATION/CAIDA/data_GCN/graph_float_20210301.pickle'
with open(path_to_protocol5, "rb") as fh:
    G_float = pickle.load(fh)

# Adding masks on graph
nx.set_node_attributes(G_float, dict_train_mask, "train_mask")
nx.set_node_attributes(G_float, dict_test_mask, "test_mask")
nx.set_node_attributes(G_float, dict_val_mask, "val_mask")
```

Figure 44: Ajout des masques des différents ensembles de données

Ensuite nous transformons le graphe du format **Pickle** au format **dgl** pour pouvoir appliquer des opérations de GCN que l'on définira plus tard. Nous ajoutons aussi les attributs extraits du dataframe panda au graphe dgl pour pouvoir les utiliser dans les opérations du GCN (nous effectuons la même sélection que pour la Baseline), sans ces ajouts d'informations directement en tant qu'attributs des noeuds du graphe, les résultats étaient vraiment mauvais.

```
# Conversion of the graph to dgl format
G_dgl_float = dgl.from_networkx(G_float, node_attrs=['label', 'train_mask', 'val_mask', 'test_mask'])

# extracting features from the dataset
data_features = dataset.iloc[:,range(3,11)]

print(data_features)

data_features_dgl = data_features.to_numpy().astype(float)

# Adding features to dgl graph
G_dgl_float.ndata["feat"] = torch.tensor(data_features_dgl)
```

Figure 45: Ajout des masques des différents ensembles de données

Équilibrage des données d'entraînement :

Étant donné que les classes ne sont pas équitablement représentées dans l'ensemble d'entraînement nous risquons de nous retrouver avec un résultat biaisé où la classe la plus présente dans l'ensemble d'apprentissage est surprésentée pendant l'apprentissage et est donc privilégiée par l'algorithme (voir ci-dessous):

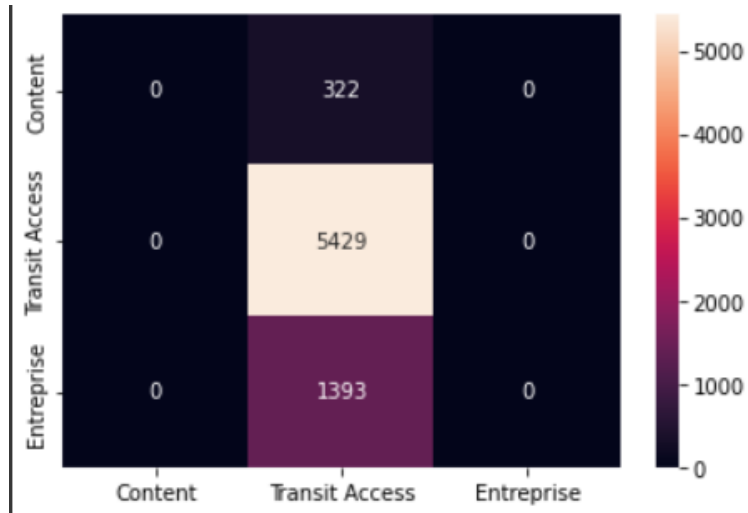


Figure 46: Resultat GCN sans équilibrage des classes

Pour rééquilibrer les classes dans l'ensemble d'entraînement, il existe plusieurs méthodes, cependant comme les données sont représentées sous forme de graphes, nous avons seulement réussi à sous-échantillonner les classes majoritaires. En effet l'équilibrage par sur-échantillonnage est très complexe sur des graphes, car pour ne pas perdre d'informations importantes, il faut ajouter de nouveaux noeuds tout en préservant la topologie du graphe. Dans cette partie, nous nous sommes donc contenté d'un équilibrage par sous-échantillonnage. Ce rééquilibrage est effectué dans le bloc `Undersampling of train set`

Définition du modèle: Nous avons défini notre modèle en nous basant principalement sur la fonction **SageConv** de la librairie **dgl.nn** qui permet de définir une opération de convolution de type GraphSAGE sur un graphe donné. Nous avons donc testé différentes architectures de GCN à l'aide de cette opération, après plusieurs essais nous sommes parvenus à l'architecture suivante.

```
class GCN(nn.Module):  
    def __init__(self, in_feats, h_feats, num_classes):  
        super(GCN, self).__init__()  
        self.conv1 = SAGEConv(in_feats, h_feats, 'pool')  
        self.conv2 = SAGEConv(h_feats, num_classes, 'pool')  
  
    def forward(self, g, in_feat):  
        h = self.conv1(g, in_feat)  
        h = F.torch.tanh(h)  
        h = self.conv2(g, h)  
        return h
```

Figure 47: Architecture GCN

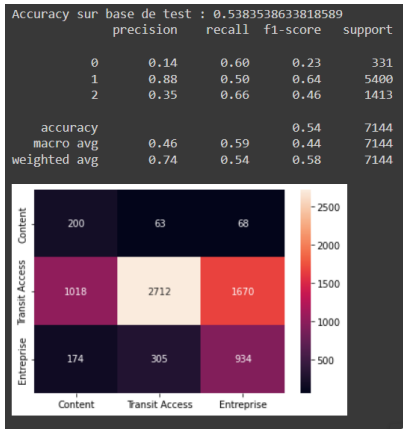
L'architecture la plus efficace que nous avons trouvée est composée de seulement deux couches de convolution de type **SageConv** avec comme fonction d'agrégation, la fonction **Pool** qui était la meilleure que nous pouvions utiliser (nous avons vu que la fonction **lstm_lstm** était la plus conseillée cependant nous n'avions pas la RAM nécessaire pour utiliser notre GCN avec cet agrégateur là).

De plus, nous avons testé plusieurs fonctions d'activation et nous avons choisi de prendre *tanh* parce que les résultats étaient meilleurs qu'avec les autres fonctions d'activation.

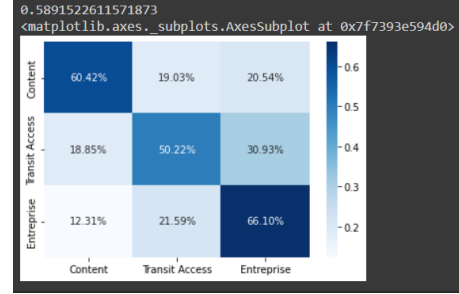
Autres hyperparamètres : Comme pour la BaseLine, nous avons agi sur d'autres hyperparamètres, nous utilisons l'optimiseur **Adam**. Pour la valeur du learning rate, nous prenons $1e-3$ pour avoir des oscillations plus importantes dans la précision au fil des epochs pendant l'entraînement. La fonction de perte utilisé pour l'entraînement est celle de *Crossentropy*.

Résultats:

En appliquant la tâche de classification avec le GCN défini précédemment et en optimisant un maximum les hyperparametres, nous obtenons les résultats suivants.



(a) Plot de la matrice de confusion et du classification report



(b) Plot de la matrice de confusion en pourcentages

Figure 48

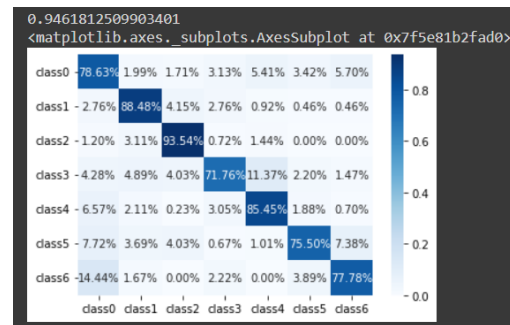
Nous pouvons voir que les résultats sont à peu près équivalents à ce que l'on obtenait avec la BaseLine avec l'oversampling. Nous obtenons un recall moyen de 59%, et il est très difficile de faire mieux. Les données du dataset CAIDA étant fiables à environ 70%, nous espérons nous rapprocher de ce résultat là, nous avons quelques explications à cela.

- Premièrement, les attributs ajoutés aux noeuds représentent seulement la topologie du graphe, nous n'avons pas ajouté d'attributs n'ayant aucun rapport avec la structure au graphe donc le réseau de neurones ne possède pas d'information supplémentaire par rapport à la BaseLine.
- Deuxièmement, il est possible que la topologie du graphe ne permettent pas suffisamment de différencier les différentes classes.

Lorsque l'on teste avec une base de données benchmark proposé par la librairie **dgl** (**Cora dataset**) avec la même structure de GCN, nous obtenons de bien meilleurs résultats.



(a) Plot de la matrice de confusion et du classification report



(b) Plot de la matrice de confusion en pourcentages

Figure 49

En effet nous obtenons un recall moyen de 81,6%, sans avoir adapté notre réseau de neurone à ce dataset en particulier. Nous pouvons donc supposer que notre GCN ne possède pas assez d'informations pour correctement classer nos données.

- Troisièmement, l'ensemble d'entraînement est drastiquement réduit à cause du sous-échantillonnage, ce qui a forcément une conséquence sur notre résultat. En effet, la classe minoritaire (**Content**) représente 5% de nos données, cela veut dire qu'après le sous-échantillonnage, il ne nous reste que 15% de notre base d'entraînement originale.

4.4 Pistes de recherche

Pour améliorer les résultats obtenus nous pourrions avoir :

- Utilisé une implémentation d'oversampling pour des graphes comme **GraphSMOTE** (paper qui date de 2021) qui utilise un modèle de prédiction d'arrêtes entraîné en parallèle avec le modèle principal pour générer des échantillons supplémentaires des classes sous-représentées
- Utilisé l'apprentissage non-supervisé pour pouvoir comparer les clusters déduits par cette méthode aux clusters présentés par CAIDA et PeeringDB. Ceci a été tenté dans le notebook `TorchLightning.ipynb`

5 Conclusion

En somme, pour étudier le trafic d'internet, nous avons du parser des fichiers provenant des bases de données de CAIDA et de PeeringDB en vue de créer les graphes et datasets correspondant aux jeux de données disponibles. Il nous a fallu ensuite sélectionner et/ou créer des nodes-features pertinents, par diverses techniques de visualisation de donnée. Enfin, le traitement des données se concluait par une mise à l'échelle de chaque node-feature qui, scalé entre -1 et 1 et avec une bonne distribution, devient une information directement traitable par le GCN.

L'analyse avec baseline nous a permis de conclure qu'un modèle est capable d'inférer la nature des AS à partir d'attributs déduits de la topologie du graphe. Cependant, les prédictions sont de mauvaise qualité lorsque les données sont déséquilibrées, ce qui était le cas avec nos données : là où certaines classes étaient majoritaires, d'autres étaient présentes en proportion négligeable. Pour l'analyse avec des MLP nous avons réussi à régler ce problème en utilisant des techniques de downsampling et oversampling, la deuxième étant significativement meilleure.

Malheureusement, même si les GCN ont présenté des performances équivalentes à la baseline dans leur version non-équilibrée, nous n'avons pas réussi à appliquer les mêmes techniques parce qu'il est difficile d'ajouter des nouveaux noeuds à un graphe tout en conservant sa topologie. Certains travaux récents présentent des solutions pour ce problème d'équilibrage de classes dans certains contextes mais nous sommes encore loin d'atteindre un consensus sur la meilleure méthode à utiliser.