

Détection d'intrusions réseaux

Jiacheng Zhou, Gaëtan Le Frioux, Lorys Hamadache

29 mars 2025

Résumé

Ce document restitue le travail de notre groupe de projet réalisé dans le cadre du cours de **SY09** - Analyse des données et Data-Mining à l'Université de Technologie de Compiègne - *UTC*. Notre groupe s'intéresse aux données d'intrusions dans les réseaux informatiques grâce au jeu de donnée appelé **KDD Cup 1999 Data set** [1].

Nous veillerons à expliquer notre démarche à chaque étape de notre étude.

1 Introduction

Commençons dans un premier temps par essayer de comprendre notre jeu de donnée et les enjeux de celui ci.

Notre jeu de donné est le **KDD Cup 1999 Data set**. Ce jeu de donnée à été utilisé pour une compétition de Data Mining *The Third International Knowledge Discovery and Data Mining Tools Competition* en 1999 d'où son nom. La tache principale de la compétition était de créer un détecteur d'intrusion, pouvant prédire le type des connexions entrantes en différenciant une "bonne" connexion d'une attaque. Ce Data set est une simulation de connexions et d'intrusions dans un réseau militaire simulé.

C'est un sujet intéressant car répondant à une problématique réelle et importante. Aujourd'hui, nos infrastructures réseaux prennent une place centrale et plus importante que jamais. Les services étatiques, la défense, les entreprises et les particuliers sont de plus en plus touchés par des intrusions réseaux. En 2017,

d'après *datasecuritybreach.fr* [2] plus de 700 Millions d'attaques ont été enregistrées. C'est une augmentation de 100% par rapport à 2015. C'est un enjeux cruciale de Défense mais aussi économique puisque, d'après *Microsoft* [3], c'est 81% des entreprises françaises qui sont touchées avec un coût moyen par violation de 800 000 euros. C'est donc un enjeu majeur aujourd'hui.

Notre but sera donc le même que celui de la compétition : c'est à dire construire un modèle nous permettant de détecter les "mauvaises" connexions aux réseaux. Ce but pourra évoluer avec par exemple la volonté de caractériser le type d'attaque ou d'attaquant.

Notre étude se compose des parties suivantes :

1. Cette Brève Introduction
2. Découverte et Analyse du Data set
3. Modèles d'apprentissage supervisé
4. Conclusion et Perspectives

2 Le Data set

2.1 Découverte

En se rendant sur la page du Data set [1], on remarque que l'on nous propose plusieurs fichiers :

- *kddcup.names* Qui est une liste des différentes features
- *kddcup.data.gz* Le data set d'entraînement complet (753 Mb)
- *kddcup.data_10_percent.gz* 10% du data set précédent pour faciliter les analyses préalables.
- *Plusieurs jeu de données sans label* Nous n'utili-

- liseros pas ces fichiers qui étaient les données à classifier pour les candidats (sans les résultats)
- *corrected.gz* Le jeu de donnée test avec la correction
 - *training_attack_types* La liste des type d'attaques (labels)

Les features sont très nombreuses et aux nombres de 41 et à la fois qualitatives et quantitatives. Cela risque de poser quelques problèmes lors de l'application d'algorithmes. Nous verrons comment gérer cela. Le data set complet contient plus de 5 Millions d'individus / connexions. C'est assez énorme et risque de beaucoup ralentir l'exécution des différents modèles et poser des problèmes de visualisation. C'est pourquoi, dans un premier temps, nous travaillerons avec le data set ne contenant que 10% des données puis appliquerons , ou non le modèle au data set complet.

Les type de connexions sont aux nombres de 23 avec par exemple un connexion "normal", ou des attaques "smurf" et "back". Ces connexions sont divisées en 5 catégories : "normal" ou attaque "dos", "u2r", "r21" et "probe".

On pourra s'intéresser aux différents niveaux de profondeurs de ces attaques en commençant par le principal, savoir si c'est une attaque ou pas, puis ça catégorie et enfin son type

2.2 Analyse

Dans un premier temps, on remarque rapidement qu'il y a de nombreuses lignes identiques dans notre jeu de données. Ce sont des données qui sont redondantes lors de l'application des modèles. Nous enlèverons ces duplicita lors de l'utilisation d'algorithme d'apprentissage. On passe de 494 021 individus à 145 586 individus uniques.

Commençons par regarder la distribution des attaques dans notre jeu de données 10%

On remarque que ce sont les attaques qui sont beaucoup dupliquées, en particulier de classe Dos et de

Connexions	TABLE 1 – Distribution des connexions.	
	Sans modification	Sans Duplicatas
Normales	97 278	87 832
Attaques	396 743	57 754

Connexions	TABLE 2 – Distribution des connexions.	
	Sans modification	Sans Duplicatas
Normales	97 278	87 832
Dos	391 458	54 572
U2r	52	52
R21	1 126	999
Probe	4 107	2 131

type smurf. Cela est logique puisque les attaques par déni de services sont extrêmement similaires et répétées massivement.

Ce jeu de données est très hétérogène entre les classes. La distribution est loin d'être uniforme. Nous avons donc des classes avec très peu de données. Cela peut rapidement nous mener à une classification erronée, par exemple en ayant peu d'erreur simplement en ignorant les classes minoritaires. Dans le cas présent c'est extrêmement problématique, avec un classificateur qui risque de considérer la majorité des connexions comme normales. Il est plus dangereux de considérer une attaque comme une connexion normale que l'inverse. Si on laisse passer une attaque, les conséquences pour le réseau peut être désastreuse mais bloquer une connexion normale est presque sans conséquence. Pour pallier à ce problème, si il se pose, on pourra réaliser un échantillonnage des classes trop majoritaires.

Un autre petit point à éclaircir est le domaine de définitions des features. Avec beaucoup de variables binaires ou souvent à zéro mais aussi des valeurs proche de la dizaine de millions, il y a une grande disparité entre les features. Afin d'aider les modèles d'apprentissage, nous normaliserons souvent les données dans l'intervalle [0,1].

Nous réalisons une ACP afin de visualiser les don-

nées à 41 dimensions sur 2. On garde les 2 axes expliquant le plus les données.

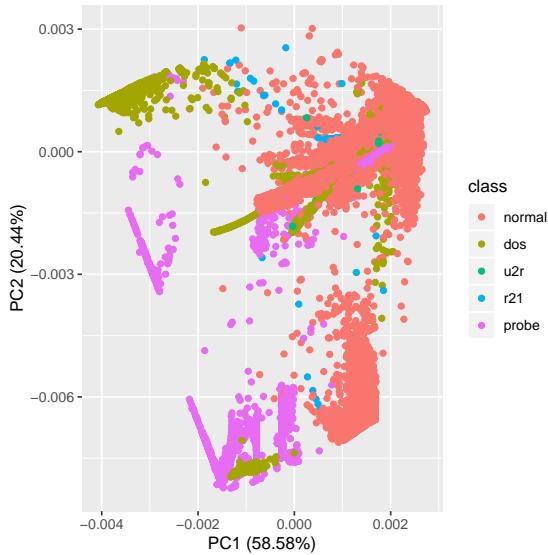


FIGURE 1 – ACP du jeu de données sur les axes principaux avec la couleur représentant le type de connexion, générée avec R.

On peut observer quelques clusters évidents mais on se rend compte qu'il ne sera pas si aisés de séparer les différents types d'attaques. Nous cherchons à réaliser une classification supervisée c'est pourquoi on utilisera des outils du domaine de l'apprentissage supervisé.

Nous avons ensuite étudié la répartition des différents paramètres en fonction des principales classes d'attaques.

Certains paramètres sont assez hétérogènes selon la classe étudiée. C'est le cas par exemple du service utilisé pour la connexion. On remarque que 98,11% des connexions utilisant le service HTTP sont des connexions normales. De plus on remarque également que 97,01% des connexions utilisant le service Private ont été classifiées comme attaque. Enfin, 100% des connexions utilisant le service echo sont des attaques et 99,10% de ces connexions sont des attaques de type Neptune (catégorie DOS).

Le flag permet lui aussi de discriminer certaines classes. En effet, on voit que 99,87% des connexions partant le flag S0 sont des attaques, et que 99,40% d'entre elles sont des attaques de type Neptune.

On remarque également l'importance du nombre d'octet envoyé depuis la destination jusqu'à la source de la connexion. En effet, on remarque que les connexions ayant un grand nombre d'octet envoyé de la destination vers la source sont plus souvent des requêtes classifiées comme normale. Par exemple, 96,51% des requêtes avec un échange de plus de 1000 octets entre la destination et la source sont des connexions normales.

Le nombre de fragment incorrect est également un bon indicateur, 100% des requêtes contenant au moins un fragment incorrect sont classifiées comme des attaques.

3 Modèles d'apprentissage supervisé

Afin de mettre en place notre apprentissage supervisé, il nous faut un jeu de données d'entraînement ainsi qu'un jeu de test. On remarque, en regardant le jeu de test fournit que celui-ci contient des types d'attaques introuvables dans le jeu d'entraînement. En effet il y a 23 types d'attaques dans le jeu d'entraînement contre 38 dans celui de test. Nous découperons le jeu d'entraînement fournit en 2 jeux distincts pour évaluer notre modèle et nous l'appliquerons ensuite au jeu à 38 types. Nous commencerons par le modèle simple des K plus proches voisins (PPV).

3.1 K plus proches voisins

Le principe de l'algorithme des K plus proches voisins est de discriminer un individu à partir des classes des K individus les plus proches au sens de la distance euclidienne. On applique ensuite un vote majoritaire

pour choisir la classe de l'individu. Nous appliquerons ce classificateur à plusieurs problèmes sur ce jeu de données. Dans un premier temps afin de distinguer les connexions dites "normales" des attaques, ensuite entre les classes d'attaques et enfin les types. Nous utiliserons la fonction intégré `knn()`.

Quelles features choisissons nous ? L'algorithme KNN utilise la notion de distance. On ne peut donc pas utiliser les features qui sont qualitatives qui de plus sont sans ordre. On mettra toutes ses features à zéro ce qui correspond à ne pas les considérer.

Notons une dimension importante pour quantifier la qualité des modèles est le nombre d'attaques classées comme connexions normales. Le but d'un tel data set est d'empêcher à l'avenir les connexions considérées comme attaques. Il est beaucoup plus problématique de laissé passer des attaques (on parlera d'attaques oubliées) que de bloquer des connexions normales (on parlera de connexions normales bloquées)

Sur le data set contenant 10% du jeu de donnée (data10%) avec 70% pour l'entraînement et 30% pour le test. On obtient les résultats suivants :

TABLE 3 – Resultats de KNN sur data10% avec 70% : entraînement, 30% : test

Nb Voisins	1	2	3	4	5
% Efficacité Globale	99.83	99.81	99.80	99.78	99.78
% Normales bloquées	0.15	0.19	0.18	0.20	0.19
% Attaques oubliées	0.18	0.19	0.23	0.25	0.26

Tous nos indicateurs de performance semble baisser avec l'augmentation du nombre de voisins considéré. L'indicateur d'attaques oubliées est aussi extrêmement important. On veut une bonne réussite générale avec le plus faible taux d'attaques oubliées. Attention a ne pas rejeter toutes connexions dans ce domaine.

Avec de tels résultats, on peut suspecter un surapprentissage de certaines classe. L'application du test sur un jeu de donnée avec la même distribution et avec des classes autant déséquilibré risque de poser problème

lors de l'application réelle.

Testons maintenant en réalisant l'apprentissage sur la totalité du jeu de donnée (data 10%) et en utilisant le jeu de test fournit lors de la compétition (corrected). Ce dernier jeu contient des types attaques non présents dans le jeu d'entraînement. On pourra ainsi voir si notre modèle généralise suffisamment.

TABLE 4 – Resultats de KNN avec data10% : entraînement et corrected : test

Nb Voisins	1	2	3	4	5
% Efficacité Globale	93.45	93.49	93.51	93.62	93.67
% Normales bloquées	0.48	0.49	0.48	0.48	0.47
% Attaques oubliées	16.46	16.323	16.29	16.00	15.88

Les résultats confirment nos suspicions. Beaucoup d'attaques sont oubliés contrairement aux connexions normales tout en gardant une efficacité décente. Il faut absolument améliorer cet indicateur. Le modèle n'est pas assez général pour fonctionner très efficacement sur des attaques non présents dans le jeu d'entraînement.

On peut essayer pour cela de rééquilibrer le jeu de données afin d'avoir d'avantage d'attaque (actuellement avec un rapport 2 pour 1 pour les connexions normales). Le modèle n'ayant pas de mal à détecter les connexions normales, il pourrait être intéressant de déséquilibrer le jeu de donnée dans l'autre sens).

Dans un premier temps, nous avons testé l'utilisation de **poids** sur l'algorithme des plus proches voisins avec la fonction `kknn` inclus dans le package du même nom. Nous avons utilisé des poids inversement proportionnels aux distances. Les résultats, pour tout k (nombre de voisins considérés) étaient extrêmement médiocres et ne dépassait pas une efficacité de 60%. C'est pourquoi nous décidons de nous concentrer sur les techniques d'oversampling et d'undersampling.

Commençons par l'undersampling : Notre but est de baisser la quantité de connexions dites normales car celles-ci sont trop nombreuses comparées aux attaques. Cela permettra au modèle de réduire le nombre d'at-

taques oubliées tout en restant efficace de façon générale.

Dans le tableau suivant nous étudierons les résultats pour différents ratio du nombre de connexions normales par le nombre d'attaque. Plus le ratio est petit , moins nous considérons de connexions normales, qui est, de base, la classe sur-représentée.

TABLE 5 – Resultats de KNN par ratio du jeu de données avec data10% : entraînement et corrected : test

Rapport Entraînement Normale/Attaque	1	0.5	0.2	0.05	0.01
% Efficacité					
Globale	93.42	94.11	94.19	94.36	92.82
% Attaques oubliées	15.25	14.37	13.64	11.36	9.86

Réduire le nombre de connexions normales a réellement un effet positif. Avec peu de différences du point de vu de l'efficacité globale, le pourcentage d'attaques oubliées diminue de façon intéressante. A partir du ratio où les attaques sont 20 fois plus nombreuses, le nombre de connexions normales devient trop faible et l'efficacité globale commence à diminuer un peu trop.

Cet under-sampling semble être une méthode efficace, mais comment améliorer encore davantage ce modèle ? Comment diminuer le pourcentage d'attaques oubliées ? Il peut être intéressant de regarder la classe des attaques qui peinent à être évaluées comme telle.

TABLE 6 – Nombre d'attaques étiquetées comme connexion normale

Classe d'attaque	dos	u2r	r21	probe	inconnues
Nombre d'erreurs	12	204	245	1383	1538

On remarque qu'il n'y a pas de problèmes à catégoriser les attaques dos comme des attaques. Les autres classes souffrent d'un trop petit nombre de données. C'est pourquoi il va falloir réaliser un over-sampling de ces classes. Quant aux attaques non présentes dans le jeu d'apprentissage, on ne peut pas faire grand chose

pour l'instant. On peut espérer que ces attaques soit proches des classes que l'on va sur-échantillonner afin que cela leurs soit aussi bénéfique.

Pour notre over-sampling, il existe de nombreuses techniques : le sur-échantillonage aléatoire où l'on duplique des individus du jeu de données ou des techniques plus complexes comme SMOTE ou ADASYN. Nous voulons rester simple mais nous pouvons faire cela de façon plus maligne. Nous avons accès au jeu de données complet que l'on utilise pas pour des raisons de temps de processing et de mémoire. On peut tirer des attaques de ce jeu de données. Malheureusement le jeu 10% inclut déjà toutes les attaques. Il est donc inutile.

On va dupliquer de façon aléatoire les individus. Cette méthode ne nous donne pas des résultats satisfaisants. En effet le nombre de attaques oubliées stagne. On n'arrive pas, pour tout k, à passer en dessous de 9% d'attaques oubliées sans que l'efficacité globale s'écrase. Ensuite nous avons tester la méthode SMOTE() d'échantillonnage. Même constat, les résultats ne sont pas grandement améliorés, tournant autour de 93% d'efficacité générale et 10% d'attaques oublié.

L'échantillonnage nous a donc permis de ne pas perdre trop en efficacité et de passer de 16% à 9.7% d'attaques oubliées. On peut expliquer ce plafond de performance par l'introduction d'attaques jamais observées dans le jeu de test. L'autre classe d'attaque posant des problèmes est la classe "probe" qui compte pour près de la plupart des erreurs pour les classes connues. Cela est de bonne augure. Cela signifie que si l'on trouve un moyen de différencier les attaques "probe" des connexions normales (à l'aide d'un autre modèle ou étude), on pourrait avoir , en ajoutant un Knn, un modèle extrêmement fonctionnel, peu d'erreur et peu d'attaque oubliées.

Grâce au travail réalisé préalablement, on peut lancer les mêmes algorithmes pour la classification des classes. Les résultats sont correctes avec les meilleurs résultats pour k= 1 et 89% d'efficacité bien que certaines classes comme probe et u2r ont un très faible

pourcentage de réussite comme on l'imaginait par notre travail préalable. Cela n'étant pas le but principal, on cherchera d'abord à améliorer la classification attaque / normal.

3.2 Naive Bayes et ADL

On s'intéresse maintenant à la classification naïve bayésienne. Commençons, comme d'habitude, par regarder le comportement en divisant le dataset 10% en 70/30.

Les résultats sont les suivants : **Efficacité Globale** 96.61% et **Attaques oubliées** 6.08%. En comparaison à l'algorithme KNN pour les mêmes jeux de données, Naive Bayes donne des résultats très moyen (99% et 0.18% attaques oubliées).

Il est possible que cette algorithme réagisse mieux au jeux de test corrected. Les résultats sont les suivants : **Efficacité Globale** 92.53% et **Attaques oubliées** 15.53%. Ce n'est pas exceptionnel et ne justifie pas l'utilisation de cette algorithme par rapport à l'utilisation de KNN. Les résultats ne s'améliorent pas avec l'échantillonnage.

Comment expliquer de tels résultats et augmenter l'efficacité ? Regardons les hypothèses d'un tel algorithme. Les hypothèses sont des hypothèses d'indépendances entre les features. Regardons la corrélation entre les features.

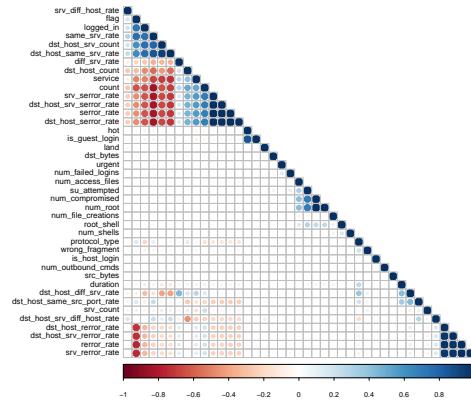


FIGURE 2 – Visualisation de la matrice de corrélation des features, générée avec R.

On remarque qu'il existe de forte corrélation entre de nombreuses features grâce à cette pyramide créée à l'aide de la fonction `corrplot` et `cor`. Les hypothèses ne sont pas respectées , d'où les résultat médiocres. En enlevant les features corrélées, on ne s'attend pas à pouvoir faire baisser le pourcentage d'attaques oubliées avec cette perte d'information.

Pouvons nous appliquer une Analyse Discriminante Linéaire par exemple ? Une ADL à cette même hypothèse, même si celle ci fonctionne souvent sans. L'hypothèse d'homoscélasticité est elle aussi largement violée, puisque les matrices de corrélations sont différentes en fonctions des classes.

EN appliquant quand même une ADL avec la fonction `lda()` (nous avons enlevé les features trop corrélé car la fonction nous rentrait des erreurs nous invitant à corriger cela) et le jeu de données préalablement traité, on remarque que bien que l'efficacité globale est élevé (90%), le nombre d'attaques non étiquetées comme tel explose, par exemple avec plus de 22% d'erreur pour les samplings utilisés pour knn. C'est pourquoi nous passons à une réflexion sur les arbres de décisions.

3.3 Arbre de décision

Les arbres binaires constituent une méthode populaire d'apprentissage supervisé grâce à sa qualité d'interprétation et le coût calculatoire limité des procédures d'apprentissage et de classement. Mais en effet, l'optimisation du critère d'impureté de cette procédure n'est faite que localement, donc une stratégie globale qui consisterait à optimiser l'ensemble de l'arbre est délicate à mettre en oeuvre pour des raisons combinatoires : on applique ici la procédure d'élagage et les forêts aléatoires.

Premièrement, nous utiliserons la fonction intégrée `tree()` pour l'arbre binaire, puis `prune.misclass()` pour élagager, enfin une `randomForest()` pour les forêts aléatoires.

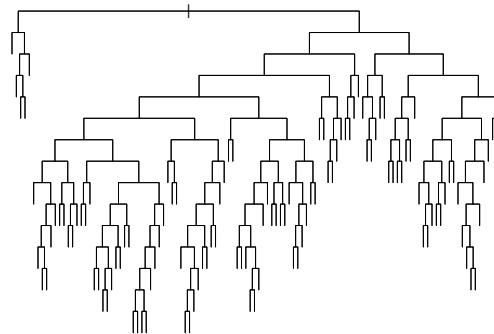
Quelles features choisissons-nous ? Dans l'algorithme des arbre de décision, pour éviter le **Overfitting**, il faudra qu'on fait une **Sélection de Features** au lieu de garder toutes les variables. A l'aide de l'algorithme de **Feature Selection** dans le bibliothèque de machine learning **Scikit** de Python, on s'obtient les 10 features qui sont "principales" : `_count ;srv_serror_rate ;srv_count ;same_srv_rate ;dst_host_same_src_port_rate ;dst_host_srv_error_rate ;dst_host_srv_count ;dst_host_count ;logged_in ;protocol_type` [4]

On prend alors des données mentionnées pour la suite.

3.3.1 Classification par connexions normales ou attaques

Dans un premier temps, sans traitement préalable, la fonction `tree()` nous retourne un arbre binaire de 125 noeuds terminaux. Graphiquement, un tel arbre est illisible, encore plus avec les textes de décisions. Chaque noeud correspond à une décision sur les features, par exemple pour le premier noeud : `same_srv_rate >= 0.495`. Vous trouverez ci dessous une représentation de cet arbre. qui nous semble overfitting.

FIGURE 3 –
Visuel de l'arbre de décision de l'analyse des attaques
avec 125 nodes terminales, générée avec R.



C'est un arbre complet, peu être même trop complet. On soupçonne un overfitting. Mais quelles sont les résultats ? Commençons comme on le fait toujours par séparer notre jeu data10% en 70 /30 pour l'entraînement. On obtient **Efficacité Globale** 99.72% et **Atttaques oubliées** 0.34%. Ce sont des résultats du même ordre que KNN. Mais le plus important reste le comportement sur le jeu de données corrected.

Les résultats sont réellement similaires aux autres modèles. Mais c'est encore très médiocre dans l'adaptation au jeu de test. Les attaques non présentes dans le jeu d'entraînement sont responsables pour plus de 60 % des erreurs. Ensuite on retrouve les attaques "probe".

TABLE 7 – Résultats de arbre de décision Normale / Attaque avec data10% : entraînement et corrected : test

KPI	Réultats en %
Efficacité Globale	92.86
Attaques Oubliées	17.28
Connexions Normales Bloquées	0.93

Cet arbre n'est pas efficace car trop proche des données. Il est clairement overfit. Comment résoudre cela ?

La première stratégie utilisable pour éviter un overfitting des arbres de décision consiste à proposer des critères d'arrêt lors de la phase d'expansion. C'est le principe du **pré-élagage**. Lorsque le groupe est d'effectif trop faible, ou lorsque l'homogénéité d'un sous-ensemble a atteint un niveau suffisant, on considère qu'il n'est plus nécessaire de séparer l'échantillon.

En utilisant `cv.tree()`, qui associé à la fonction `prune.tree()`, réalise une validation croisé des performances l'arbre en fonction du nombre de noeuds coupés.

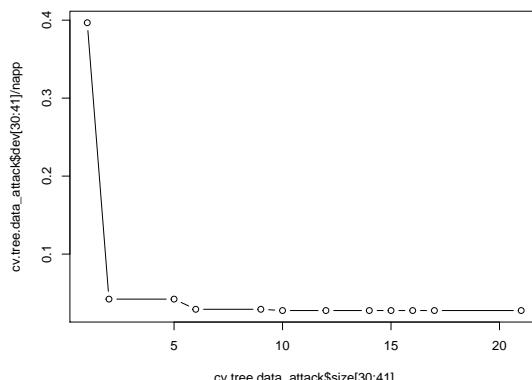


FIGURE 4 – La tendance de déviation du data selon le nombre de noeuds, générée avec R.

D'après ce graphique, à partir de 6 noeuds, la deviance (goodness of fit - qualité du modèle) atteint son minimum et reste constant jusqu'à nos 120 noeuds maximum. Ainsi puisque sa deviance pour l'arbre avec élagage à 6 noeuds est optimal et la même que pour

125 noeuds, on continuera notre étude avec un élagage à 6 noeuds. Pour des facilités de représentations, nous n'afficherons pas les arbres de façon proportionnel mais de façon uniforme.

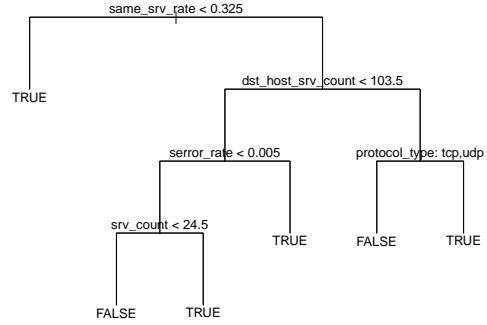


FIGURE 5 – Arbre de décision élagagé de l'analyse des attaques avec 6 noeuds terminales, générée avec R.

TABLE 8 – Résultats de arbre de décision 6 noeuds Normale / Attaque avec data10% : entraînement et corrected : test

KPI	Réultats en %
Efficacité Globale	91.34
Attaques Oubliées	21.63
Connexions Normales Bloquées	0.69

L'erreur de l'arbre élagagé est plus grande que celle de non élagagé. Ce n'est toujours pas acceptable et pas du tout suffisant.

3.3.2 Classification par classe d'attaques

Dans un premier temps, on est encore en arbre de décision pour la classification. On réitère la démarche réalisé préalablement ? On se retrouve avec 136 noeuds, un nombre bien trop grand pour ne pas faire de l'overfitting. Regardons quand même les résultats

TABLE 9 – Résultats des arbre de décision avec 136 noeuds pour analyse de classes avec data10% : entraînement et corrected : test

Réel Prédiction	normal	dos	u2r	r21	probe	Efficacité test	R P	normal	dos	u2r	r21	probe	Efficacité
normal	47433	255	0	29	196	99.00 %		47671	237	0	0	5	99.49%
dos	299	21365	0	0	56	98.37 %	normal	510	21210	0	0	0	80.61%
u2r	34	2	2	1	0	5.13 %	dos	39	0	0	0	0	0 %
r21	1824	280	0	154	80	6.59 %	u2r	2322	6	0	0	80	0 %
probe	22	37	0	66	1144	5.20 %	r21	378	381	0	0	510	40.19%
autre	3008	543	0	28	443	0 %	probe	3606	217	0	0	199	0 %
Total						90.7 %	autre						89.8 %
							Total						

Ce sont de très mauvais résultats par classe. On essaie alors de réduire le nombre de noeuds. D'après l'analyse d'élagage, on obtient que le meilleur nombre de noeuds est 8, mais quand on réalise un arbre de 8 noeuds.

nombre d'individus de ces classes.

Réel Prédiction	normal	dos	u2r	r21	probe	Efficacité test	R P	normal	dos	u2r	r21	probe	Efficacité
normal	47671	237	0	0	5	99.49%		47671	237	0	0	5	99.49%
dos	510	21210	0	0	0	80.61%	normal	510	21210	0	0	0	80.61%
u2r	39	0	0	0	0	0 %	dos	39	0	0	0	0	0 %
r21	2322	6	0	0	80	0 %	u2r	2322	6	0	0	80	0 %
probe	378	381	0	0	510	40.19%	r21	378	381	0	0	510	40.19%
autre	3606	217	0	0	199	0 %	probe	3606	217	0	0	199	0 %
Total						89.8 %	autre						89.8 %
							Total						

Les colonnes "u2r" et "r21" sont tous nulles à cause du manque d'individus pour l'entraînement de l'arbre. Cela montre que c'est un très mauvais modèle qui ne peut pas être exploiter comme tel. Voyons maintenant un autre algorithme de classification en arbres.

3.3.3 Random Forest

Random forest est une méthode basé sur les arbres de décision. Le principe est d'apprendre un grand nombre d'arbre de décision aléatoirement et d'effectuer ensuite un vote majoritaire pour choisir la classe. Cette méthode est assez complexe en terme de nombre de paramètres du modèle et de temps d'apprentissage. Cependant bénéficiant d'une grande quantité de données on peu tout à fait apprendre un modèle avec la méthode Random Forest sans craindre de sur-apprentissage.

On analyse ensuite en utilisant les forêts aléatoires.

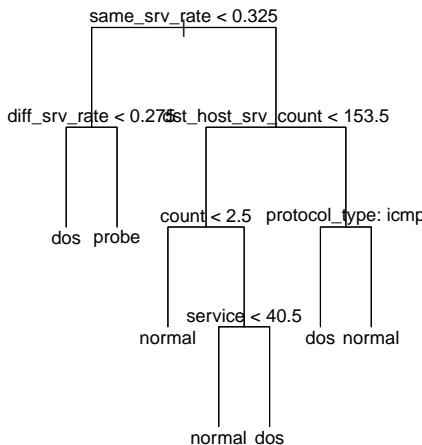


FIGURE 6 – Arbre de décision avec élagage de l'analyse des classifications des attaques avec 8 noeuds terminaux, générée avec R.

On peut remarquer que dans l'arbre, toutes les classes ne sont pas représentées. On ne retrouve que les classes dos / probe et normal. Cela est du au faible

TABLE 11 – Résultats des forêts aléatoires pour analyse des attaques avec data10% : entraînement et corrected : test

KPI	Résultat en %
Efficacité Globale	92.50
Attaques Oubliées	18.48
Connexions Normales Bloquées	0.85

Ce sont encore une fois des résultats assez mauvais

et proche de tout ce que l'on a vu précédemment. Ces résultats n'évoluent pas énormément en changeant le nombre d'arbres de la forêt.

TABLE 12 – Résultats des forêts aléatoires pour analyse des classes avec data10% : entraînement et corrected :

test

R P	normal	dos	u2r	r21	probe	Efficacité
normal	47526	152	2	48	185	99.19%
dos	337	21300	0	0	83	98.07%
u2r	36	0	0	3	0	0 %
r21	2006	5	5	299	13	12.84%
probe	14	46	0	3	1206	99.19%
autre	2856	610	2	25	529	0%
Total						90.01%

On retrouve ce même problème avec la classe u2r. Comparons rapidement les méthodes à bases d'arbre avant de rechercher la meilleure stratégie pour réaliser notre **pare-feu** anti-attaque.

TABLE 13 – Comparaison des erreurs avec data10% : entraînement et corrected : test

Models	Decision Tree	Prune Tree	Random Forest
Binaire Attaque %	91.34	92.86	92.50
Attaques oubliées %	21.63	17.28	18.48
Par Classe %	93.00	89.8	90.01

Les résultats sont assez similaires entre les modèles bien que celui utilisant des arbres avec élagage (Prune tree) soit meilleur à tout les aspects. Ils sont bien moins efficace que KNN mais bien meilleur que Naive Bayes. Ces algorithmes ont beaucoup de mal à classifier les attaques u2e et r21 en tant qu'attaques alors que Knn avait des problèmes avec la classe probe que les arbres de décisions classifient correctement.

3.3.4 Un modèle composé ?

Rappelons le but d'une telle étude. Le but est de créer un modèle pouvant autoriser les connexions normales et rejeter les attaques. Ensuite nous savons qu'il existe des attaques que notre modèle n'aura jamais ren-

contrées, comme c'est le cas avec le jeu de test corrigé. Les nouvelles attaques doivent servir à réentraîner le modèle afin que cela ne se reproduise plus. Voici un schéma représentant cela.

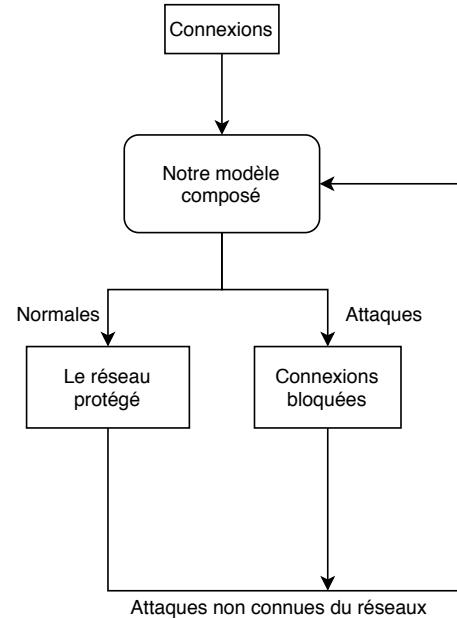


FIGURE 7 – Utilisation de notre modèle, créé avec draw.io.

Nous savons que le modèle KNN fonctionne plutôt bien mais à quelques problèmes à catégoriser les attaques probe comme des attaques. Bien qu'ils fonctionnent moins bien, nos arbres de décisions, en particulier avec randomForest, n'ont pas de problèmes avec ce type d'attaque. Peut-il être intéressant de combiner ces 2 modèles. Ils seraient combinés de la façon suivante.

Dans un premier temps, on entraîne le KNN optimal trouvé dans la partie dédiée. C'est à dire avec $k = 1$ et un sous-échantillonnage avec un rapport de 0.01 pour le nombre de connexions normales / nombre d'attaques. Une fois celui-ci entraîné, on fait de même avec le randomForest de la partie précédente. Celui-ci n'est pas entraîné exactement sur les mêmes données (features sélections et pas d'échantillonnage).

Cela étant fait, on teste le modèle comme sur la figure ci-dessous. Pour cela à partir du jeu de test et du

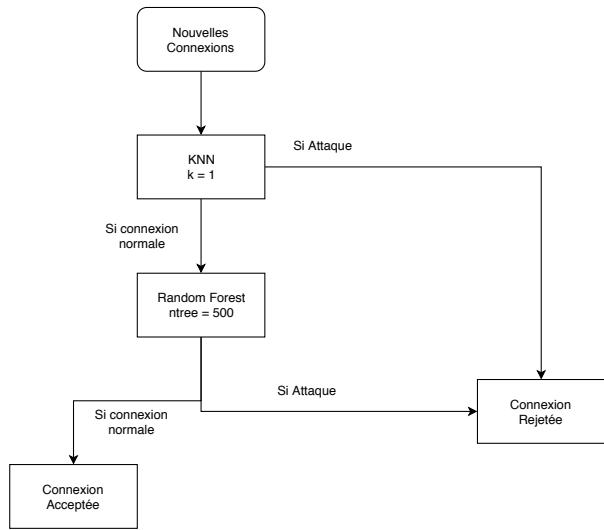


FIGURE 8 – Combinaison des modèles, crée avec draw.io.

modèle knn entraîné, on prédit si les connexions du jeu de test sont des attaques ou non. On considère alors que si la connexion est considérée comme une attaque, alors le modèle KNN a raison et c'est une attaque. Sinon on obtient la probabilité que cela soit une attaque grâce au modèle RandomForest. Si la probabilité est inférieur à 0.5, cela signifie que le modèle RandomForest considère que c'est une connexion normale comme KNN. Sinon le RandomForest est en désaccord avec KNN. Comment les départager ? Puisque l'on a les probabilités pour le RandomForest, on peut jouer sur celles ci et voir à partir de quelle probabilité entre 0.5 et 1 nous considérons le RandomForest au détriment du KNN.

On remarque rapidement que l'efficacité globale ne change pas beaucoup. Testons donc l'influence des différentes probabilités sur la quantité d'attaques oubliées.

On remarque que, d'après le graphique, la meilleur probabilité frontière est de $P = 0.6$. Pour récapituler cela signifie que : Lorsqu'une connexion entre, si elle est considérée comme une attaque par KNN celle ci est bloquée, sinon on regarde la probabilité que ce soit une attaque d'après le modèle RandomForest. Si cette probabilité est supérieur à $P = 0.6$, alors c'est une attaque et elle est bloquée. Sinon elle est acceptée.

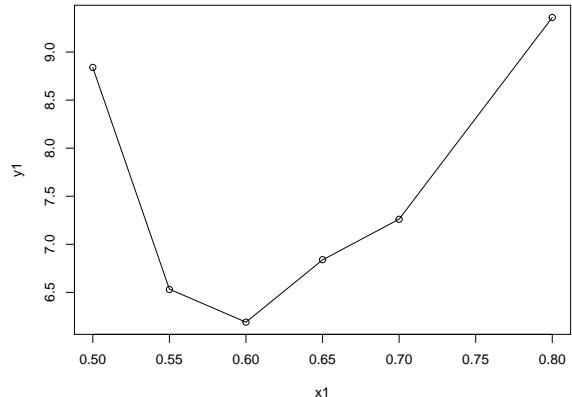


FIGURE 9 – Pourcentage d'attaques oubliées en fonction de la probabilité frontière.

Maintenant voyons voir les résultats de ce modèle combiné.

Efficacité globale : 94.27 %

Pourcentage d'attaques oubliées : 6.19 %

Pour rappel, en comparaison, la plupart des modèles ne passaient pas sous la barre des 16% d'attaques oubliées et KNN, après énormément d'échantillonnage atteignait tout juste 9% d'attaques oubliées pour une moins bonne efficacité globale.

Cette combinaison de modèle semble fonctionner à merveille. On arrive à garder une super efficacité globale tout en réduisant fortement le nombre d'attaques oubliées. On est ensuite limité par les attaques inconnues du jeu d'entraînement. Avec d'avantages de données d'attaques, les modèles n'en seront que meilleur. Cette combinaison KNN / Random Forest est le meilleur modèle que l'on est eu jusqu'à là, et de loin.

3.4 Conclusion

En conclusion, on peut dire que ce data set est extrêmement intéressant. De part son intérêt pratique, on a réellement l'impression de chercher un type de mo-

dèle particulier et on ne fait pas juste un lancement de différents modèles en comparant les erreurs. De plus, le fait que le jeu de données ne comporte que très peu d'attaque (hors dos) et que le jeu de test contiennent des attaques inconnus rend les choses beaucoup plus intéressantes. C'est en étudiant les modèles un à un que l'on s'est rendu compte de la faiblesse de certains sur la prédiction de certaines classes et c'est comme cela que l'on en arrive à combiner les modèles pour ne garder que le meilleur de chacun.

Il existe de nombreuses façons de poursuivre cette étude. Dans un premier temps, il serait possible de pousser un peu plus loin la réflexion de combinaison de modèle. En effet c'est en regardant les erreurs par classes que l'on a compris que KNN et RandomForest pouvaient être associé pour de meilleurs résultats. On peut donc faire la même chose avec d'autres modèles mais surtout en regardant les erreurs sur les types d'attaques, que nous n'avons pas exploités. Ensuite nous pouvons placer des filtres à notre modèle. D'après les études préliminaires sur les features, certaines features impliquent des classes d'attaques. C'est un domaine qu'il pourrait être bon d'explorer. Enfin, avec l'avancée des réseaux de neurones il serait dommage de passer à coté de ces modèles, souvent ultra-efficaces.

Pour les personnes souhaitant s'inspirer de notre travail ou récupérer le code ou les données pré-traités : [Drive](#).

Merci de votre lecture

Références

- [1] University of California, Irvine . Le DataSet sur le site de la compétition et de l'université associé. *Lien : [DataSet](#)*.
- [2] DataSecurityBreach. 700 millions d'attaques informatiques enregistrées en 2017. *Lien : [DataSecurityBreach](#)*.
- [3] Microsoft. Cybersécurité : 5 chiffres à connaître. *Lien : [Expetiencies.Microsoft](#)*.
- [4] 机器学习实战 - Scikit 决策树分类算法 *Lien : [机器学习实战 - 决策树分类算法](#)*.
- [5] Documentation R - Package tree *Lien : [Lien](#)*.