



MISC

LE MAGAZINE DE LA SÉCURITÉ INFORMATIQUE MULTIPLATEFORME !

NOV.
DÉC.
2018

FRANCE
MÉTRO : 12,90 €
DOM/TOM : 13,90 €
BEL/LUX/PORT.
CONT. : 13,90 €
CH : 18 CHF
CAN : 18 \$CAD

NOUVELLE FORMULE !

INTERVIEW

Entretien de Julien Cornebise,
expert en Machine Learning

HORS-SÉRIE N°18

MACHINE LEARNING & SÉCURITÉ



OÙ EN SOMMES-NOUS ?

► APPRENTISSAGE MACHINE : LES BASES

Faites vos premiers pas et vos premiers codes Python en Machine Learning

► DÉTECTION D'INTRUSION

Découvrez comment Keras et TensorFlow peuvent vous y aider

► DEEP LEARNING

Classification de malwares : utilisez l'IA pour l'analyse de sentiments dans des textes

► CLUSTERISATION DE MALWARES

Techniques de classification automatique de malwares pour générer vos propres règles Yara

L 16844 - 18 H - F : 12,90 € - RD



COMPRENDRE

Découvrez la prise de décision grâce à l'apprentissage par renforcement

EXPÉRIMENTER

Théorie des graphes : peut-on définir et trouver le centre du Web ?

CSAW'18

EUROPE
GRENOBLE INP - ESISAR
8-10 NOVEMBER 2018



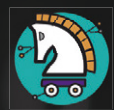
EUROPEAN FINALS

Meet the future of Cybersecurity!

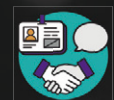
CSAW is comprised of hardware and software Cybersecurity competitions, keynotes, as well as a career fair



Capture The Flag: 36 hour long CTF competition



Embedded Security Challenge: IoT smart light bulb hacking so as to leak secret information by means of Side-Channel attacks



Applied Research: Cybersecurity Best Paper Award



RED: High school Cybersecurity competition

Take the opportunity to participate and meet with Cybersecurity experts from industry and academia.

Participants will be able to compete during a special Cybersecurity competition. Participation is free but registration is required at:

www.csaw.io/europe



Retrouvez toutes nos publications



MISC Hors-Série est édité par Les Éditions Diamond

10, Place de la Cathédrale – 68000 Colmar – France

Tél. : 03 67 10 00 20 / Fax : 03 67 10 00 21

E-mail : cial@ed-diamond.com

Service commercial : abo@ed-diamond.com

Sites : <https://www.miscmag.com> – <https://www.ed-diamond.com>

Directeur de publication : Arnaud Metzler

Chef des rédactions : Denis Bodor

Rédacteur en chef : Cédric Foll

Rédacteur en chef adjoint : Émilien Gaspar (gapz)

Secrétaire de rédaction : Aline Hof

Responsable service PAO : Kathrin Scali

Réalisation : Kathrin Scali et Thomas Pichon

Responsable publicité : Tél. : 03 67 10 00 27

Service abonnement : Tél. : 03 67 10 00 20

Impression : Loire Offset Titoulet, Saint-Etienne

Distribution France : (uniquement pour les dépositaires de presse)

MLP Réassort :

Plate-forme de Saint-Barthélemy-d'Anjou. Tél. : 02 41 27 53 12

Plate-forme de Saint-Quentin-Fallavier. Tél. : 04 74 82 63 04

Service des ventes :

Abomarque : 09 53 15 21 77

IMPRIMÉ en France - PRINTED in France

Dépôt légal : A parution

N° ISSN : 1631-9036

Commission Paritaire : K 81190

Périodicité : Bimestrielle

Prix de vente : 12,90 Euros

Première parution : Novembre 2018

La rédaction n'est pas responsable des textes, illustrations et photos qui lui sont communiqués par leurs auteurs. La reproduction totale ou partielle des articles publiés dans MISC est interdite sans accord écrit de la société Les Éditions Diamond. Sauf accord particulier, les manuscrits, photos et dessins adressés à MISC, publiés ou non, ne sont ni rendus, ni renvoyés. Les indications de prix et d'adresses figurant dans les pages rédactionnelles sont données à titre d'information, sans aucun but publicitaire.

Les articles non signés contenus dans ce numéro ont été rédigés par les membres de l'équipe rédactionnelle des Éditions Diamond.

CHARTRE DE MISC :

MISC est un magazine consacré à la sécurité informatique sous tous ses aspects (comme le système, le réseau ou encore la programmation) et où les perspectives techniques et scientifiques occupent une place prépondérante. Toutefois, les questions connexes (modalités juridiques, menaces informationnelles) sont également considérées, ce qui fait de MISC une revue capable d'appréhender la complexité croissante des systèmes d'information, et les problèmes de sécurité qui l'accompagnent. MISC vise un large public de personnes souhaitant élargir ses connaissances en se tenant informées des dernières techniques et des outils utilisés afin de mettre en place une défense adéquate. MISC propose des articles complets et pédagogiques afin d'anticiper au mieux les risques liés au piratage et les solutions pour y remédier, présentant pour cela des techniques offensives autant que défensives, leurs avantages et leurs limites, des facettes indissociables pour considérer tous les enjeux de la sécurité informatique.

Retrouvez-nous sur :



@MISCmag
@editionsdiamond



@MISCleMag



<https://connect.ed-diamond.com>



<https://www.miscmag.com>
<https://www.editions-diamond.fr>

<https://www.ed-diamond.com>

www.miscmag.com

ÉDITO

De l'hermétisme au marketing

Qu'il s'agisse de lessive lavant plus blanc que blanc ou d'une énième vulnérabilité cataclysmique, la résultante invariable du marketing est de produire, à un moment, du faux. Prenons le quotidien de qui-conque ferait un minimum de veille du côté des vulnérabilités : dans les méthodes à l'ancienne, suivre les listes de diffusion comme full-disclosure, bugtraq ou oss-sec par exemple n'est pas chose aisée tant la pluie de CVE est quasi-permanente et l'information toujours partielle (même si quelques advisory et discussions de qualité sont là pour rattraper l'ensemble). Du côté de Twitter, si l'on arrive à mettre de côté les petites querelles du moment, on y trouve une quantité intéressante d'informations, bon nombre d'experts en sécurité francophones et anglophones l'utilisant au quotidien. Jusque-là, faire la part des choses, distinguer ce qui est pertinent ou non, se faisait dans un flux permanent avec ces hauts (POC sur full-disclosure) et ces bas (hoax). Puis est arrivé le marketing : nom, site, logo, script/outil, vidéos, raz de marée sur Twitter, papiers dans les journaux grand public, annonce d'une demi-fin du monde, et cela pour **une** vulnérabilité (dans certains cas même quasi inexploitable). Et voilà que l'on a basculé dans le rythme infernal de la course à la célébrité pour les vulnérabilités. Certes, pour l'initié (toi, lecteur de MISC), tout cela s'apparente essentiellement à une dégradation du rapport signal/bruit dans la veille quotidienne.

Mais où est le faux ? D'une part, il n'y a pas de corrélation directe entre l'impact de la faille et la célébrité qui lui a été associée. Il peut s'agir parfois d'un travail élégant ou étonnant, ou qui touche simplement un élément important. Il est très difficile de quantifier le nombre de vulnérabilités critiques qui sont passées sous silence, tout comme la course aux vues/retweets/etc. pousse à exagérer l'impact réel. D'autre part, quand l'information arrive dans un média grand public, qu'elle soit traitée correctement ou non, il est souvent fait complètement abstraction des enjeux réels de la sécurité, qui sont souvent bien loin des failles à la mode. Ce ne sont que des exemples très simples de ce que peut produire le marketing, on peut facilement imaginer la confusion supplémentaire que cela rajoute d'une manière générale. Comme le disait si bien Nietzsche : « *le demi-savoir triomphe plus facilement que le savoir complet : il conçoit les choses plus simples qu'elles ne sont, et en forme par suite une idée plus saisissable et plus convaincante* ».

Alors certes, il y a des failles qui ont connu un grand succès pour de bonnes raisons, mais est-ce bien raisonnable de se prendre au jeu de la communication ? Si le travail produit est intéressant, la qualité ne se suffit-elle pas à elle-même ?

Émilien Gaspar / gapz / eg@miscmag.com

MISC HORS-SÉRIE N°18

3

SOMMAIRE 18

MISC HORS-SÉRIE N°



08

ACTUS

- 08 Revue de livres...
- 10 Entretien avec Julien Cornebise, expert en machine learning



16

MALWARE

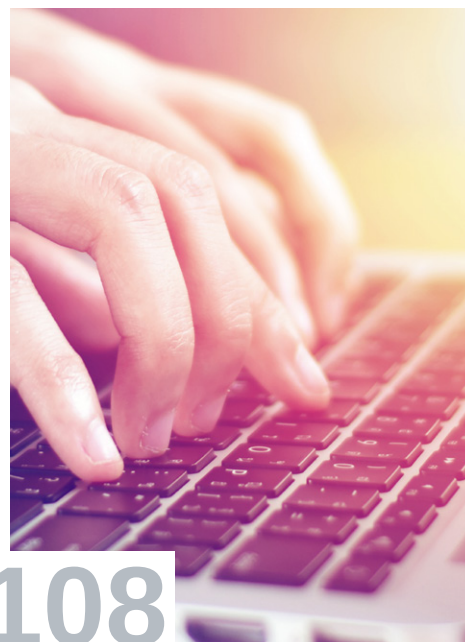
- 16 Aux voleurs

26

DOSSIER : MACHINE LEARNING & SÉCURITÉ



- 28** Machine Learning : un (rapide) tour d'horizon
- 42** ML & sécurité : les opportunités
- 51** Vers une détection d'intrusion dynamique et continue en utilisant les réseaux de neurones
- 62** Machine Learning en Python : clusterisation à la rescousse des hunters (de malware)
- 74** Automeans, ou comment éviter le « k par k » avec K-means
- 84** Analyse des sentiments avec le Deep Learning
- 97** YaDiff : de l'Intelligence Artificielle pour comparer les codes binaires



108

CODE

- 108** Voyage en C++ie : un diamant d'innocence



114

SCIENCES

- 114** Introduction à l'apprentissage par renforcement, sans code et presque sans maths
- 122** Graphes géants creux : comment définir le centre du Web

FACILITEZ-VOUS LA VEILLE TECHNO

Un accès à
+ de 7000 articles

Disponibles pour :



Consultez les
numéros d'hier et
ceux d'aujourd'hui

Un outil de
recherche

Un affichage par
numéro paru

Les magazines
standards et leurs
hors-séries

Des articles
triés par
domaines

connect.ed-diamond.com

LA PLATEFORME DE DOCUMENTATION NUMÉRIQUE DES ÉDITIONS DIAMOND

3104 articles dans GNU/Linux Magazine
253 articles dans Hackable
1872 articles dans Linux Pratique

1124 articles dans Linux Essentiel
1036 articles dans MISC
189 articles dans Open Siliconium
749 articles dans Unix Garden

Accueil » MISC

BIENVENUE SUR LA PLATEFORME DE DOCUMENTATION NUMÉRIQUE

DOSSIER : ENVIRONNEMENTS D'EXÉCUTION SÉCURISÉS : DE SGX À TRUSTZONE

MISC n° 099

VOUS AVEZ DIT ENVIRONNEMENTS D'EXÉCUTION SÉCURISÉS ?

Derrière ce titre se cache une actualité très simple : la montée en puissance des extensions de sécurité de processeurs pour la création d'environnements d'exécution sécurisés. Mais de quoi s'agit-il au juste ?

LES ARTICLES DE MISC N°99 - SEPTEMBRE/OCTOBRE 2018

Edito

MISC n° 099 | septembre 2018 | Cédric Foil

Sécurité

C'est la rentrée, le travail reprend avec force et vigueur !

> Lire l'extrait

Attaques sur la technologie TrustZone d'Intel SGX

MISC n° 099 | septembre 2018 | Geoffrey Guibon

Sécurité

Un aperçu de la technologie TrustZone a été donné dans l'article précédent expliquant les rôles de cette technologie. Ce deuxième...

> Lire l'extrait

Vous avez dit environnements d'exécution sécurisés ?

MISC n° 099 | septembre 2018 | gapz

Sécurité

Derrière ce titre se cache une actualité très simple : la montée en puissance des extensions de sécurité des processeurs pour la création...

> Lire l'extrait

Développer une application sécurisée avec Intel SGX

MISC n° 099 | septembre 2018 | Alexandre Adamski

Sécurité

Cet article propose une introduction à la technologie Intel SGX à travers l'exemple du développement d'une application de gestion de mots de passe.

> Lire l'extrait

Attaques par canaux auxiliaires : évaluations de sécurité à court et long terme

MISC n° 099 | septembre 2018 | François-Xavier Standaert

Sécurité

Connues depuis la fin des années 1990, les attaques par canaux auxiliaires sont devenues un élément important de la sécurité des systèmes.

> Lire l'extrait

EDR : Endpoint Detection & Response

MISC n° 099 | septembre 2018 | Laurent Oudot

Sécurité

La protection des points de terminaison (stations de travail, serveurs, mobiles) devient un axe majeur de la cybersécurité.

> Lire l'extrait

connect.ed-diamond.com



ET L'ACCÈS À LA DOCUMENTATION !

The screenshot shows the OpenSilicium website interface. At the top, there are links for 'Identifiez vous' and 'S'inscrire', and a search bar labeled 'Votre recherche'. Below the navigation bar, there's a section titled 'AMÉRIQUE DE MISC !' featuring an image of a smartphone with a chain around it. To the right of this image is the MISC logo and a 'Lire l'extrait' button. Below this, there's a 'RECHERCHER' section with a search bar and a list of articles. Further down, there's an 'ACCÈS PAR NUMÉRO' section with a grid of numbers (099 to 076) and a 'NUMÉROS HORS SÉRIE' section with a grid of numbers (017 to 002). At the bottom, there's an 'ACCÈS PAR DOMAINE' section with a grid of topics (Audio/Vidéo, Brèves, Bureautique, Code, Domotique, Droit, Électronique, Embarqué, Graphisme, Hacks, Humour et Critique, IA, Jeux, Mobilité, Radio et wireless, Réseau, Rétro, Robotique, Sécurité, Société, Système, Témoignage, Tests et prise en main, Web). Arrows point from the text on the right to specific elements on the website.

Sécurité informatique,
Open Source, Linux,
Électronique, Embarqué...



c'est la solution pour
vous et votre équipe !

À découvrir sur :
connect.ed-diamond.com

Pour vous abonner :
www.ed-diamond.com

Accès multi-lecteurs
possible pour :
Pro, R&D, Enseignement...

En savoir plus :
Tél. : +33 (0)3 67 10 00 28
E-mail : connect@ed-diamond.com

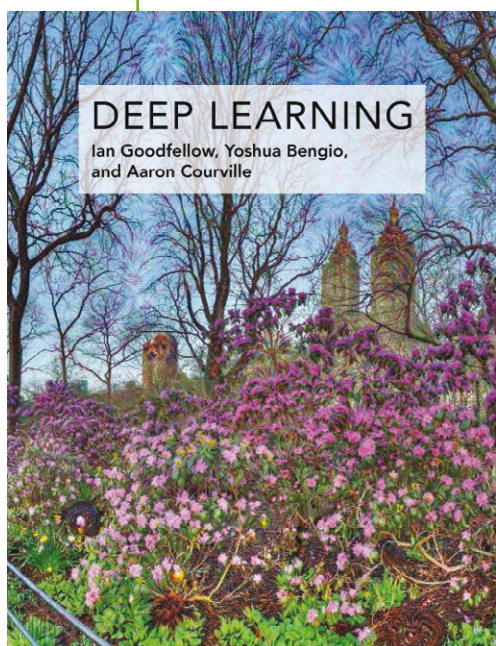
REVUE DE LIVRES...

Reda DEHAK

La littérature regorge de livres traitant le sujet de l'apprentissage automatique, le Machine Learning. Cet ensemble de livres peut être décomposé en deux parties :

- La première partie regroupe les livres traitant le sujet en abordant le côté pratique sans trop insister sur la théorie, ils misent sur la partie programmation en utilisant les différentes bibliothèques connues du domaine.
- La deuxième partie moins connue chez les novices, mais reconnue comme des livres de référence du domaine chez les experts.

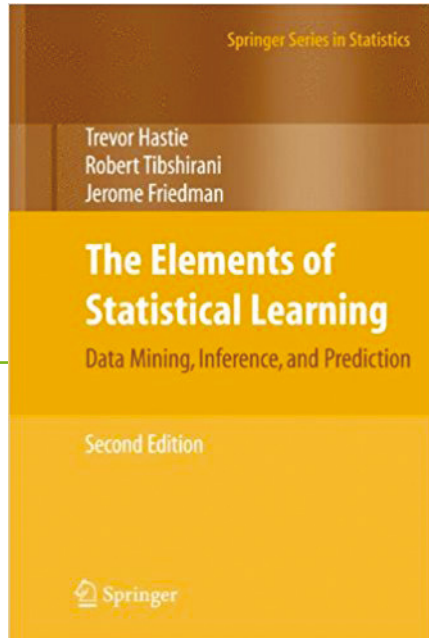
Nous allons essayer de mettre en lumière trois livres de référence dans le domaine de l'apprentissage automatique à travers cette petite revue. Ces livres abordent le côté théorique du domaine, pour la compréhension de leur contenu le lecteur devrait avoir un minimum de connaissance en mathématiques, probabilités et algèbre linéaire.



DEEP LEARNING

Ce livre se focalise sur l'une des techniques les plus répandues en ce moment à savoir les réseaux de neurones profonds (Deep Learning). Un rappel des notions de mathématiques, de l'algèbre linéaire, de l'optimisation ainsi que les bases de l'apprentissage automatique nécessaires à la compréhension de la suite sont présentés dans les cinq premiers chapitres. Le livre aborde les réseaux de neurones classiques, les versions profondes ainsi que les réseaux récurrents et les modèles génératifs. Vous trouverez l'ensemble des connaissances théoriques utilisées dans ce domaine. Pour une meilleure compréhension, l'étude de ce livre doit être accompagnée d'un livre pratique sur la programmation et l'application des réseaux de neurones profonds en utilisant l'une des bibliothèques spécialisées dans ce domaine : TensorFlow, Keras, Caffe, CNTK, PyTorch, Mxnet... comme par exemple le livre de Aurélien Géron [1].

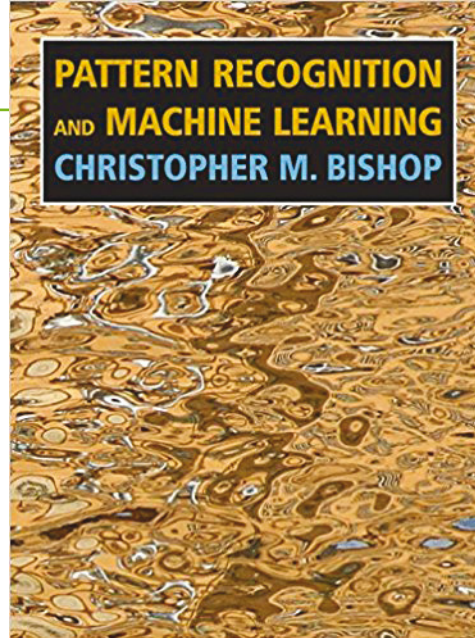
- **Auteurs** : Ian Goodfellow, Yoshua Bengio et Aaron Courville
- **Éditeur** : The MIT press
- **Parution** : 2016
- **Nombre de pages** : 775
- **Prix** : 74 €



THE ELEMENTS OF STATISTICAL LEARNING: DATA MINING INFERENCE AND PREDICTION

Le sujet de l'apprentissage automatique est abordé d'un point de vue statistique, ce livre est un excellent complément pour le Bishop. Des méthodes peu détaillées dans le Bishop comme le Boosting, le Bagging et les Random Forest sont très bien étudiées dans cet ouvrage. Cet ouvrage met l'accent sur les principaux problèmes statistiques liés à l'apprentissage automatique (dimensionnalité, biais, variance, etc.) qu'une grande majorité de débutants néglige en construisant leurs solutions. ■

- **Auteurs** : Trévor Hastie, Robert Tibshirani et Jérôme Friedman
- **Éditeur** : Springer
- **Parution** : 2009
- **Nombre de pages** : 745
- **Prix** : 56.99 €
- **Voir** : <https://web.stanford.edu/~hastie/ElemStatLearn/>



PATTERN RECOGNITION AND MACHINE LEARNING

Malgré son ancienneté, ce livre reste une référence dans le domaine. Le contenu est très académique et destiné à des chercheurs ou des ingénieurs. Il aborde un large spectre de méthodes et techniques bayésiennes. Le contenu est autosuffisant, malheureusement, à cause de l'ancienneté, les résultats et méthodes récentes du domaine ne sont pas abordés ou très peu détaillés. ■

- **Auteur** : Christopher M. Bishop
- **Éditeur** : Springer
- **Parution** : 2006
- **Nombre de pages** : 738
- **Prix** : 74.85 €

RÉFÉRENCE

- [1] « Hands-On Machine Learning with Scikit-Learn and TensorFlow : Concepts, Tools, and Techniques to Build Intelligent Systems », Aurélien Géron, 2017



ENTRETIEN AVEC JULIEN CORNEBISE,

EXPERT EN MACHINE LEARNING

Propos recueillis par Robert ERRA

Julien Cornebise est un expert francophone du Machine Learning qui a été l'un des premiers chercheurs de DeepMind, entreprise connue pour avoir notamment développé AlphaGo (une IA qui battra les meilleurs joueurs de Go au monde). Il a accepté de répondre à nos questions afin de nous faire découvrir le parcours d'un chercheur en Machine Learning.

MISC : Peux-tu dire quel est ton poste actuel ?

Tout d'abord, merci de m'héberger dans les colonnes de ce hors-série de MISC. Je suis *Director of Research*, chez Element AI, en charge de notre bureau de Londres que nous avons ouvert en janvier

2018. J'ai le plaisir d'y construire et mener une équipe d'ingénieurs et de chercheurs passionnés afin d'avancer nos efforts en « AI for Good » : aider les ONGs, organisations internationales, et acteurs locaux à utiliser pour leurs missions les outils créés par la communauté du Machine Learning.

MISC : Quel est ton parcours pendant ces 10 dernières années ?

En bref, université, industrie, recherche, et applications, avec un fil rouge : comprendre et étendre des algorithmes pour l'inférence probabiliste, afin de leur donner le plus d'impact concret.

Version longue : doctorat obtenu en 2009 en statistique computationnelle à Paris 6 et Telecom ParisTech, deux postdocs de 2009 à 2011 (SAMSI, NC et UBC Vancouver, puis University College London), et une année de consultant indépendant pour l'industrie, avant de rejoindre en 2012 ce qui était alors une petite startup de 36 employés : DeepMind Technologies, désormais Google DeepMind. J'y suis devenu le 5ème ou 6ème chercheur et y ai passé quatre années : deux ans en recherche fondamentale sur les modèles génératifs, les mécanismes mémoriels et l'incertitude dans les réseaux de neurones, puis, après l'acquisition par Google, j'ai co-créé et mené l'équipe de recherche appliquée à la santé, en collaboration avec cliniciens et hôpitaux, dont les premiers résultats viennent d'être publiés dans Nature Medicine.

Fin 2016, j'ai quitté DeepMind, voulant voir où je pouvais avoir le plus d'impact, et suis devenu volontaire pour Amnesty International, les aidant sur leur premier projet de Machine Learning. Cherchant alors le meilleur véhicule pour systématiser de telles collaborations, j'ai rejoint Element AI en janvier 2018 et créé une équipe à Londres précisément dans ce but.

MISC : Comment as-tu découvert le Machine Learning ?

En école d'ingénieurs, un professeur et mentor fascinant m'a passionné pour l'algorithmique. Réalisant qu'hélas peu de problèmes réels sont réellement déterministes, je me suis alors tourné vers des algorithmes basés sur des méthodes plus probabilistes : la statistique computationnelle, et le Machine Learning, domaines dans lesquels j'ai alors effectué mon DEA et mon doctorat. Pour être précis, je pourrais entrer dans la conversation périlleuse sur la nuance entre la statistique et le Machine Learning, sur laquelle tout le monde n'est pas d'accord. En université, la première est souvent (mais pas toujours) hébergée dans les départements de mathématiques, le second dans les départements d'informatique. La différence est peut-être dans l'emphase algorithmique du second – mais après tout peu importe : les deux ont en commun l'union de mathématiques et de données, et une volonté d'être applicables au plus près des problèmes réels.

”

Je suis devenu le 5ème ou 6ème chercheur chez DeepMind et y ai passé quatre années : deux ans en recherche fondamentale sur les modèles génératifs, les mécanismes mémoriels et l'incertitude dans les réseaux de neurones.

MISC : Peux-tu nous dire ce qui te passionne dans le ML ?

L'approche « méta » derrière le Machine Learning est incroyablement riche : ne pas programmer pour faire, mais programmer pour apprendre à faire. La beauté en est double : mathématique et applicative. Mathématique, car ces algorithmes, lorsqu'ils sont proprement justifiés (ou, à défaut, « intuités »), reposent souvent sur des interprétations probabilistes particulièrement élégantes – une passion de chercheur. Et applicative, car le but de cet apprentissage est justement d'être applicable de façon immédiate à de très nombreux problèmes différents, pour peu qu'une quantité suffisante de données soit disponible – une passion d'ingénieur.

”

**Ne pas
programmer
pour faire, mais
programmer pour
apprendre à faire.**

MISC : Quelques mots sur Element AI ?

Element AI est une compagnie d'intelligence artificielle qui avance la recherche de pointe sur l'IA et la transforme en produits à l'échelle qui rendent les entreprises plus sûres, plus robustes, et plus agiles. Co-fondée par Jean-François Gagné, Nicolas Chapados et Yoshua Bengio en 2016, elle a annoncé en juin 2017 une levée de fond de Série A de \$102M USD, une ronde historique pour le domaine technologique et le plus grand dans l'histoire canadienne. Ayant grandi de 8 à 370 employés en 22 mois, dont 80 PhDs, elle est désormais basée dans 5 hubs majeurs : Montréal, Toronto, Londres, Séoul et Singapour.

MISC : Peux-tu nous parler du courant « AI for Good » ?

Le terme est très vague, mais est utilisé par les Nations Unies et a donc le mérite de servir de point de référence, avec des variantes comme « AI for Social Good ».

Pour citer Jeffrey Hammerbacher, « Les meilleurs esprits de ma génération cherchent comment pousser

les gens à cliquer sur des publicités ». C'est une perte incroyable pour notre société. Heureusement, beaucoup de membres de la communauté scientifique et technologique ne demandent qu'à mieux faire – AI for Good ouvrant peut-être un chemin dans cette direction.

Soyons clairs : je suis un scientifique, je n'ai aucune prétention de savoir comment résoudre les grands problèmes géopolitiques – et je ne veux surtout pas tomber dans le cliché « Laissez-moi faire, je vais sauver le monde avec [technique du jour] ».

En revanche, il y a des personnes qui ont passé des décennies, des carrières entières, voire risqué leur vie pour découvrir ce qui marche, ce qui ne marche pas, quelles erreurs à ne pas faire. On les trouve dans les ONG, les organisations internationales, certains gouvernements, et les acteurs locaux. Pouvons-nous ensemble trouver un moyen de mettre modestement nos outils et nos techniques à leur service, pour faciliter quelques aspects de leur mission et leurs efforts, là où cela peut être utile? Sans mégalomanie, en ayant conscience de nos limites techniques, mais en repoussant les limites d'idées !

MISC : Peux-tu nous parler d'un projet que tu as réalisé dans ce courant ?

Les durées des projets qui ont un réel impact sont typiquement assez longues, et je préfère parler de projets après leur conclusion que pendant leur développement. Je crois sincèrement dans le bénéfice de faire avant de parler. Il peut être tentant et facile pour des organisations de

signaler leur vertu en annonçant le lancement de projets « for good » à l'état embryonnaire – a fortiori combinés à l'excitation actuelle autour de l'intelligence artificielle. Et si nous mettons la barre plus haut, pour qu'il devienne alors standard d'obtenir un résultat avant de clamer victoire ? Le rapport signal/bruit deviendra alors bien supérieur, et ce sera tellement plus inspirant, que l'on peut alors espérer un effet boule de neige.

MISC : Quelles sont selon toi les limites du ML et plus généralement de l'IA ?

Les limites actuelles sont multiples, ce qui explique l'extrême activité de recherche dans le domaine.

Par exemple, l'apprentissage supervisé par réseaux de neurones requiert actuellement une quantité

”

Pour citer

Jeffrey

Hammerbacher,

« Les meilleurs

esprits de ma

génération

cherchent comment

pousser les gens

à cliquer sur

des publicités ».

C'est une perte

incroyable pour

notre société.

Le fournisseur principal de solutions en IA

Nous faisons avancer la recherche de pointe en intelligence artificielle et en développons des produits modulables pour rendre les entreprises plus sécuritaires, plus efficaces et plus agiles.



considérable de données étiquetées : le « transfert learning » et le « multi-task learning » ou « meta-learning » cherche à réduire le nombre d'exemples nécessaires pour chaque tâche en répartissant l'apprentissage sur plusieurs jeux de données ou tâches indépendantes. L'apprentissage semi-supervisé, voire non supervisé, cherche à utiliser des données non étiquetées, souvent plus nombreuses.

Les réseaux de neurones sont très performants sur des données « denses » et très structurées telles qu'images, sons, vidéos. Leur extension à des structures plus arbitraires, telles que des graphes, en est encore à ses balbutiements, avec des travaux récents prometteurs.

L'apprentissage par renforcement, où un agent interagit directement avec l'environnement, est derrière beaucoup des succès marquants de ces cinq dernières années : AlphaGo/AlphaZero, joueur Atari, OpenGym etc. Mais combien d'applications réelles permettent une telle inte-

raction – ou bien ont un simulateur parfait sur lequel l'agent puisse apprendre ?

Et finalement, alors que ces outils deviennent utilisés de plus en plus largement, se posent des questions clés : comment éviter de reproduire, voire de renforcer des biais déjà présents dans les données sur lesquelles l'algorithme apprend ? Voir par exemple les travaux de Lum et Isaac du Human Rights Data Analysis Group sur la police préventive.

Parallèlement, sur une note bien moins technique que sociale, je pense qu'il y a actuellement une « hype » malsaine autour du terme « intelligence artificielle ». Ce n'est pas étonnant : ce thème résonne avec la volonté de créer un être à son image, un rêve commun à de nombreuses cultures – on le retrouve dans la Genèse, dans le mythe du Golem, dans le Frankenstein de Mary Shelley... Il est important de séparer le fantasme de la réalité, et je pense que les scientifiques et ingénieurs ont le devoir d'aider à ancrer la conversation dans le réel.



Il est important de séparer le fantasme de la réalité, et je pense que les scientifiques et ingénieurs ont le devoir d'aider à ancrer la conversation dans le réel.

MISC : Un conseil pour qui veut se lancer dans le ML ou l'IA ?

Être curieux, expérimenter, chercher, lire beaucoup, et surtout, savoir ne pas foncer tête baissée dans la dernière mode. Il est bien sûr extrêmement utile d'acquérir l'intuition en expérimentant, ce qui est nécessaire pour des applications couronnées de succès, et rendu très facile par les nombreuses bibliothèques récentes (PyTorch, SkLearn, Tensorflow, etc.). Mais prenez toujours le temps de situer les derniers algorithmes par rapport à leurs prédécesseurs. Vous en tirerez une compréhension bien plus profonde, tant de leurs limitations que de leurs forces, et une souplesse pérenne. Comme en sécurité : un praticien comprenant la stack de JavaScript jusqu'à l'ASM x86, d'HTTP jusqu'aux IRQ, fera toujours des choses bien plus intéressantes qu'un script kiddie. ■

ACTUELLEMENT DISPONIBLE

NOS HORS-SÉRIES



NE LE MANQUEZ PAS

CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

CHANGENT....

DÉCOUVREZ LEUR NOUVELLE FORMULE !

QUOI DE NEUF ?

- » un dossier offrant un tour d'horizon complet d'une technologie ou d'un domaine en particulier
- + des retours d'expériences sous la forme d'interviews ou de portraits
- + un contenu plus varié permettant de se familiariser avec diverses techniques et tendances à travers des décryptages ou des articles repères
- + des sujets liés aux dernières actualités technologiques
- + un format offrant un meilleur confort de lecture
- » et toujours 128 pages pour faire le plein de connaissances !



AUX VOLEURS

Paul JUNG

Les voleurs de mots de passe ne sont pas une nouveauté, ni même, pour la plupart quelque chose d'exceptionnel d'un point de vue technique. Mais le nombre fait la force. Découvrons ce petit monde.

INTRODUCTION

Les voleurs de mots de passe, communément appelés par les éditeurs d'anti-virus PWS (*PassWord Stealers*), PSW (*PaSsWords*) ou encore *Information stealer* sont présents depuis plus d'une décennie et ils sont légion. Ces logiciels, comme Agent Testla, LokiBot ou Pony (appelé aussi FareIT), sont fortement utilisés, répandus, et personne ne semble s'en soucier réellement. Nous allons dans cet article, et d'autres qui suivront, étudier cette petite industrie et passer en revue cette famille de malwares. Ces malwares sont variés et inégaux, des plus connus de qualité professionnelle aux plus extravagants, en passant bien sûr par ceux de triste qualité dont on se demande vraiment si un collégien ne les a pas écrits. Nous allons voir comment ils fonctionnent, leurs spécificités, leurs lacunes, et surtout comment les identifier.

Étonnement peu de gens semblent s'intéresser à ces petites entreprises ; elles sont évidemment moins médiatisées que les outils utilisés par les groupes APT (*Advanced Persistent Threat*). Mais si l'on regarde les statistiques d'incidents, leur place n'est pas usurpée.

Un PWS n'est pas un RAT (*Remote Access Tool*) : il faut faire cette distinction, le but d'un PWS est d'être exécuté par la victime, de récupérer tout ce qui est possible en termes d'identifiants de connexions, éventuellement des numéros de série de logiciels voire même des portefeuilles de monnaies virtuelles. Enfin ils s'occupent d'exfiltrer toutes ces données. Certains évidemment sont doublés de keylogger, font des captures d'écran à intervalle régulier et peuvent lancer un payload additionnel, mais dans tous les cas ils sont autonomes.

Tristement, personne ne s'inquiète réellement quand un antivirus se déclenche pour ce genre de menace pourtant le mal est fait.

MAMAN, C'EST QUOI UN PWS?

Un PWS, comme nous l'avons vu, va voler des mots de passe. Pour la plupart ce sont de simples exécutables lancés en userland avec les droits de l'utilisateur courant. Il n'y aura donc pas d'exploitation de 0 days ni de choses techniques croustillantes. Ces maliciels vont récupérer et décoder la configuration et/ou les clés de licence de divers logiciels depuis les fichiers de configuration ou le registre Windows. Ils peuvent aussi récupérer les cookies de sessions et les mots de passe directement dans le processus du navigateur. Le nombre de logiciels couverts est étonnant.

Généralement, une fois ces informations récupérées ainsi que quelques informations sur la machine (antivirus, *fingerprint* du système d'exploitation et de l'utilisateur), ces données sont directement exfiltrées.

Plusieurs modes d'exfiltration sont fréquemment utilisés par ces logiciels. Par mail ou FTP, dans ce cas,

la connexion part directement vers un serveur externe ; mais heureusement ces deux premiers cas n'ont généralement aucune chance de fonctionner au sein d'une entreprise ayant un minimum d'infrastructure de sécurité (firewall et proxy). Le troisième mode utilisé, lui, fonctionne généralement partout. Le maliciel va uploader ses résultats sur un site web dédié. Ce site web se nomme un « *panel* », il permet au malware d'envoyer ses données, mais il permet aussi à l'attaquant d'aller récolter lesdites données.

Sur ce genre de maliciel, la plupart du temps le panel est en PHP, et la qualité du code n'est pas forcément meilleure de ce côté. Il n'est pas rare d'y trouver des vulnérabilités, que ce soient des injections SQL, des uploads arbitraires ou de simples *file listing*. Et pour trouver les sources de ces panels, dans de nombreux cas un **panel.zip** oublié attend patiemment sur le serveur.

Côté malware, il n'y a aucune compétence logicielle requise pour les

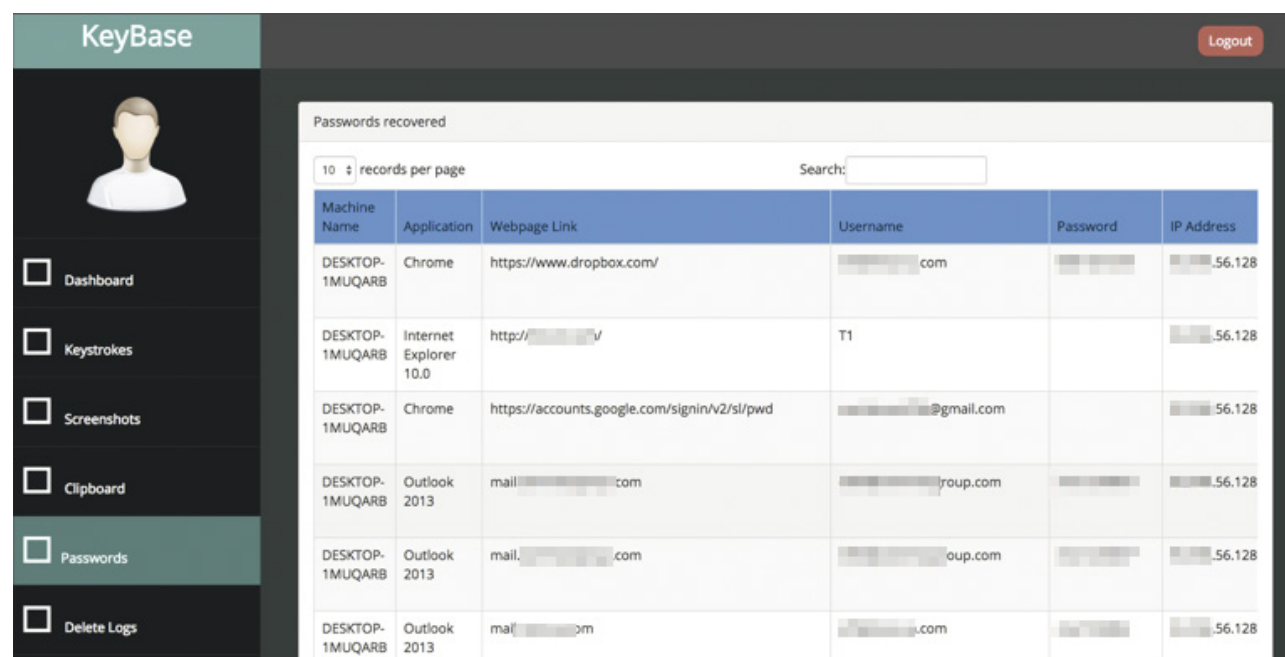


Fig. 1 : Panel d'administration du stealer KeyBase.

opérateurs. Dans la plupart des cas, pour une somme modique, ces logiciels sont livrés clés en main avec ce que l'on appelle un « *builder* ». Le builder permet de générer le PE, éventuellement déjà packé et même, pour certains, le kit permet directement de générer le document office avec macro permettant l'infection de la machine.

SCHÉMA D'INSTALLATION CLASSIQUE

Le schéma d'installation de ce type de logiciel n'a rien d'innovant, il n'est pas différent de n'importe quel autre malware. Celui-ci s'installe en utilisant les voies classiques, e-mail avec lien ou pièce jointe, exploit kit, bref rien de neuf de ce côté-là. À noter cependant que les voleurs de mots de passe ne sont pas forcément utilisés seuls, il n'est pas rare qu'ils soient utilisés avec d'autres malwares. Soit le malware, généralement un bancaire, *download* et exécute un voleur de mot de passe, soit ledit voleur s'occupe de télécharger et d'installer le malware bancaire. En effet, il n'est pas rare de voir un voleur de mot de passe comme Pony installer un bancaire Vawtrack ou à l'inverse un bancaire Ramnit lançant un PWS Azorult [INFECT].

QUI SONT-ILS ?

Et c'est là que le problème commence. Depuis 2016, quand j'ai commencé à m'intéresser à ces petits bouts de code, j'ai pu répertorier, que ce soit via la lecture de forums plus ou moins obscurs ou via l'analyse de samples, près d'une quarantaine de codes malicieux différents. Et ce qui est encore plus choquant au demeurant est la facilité d'accès et l'apparente impunité dont semblent jouir leurs auteurs. En effet, il n'est pas rare que ces malwares soient directement vendus en ligne sur des sites dédiés, sous couvert d'outils de récupération de mots de passe avancés ou de surveillance de votre famille ou d'employés.

Techniquement, certains sortent du lot par leur originalité, leur robustesse ou certaines spécificités. Certains sont sur le terrain depuis plus de 10 ans désormais. Ils sont écrits pour un bon nombre en .net. Leur qualité est inégale.

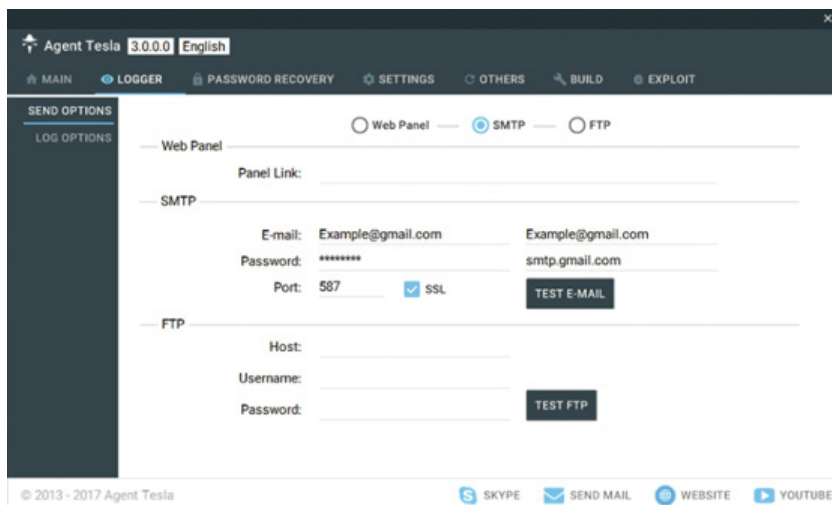


Fig. 2 : Builder du malware Agent Tesla.

L'ÉCONOMIE SOUTERRAINE ET LES RÉCLAMES

Pour acquérir ces logiciels, trois options sont possibles.

Premièrement, le site de vente en ligne. Certains comme Sentinel, Agent Tesla ou Ardamax n'hésitent pas à vendre en ligne directement ces produits en échange de monnaie virtuelle ou de services de transfert d'argent. Généralement, ces sites font passer le produit pour un outil de surveillance de sa famille ou de ses employés voire une solution de récupération de mots de passe (qui, en plus, peut exfiltrer par mail avec des captures d'écran toutes les 10 minutes, un produit idéal !).



Fig. 3 : Noms des stealers classés par occurrence dans les feeds de threat intelligence.



Fig. 4

La deuxième solution d'acquisition est l'obscur forum « underground » qu'il soit de carding ou pas, avec ou sans passage via Tor. C'est de loin la solution la plus amusante à découvrir. Les vendeurs rivalisant entre eux, on retrouve des publicités de qualité quasi professionnelle vantant les mérites des produits mettant en avant leurs qualités qui sont généralement la surface de vol de mot de passe et

l'indéfectibilité du produit. Le tout accompagné de démonstrations voire de promotions étonnantes sur [YouTube].

La troisième et dernière solution est d'utiliser une version ancienne qui a leaké ou d'utiliser directement une version craquée des produits. Après tout tant qu'on est dans la piraterie... Voilà comment fonctionne ce petit écosystème.

Pour en revenir aux malwares eux-mêmes, bien que tous ne soient pas remarquables, certains reviennent souvent. Il apparaît alors judicieux de les connaître et de pouvoir les identifier mieux que le classique « generic.trojan » de l'antivirus.

Nous allons voir en détail Autolog/Sentinel, un de ceux-ci et, dans un prochain article, nous continuerons de voir certains d'entre eux en détail.

ANALYSE D'AUTOLOG

Présentation

Autolog est un voleur de mots de passe sorti en 2016. Au premier abord, la publicité des vendeurs est dithyrambique sur les capacités du produit. Les plus récentes versions d'Autolog sont présentées sous le nom de Sentinel. L'outil est présenté comme écrit en Python et Auto-IT.

Unpacking

Les samples analysés sont packés avec un packer en .net qui s'unpack assez facilement avec DnSpy. L'astuce mise en œuvre pour le packing est un lancement direct de code en assembly .net qui charge un module additionnel qui finalement lance le payload final avec un simple « process hollowing » de type RUNPE.

```
// \u0094\u008D,\u0094\u008C
// Token: 0x06000020 RID: 32 RVA: 0x00002A9C File Offset: 0x00000C9C
[MethodImpl(MethodImplOptions.NoInlining)]
private static void \u0094\u008F()
{
    try
    {
        \u0094\u0098.\u0094\u009C<object, MethodBase>(\u0093\u0091.\u0093\u0094<Assembly>(\u0094\u0092.\u0094\u0095, 262, 256), null, null, 941, 'φ');
    }
    catch
    {
    }
}
```

Emplacement de l'évaluation du code .net.

Afin de dépacker celui-ci, effectuer un premier break sur le chargement de modules (dans **DnSpy Start > No Break > Debug > Modules**). Ce break permet de voir apparaître le module qui s'occupe de charger le second stage en mémoire.

Il suffit dès lors de breaker la fonction PEStart pour obtenir un buffer contenant le stage 2. DnSpy permet de sauver directement le buffer.

```
// wiki.PE
// Token: 0x06000053 RID: 83 RVA: 0x00004348 File Offset: 0x00002548
public static bool PEStart(string hostProcess, byte[] buffer, string commandLine)
{
    int num = 0;
    do
    {
        PE.processId = PE.DoInjectPE(hostProcess, buffer, commandLine);
        num++;
        Thread.Sleep(1000);
        try
        {
            Process.GetProcessById(PE.processId).Dispose();
        }
    }
}
```

```

    catch
    {
        PE.processId = -1;
    }
}
while (PE.processId == -1 && num = < 5);
return false;
}

```

Appel de la fonction exécutant le RUNPE.

Le fichier récupéré depuis ce buffer est un exécutable packé avec le packer open source UPX. Il est rapidement identifiable avec la commande **file** et la commande **UPX -d** permet tout aussi rapidement de le dépacker.

Il apparaît alors que cet exécutable est un script Auto-IT compilé. La présence de code Auto-IT est aisément détectable avec un simple « strings ».

```

$strings -el stage2_noupX.bin | grep -i autoit | head
AutoIt v3
>>>AUTOIT NO CMDEXECUTE<<<
/AutoIt3OutputDebug
/AutoIt3ExecuteLine
/AutoIt3ExecuteScript
AutoIt
Software\AutoIt v3\AutoIt
#OnAutoItStartRegister
AUTOITSETOPTION
AUTOITWINGETTITLE

```

Détection de chaînes de caractères Auto-IT.

[Auto-IT] est un langage de scripting qui ressemble à du BASIC. Il est généralement utilisé pour de l'automatisation de tâches ou des *logins scripts* sous Windows. Ce langage a l'avantage de pouvoir être compilé en exécutable et avoir un code qui fonctionne de Windows XP à Windows 10. Ce langage est, de plus, capable d'appeler nativement les API de Windows et des fonctions depuis des DLL. Enfin, il permet d'évaluer et d'exécuter du code à l'instar d'un **eval()** en PHP ou JavaScript. Bref, il a toutes les qualités requises pour la création de malwares.

Pour retrouver le code natif à partir de l'exécutable PE, on utilisera l'outil **[exe2auto]**. Cet outil permet avec un simple drag'n drop de récupérer directement le code Auto-IT.

Analyse

L'entièreté du code d'**[Autolog]** est offusqué en créant au vol les chaînes de caractères. On va retrouver les mêmes méthodes que dans un script malveillant en PHP ou JavaScript.

Les variables et fonctions utilisent toutes la même nomenclature, et toutes les chaînes de caractères sont stockées dans un fichier compagnon avec une extension faisant penser à un certificat x509 « stub.au3.509 ». À l'intérieur de celui-ci, les chaînes sont simplement encodées en hexadécimal avec un séparateur.

Tout le code utilise la construction suivante.

La base est une simple routine de décodage qui convertit les données du fichier avec extension .509 en chaîne de caractères.

```
Func a4600006061($a4600006061)
  Local $a4600006061_
  For $x = 1 To StringLen($a4600006061) Step 2
    $a4600006061_ &= Chr(Dec(StringMid($a4600006061, $x, 2)))
  Next
  Return $a4600006061_
EndFunc
```

Routine de décodage hexadécimal vers ascii.

On peut évidemment reconvertir facilement le fichier compagnon avec une simple ligne de commandes.

```
$cat stub.au3.509 | sed s' /\[04Et/0A/g' | xxd -r -p | head
struct;long Left;long Top;long Right;long Bottom;endstruct
uint cbSize;uint fMask;uint fStyle;dword clrFore;dword clrBack;ptr lpText;uint cch;
int iImage;hwnd hwndChild;uint cxMinChild;uint cyMinChild;uint cx;handle
hbmBack;uint wID;uint cyChild;uint cyMaxChild;
uint cyIntegral;uint cxIdeal;lparam lParam;uint cxHeader
@OSVersion
WIN_XP
;
;uint uChevronState
dword vkCode;dword scanCode;dword flags;dword time;ulong_ptr dwExtraInfo
1
```

Décodage du fichier compagnon.

Mais cela ne permet malheureusement pas de comprendre le code. De plus, si en effet certaines chaînes sont décodables directement, toutes les chaînes de caractères ne le sont pas. Certaines chaînes ont un codage supplémentaire. Dans le fichier précédemment décodé, on peut aisément trouver certaines chaînes qui semblent être des chaînes en base64, mais dans certaines, fait étrange, on retrouve un caractère alphanumérique après le padding.

```
WQ05SaWBXa1JycTB2YuFmbyViO=A
```

En relisant le code, on découvre une seconde routine qui pour certaines chaînes de caractère échange les caractères deux à deux avant de décoder le base64.

```
Func a5a60605046($a1f2ca04135)
  If NOT IsDeclared("SSA5A60605046") Then
    Global $a582cd0040e = a4600006061($cw[2479]), $a522ce04017 = a4600006061($cw[2480]),
    $a0a2cf05619 = a4600006061($cw[2481])
    Global $ssa5a60605046 = 1
  EndIf
  Local $a2c2cb03532, $a182cc05a49
  For $a2c2cb03532 = Number($a582cd0040e) To StringLen($a1f2ca04135) Step Number($a522ce04017)
    $a182cc05a49 = $a182cc05a49 & STRINGREVERSE(StringMid($a1f2ca04135, $a2c2cb03532,
    Number($a0a2cf05619)))
  Next
  Return $a182cc05a49
EndFunc
```

Routine de décodage swap base64.

WQ05SaWBXa1JycTB2YuFmbyViO=A donne **QW50aSBWaXJ1cyBTY2FubmVyOiA=** qui veut alors dire « Anti Virus Scanner: » en base64.

La recette CyberChef suivante en vient rapidement à bout.

```
Swap_endianness('Raw',2,true)
From_Base64('A-Za-z0-9+/' ,true)
```

Et c'est ainsi dans tout le code, la plupart des chaînes, telles un patchwork, sont construites en utilisant ces fonctions de décodage. De plus, les chaînes, dans un souci d'offuscation, sont construites sur la même ligne puis concaténées.

```
$a0fb840302f = a4600006061($cw[1734]), $a2db8502d19 = a4600006061($cw[1735]), $a2ab8600a1b =
a4600006061($cw[1736]), $a45b870421d = a4600006061($cw[1737]), $a34b8805427 = a4600006061($cw[1738]),
$a2eb8901707 = a4600006061($cw[1739]), $a5ab8a03112 = a4600006061($cw[1740]), $a4ab8b0631b =
a4600006061($cw[1741]), $a59b8c04c23 = a4600006061($cw[1742]), $a22b8d04611 = a4600006061($cw[1743]),
$a00b8e00e1a = a4600006061($cw[1744]), $a2bb8f02511 = a4600006061($cw[1745]), $a3dc8000f14 =
a4600006061($cw[1746]), $a60c8101021 = a4600006061($cw[1747]), $a05c8203312 = a4600006061($cw[1748])
```

Concaténation d'une simple chaîne de caractères dans la variable a0fb840302f.

Elles sont soit directement utilisées comme dans cet exemple :

```
Sleep(Number($a0fb840302f))
```

Soit elles représentent du code Auto-IT qui est réévalué avec le mot clef « Execute » :

```
Global Const $a0a44305f23 = $a4944400725 & $a3e44505e2d & $a5344602c1e &
((Execute($a0444701658) = $a444480175a) ? "" : $a5b4490050f & $a2b34802149 &
$a2b44a0464f)
```

Le code utilise de manière intensive cette logique et est ainsi offusqué sur près de 3000 lignes rendant son analyse statique assez complexe. Pour compliquer la vie de l'analyste, au démarrage après évaluation des chaînes, le fichier compagnon avec l'extension .509 contenant toute la configuration est supprimé par le malware.

Heureusement, le code Auto-IT peut être exécuté facilement dans le débogueur propriétaire rendant alors son analyse facile.

Capacités

Autolog est donc un simple malware avec comme particularité son écriture en Auto-IT. Il dispose cependant des mêmes capacités que ses semblables.

Parmi les capacités de ce malware, on peut noter :

- une répllication dans le répertoire utilisateur couplée à une persistance simple dans la clef **HKCU\Software\Microsoft\Windows\CurrentVersion\Run** et RunOnce nommé humblement « immortal process » par son créateur ;
- un Keylogger Auto-IT natif (et colorisé !) en utilisant directement la fonction **RegisterCallback** ;
- des captures d'écrans ;

- information sur le host (CPU, GPU, nom, IP externe via une requête web sur <http://icanhazip.com/>) ;
- le rapport des solutions de sécurité (antivirus) installées via une query wmi ;
- une exfiltration des données via e-mails ou FTP ;
- la désactivation du firewall (quand les droits le permettent) ;
- la détection de machines virtuelles en se basant sur la détection des processus compagnons.

À noter, l'amusante typo dans la version dont je dispose :

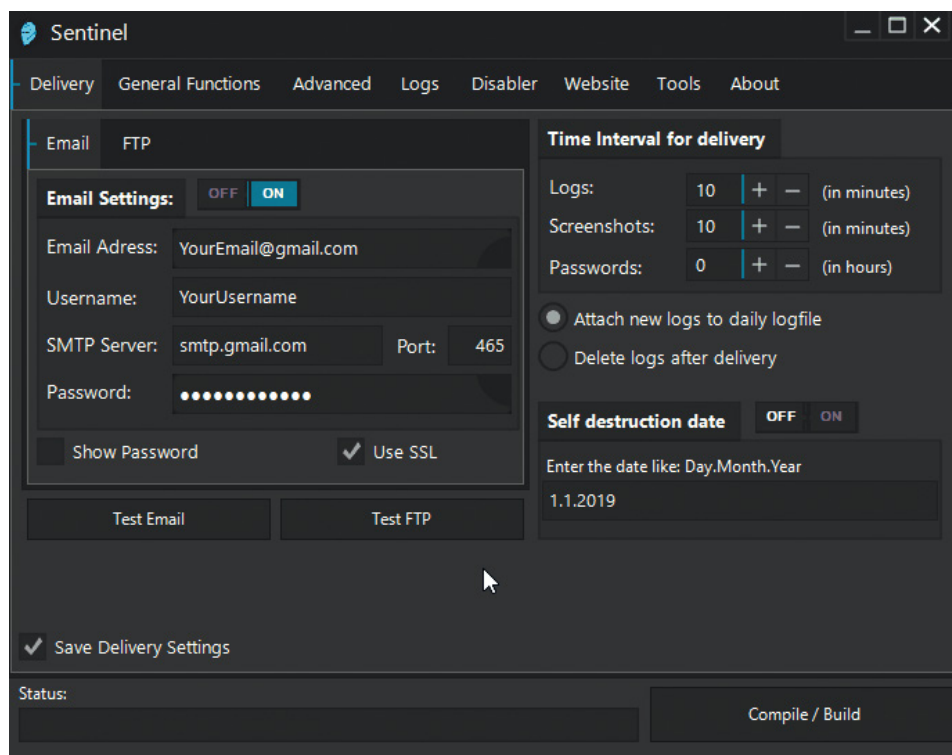
```
If WinGetText(a3a6050431b(str_unobf($a361c100f05))) = $a4b1c204105 Then Exit ; Program Manager = 0
If ProcessExists(a3a6050431b(str_unobf($a061c305816))) Then Exit ; VboxService.exe
If ProcessExists(a3a6050431b(str_unobf($a151c405c1b))) Then Exit ; VMwaretray.exe
If ProcessExists(a3a6050431b(str_unobf($a1b1c502c62))) Then Exit ; vpc.exe
If ProcessExists(a3a6050431b(str_unobf($a211c601c28))) Then Exit ; VBoxTray.exe
If ProcessExists(a3a6050431b(str_unobf($a0d1c701d59))) Then Exit ; VBoxexe
If ProcessExists($a451c800c24) Then Exit ; VmWareTools.exe
If ProcessExists($a1d1c901e23) Then Exit ; VmwareService.exe
EndFunc
```

Routine de détection de la virtualisation.

En effet, le «.» de **Vbox.exe** a été oublié dans la construction de la chaîne obfusquée « kVvJGe4VQZ== ».

Mais il y a un point étonnant quand on débogue le code d'Autolog, on ne trouve aucune trace de routine permettant de capter des mots de passe, et pour cause. En s'exécutant, Autolog va télécharger un exécutable additionnel en http et l'exécute afin d'effectuer la captation des mots de passe. Dans les samples analysés, le module additionnel n'est autre que **[LaZagne]** un projet open source de captation de mots de passe qui a été compilé avec py2exe. Les mots de passe collectés sont exfiltrés et le rapport mentionne en toute transparence l'utilisation de LaZagne. Note amusante, le code compilé fait 15 Mo, ce qui est plus que conséquent pour un « module » additionnel d'un malware de 800ko.

Fig. 5 : Builder du malware Sentinel.



Un peu de commerce

On peut trouver Autolog/Sentinel pour la modique somme de 15\$ mensuels jusqu'à 45\$ pour la licence à vie sur différents sites ayant plus ou moins pignon sur rue, que ce soit en vente **[directe]** ou via divers **[forums]**. Pour ce prix, l'utilisateur dispose du builder lui permettant de générer le malware sans aucune compétence particulière.

CONCLUSION

Les voleurs de mots de passe sont une « race » de malwares assez méconnue du grand public. Certes, la qualité de certains codes les rend beaucoup moins attractifs et visibles que les codes APT, cependant ils sont tout aussi efficaces, couramment utilisés et d'un autre côté assez amusants à reverser. ■

REMERCIEMENTS

Un merci tout particulier à Serge et Xanax pour leur relecture et leurs bons conseils, et merci à Futex pour tous ces moments drôles d'unpacking.

Si vous êtes intéressé par des échanges à propos des stealers, n'hésitez pas à nous contacter, moi ou mes collègues du CERT-XLM à cert@excellium-services.com

RÉFÉRENCES

[YouTube] <http://thanat0s.trollprod.org/2018/03/conservation-nationale/>

[INFECT] <https://www.malware-traffic-analysis.net/2018/01/30/index.html>

[Auto-IT] <https://www.autoitscript.com/site/autoit/>

[exe2auto] <http://domoticx.com/autoit3-decompiler-exe2aut/>

[Autolog] <https://github.com/Th4nat0s/Yaramoi/tree/master/config/Autolog>

[LaZagne] <https://github.com/AlessandroZ/LaZagne>

[directe] <https://ranger-exploit.com/product/sentinel-keylogger/>

[forum] <https://hackforums.net/showthread.php?tid=5734981>

MACHINE LEARNING & SÉCURITÉ

AU SOMMAIRE :

28 Machine Learning : un (rapide) tour d'horizon

42 ML & sécurité : les opportunités

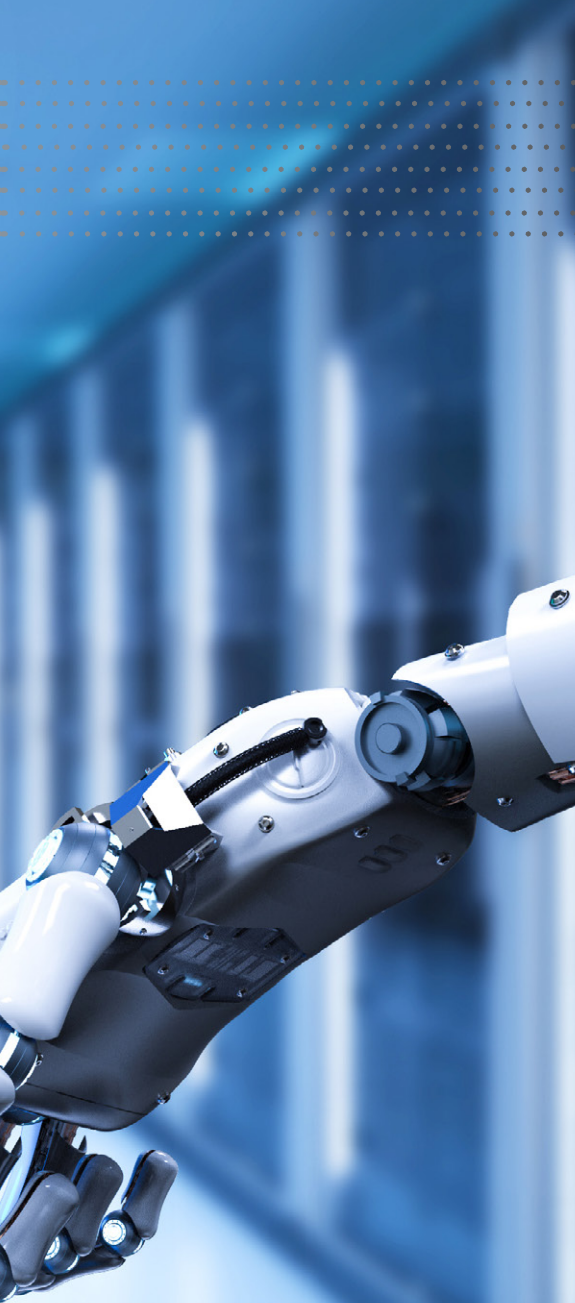
51 Vers une détection d'intrusion dynamique et continue en utilisant les réseaux de neurones

62 Machine Learning en Python : clusterisation à la rescousse des hunters (de malware)

74 Automeans, ou comment éviter le « k par k » avec K-means

84 Analyse des sentiments avec le Deep Learning

97 YaDiff : de l'Intelligence Artificielle pour comparer les codes binaires



Se promener au FIC 2018 permettait de se rendre compte, jusqu'à la nausée, qu'un invité de marque était présent

sur pas mal de stands : le *Machine Learning* (ML), catalogué comme une technologie de rupture par le MIT. Alors le ML est une révolution ? Est-ce réservé aux GAFAM ? Faut-il une armée d'ordinateurs et un doctorat en mathématiques pour calculer votre premier réseau de neurones ? Un peu d'histoire : le CRAY I (années 70) affichait 160 millions d'opérations flottantes par seconde pour 9 millions. À 3000 euros, on a une carte GPU à 110 Teraflops ! 687500 fois plus d'opérations à la seconde. C'est un pic, même à 10% c'est colossal. Convaincu ? Regardons le coût : 3000 euros, c'est 3000 moins cher qu'un CRAY I.

La vraie révolution ce n'est pas que puissance de calcul, grande quantité de données et algorithmes sont disponibles (les deux « moteurs » du ML, l'algorithme de rétropropagation de gradient et la différentiation automatique sont anciens), mais que le ML s'est démocratisé ! Une carte GPU ou quelques euros suffisent, des calculs inimaginables il y a 20 ans sont aujourd'hui faisables chez vous, *at home* !

Le ML s'invite donc dans la sécurité ? Oui, à la fois pour de bonnes et de mauvaises raisons. Mauvaise raison ? « Le ML est à la mode et on dit qu'on en fait sinon on coule ». Bonne raison ? Il est facile d'obtenir des données et des algorithmes faciles à utiliser (grâce à : Keras, TensorFlow, PyTorch et Scikit-Learn). Un projet de ML est un processus itératif d'expérimentation(s). Embaucher un *features engineer* (nouveau métier) ? Oui, ou alors : lisez ce hors-série et apprenez par vous-

même : appropriiez-vous le ML. Pour avoir des résultats, il vous faudra (re) faire un peu de maths et apprendre à comprendre vos données. L'autre impact du ML est là : vous obliger à comprendre les données.

Y a-t-il un risque ? Oui. Donner à « manger » des données à un algorithme et prendre pour Vérité ce qui en sort ? Un exemple ? Henderson, créateur de cette parodie de religion qu'est le pastafarisme avance que « *le réchauffement planétaire, [...] et les autres catastrophes naturelles sont une conséquence directe du nombre décroissant de pirates depuis les années 1800* ». Il « prouve » qu'alors que le nombre de pirates diminuait, la température mondiale augmentait. Remplacez juste « *réchauffement* » par « *dette publique* » (~ 2200 milliards !) et « *pirates* » par « *hérissons* » : vous avez le même type de courbe : la disparition progressive des hérissons en France fait exploser la dette de notre pays !

Oyez, oyez, la vraie disruption, c'est la *dé-mo-cra-ti-sa-tion* du ML : sceptiques ? Lisez donc [1]. Ce n'est plus réservé aux GAFAM ! Le ML comme arme ultime ? Non. Parce qu'il y aura comme souvent une phase de transition (avec confusion et arnaques), mais on arrivera vite (soyons optimistes), à une utilisation intelligente du ML dans le monde de la sécurité. Mais la *dé-mo-cra-ti-sa-tion* est là pour tous : les « méchants » vont s'y mettre, si ce n'est déjà fait ! Grand merci aux auteurs et relecteurs, et n'écrasez plus de hérissons ! ■

Robert ERRA,
Directeur du LSE (EPITA)

[1] <https://www.technologyreview.com/s/611858/small-team-of-ai-coders-beats-googles-code/>



MACHINE LEARNING : UN (RAPIDE) TOUR D'HORIZON

Robert ERRA

Le Machine Learning (ML) qu'on peut traduire par apprentissage automatique ou apprentissage machine (ou encore apprentissage statistique il y a encore quelques années) est catalogué comme une des 10 technologies de rupture par la Technology Review, célèbre revue du MIT. Devenue une expression fétiche il semble qu'aucun domaine ne va y échapper et il était inévitable que la sécurité s'y intéresse, reste à le faire de manière intelligente. Nous nous proposons dans cet article de faire un tour d'horizon des possibilités qu'offre le ML. Cet article devrait vous aider à faire vos premiers pas, à comprendre plus facilement les articles de ce hors-série et enfin, à permettre au lecteur d'évaluer l'intérêt d'une solution de sécurité qui se vante de faire du ML.

« If a machine is expected to be infallible, it cannot also be intelligent. » - Alan Turing

Le Machine Learning est classé par la célèbre revue *Technology Review* du MIT comme une des 10 technologies de rupture [1] et ce, depuis plusieurs années. Soit. Est-ce une mode ? Oui. Et non. C'est une véritable révolution, mais cela n'empêche pas que certaines solutions proposées à la communauté cyber sont probablement inutiles et dues à un effet de mode, c'est inévitable. Dans les domaines du traitement d'images, de la vision, du traitement ou reconnaissance de la parole ou

encore de la traduction automatique, les succès sont nombreux et impressionnants. Récemment, Google a fait sensation avec AlphaGo Zero, version améliorée de AlphaGo [4], ce sont des logiciels de jeu de Go qui sont devenus meilleurs que la plupart des joueurs professionnels développés par sa filiale DeepMind. AlphaGo a donné bien des soucis aux meilleurs joueurs de cet ancestral jeu.

Le ML devrait devenir très vite un des outils de base :

- dans la détection de fraudes dans des opérations bancaires, des opérations en bourse ;
- dans la détection d'anomalies dans les déclarations des impôts ;
- dans la santé ;
- et dans la sécurité du numérique, évidemment.

Enfin, si vous avez lu l'édito, l'auteur pense que la vraie révolution c'est la *démocratisation* du ML : non, ce n'est pas réservé aux GAFAM, tout le monde peut aujourd'hui utiliser le ML et développer une application assez rapidement. Pourquoi ? Parce que :

- Les outils open source comme Scikit-Learn [2] (ou Weka si vous n'aimez pas Python) sont disponibles et relativement aisés à prendre en main.
- Les frameworks et les bibliothèques, en plus de Scikit-Learn devenu incontournable, sont terriblement efficaces et nombreux (citons Tensorflow, Keras, Pytorch ou Theano) et relativement aisés à manier, y compris pour le Deep Learning. Chacun a ses avantages et ses inconvénients.
- Les données sont faciles à trouver (surtout si ce sont les vôtres !). Il suffit sinon d'aller sur un des sites référencés à la fin de cet article.

ImageNet par exemple, publié par Stanford, contient plus de 15 millions d'images labellisées.

- La puissance de calcul est disponible. Même sans carte GPU on peut faire des choses inimaginables il y a encore 20 ans sur un simple PC (tous les exemples présentés ici ont été calculés en un temps raisonnable sans carte GPU).
- De quoi parle cet article (et la plupart des articles de ce hors-série) ? Il s'agit avant toutes choses de :
- présenter les grandes lignes de problèmes qu'on peut aborder avec le ML ;
 - de donner des exemples de ce qu'on peut faire (ou pas) ;
 - et, *last but not least*, devant une offre commerciale de ML en sécurité d'aider le lecteur à décider si l'outil est pertinent ou pas pour son problème, voire s'il y a vraiment du ML dans le produit.

Évidemment, nous ferons des impasses, mais les références, et les autres articles de ce hors-série devraient aider le lecteur à aller plus loin. Nous donnerons des exemples classiques et d'autres moins classiques.

INTELLIGENCE ARTIFICIELLE ET APPRENTISSAGE MACHINE ?

L'intelligence artificielle (IA) est la discipline qui a pour objectif de développer des *programmes informatiques capables de réaliser des tâches sans intervention humaine après avoir été entraînés à partir de grandes quantités de données*. On trouve cette définition dans le

Courrier International du 26 juillet 2018. Wikipédia donne comme définition « *l'ensemble de théories et de techniques mises en œuvre en vue de réaliser des machines capables de simuler l'intelligence* ».

La naissance de l'IA (comme expression et comme discipline) date de la Conférence de Dartmouth de 1956 organisée par Minsky, McCarthy, Shannon et Rochester. Cette discipline a connu des hauts et des bas, les plus vieux d'entre nous se rappellent les années 80 où les systèmes experts étaient à la mode et qui ont finalement quasiment disparu.

Que dit Wikipédia de l'apprentissage automatique ? On trouve « *l'apprentissage automatique (en anglais ML, littéralement « l'apprentissage machine ») ou apprentissage statistique, champ d'étude de l'intelligence artificielle, concerne la conception, l'analyse, le développement et l'implémentation de méthodes permettant à une machine (au sens large) d'évoluer par un processus systématique, et ainsi de remplir des tâches difficiles ou problématiques par des moyens algorithmiques plus classiques* » [3].

Il existe une définition plus formelle, due à Tom Mitchell (1997, cité dans [8]) : « *Étant donné une tâche T et une mesure de performance P , on dit qu'un programme informatique apprend à partir d'une expérience E si les résultats obtenus sur T , mesurés par P , s'améliorent avec l'expérience E* ». Le mot important, mais hélas ambigu est « apprend ». Ainsi, le ML est une branche de l'IA.

Depuis quelques années il y a un renouveau, surtout du ML. Mais pourquoi constate-t-on cet incroyable engouement ces dernières années ?

Il y a plusieurs facteurs :

- Une abondance de données : il est relativement facile aujourd'hui de récupérer un grand nombre de données, avec un coût de stockage très bas ; on pense que 90 % des données actuellement stockées dans le monde ont été créées ces deux dernières années.
- La puissance de calcul et un coût du calcul qui sont devenus très intéressants : on peut, avec ou sans carte graphique GPU, avec ou sans le cloud, disposer de plusieurs Teraflops sans un budget prohibitif ; une carte graphique de 1000 euros vous donne déjà une puissance de calcul considérable et une carte de 3000 euros vous donne une puissance colossale (voir l'édito).

Ces deux facteurs expliquent par exemple pourquoi une technique comme les réseaux de neurones (inventés il y a quelques décennies) sont devenus certes incontournables,

mais surtout opérationnels dans certains domaines comme l'analyse d'images, de vidéos ou de la voix. Mais il y a deux autres facteurs peut-être tout aussi importants :

- Le besoin de nouvelles approches algorithmiques : l'algorithmique classique, non encore morte rassurons-nous, peine à résoudre des problèmes qui se sont fait jour comme la détection d'un objet dans une image ou une vidéo, ou l'identification de locuteur, alors que le ML fait des prouesses dans ce domaine.
- Et la vraie révolution : le fait que tout le monde peut faire du ML chez soi ! En grande partie grâce aux outils open source disponibles en quelques minutes et cités précédemment.

Par « viralité », le ML s'impose doucement dans les domaines de la santé, de la recherche de fraude ou (hélas) dans la *High Frequency Trading* ou les jeux (même le poker y

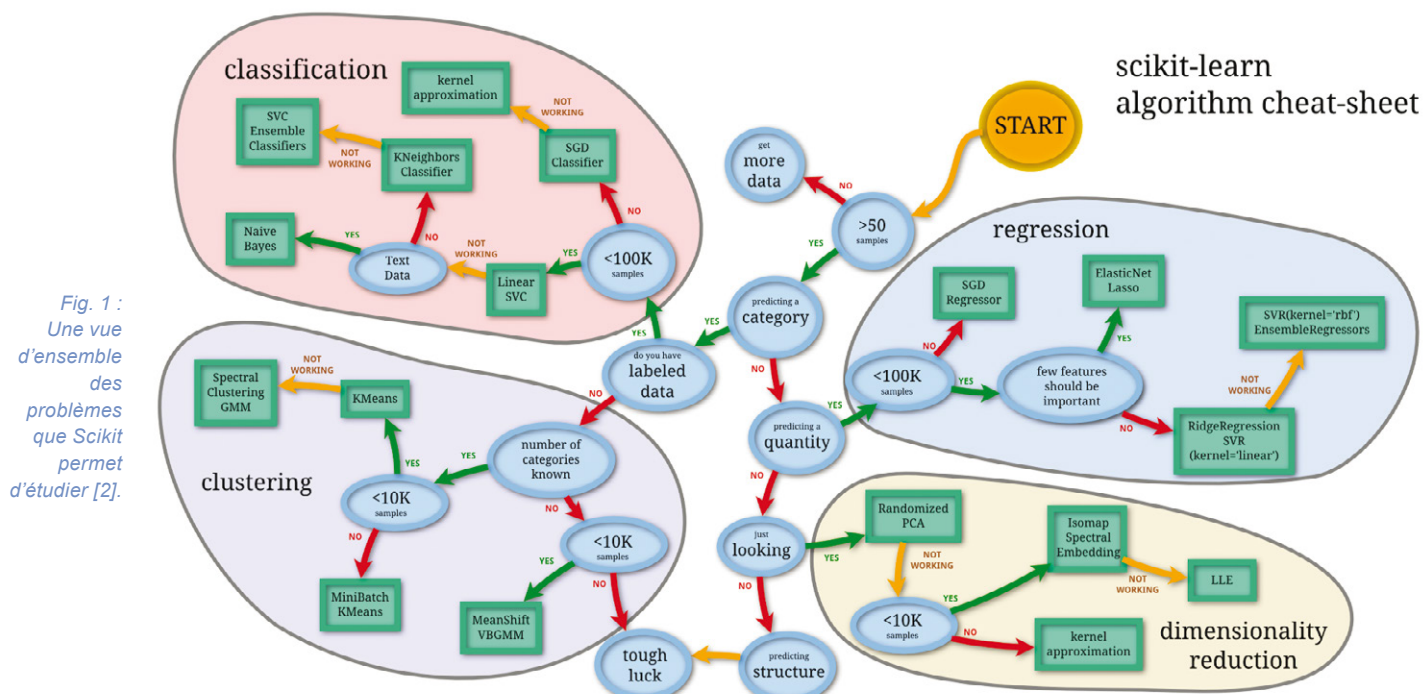
est passé récemment). Devenu une expression fétiche il semble qu'aucun domaine ne va y échapper. Il était inévitable que le monde de la sécurité s'y intéresse.

COMPRENDRE LE MACHINE LEARNING EN UNE IMAGE

Quelles sont les problématiques et les algorithmes du ML ? La figure 1, tirée de [2], donne une vue d'ensemble de ce qu'on trouve dans Scikit-Learn :

- clusterisation ;
- classification ;
- régression ;
- réduction de dimension.

Théoriquement, les algorithmes du ML vous aident à apprendre des connaissances sur un jeu de données. Comme le verbe « to learn »



peut se traduire par « apprendre », mais aussi par « étudier », on peut faire remarquer que tout projet de ML vous oblige aussi à comprendre vos données.

Attention : dans la pratique cependant, il vous faut un jeu de données assez large. Et si celui-ci est trop large, certains algorithmes vous seront en fait interdits.

Avant de faire un tour d'horizon des trois premières de ces différentes problématiques, intéressons-nous un peu à quelques algorithmes du ML.

RÉGRESSION LINÉAIRE POUR PRÉDIRE : COMME M. JOURDAIN...

Si vous avez fait des études scientifiques, comme M. Jourdain faisait de la prose sans le savoir, vous avez probablement déjà utilisé un des algorithmes du ML qui résout le problème de la régression linéaire, encore appelée droite des moindres carrés. Nous allons montrer sur un exemple ce que donne cet algorithme (qui date en gros du 19ème siècle). Au lieu de prendre un des exemples de Scikit, nous allons utiliser la liste des records du monde successifs du 100m.

En regardant la liste des records (sur Wikipédia par exemple), on peut avoir envie de prédire ce que pourrait être le record dans 10, ou 20 ans. Pour ce faire, on doit choisir un modèle. Nous allons simplifier en prenant un modèle linéaire, i.e. on cherche une relation du type : $y = A \cdot x + B$. Où x représente la « date » et y le record (en s). La figure 2 montre la liste des records (points rouges), nos données. Appelons X et Y notre jeu de données et n la taille : $X = \{x_1, x_2, \dots, x_n\}$ et $Y = \{y_1, y_2, \dots, y_n\}$. Comment choisir la « meilleure » droite parmi l'infinité de droite ? On choisit de prendre la droite $y = A \cdot x + b$ tels que A et B minimisent l'erreur quadratique totale (la fonction « coût ») : la somme de tous les $(y_i - (A \cdot x_i + B))^2$.

Effectuer une régression linéaire c'est trouver la droite qui minimise l'erreur totale précédente : la somme des carrés des différences. Il existe de nombreux algorithmes, en fonction de l'ordre de grandeur de n par exemple.

Mais laissons faire Scikit. On commence par charger ce qui nous est nécessaire

```
import matplotlib.pyplot as plt    # Pour faire de jolies figures
import numpy as np                 # Utile pour déclarer des tableaux pour certains
                                   # modèles mais dans ce cas précis on pourrait
                                   # s'en passer
from sklearn import linear_model  # Nécessaire
```

Déclarons les données :

```
01 X_date=np.array([[1968], [1983], [1987], [1988], [1991.5], [1991.66], [1994], [1995],
[1999], [2002], [2005], [2006.4], [2006.5], [2006.66], [2007], [2008.4], [2008.66], [2009]])
02 Y_record =np.array([9.95, 9.93, 9.83, 9.92, 9.90, 9.86, 9.85, 9.84, 9.79, 9.78, 9.768,
9.766, 9.763, 9.762, 9.74, 9.72, 9.69, 9.68])
```

En ligne 1, on déclare les dates, en mettant une valeur réelle correspondant (approximativement) au mois de l'année du record, ainsi août 2006 devient 2006.66. Nous n'avons pris que les records électroniques (1968). Nous avons laissé le « record » de 1987 annulé pour dopage. Nous laissons au lecteur le soin de refaire les calculs sans ce « record ».

On choisit le modèle (ligne 03), nommé ici RL, on calcule le modèle (i.e. on apprend) ligne 4 :

```
03 RL=linear_model.LinearRegression() # RL est le nom de notre modèle
04 RL.fit(X_date,Y_record)
```

Les coefficients A et B de la droite calculés sont obtenus par :

```
A=RL.coef_[0]
B=RL.intercept_
print("Pente=",A, " Constante=",B)
```


Ce qui donne dans notre cas une pente A de -0.006664... et un coefficient B de 23.12... On peut voir la droite sur la figure 1.

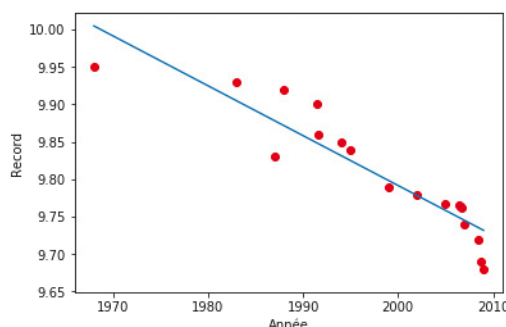
Que peut-on faire avec cette droite ? Mais « prédire » pardi ! Avec notre modèle qui a appris, quel serait par exemple le record en 2018, en 2019, ou en 2100 ? Il suffit de demander avec le code suivant :

```
05 RL.predict([[2018],[2019],[2020],[2030],[2100],[2119]])
```

renvoie **array([9.671..., 9.665..., 9.658..., 9.591..., 9.125..., 8.998...])**, qui peut se lire : l'algorithme (la droite) prédit qu'en 2018 le record aurait dû être de 9.671.. et en 2119 on passe sous les 9 secondes pour la 1ère fois : 8.998...

Évidemment, ce n'est qu'un modèle, qui souffre de sa « linéarité » et en 2018 (même si l'année n'est pas terminée) le record est toujours de 9.68. Si le lecteur refait les calculs en supprimant le faux « record » de 1987, on obtient **array([9.669..., 9.662..., 9.655..., 9.586..., 9.101..., 8.969...])**. C'est meilleur, comme quoi : tricher ce n'est pas bien !

Fig. 2 : Records (points rouges, en s) et dates (réelles) des records en abscisse, avec la droite de régression linéaire calculée.



CLUSTERISATION ET CLASSIFICATION

Clusterisation

De manière très informelle, la classification automatique (en Français, *clusterisation* en Anglais) consiste à regrouper en classes des objets (exclusivement numériques dans notre cas) qui sont les plus similaires et à séparer ceux qui sont différents ou dissimilaires. On désire donc partitionner en classes, si possible, ces classes doivent être distinctes, chaque classe contenant des objets qui se ressemblent, on utilisera dans la suite l'anglicisme *clusterisation*. Le critère de regroupement est réalisé grâce à une fonction de similarité qui mesure la ressemblance entre deux objets.

Classification

Prenons l'exemple des malware. Le but étant de placer un programme malveillant (nouvellement récupéré) dans un groupe afin que celui-ci soit plus proche d'eux que ceux situés à l'extérieur du groupe. La classification dans cette problématique consiste à labelliser un programme inconnu en l'associant à un cluster prédéfini de malware. En clair : on veut labelliser un programme inconnu,

mais considéré comme malveillant en l'associant à un ensemble de programmes pré-labellisés. Chaque ensemble de programmes représente une classe/un cluster ou une catégorie de programmes. L'exemple le plus simple de classification est la classification binaire des programmes en bénins (*goodwares*) ou malveillants (*malware*). Une fois la classe choisie (le label) d'une variante, on peut « élire » un ou plusieurs représentants et comparer plus finement cette variante aux représentants, ceci permet de classifier (en général) une nouvelle variante, ce qui permet éventuellement de comprendre les mécanismes d'infection, d'attaque et de propagation de cette nouvelle variante, mais on peut espérer aussi ainsi avoir envie de détecter plus rapidement un malware « différent ».

Algorithmes de clusterisation

Il existe un grand nombre d'algorithmes de clusterisation, que nous pouvons regrouper en plusieurs catégories :

- Algorithmes supervisés : nous avons un jeu de données déjà labellisées. On dit quelquefois que l'algorithme apprend « par l'exemple ». Il faut alors éviter le sur-apprentissage (*overfitting*).
- Algorithmes non supervisés : des données, mais non labellisées. Des algorithmes existent, la difficulté résidant alors dans la validation.
- On peut y ajouter les algorithmes semi-supervisés : les données sont peu voire très peu labellisées, dans ce cas, le problème devient plus difficile.

Le tableau suivant résume la situation pour la clusterisation (pour les deux premières catégories) :

	Algorithmes non supervisés	Algorithmes supervisés
Nombre de clusters K	Inconnu	Connu
Données	Non labellisées	Labellisées
Exemple d'algorithmes	K-means, DBSCAN, Ward, etc.	KNN, SVM, Réseau de Neurones, Arbre de décision, etc.

Arbres de décision

Un arbre de décision est un algorithme supervisé qui permet, entre autres, de clusteriser des données (labellisées). Donnons un exemple « léger » avec le jeu de données « Iris », intégré à Scikit-Learn. Ces données expérimentales concernent la longueur (*length*) et la largeur (*width*) des sépales et des pétales de 150 fleurs de type iris, de trois espèces différentes (voir le code), chaque espèce ayant exactement 50 représentants.

Voyons ce que cela donne (sans trop entrer dans les détails), on trouve le code équivalent dans de nombreux livres et sur un grand nombre de sites web. On commence par importer ce dont on a besoin :

```
from sklearn.datasets import load_iris          # pour pouvoir charger le
jeu de données IRIS
from sklearn.tree import DecisionTreeClassifier # pour utiliser l'algorithme
qui calcule l'arbre de décision
from sklearn.tree import export_graphviz        # Pratique: pour visualiser
l'arbre (dot de Graphviz)
iris=load_iris()                               # On charge les données :
simple !
print(iris)                                    # on laisse au lecteur le
soin de voir la structure
#
X_data=iris.data[:,2:]                         # On choisit juste la longueur et la largeur des
pétales
y_target=iris.target                          # Contient le type des iris: 0=setosa,
1=versicolor, 2=virginica
print(y)                                       # juste pour vérifier que tout se passe bien
```


On définit le type d'arbre de décision qu'on désire, et on le calcule, ici on utilise le critère d'entropie :

```
arbre_class=DecisionTreeClassifier(max_depth=3, criterion='entropy', random_state=1)
arbre_class.fit(X_data,y_target)
```

Enfin, on crée et on exporte le fichier dot :

```
export_graphviz(arbre_class, out_file="iris_arbre.dot", feature_names=['Longueur des pétales', 'Largeur des pétales'], class_names=iris.target_names, rounded=True, filled=True)
```

On récupère le fichier **iris_arbre.dot**, qu'on traite avec la commande `dot` de Graphviz, ce qui donne la figure 3, facile à lire : le premier critère de séparation est « Largeur des pétales ≤ 0.8 », les cinq feuilles de l'arbre (fond coloré) sont les classes calculées par l'algorithme ! On voit qu'au premier niveau, la classe **setosa** est correctement regroupée (**value=[50,0,0]**) avec une entropie de 0. Les autres classes ne sont pas exactement regroupées, mais avec peu d'erreurs. Le nombre (maximum) de classes est indiqué par **max_depth**, on laisse le lecteur vérifier qu'avec **max_depth=2**, on aurait eu 3 classes et qu'avec **max_depth=4**, on a plus de classes, mais elles sont « pures ».

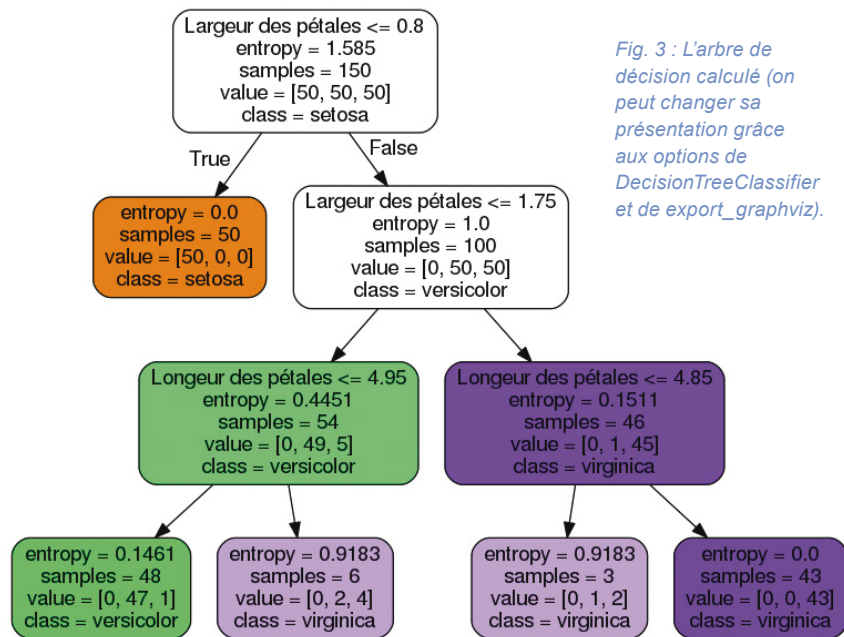


Fig. 3 : L'arbre de décision calculé (on peut changer sa présentation grâce aux options de `DecisionTreeClassifier` et de `export_graphviz`).

DEEP LEARNING : APPRENTISSAGE PROFOND PAR LES RÉSEAUX DE NEURONES ARTIFICIELS (RNA)

Deep Learning se traduit par « apprentissage profond ». C'est la star actuelle du ML. La recherche sur cette technique est colossale depuis quelques années, notamment parce que les GAFAM s'y sont intéressés très tôt. Cela n'a pas toujours

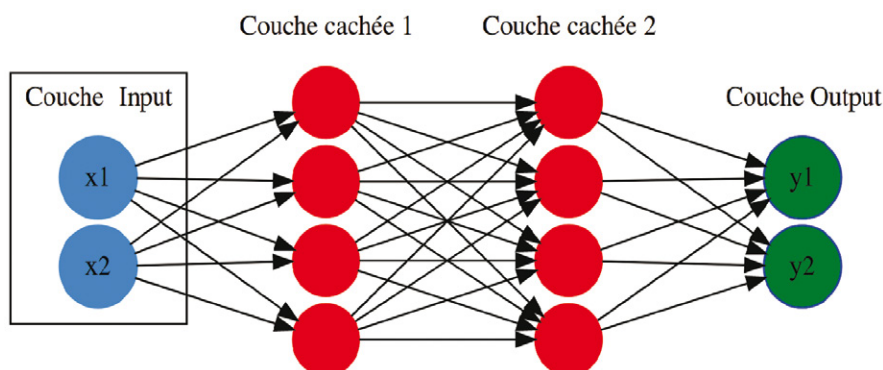


Fig. 4 : Une représentation d'un réseau de neurones artificiels.

été le cas ! La légende dit que les 3 « conspirateurs » du Deep Learning, à savoir G. Hinton, Y. LeCun et J. Bengio se sont vus refuser, par les organisateurs, la création d'un workshop en 2007 lors de la grande messe de l'IA : la conférence NIPS. Ils ont donc créé un workshop « sauvage » qui a attiré plus de 300 personnes. L'article [6] co-écrit par ces 3 conspirateurs du ML décrit parfaitement ce qu'on appelle le Deep Learning.

L'histoire commence dans les années 50 lorsque F. Rosenblatt crée le perceptron, en fait un réseau de neurones à une seule couche. Les articles de ce hors-série présentent diverses architectures de RNA. Un exemple d'architecture de réseau de neurones est donné en figure 4 :

- il comporte une couche d'entrée (deux composantes) ;
- une couche de sortie (ici deux composantes) ;
- deux couches intermédiaires (dites cachées, *hidden*) ;
- quelquefois on peut introduire un biais (voir l'article de A. Boucekif).

Votre premier réseau de neurones : la fonction XOR

Donnons de suite un exemple simple : l'approximation de la fonction XOR, et construisons notre premier réseau de neurones artificiels pour cette fonction avec Keras. Pourquoi faire ? Disons par exemple que vous voulez obfusquer votre code, ou juste coder votre premier réseau de neurones. Ce genre de techniques devrait obliger tôt ou tard les chercheurs en sécurité à attaquer un nouveau domaine le « *Reverse Machine Learning* ».

Nous choisirons un réseau de neurones avec une seule couche cachée, deux entrées et évidemment une seule sortie. La couche intermédiaire étant dans notre exemple composé de 16 neurones.

On commence par charger ce dont nous avons besoin :

```
# En Python 3, installer évidemment Scikit-Learn, Keras, TensorFlow, etc. voir [9, 13]
01 import numpy as np
02 from keras.models import Sequential
03 from keras.layers.core import Dense
```

Ensuite, on déclare les données :

```
# Les 4 différentes entrées de la fonction XOR, sous forme de tableau Numpy
entraîn_data = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")
# Les 4 sorties attendues, dans le bon ordre évidemment
cible_data = np.array([[0],[1],[1],[0]], "float32")
```

On déclare la structure du réseau de neurones (le nombre de couches, les fonctions d'activation, voir la section 5.2), par exemple :

```
# SGD : algorithme d'optimisation numérique (il y en a d'autres : adam ...)
01 model = Sequential()
02 model.add(Dense(16, input_dim=2, activation='relu'))
03 model.add(Dense(1, activation='sigmoid'))
04 model.compile(loss='mean_squared_error', optimizer='sgd', metrics=['binary_accuracy'])
```

On peut résumer à l'écran les choix retenus avec `model.summary()`, un exemple est donné dans l'article de A. Boucekif.

Maintenant, on peut « entraîner » le modèle sur nos données (et imprimer les résultats intermédiaires) :

```
06 model.fit(entraîn_data, cible_data, nb_epoch=10, batch_size=4, verbose=2)
07 print(model.predict(entraîn_data).round())
```

Ce qui donne comme résultats :

```
Epoch 1/5
- 0s - loss: 0.2266 - binary_accuracy: 1.0000
Epoch 2/5
- 0s - loss: 0.2255 - binary_accuracy: 1.0000
```



```

Epoch 3/5
- 0s - loss: 0.2242 - binary_accuracy: 1.0000
Epoch 4/5
- 0s - loss: 0.2231 - binary_accuracy: 1.0000
Epoch 5/5
- 0s - loss: 0.2223 - binary_accuracy: 1.0000
[[ 0.]
 [ 1.]
 [ 1.]
 [ 0.]]

```

On voit que la sortie est un vecteur de flottants, on peut comme on l'a dit les interpréter comme des probabilités ou comme une valeur de la fonction XOR. Les hyperparamètres d'un modèle sont les paramètres qui ne sont pas optimisés par l'algorithme et donc, choisis par l'utilisateur. Ici, c'est : le nombre de couches cachées, la taille de chaque couche, etc. Il faut souvent « jouer » avec, et ce peut être délicat, pour trouver ce qui permet de converger le plus vite. On laisse au lecteur le soin de faire des expériences : ajouter une couche cachée ou deux, changer la taille, etc. Le lecteur qui voudrait calculer un réseau de neurones à partir des données du record du monde du 100m se rendra vite compte que ce n'est pas facile d'avoir du premier coup les bons hyperparamètres.

Bon OK, mais comment ça marche ?

La figure 5 décrit les calculs d'un réseau plus simple :

- En Input : un vecteur $X=\{x_1, x_2\}$.
- En Output : un vecteur $Y=\{y_1, y_2\}$.
- Une (seule) couche cachée (pour simplifier).
- Le lien entre deux couches est représenté par une matrice de poids, ici W et M (équations 2 et 3 de la figure 5). Le lecteur attentif aura remarqué que les éléments des matrices sont indicés de manière inhabituelle, c'est pour mieux formaliser la suite.
- W : relie la couche d'Input et la couche cachée.
- M : relie la couche cachée et la couche de Output.
- La fonction F est ce qu'on appelle la fonction d'activation, ici utilisée à la fois comme fonction scalaire et vectorielle pour simplifier les notations : on utilise souvent la fonction **sigmoid**, la fonction **ReLU** (voir les figures de l'article de A. Boucekif) ou encore la tangente hyperbolique, voir [9].
- Ainsi, on définit $Z=(z_1, z_2)$, le vecteur intermédiaire (comme indiqué sur la figure 5) comme le résultat de la fonction F appliquée à chacune des composantes du vecteur défini comme le produit de la matrice W par le vecteur X (équation 3 de la figure 5).
- On définit de même le vecteur de sortie Y comme le résultat de la fonction F appliquée à chacune des composantes du vecteur défini comme le produit de la matrice M par le vecteur Z (équation 4 de la figure 5).
- L'équation 6 résume les relations entre les entrées et les sorties et les matrices. Un réseau plus efficace serait obtenu en ajoutant des biais, voir l'article de A. Boucekif.

$$F(x) = \frac{1}{1 + e^{-x}} \quad (1)$$

Fig. 5 : Les équations d'un réseau de neurones avec deux entrées, deux sorties et une seule couche cachée.

$$M = \begin{bmatrix} m_{11} & m_{21} \\ m_{12} & m_{22} \end{bmatrix} \quad (2)$$

$$W = \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \end{bmatrix} \quad (3)$$

$$\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = F \left(W \times \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \right) = \begin{pmatrix} F(w_{11}x_1 + w_{21}x_2) \\ F(w_{12}x_1 + w_{22}x_2) \end{pmatrix} \quad (4)$$

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = F \left(M \times \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} \right) = \begin{pmatrix} F(m_{11}z_1 + m_{21}z_2) \\ F(m_{12}z_1 + m_{22}z_2) \end{pmatrix} \quad (5)$$



■ Le lecteur attentif aura compris que les sorties d'un tel réseau sont réelles et comprises entre 0 et 1. Cela peut s'interpréter comme une probabilité (la somme de toutes les sorties doit alors être égale à 1 modulo les erreurs dues à l'arithmétique flottante) ou une valeur. Dans le cas où on désire autre chose, un entier entre 0 et 9 par exemple, il faut ajouter en sortie un quantificateur ou un « catégoriseur ». En Keras c'est très simple grâce à **np_utils.to_categorical** par exemple. En Tensorflow, on peut utiliser la fonction « softmax », fonction d'activation qui retourne les probabilités que l'exemple appartienne à chaque classe prédéfinie [8].

Convergence

Nous avons donc la mécanique, elle est la même quel que soit le nombre de couches, quel que soit le nombre d'éléments en entrée et en sortie : si on a n composantes en entrée et m composantes en sortie, on part

du vecteur d'entrée $X = \{x_1, x_2, \dots, x_n\}$ et couche après couche on fait le produit matrice vecteur et on applique la fonction d'activation à chaque composante et ce, jusqu'à obtenir le vecteur de sortie $Y = \{y_1, y_2, \dots, y_m\}$.

Oui, mais ça c'est une fois que nous avons le réseau de neurones, c'est-à-dire les poids de chaque matrice reliant une couche et la suivante. Mais comment le choisir ? C'est là où intervient l'optimisation numérique, on ne le dira jamais assez : l'optimisation numérique est au cœur du ML.

Revenons à notre exemple :

- On commence par initialiser les différentes matrices (ici W et M), voir [9] pour les différentes possibilités. Si vous débutez, choisissez une initialisation purement aléatoire ou laissez faire votre framework.
- Pour le jeu de données d'entraînement on calcule la sortie, généralement au début on aura une valeur très différente de celles attendues, appelons $\hat{Y} = \{\hat{y}_1, \hat{y}_2\}$ cette sortie « erronée ».

- Ainsi, pour chaque donnée (x_1, x_2) du jeu d'entraînement X on définit l'erreur quadratique locale par : $(y_1 - \hat{y}_1)^2 + (y_2 - \hat{y}_2)^2$.

- On appelle J la somme de toutes ces erreurs, (nous avons supprimé les indices des poids pour simplifier) :

$$J(W, M) = \sum_{(x_1, x_2) \in X} (y_1 - \hat{y}_1)^2 + (y_2 - \hat{y}_2)^2$$

c'est la fonction coût : le « jeu » consiste à minimiser la fonction (dans le cas d'un réseau avec une seule couche cachée).

Il s'agit donc de trouver des matrices W et M qui minimisent l'erreur J ! C'est la partie délicate du calcul. Aux yeux de l'auteur, c'est à partir du moment où on a réussi à gérer ce problème sur des réseaux de neurones « profonds » (soit multicouches) sur de grands jeux de données que le Deep Learning est devenu un vrai outil industriel et non plus « académique ». Tout framework de ML contient des algorithmes de minimisation, basés sur des méthodes numériques dites de descente de gradient, le ML a d'ailleurs redonné une vigueur à ce domaine de recherche. On peut citer, le *gradient stochastique*, l'algorithme Adam, les variantes de type Nesterov ou encore LBFGS, pour *Limited Memory BFGS*. Le choix de l'algorithme peut être délicat et dépend fortement du type de réseau de neurones choisi, de la taille des différents paramètres, et des données évidemment. On renvoie à [9] et [11].

Une fois choisi l'algorithme de minimisation, pour toute méthode numérique de minimisation via les vecteurs gradients, il faut calculer le gradient ! Pour un réseau de neurones multicouches, cela peut devenir très vite difficile, voire impossible. Mais,

ouf, les frameworks sont capables de faire de la différentiation automatique (le lecteur attentif aura remarqué que le code précédent ne comporte rien sur les gradients, le framework se charge de tout). C'est une des autres clés de l'efficacité des frameworks actuels.

Une fois qu'on peut calculer le gradient, on peut itérer la méthode de minimisation choisie et faire converger le réseau itération après itération, en Deep Learning on parle d'époques (*epochs*). Propagez les modifications à travers tout le réseau se fait grâce à l'algorithme de rétropropagation *du gradient* (*Backpropagation*), la « châtaigne » évoquée dans l'article de ce hors-série sur YaDiff.

L'invention de cet algorithme est vraiment ce qui permet aux algorithmes calculant les différents poids d'un réseau de neurones multicouches de converger. Et il a déjà plus de 30 ans ! Nous n'en dirons pas plus, une bonne présentation de ces sujets se trouve dans [8,9,13].

UN EXEMPLE CLASSIQUE : LE JEU DE DONNÉES DU MNIST

Donnons un exemple complet pour convaincre les sceptiques (il y en a encore ?) que c'est devenu relativement facile, rapide et efficace. Pour nuancer un peu ces propos, disons que quand ça marche, c'est relativement facile, rapide et efficace. Le jeu (complet) de données MNIST est composé d'images de chiffres écrits à la main, donc quelquefois avec des ambiguïtés, il est très utilisé en ML, des milliers de sites en parlent. Il est composé de 60000 images d'entraînement et de 10000 images de tests [10]. La version Scikit contient 1797 images, il est facile de se procurer le jeu complet dans différents formats, voir [10].

Commençons par charger ce dont nous aurons besoin (pour alléger le texte nous ne donnons pas le résultat de tous les « print ») :

```
01 import numpy as np
02 from sklearn.datasets import load_digits
03 from sklearn import metrics
04 from sklearn.neural_network import MLPClassifier
05 data = load_digits()
06 print("Nombre d'échantillons : %d" % len(data.target))
07 print("Catégories en sortie: %s" % data.target_names)
08 digits = data.data
09 print("Vecteur de caractéristiques: %s" % digits)
10 labels = data.target
11 print("Labels en sortie: %s" % labels)
```

Pour mesurer la qualité des résultats
Pour construire un réseau de neurones
On charge le jeu de données
Dix catégories en sortie : de 0 à 9
On extrait les données images
Pour avoir une idée du format
On extrait les labels de chaque image

Comme dans ce jeu de données nous n'avons pas de données de tests, nous choisissons au hasard des images du jeu de données (ici 100-78=22% des images) :

```
12 X_entrain = np.random.rand(len(labels)) < 0.78
13 digits_entrain = digits[X_entrain]
14 labels_entrain = labels[X_entrain]
15 digits_test = digits[~X_entrain]
16 labels_test = labels[~X_entrain]
17 print("Nombre d'échantillons d'entraînement: ", digits_entrain.shape[0])
18 print("Nombre d'échantillons de test: ", digits_test.shape[0])
19 print("Dimension du vecteur de caractéristiques: ", digits_entrain.shape[1])
```

On choisit une partie des images pour tester
les échantillons choisis pour l'entraînement
leurs labels
les échantillons choisis pour le test
leurs labels

Lors de l'exécution, nous avons eu 1450 images d'entraînement et donc 347 images de test. Nous définissons ensuite le réseau de neurones (ligne 20, avec 9 couches cachées, juste pour voir, et « LBFGS » comme algorithme de minimisation) :

```
# On définit le réseau de neurones
20 mon_modele = MLPClassifier(solver='lbfgs', alpha=1e-5,
                             hidden_layer_sizes=(28, 2*28, 4*28, 8*28, 16*28, 8*28, 4*28, 2*28, 28), random_state=2)
# On l'entraîne avec le jeu de données d'entraînement
```

```

21 mon_modele.fit(digits_entrain, labels_entrain)
# On veut tester donc on calcule les "prédictions" sur le jeu de données de test
22 labels_pred = mon_modele.predict(digits_test)
# On affiche des données sur la qualité
23 print(metrics.classification_report(labels_test, labels_pred))
# On affiche la matrice de confusion [ce qu'on ne peut faire si le jeu de données est très grand]
24 print(metrics.confusion_matrix(labels_test, labels_pred))

```

La moyenne sur la précision obtenue est de 96 %, ce qui est excellent. La matrice de confusion obtenue est la suivante :

```

[[29  0  0  0  0  0  0  0  0  0]
 [ 0 29  0  0  2  0  0  0  1  0]
 [ 0  0 39  1  0  0  0  0  0  0]
 [ 0  0  0 35  0  0  0  0  0  0]
 [ 0  0  0  0 31  0  0  0  0  0]
 [ 0  0  0  0  0 37  0  0  0  0]
 [ 0  0  0  0  0  0 41  0  0  0]
 [ 0  2  0  0  0  0  0 26  0  1]
 [ 0  0  0  1  0  1  0  0 33  0]
 [ 0  0  0  0  0  0  0  0  1 37]]

```

Cette matrice (quand on peut l'afficher) sert à mesurer la qualité d'un système de clusterisation, elle se lit de la manière suivante :

- chaque élément diagonal indique combien d'images de test ont été correctement « reconnues » ;
- chaque élément hors diagonal correspond à des erreurs, ainsi sur la 1ère ligne, qui correspond au chiffre '0', aucune autre image des chiffres allant de 1 à 9 n'a été mal reconnue. En revanche, sur la 2ème ligne, qui correspond au chiffre '1', 2 images de test du chiffre '4' et une image de test du chiffre '8' ont été reconnues comme étant un '1'.

Le lecteur vérifiera que sur 347 images de test, seules 8 images n'ont pas été correctement reconnues. Un petit exercice : essayer, en jouant sur les hyperparamètres de faire mieux que 96 %.

Remarque : on peut « inverser » un tel réseau de neurones artificiels. Du coup, on obtient un réseau qui prend en entrée une catégorie, de '0' à '9' et renvoie ... (vous suivez ?) une image ! Voir **[13]** pour des exemples en Keras.

VOUS AVEZ UN PROJET DE ML EN TÊTE ? QUELQUES CONSEILS

Imaginons que vous ayez un projet de ML en tête. Résumons donc la méthodologie classique en quelques mots :

- À partir des données (préparées, voir la suite) dont on dispose, on choisit un modèle et on « apprend » les paramètres de ce modèle, on entraîne l'algorithme (en espérant qu'il converge) sur le jeu de données choisi.
- À partir du modèle calculé, on peut alors « prédire ».

Alors, à quoi devez-vous vous attendre comme problèmes à résoudre ? Citons quelques-uns de ces problèmes :

- Vous devrez récupérer des données. Prenons comme exemple la clusterisation d'un grand nombre de malware non labellisés. Vous pouvez commencer avec « theZoo », petit, mais propre.
- Vous devrez les nettoyer (enlever ou gérer les données incomplètes, enlever les données redondantes, etc.), par exemple vous pouvez choisir de supprimer les différentes copies d'un malware qui ont juste un nom différent, si vous en avez.
- Vous devrez *normaliser* vos données, par exemple pour un jeu de données de malware, si vous calculez le nombre de sections et l'entropie de chaque section, si vous avez un malware à plus de 50 sections, cette valeur risque de « dévorer » les autres.
- Vous devrez aussi les « explorer », *i.e.* commencer à les comprendre. On pourrait presque dire qu'il vous faudra les apprivoiser.

- Vous devrez choisir un vecteur de caractéristiques (très délicat), la qualité et la pertinence de ce vecteur vont fortement influencer la qualité de vos résultats. Pour un malware, on peut commencer avec : sa taille (en octets), le nombre de sections (éventuellement leur taille), l'entropie du fichier, l'entropie de chaque section (éventuellement), le nombre de symboles de la table d'import, le nombre de symboles de la table d'export, le nombre de ressources, etc.
- Vous devrez choisir une distance (pour les algorithmes comme K-means, DBSCAN, KNN et pas mal d'autres), on peut commencer par choisir la distance euclidienne classique. Attention : si vous avez bien une distance entre les vecteurs de caractéristiques, cela ne vous donnera qu'une similarité entre les malware.
- Vous devrez maîtriser le « passage à l'échelle » (*scalability*) : c'est-à-dire maîtriser la complexité pratique des algorithmes. Un exemple : un algorithme de clusterisation comme WARD ou DBSCAN ne passe pas à l'échelle, en clair : si votre ensemble de données est très grand (une taille de quelques centaines de millions) vous ne pourrez utiliser ni Ward ni DBSCAN. Dans l'article sur les graphes géants creux, on appelle ce problème le « mur du Big Data ». Si vous êtes malchanceux, il vous faudra coder votre propre algorithme, voir [12].
- Vous devrez choisir (ou calculer) les hyperparamètres d'un algorithme : ce sont les paramètres non optimisés par l'algorithme. Ils doivent donc être soit choisis par l'utilisateur, soit calculés de manière différente. Cela peut être très complexe, et très coûteux. Par exemple, pour un

réseau de neurones : combien de couches cachées faut-il choisir ? Et pour chaque couche, quelle taille ? Pour l'algorithme des moyennes mobiles (k-means) : quelle est la valeur de k, le nombre de clusters ? Le lecteur intéressé pourra consulter la documentation Scikit-Learn de RandomizedSearchCV et de GridSearchCV.

- Vous devrez probablement aussi réviser (un peu ou beaucoup) vos maths : statistiques et probabilités, algèbre linéaire, calcul différentiel, algorithmes numériques de gradient pour la minimisation, etc.
- Et, *last but not least*, vous devrez choisir vos algorithmes, en fonction de vos données, de vos attentes, de votre puissance CPU, etc. Bon là, l'auteur vous laisse faire !

Ah oui : évidemment vous devrez avoir une petite idée de ce que vous voulez mettre en évidence, il n'y a pas de magie. Et n'oubliez pas : le ML est essentiellement un processus itératif, vous devrez revenir sur vos choix en fonction des résultats préliminaires, refaire des tests, revenir sur vos hyperparamètres, etc.

CONCLUSION

Cet article n'a fait qu'effleurer le vaste sujet du ML. Il fallait faire des choix : nous n'avons pas abordé la réduction de dimension (ni la malédiction de la dimension), via l'ACP ou via les autoencodeurs (voir le chapitre sur ce sujet dans [8]), les algorithmes de type SVM, le KNN, les *random forests* (qui généralisent les arbres de décision), les réseaux antagonistes génératifs, les machines de Boltzman, ni les techniques de noyaux (*kernel trick*), ni le « *machine ensemble* » (très intéressant, permet à partir de

plusieurs « mauvais classifieurs » d'en construire un meilleur) et rien sur la disparition et l'explosion des gradients. Les réseaux de neurones récurrents (y compris les LSTM) sont présentés dans un autre article.

Il resterait aussi à présenter l'ingénierie de caractéristiques (traduction libre de *features engineering*) qu'on peut définir comme l'ensemble des procédés qui, à partir des connaissances du domaine, définit les « bonnes » caractéristiques qui vont permettre aux algorithmes du ML de fonctionner. Autant dire que c'est plus un artisanat qu'un art. Wikipédia dit : « *Feature engineering is the process of using domain knowledge of the data to create features that make machine learning algorithms work.* ».

Une grande partie des articles de ce hors-série complète ce tour d'horizon et le lecteur intéressé trouvera dans les références de quoi aller (beaucoup) plus loin.

Alors, le ML comme arme ultime contre les « bad guys » ? Non. Comme toute technique, le ML n'a rien de magique. Enfin, on peut parier que celui-ci sera vite adopté par ces « bad guys », comme ce fut le cas de la cryptographie, mais ça, c'est une autre histoire... █

REMERCIEMENTS

Merci à mes collègues du LSE, Abdessalam, Reda, Alexandre et Marc, à Marwan (ex-collègue), à Sébastien (pour sa relecture) et à l'équipe de MISC pour avoir accepté l'idée de ce hors-série.



Les références de cet article sont disponibles sur : <https://www.miscmag.com/>.

ACTUELLEMENT DISPONIBLE

LINUX PRATIQUE HORS-SÉRIE N°43

LES HORS-SÉRIES CHANGENT DE FORMULE !



NE LE MANQUEZ PAS
CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :
<https://www.ed-diamond.com>





ML & SÉCURITÉ : LES OPPORTUNITÉS

Raphaël DE CORMIS et Virginie GALINDO

Machine Learning, l'ingrédient indispensable à la bonne recette de l'innovation ? Revoyons ensemble quels sont les vrais domaines où ce nouvel outil transformera réellement le paysage de la sécurité.

INTRODUCTION

Le *Machine learning*, souvent confondu avec son domaine plus large qu'est l'Intelligence Artificielle reste le sujet qui aura fait couler le plus d'encre sur les deux dernières années. Révolutionnaire, disruptif, outil de transformation numérique, les qualificatifs n'ont pas manqué. Certes, le *Machine Learning*, est vecteur de transformation, sans aucun doute, il donne accès à un certain nombre de services nouveaux, il en améliore d'autres, il est un excellent

prétexte à explorer de nouveaux domaines et résoudre des problèmes dont les masses de données nous perdent. Néanmoins, au moment de comprendre quel est l'impact réel qu'aura cette technologie, il faut garder raison, et se demander quels sont les ingrédients nécessaires pour qu'un vrai cas d'usage bénéficie de cette technologie. Dans cet article, nous explorerons les domaines de la sécurité qui pourront bénéficier de l'arrivée de cette technologie, nous adressons également quelques domaines qui nous paraissent importants de surveiller.

À LA CONQUÊTE DE LA DONNÉE

Le Machine Learning, on le sait est un outil annoncé comme stratégique pour le classement de données, la prédiction de scénarios, et la recommandation. Sur la base de données issue d'une activité particulière (les flux de marchandises d'un magasin, le panier d'achat des clients, le chiffre d'affaires du magasin, la rotation des employés, la pluie et le beau temps), le logiciel, est capable de détecter des schémas de corrélation ou des schémas répétitifs et donc de faire des prédictions de vente, et alimenter une stratégie de gestion de stock. Cet outil n'est pas magique, il repose sur l'usage de formules mathématiques et de logiciels, qui sont confrontés à des jeux de données (souvent appelés *data set*). Ces données doivent être nettoyées, complétées, le logiciel est ensuite entraîné, puis, une fois calibré correctement, il pourra prédire (un chiffre d'affaires, un pic de clientèle), et aider à la décision (le réassort, la gestion du personnel...).

On l'aura compris, dans ce scénario, la donnée clé, c'est... la

donnée. Puisque nous nous concentrons dans cet article sur les aspects de sécurité, une façon d'aborder le sujet est de regarder le monde à travers le prisme des données. Là où il y aura des données, il y aura la possibilité de créer des services de Machine Learning. Et ces données, nous le savons, peuvent être dans trois états. C'est-à-dire, soit gentiment stockées sur un serveur, attendant une mort ou une exploitation certaine, en train d'être activement utilisées par une application ou alors en train d'être échangées entre deux personnes, serveurs, machines... C'est le fameux « *data at rest, at run, et in transit* ». Maintenant que nous avons posé les bases de la conquête de la donnée, et des différents états qui la caractérisent, partons vraiment à la recherche des données.

Les données business et 'méta'

Pour des raisons de simplicité, nous nous focaliserons sur le scénario d'une entreprise. Que se passe-t-il dans une entreprise ? Les données sont générées de deux façons. L'activité de l'entreprise produit de la donnée. Les applications métiers embarquées dans un système d'information, stockent des informations sur l'état de l'entreprise, génèrent des événements, font circuler des données (mails, fichiers de production, résultats financiers, factures, payes...). Les éléments qui permettent de stocker ces données sont des serveurs privés ou des serveurs publics (laissons-nous aller à l'anglicisme *Public cloud*). Par ailleurs, le système informatique et le cloud public d'une entreprise sont à taille variable, dans le sens où des utilisateurs se connectent sur le réseau via des machines, puis dis-

paraissent, des machines virtuelles sont instanciées, puis éteintes. Ces données d'activités du SI et du cloud, sont souvent capturées de manière ordonnée dans des logs. Cette masse de données, que nous pouvons appeler « méta » puisqu'elles sont caractéristiques de l'activité du SI et du cloud, a des formats souvent simples et standards. Ces logs souvent identifiés par l'entreprise comme importants, sont classés, accessibles moyennant une protection pour les plus sensibles d'entre eux. Nous avons donc ici, une belle source de données produites par l'activité d'une entreprise. Il y a sans doute matière à utiliser la magie du Machine Learning pour adresser cette masse grandissante de données, qui atteint nos limites de traitement et d'analyse actuelle, et continuer à fournir un service de sécurité. RSSI, restez à l'écoute !

Les données éparées ou non contrôlées

Les applications, le SI et le cloud n'ont pas l'exclusivité de la génération de données dans une activité d'entreprise. Les utilisateurs et les machines qu'ils utilisent, PC portable, smartphone, grille-pain connecté, ont aussi leur mot à dire. En effet, lors d'un accès à distance au réseau de l'entreprise, via un PC portable, un utilisateur crée un contexte de connexion, lié à sa position, le navigateur qu'il utilise, l'heure de connexion. C'est une donnée qui n'est pas toujours proprement capturée, qui est éparée, mais qui pourtant est, elle aussi, caractéristique de l'activité de l'entreprise, ou plus exactement de l'employé pour le compte de l'entreprise. Le même genre de donnée éparée est également pro-

duit par les utilisateurs d'un service délivré par cette même entreprise. Un client qui se connecte laisse un certain nombre de traces, dans un contexte qui lui est propre. Ces données générées par les utilisateurs ou les employés sont nombreuses, elles ont en général des formats divers, elles sont générées sur des canaux divers, et ont la caractéristique de ne pas être sous le contrôle de l'entreprise ou le fournisseur de service. À cause de leur caractère périphérique, de leur diversité, et parfois de leur valeur perçue comme faible (car non essentielles à la pérennité de l'activité), ces données sont rarement capturées et exploitées - on n'a pas dit jamais. Cependant, elles représentent une source intéressante d'information aussi bien sur des questions de sécurité, que sur des questions d'usage et de consommation de service. On l'aura compris, cette source de données peut avoir une valeur dans le cas d'un traitement de Machine Learning, puisqu'elle permettrait d'étendre les possibilités de modélisation d'une activité. Cependant, pour des raisons pratiques, leur collecte, leur nettoyage et leur utilisation restent à la marge, dans la plupart des cas d'usage que nous avons repérés.

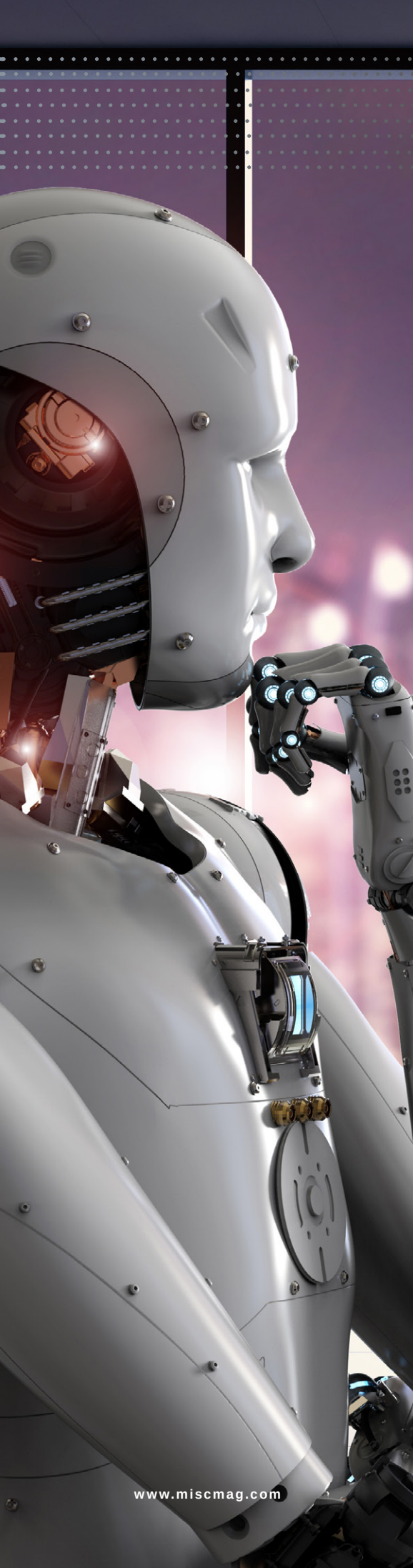
Et maintenant qu'on sait où se cache la donnée, que fait-on ?

Pour ceux qui n'auraient pas suivi, le Machine Learning requiert d'avoir de la donnée. Cette donnée doit être fiable, fraîche, vraie et disponible en quantité. En effet, la plupart du temps, il est nécessaire d'avoir un jeu de données d'entraînement, qui permettra au logiciel de Machine Learning, de s'entraîner, de détecter

des motifs récurrents (*pattern*, quoi), de trouver des corrélations. Puis un jeu de données est utilisé afin de tester la qualité du modèle. Par ailleurs, ces données passent souvent par une étape de nettoyage, pour que les formats soient utilisables, les catégories de données bien qualifiées... Saluons donc au passage le travail des data scientists, qui passent leur journée à nettoyer, qualifier, trier, ajuster ces jeux de données, et qui restent les magiciens du Machine Learning. Une fois ces données prêtes, le framework de Machine Learning choisi, que peut-on espérer de ces activités de Machine Learning ? Plusieurs choses, en fonction de la maturité. Un service de Machine Learning peut faire du classement (les photos de Tata Huguette*, avec les photos de Tata Huguette dans le dossier numérique qui va bien), de la détection d'anomalie (Paul* se rend au travail à 8h04 alors qu'on lui reconnaît une tendance à arriver vers 9h45), ou de la prédiction (le service de paiement opéré par Louis* va tomber en rade bientôt, bientôt). Classement, détection d'anomalie, prédiction : ce catalogue de services paraît assez magique, et utilisable pour à peu près tout et n'importe quel service. Outils de séduction technologique, le Machine Learning a un pouvoir d'attraction fort auprès des investisseurs, analystes marché, et stratèges, depuis quelques années. La tentation de jouer la carte du *AI-washing* est grande. Qui n'a pas entendu dire que la start-up TrucTruc fait du Machine Learning, alors qu'en grattant un peu, on découvre que le service comporte simplement une part d'automatisation et de l'application de règles prédéfinies. Point d'apprentissage, ni de prédiction, ici, mais juste un système bien décrit et déterministe. Autre travers. L'usage déclaré de Machine

Learning, est parfois remplacé par un système humain, qui prétend être un chatbot, ou un logiciel d'annotation (ou tag) de photo et contenu visuels. Ce business s'inscrit dans la succession des fermes à clics, qui génèrent du trafic sur les réseaux sociaux. Dernière mise en garde. On confond souvent Machine Learning et Big Data. Rappelons ici, que le Big Data sous-entend que les données sont concentrées, visualisées et analysées, certes, mais que les résultats sont obtenus manuellement.

Par ailleurs, l'usage du Machine Learning a un coût - quelle surprise ! Déployer du Machine Learning, requiert d'avoir des data scientists qualifiés, une infrastructure pour stocker les données de qualité, un accès à des logiciels performants, et/ou un budget important pour consommer les services prêts sur étagère. On peut effectivement jouer à un prix raisonnable avec les APIs de Machine Learning mis à disposition par des fournisseurs de services des géants du cloud ou de nombreuses startups. Les services classiques de type reconnaissance d'images fournissent un service suffisant pour des cas d'usage simples (classement de photos d'anniversaire en album par exemple). Mais l'investissement devient tout de suite plus important quand il s'agit de faire du travail plus sur mesure, dédié à un cas d'usage innovant ou très marqué par le métier (classement des champignons vénéneux par exemple). Évidemment, s'il y a un investissement, il est essentiel de pouvoir suivre le retour sur investissement (point #ROI), et donc d'avoir des pistes solides sur ce que le Machine Learning apporte vraiment à un service déjà en place, par exemple. En résumé, il est donc important, lorsqu'on observe les opportunités



associées au Machine Learning, de ne pas se laisser aveugler par un tourbillon de buzz words, et de bien qualifier quels sont les usages pertinents et réels du Machine Learning.

(*les prénoms ont été changés)

Petit exercice d'application : les scénarios d'attaque transformés par le Machine Learning

Bien. Te voici, lecteur, armé de notions de base pour comprendre quand et comment le Machine Learning pourrait rendre le monde meilleur. Nous sommes chez MISC, il s'agit donc de parler sécurité. Nous te proposons donc une première façon de mettre en pratique les logiques de l'intersection utile et utilisable du Machine Learning et de la cybersécurité. Si l'on observe la liste des attaques classiques, basiques, nous pouvons d'ores et déjà dire que toutes ne sont pas éligibles à être mieux détectées ou contrebalancées avec du Machine Learning d'aujourd'hui. Nous donnons ici quelques exemples - attention, ils sont non exhaustifs, et le coût de leur mise en place n'est pas discuté, ici. Il existe 3 attaques pour lesquelles le Machine Learning peut être un atout : l'attaque directe, lorsqu'un attaquant a accès physiquement à une machine, lui permettant d'explorer, copier et altérer des données, le *spoofing*, lorsqu'un attaquant endosse l'identité d'une autre personne, via un e-mail, une empreinte digitale, une adresse IP ou MAC et l'escalade de privilèges, lorsqu'un attaquant est capable, sans autorisation, de s'attribuer un niveau d'accès ou un privilège supérieur. Le facteur commun entre ces attaques, est que l'attaquant une fois dans la place,

aura sans doute un comportement différent d'un utilisateur normal. Une solution à base de Machine Learning pourrait scanner les logs d'activité, et détecter des changements d'habitude dans cette activité, et ainsi détecter une attaque directe, une attaque de *spoofing* ou une escalade de privilèges. Toute la mise en place de cette détection réside dans la capacité à capter les données du système que l'on souhaite protéger. La difficulté dépendra de la diversité des points composants le réseau, du nombre d'utilisateurs, de la diversité des points d'accès, ainsi que de la capacité à détecter des schémas d'utilisation dits « normaux ».

Une autre série de prévention gagne en efficacité, et se retrouve comme argument commercial fort pour les fournisseurs de solution de messagerie en mode SaaS. Il s'agit du *phishing*, lorsqu'un attaquant récupère les informations personnelles et sensibles d'un utilisateur, par le biais d'un site web malicieux, sur lequel l'utilisateur se connectera, ou remplira un questionnaire. Une des raisons pour lesquelles les outils anti-*phishing* peuvent être considérés comme matures et commercialisables, réside dans le fait que les fournisseurs de messagerie, tels qu'Office 365 ou GSuite, brassent une quantité incroyable de mails et messages, et sont donc à même de détecter des suggestions de sites web malfamés. Ici, c'est la quantité de données qui fera la différence sur l'efficacité réelle de la solution. En Machine Learning, le précepte qui dit qu'on ne prête qu'aux riches s'applique assez bien.

Une des grandes batailles des internets reste encore la chasse aux virus et applications malveillantes. Il se trouve qu'une nouvelle famille de virus écume nos systèmes d'informa-

tions, nos machines, ce sont les « Malware multi-vecteurs et polymorphiques ». Cette nouvelle classe de virus est réputée pour sa capacité à mettre en place une combinaison d'attaques, et qui plus est, change de forme au cours de sa propagation. Dans ce cas, la reconnaissance de la simple signature du virus (une empreinte représentative de sa taille, ses fonctions...) ne suffit plus. Ici encore, le Machine Learning peut aider, basé sur un catalogue de virus connus, un suivi des morphings que le virus s'applique.

C'est bien beau tout ça, mais en pratique ?

En pratique, il existe plusieurs domaines et applications réelles qui bénéficient immédiatement et de manière réelle de la capacité du Machine Learning. Revoyons ceux que les auteurs ont trouvés pertinents. Disclaimer, bien que travaillant pour une entreprise fournissant des solutions de sécurité numérique, ayant investi dans le Machine Learning, les auteurs se sont appuyés sur leur veille objective et minutieuse pour détecter les domaines dynamiques, face au Machine Learning. Ils se sont notamment intéressés à tous les domaines faisant preuve d'annonces importantes d'investissements majeurs, ou de reconnaissance de prix. Voici, en quelques paragraphes les tendances qu'ils ont pu noter.

LES DOMAINES DYNAMIQUES, BÉNÉFICIAIRES RÉELLEMENT DU MACHINE LEARNING

Reconnaître les personnes et les machines

» Le boom de la biométrie dans l'identification

Le vol d'identité reste un vecteur fort d'attaques réussies. Depuis quelques années, la biométrie est venue en renfort pour résoudre cette question de l'identité, en facilitant l'accès à un deuxième facteur d'authentification de type « something that you are » dans des scénarios de reconnaissance d'identité. Le fait de présenter une empreinte digitale ou une photo de son visage (dit, reconnaissance faciale) vient compléter les classiques login/password ou autre premier facteur d'authentification. Néanmoins, la comparaison de deux empreintes digitales ou de deux visages reste une opération qui prend du temps, et les fournisseurs de solutions doivent parfois sacrifier un peu de l'exactitude de leurs résultats, au profit d'une réponse plus rapide, un accès au service plus immédiat. L'arrivée du Machine Learning ouvre de nouvelles perspectives sur la capacité d'effectuer ces comparaisons de données biométriques avec plus de précision et de rapidité. Un logiciel bien entraîné manipule bien plus facilement la signature biométrique qu'un logiciel traditionnel, il possède également une capacité à reconnaître les situations d'incident avec plus de précision – les fameux FRR pour *False Rejection Rate*, le taux d'identification rejetée par erreur, et FAR pour *False Acceptation Rate*, le taux d'identification acceptée par erreur. Deux applications immédiates se sont emparées de cette vertu. Le premier est le smartphone et la reconnaissance faciale à distance. Par exemple, on retrouve



dans les derniers modèles de smartphone une reconnaissance faciale augmentée grâce au Machine Learning (Face ID chez Apple pour citer les plus connus, et des fonctionnalités similaires dans les prochains Flagship S9 de Samsung et Pixel 3 de Google). Le second est la vérification d'identité dans une foule, par exemple, au cours d'un événement sportif. La suprématie du matching (action de comparer deux empreintes biométriques) se fait ressentir dans ce cas là, dit « 1 to many », ce cas d'identification requiert de comparer une empreinte digitale à toutes celles contenues dans une base complète d'empreintes. Dans ce cas-là, il s'agit de repérer l'empreinte digitale ou la photo jumelle le plus vite possible et sans erreur.

» Le silence de l'authentification

La vérification d'identité, on l'a vu, peut être enrichie à l'aide du Machine Learning. Il existe un élément qui participe aussi à l'accès au service, il s'agit de l'authentification. Dans ce cas, l'identité de l'utilisateur n'est pas particulièrement requise, mais il s'agit de donner accès au bon utilisateur accès à un service. Jusqu'ici, le login/password était le moyen d'accès roi. Il est complété depuis quelque temps par ce que l'on appelle une authentification silencieuse. De quoi s'agit-il ? Il s'agit de créer un profil de l'utilisateur, sur la base des traces numériques qu'il ou elle laisse. Cette authentification silencieuse peut ainsi s'appuyer sur le modèle du téléphone utilisé, le réseau Wifi utilisé, la localisation, la configuration du navigateur. Bref, il existe une infinité d'empreintes numériques qui pourraient participer à la création de ce profil. L'adoption de ce type d'authentification est motivée par le fait que cette authentification est silencieuse (on peut aimer, ou pas), et continue (elle peut être jouée silencieusement à n'importe quel moment). Il existe un champion incontestable de ces profils virtuels, ThreatMetrix, qui possède la plus grande base de données d'informations relatives aux utilisateurs des internets. La liste des comptes créés, quelques informations de réputation, mais également des informations relatives au comportement des utilisateurs (le mouvement de la souris, la façon de taper sur un clavier. Threatmetrix propose ses services en mode API, sur la base de Machine Learning, pour pouvoir formuler des diagnostics de risque de manière efficace et rapide. Un autre acteur se dégage également, il s'agit de AuthZero, le champion de l'authentification et de l'intégration facile et rapide qui permet de mixer cas d'usage (B2B, B2C, entreprise), stack technologique, modes de déploiement (public, privé, SaaS) et personnalisation des règles (*connection home office*, en déplacement, pour le compte d'un collègue). C'est sur cette dernière dimension que la détection d'anomalie intervient.

L'intégrité des infrastructures

» La détection des scénarios bizarres

La détection d'anomalie est en effet l'un des cas d'usage les plus communs en Machine Learning, appliquée à la cybersécurité, il s'agit d'analyser les réseaux ou les accès en temps réel et de révéler les activités suspectes ou tout du moins hors du commun :

- transfert de gigas données d'un serveur d'un serveur à un compte perso Dropbox ;

- tentative d'accès systématique sur des serveurs cloud via différents ports de connexions et différentes identités ;
- téléchargement d'exécutable de sites douteux.

Les données d'activités sont collectées à l'aide de connecteurs au SOC (*Security Operations Center*) déjà en place dans l'entreprise pour bénéficier de la centralisation et la normalisation des logs applicatifs et d'infrastructure. Par ailleurs, les grandes plateformes SaaS (de CRM, de bureautique, de stockage) offrent des APIs qui permettent de récupérer leurs journaux d'activités.

La différence entre la probabilité d'un fonctionnement nominal (estimation Bayésienne) et la réalité permet de déclencher une alerte dont le niveau de criticité s'ajuste en temps réel.

Ces alertes sont affichées dans un panneau de contrôle, le plus souvent graphique, avec les indicateurs d'activités ou peuvent être envoyées sur différents canaux de communication.

Si la détection d'attaques (comme événement anormal de l'utilisation de l'IT de l'entreprise) peut être automatisée à l'aide du Machine Learning, la prise de décision n'est pas encore complètement autonome. Jusqu'ici la plupart des solutions proposent le déclenchement d'actions sur règle à configurer manuellement.

Il est intéressant de noter que la plupart de ces solutions sont fournies en mode SaaS afin d'entraîner les modèles Machine Learning en bénéficiant des puissances de calcul du cloud. La promesse d'un déploiement rapide et facile est aussi très largement mise en avant. On notera que le grand champion de ce créneau, la boîte qui monte sur ce domaine est DarkTrace.

Par ailleurs, les grands fournisseurs de solutions cloud commencent à offrir offrent ce type de services avec des niveaux d'intégration de plus en plus poussés. Leur vue transversale sur l'ensemble des environnements client qu'ils hébergent et les quantités de logs qui y sont produites leur permettent de construire des modèles très aboutis. On peut imaginer qu'à terme ces options seront incluses par défaut avec le traitement des logs (et au même prix).

» Attaques de l'intérieur, de l'extérieur

La protection des points d'accès (originellement des postes de travail, plus récemment des mobiles et à terme de tous les objets des réseaux IoT) est un autre domaine dans lequel émergent de nouveaux acteurs dopant leur solution au Machine Learning.

Là encore, les logiciels sont disponibles *as-a-service* avec un client léger qui transmet les données au cloud pour être traitées massivement. Contrairement aux antivirus classiques, la signature des binaires est obtenue à l'aide d'un réseau de



neurones qui facilite la recherche et la comparaison tout en permettant la détection des mutations des virus polymorphes. Arrêtons-nous un instant pour comprendre comment et pourquoi le Machine Learning à base de réseaux de neurones fonctionne particulièrement bien ici. Mettons que nous soyons à la recherche de photos de lapins. Une solution traditionnelle de classification de données travaillera sur la base d'un hash d'une photo représentant un lapin, *hash* obtenu à partir des pixels de la photo de lapin. Le réseau de neurones, travaille lui, sur un vecteur de sortie, une valeur, qui représentera toujours la forme lapin quelle que soit la qualité, la forme du lapin. C'est cette capacité, qui permet au réseau de neurones de détecter avec fiabilité et rapidité des choses, dans des situations de reconnaissance. Le lecteur qui cherchera une référence fournissant une technologie dans ce domaine pourra se tourner vers l'entreprise californienne CrowdStrike.

Ces solutions sont souvent accompagnées de services de recherche de failles et de veille sécuritaire qui sont encore largement effectuées par des équipes humaines et de hacking

éthique. La restitution des informations se fait classiquement par un tableau de bord analytique avec des fonctionnalités de recherche donnant une vue sur la santé du système d'information.

» Attaques des interfaces

De même, la prévention des attaques automatisées sur les sites web (mais aussi sur les applications mobiles ou les APIs ouvertes) bénéficie des avancées du Machine Learning. L'objectif majeur est de faire la différence entre un humain qui naviguerait en toute bonne foi et un bot malicieux qui tenterait de forcer un compte, de perturber les stocks d'un site d'e-commerce, d'aspirer les données d'un site, de tester des numéros de cartes bancaires volés ou de frauder les campagnes webmarketing. Le croisement de plus d'une centaine de dimensions concernant le réseau, les applications et les utilisateurs ainsi que le fingerprinting du point d'accès permet de partitionner les humains et les robots (populairement appelé le clustering). L'entreprise PerimeterX s'impose comme la petite boîte qui monte, qui monte (et qui lève des millions) sur le sujet.

Par contre, la prise de décision face à une attaque est présentée comme intelligente alors qu'en réalité elle est basée sur des scores et des règles de seuil éditées manuellement.

LES PROCHAINES BATAILLES...

Nous avons abordé jusque-là les cas pratiques, et les domaines pour lesquels le Machine Learning était déjà à l'œuvre. Ces cas d'usage vont continuer à s'étendre. Et de notre point de vue, certaines batailles s'annoncent déjà intéressantes. Voici

quelques sujets sur lesquels toute personne intéressée par l'intersection entre sécurité et Machine Learning devra regarder avec attention.

Un cran plus loin, l'inspection des paquets

Certaines solutions de *Deep Packet Inspection* communiquent sur de nouveaux cas d'usage en sécurité à l'aide du Machine Learning. À partir de l'analyse de flux en temps réel, basée sur la classification des applications, les métadonnées de trafic et la détection des protocoles, ces solutions permettent la détection de malware entrants et de données sensibles sortantes. Mais les outils analytiques pour créer automatiquement les règles de détection d'intrusion ne semblent pas encore développés, cela ne saurait tarder...

Les attaques inattendues

La littérature est peu bavarde sur les attaques menées grâce à du Machine Learning. Évidemment, cela ne veut pas dire que ces attaques n'existent pas, il s'agit juste d'une question de temps, avant que nous en entendions parler. Néanmoins, le jeu de chat et de souris de l'attaque et de la défense est également un jeu d'anticipation. On a vu du côté Symantec, récemment, une annonce concernant l'*adversarial Machine learning* [1]. Le principe est assez simple. Deux logiciels de Machine Learning se défient en attaque et défense, l'un cherchant à créer une attaque destinée à ne pas être connue par l'autre. On peut donc imaginer que le Machine Learning entrera bientôt en scène.

Polymorphic viruses

On l'a dit, les virus polymorphiques sont un domaine où le Machine Learning pourra rendre des services, pour détecter les formes mutées des virus. IBM a récemment annoncé une expérience permettant de générer des virus, aux formes mutantes, qui se transformeraient selon un schéma indétectable par les logiciels entraînés de Machine Learning [2]. Ici encore, machine contre machine, on essaie d'anticiper ce que pourrait faire un adversaire pour contrer les défenses mises en place, à base de Machine Learning. La souris joue donc au chat.

Massive social engineering

De la même manière que dans les scénarios de préventions, le Machine Learning est utilisé pour extraire les signaux précurseurs d'une attaque à partir de la masse de renseignements ouverts (*Threat hunting*), on peut supposer que pour construire de nouvelles attaques, les mêmes méthodes peuvent servir à réaliser des scénarios de social engineering très poussés.

Des sociétés proposent en effet d'identifier des modèles d'attaquants à partir de leurs tactiques, techniques et procédures (*Tactics Techniques Procedures / TTPs*), de leurs cibles habituelles ou de leur zone d'opération.

Symétriquement, un attaquant pourrait flouer sa victime plus facilement en corrélant les données la concernant, en mimant les modes d'interaction de ses proches ou de ses collègues, en s'immisçant dans ses habitudes. C'est ce que les experts en social engineering font déjà [3]. Alors qu'il est très difficile de dis-

tinguer une voie de synthèse d'une voix humaine, il sera aussi compliqué de reconnaître un attaquant qui utilisera le Machine Learning pour l'assister.

La question de la vie privée dans le contexte de collecte de données

La question de la donnée vient de subir un tournant considérable, du moins en Europe avec le Règlement Général sur la Protection de Données Personnelles. Les effets se font également sentir outre Atlantique et d'autres pays, comme le Japon et l'Inde, suivent le même chemin, d'une meilleure régulation de la collecte, l'usage de la donnée et le transfert des données. Pour le moment, l'usage de la donnée, dans le cadre du Machine Learning est encadré grâce à des contrats, s'il s'agit de B2B ou de conditions d'usages de services, s'il s'agit de B2C. La rétention des données n'est pas essentielle dans le processus d'entraînement des données, il serait donc facile de se conformer à la loi, mais la collecte doit être continue, si possible, afin de s'assurer que les patterns détectés restent valides, dans le temps. Un autre phénomène pourrait venir peser sur le sujet de l'opt-in et du contrôle des données par les utilisateurs, il s'agit de la mise en commun des jeux de données. En effet, les jeux de données, nettoyés, contextualisés peuvent représenter une valeur non négligeable. Certains jeux de données sont déjà mis à disposition sur des plateformes, de partage entre data scientists. Il se pourrait qu'un jour, nos données collectées se retrouvent entre les mains de data scientists pour entraîner des modèles dont nous n'avons pas

connaissance. Un meilleur suivi de la traçabilité des jeux de données pourrait être une prochaine bataille de la vie privée des utilisateurs de service.

L'autre volet de la vie privée qui pourrait se trouver ébréché par ces avancées sur le sujet Machine Learning est celui de l'observation de l'activité des utilisateurs sur les systèmes. Les employés utilisant un système d'information sous le contrôle d'un logiciel de Machine Learning ont-ils vraiment envie de voir leurs contenus scrutés par les méthodologies de *Deep Packet Inspection* ? À l'heure du mélange des outils entre vie privée et vie professionnelle, cette question doit sans doute être traitée avec une grande transparence.

CONCLUSION

Dans cet article, nous avons revu les éléments clé du Machine Learning, et la façon dont cet outil pourrait venir modifier le paysage de la cybersécurité, du côté de la défense. Au-delà des aspects théoriques et pratique, il est intéressant de noter les flux d'argent qui circulent autour de cette question de Machine Learning et intelligence artificielle.

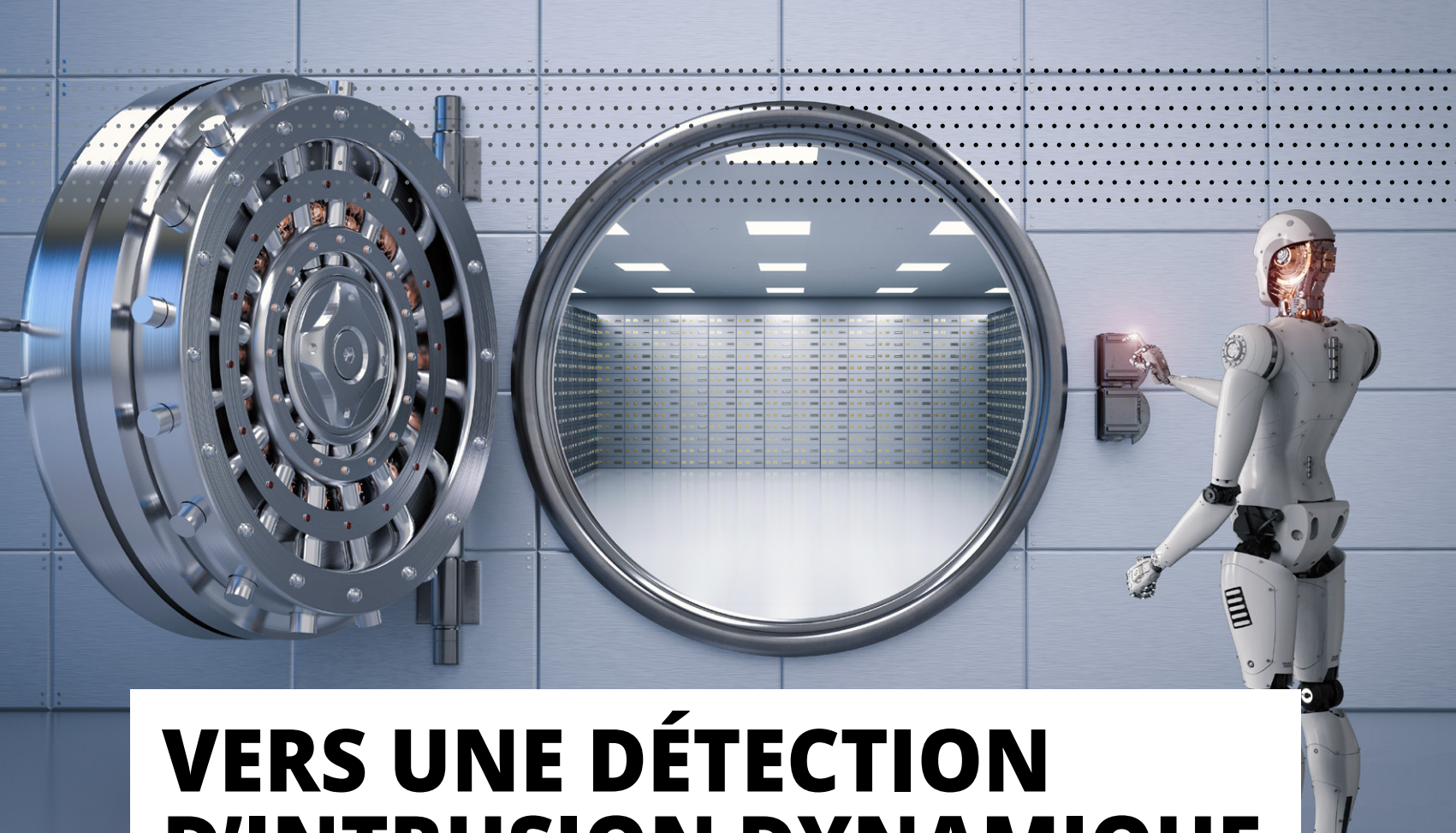
À titre d'exemple, le rapport de CB Insight AI100, liste les 100 startups qui impactent le monde du Machine Learning (ici appelé *Artificial Intelligence*), il contient 11 startups qui présentent déjà des caractéristiques de licornes. On notera que dans la catégorie cybersécurité, ce sont plus d'un milliard de dollars investis [4]. Ceci laisse présager de la dynamique à venir de ce marché.

Un facteur décisif dans le succès de ces startups, repose sur leur capacité à adresser un marché complexe avec des solutions simples. Malgré la diversité des solutions, la complexité des systèmes à surveiller, des attaques à détecter, ces startups qui gagnent la bataille économique du *Machine learning*, sont celles qui promettent une intégration facile avec les systèmes de sécurité existants, des diagnostics et plan d'action en temps réels, sous forme de dashboard faciles à lire.

Finalement, l'analyse de l'intersection entre Machine Learning, et cybersécurité, donne l'impression forte que le potentiel de cette technologie n'a pas encore été exploité, ni du côté de la défense, ni du côté des attaques. Préparons-nous à voir fleurir dans les années à venir de nouveaux systèmes de défense, puissants et efficaces, faisons confiance à la créativité du monde de la sécurité. ■

RÉFÉRENCES

- [1] <https://www.blackhat.com/us-18/sponsored-sessions/schedule/index.html#how-to-use-machine-learning-to-discover-new-targeted-attacks-12239>
- [2] <https://securityintelligence.com/deeplocker-how-ai-can-power-a-stealthy-new-breed-of-malware/>
- [3] <https://www.darkreading.com/risk/social-engineers-show-off-their-tricks/d/d-id/1332544>
- [4] <https://www.cbinsights.com/research/artificial-intelligence-top-startups/>



VERS UNE DÉTECTION D'INTRUSION DYNAMIQUE ET CONTINUE EN UTILISANT LES RÉSEAUX DE NEURONES

Charles MURE, Félix MOLINA, Mustafizur SHAHID et Gregory BLANC

La détection d'intrusion suppose souvent que l'on connaisse ce qu'est une intrusion (détection par signature) ou ce qui relève d'une activité normale (détection d'anomalie). Cependant, le volume de données à traiter dans un réseau et leur complexité brouillent souvent la frontière entre ces deux principales catégories de flux réseau. Les techniques d'apprentissage profond portent la promesse d'une découverte facilitée des caractéristiques qui permettraient de distinguer flux réseau légitimes et intrusions. Cet article se propose d'aborder de manière pratique des résultats de recherche récents en utilisant des outils libres comme Keras et TensorFlow.

À ce jour, de nombreux travaux de recherches, notamment concernant des NIDS (*Network Intrusion Detection System*, i.e. des systèmes de détection d'intrusion réseau), ont été conduits dans des conditions spécifiques en termes de systèmes, algorithmes, jeux de données utilisés, limitant l'extrapolation des résultats à une cible plus large de systèmes. Ces recherches se basent sur des méthodes d'apprentissage statistique permettant de concevoir des algorithmes encore jamais implémentés dans un système réel, permettant de détecter de nouvelles

attaques et de répondre aux problématiques dues au retard constant des outils de sécurité sur les menaces auxquelles les entreprises doivent faire face (vol de données, espionnage, etc.).

Dans cet article, nous nous proposons d'implémenter un système de détection complet basé sur des méthodes d'apprentissage profond et à partir de résultats de recherches parus récemment. Les travaux auxquels nous faisons référence décrivent la manière théorique d'implémenter un tel système. Nous nous proposons de mettre en œuvre les modèles issus de ces travaux, en y apportant les modifications nécessaires au bon fonctionnement et à la déployabilité du système.

Notre objectif est donc la mise en application de résultats de recherches dans un système capable de fonctionner en temps réel sur le réseau. La partie d'apprentissage profond (Deep Learning) du système sera implémentée en suivant les indications des différents articles de recherche et en utilisant le framework TensorFlow ainsi que la bibliothèque Keras permettant d'implémenter rapidement des algorithmes d'apprentissage automatique (Machine Learning). Nous nous intéresserons aux problématiques de gestion des différentes entrées du système de détection, et à celle de la continuité de la surveillance réseau au travers de la mise en place d'un système de traitement de flux de données.

Les données analysées par un IDS proviennent soit des données de paquets entiers, soit des flux de paquets. Dans notre cas, l'IDS considère uniquement les flux, ce qui réduit la quantité de données à analyser. L'implémentation du système a été réalisée en Python et utilise la bibliothèque scikit-learn, très facile à prendre en main et à installer [17].

MOTIVATION

Les réseaux gagnant toujours plus en taille et en complexité, il devient dès lors difficile pour un administrateur, voire un opérateur, de surveiller les activités qui s'y produisent et de détecter des anomalies, des incidents ou des activités malveillantes. La quantité de données à traiter, et leur provenance parfois imprécise, pourraient bien se résumer au fameux adage « chercher une aiguille dans une botte de foin ». Dans ce contexte, les algorithmes de Machine Learning peuvent aider à traiter l'immense quantité de données générées par le réseau afin de détecter les attaques. De nombreux travaux ont été réalisés pour mettre au point des systèmes de détection d'attaques en utilisant le Machine Learning [1]. Pouvoir décider qu'une donnée représente un comportement malveillant ou est indicatrice d'un comportement légitime relève de certaines caractéristiques (*features*) de celle-ci qui vont être traitées par l'algorithme de Machine Learning. Déterminer quelles sont les caractéristiques à utiliser est une étape souvent cruciale lors du développement d'un système de détection basé sur des algorithmes de Machine Learning.

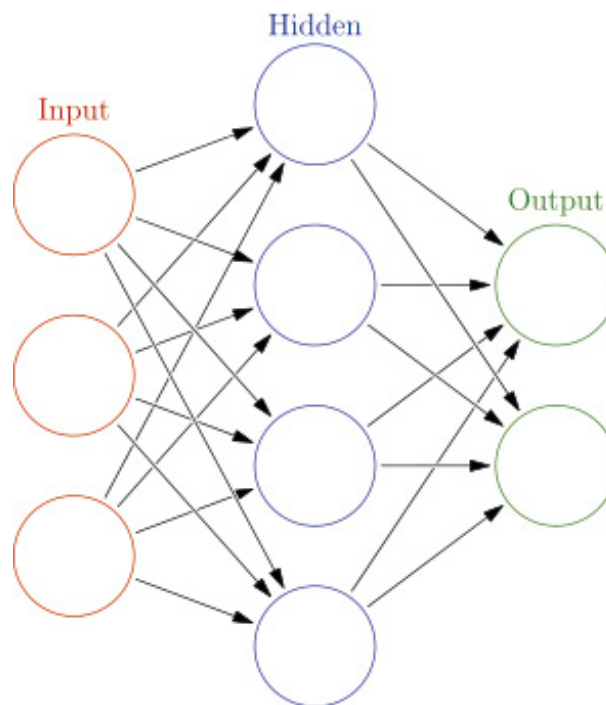


Fig. 1 : Exemple de réseau de neurones simple consistant en une couche d'entrée, une couche de sortie, et entre les deux, une couche de neurones convolutionnels (couche cachée), voir aussi l'article de A. Boucekif.

Les travaux présentés dans cet article tirent profit des algorithmes de Deep Learning, notamment les réseaux de neurones convolutionnels ou CNN (*Convolutional Neural Network*). Les réseaux de neurones sont des modèles d'apprentissage automatique inspirés du fonctionnement des neurones biologiques [2]. Un réseau de neurones (voir figure 1) est constitué de plusieurs couches : une couche d'entrée (*input layer*) constituée des caractéristiques décrivant la donnée (par exemple pour du trafic réseau, taille des paquets, nombre de paquets), une ou plusieurs couches cachées (*hidden layer*), et une couche de sortie (*output layer*) qui émet le résultat de la classification (malveillant ou légitime dans le cas de la détection d'intrusion). Chaque couche prend ses entrées de la sortie de la couche pré-

cédente. Chaque neurone d'une couche est ainsi connecté à tous les neurones de la couche précédente. Au niveau d'un neurone, un poids différent est associé à chaque entrée. Les données fournies à l'entrée traversent ainsi le réseau de neurones de couche en couche jusqu'à la couche de sortie où le résultat est émis. Le Deep Learning est un sous-domaine du Machine Learning et fait référence aux réseaux de neurones profonds, c'est-à-dire, les réseaux de neurones constitués d'au moins deux couches cachées [3].

Les algorithmes de Deep Learning permettent d'obtenir de meilleurs résultats que les algorithmes de Machine Learning classiques. En effet, les réseaux de neurones profonds sont capables de détecter des motifs beaucoup plus complexes permettant ainsi de réduire le taux de faux positifs dans le cas d'un IDS. L'autre avantage du Deep Learning est qu'il n'est plus nécessaire de déterminer les features à l'avance. En effet, les réseaux de neurones profonds peuvent à partir d'une représentation très complexe de la donnée sélectionner automatiquement, lors de la phase d'apprentissage, les features les plus pertinentes, ce qui leur permet de trouver des « motifs » afin de classer la donnée. Les CNN ont premièrement été conçus pour résoudre des problèmes de reconnaissance visuelle par ordinateur et peuvent être optimisés pour résoudre des tâches de classification d'images [4]. En effet, résoudre un problème aussi simple que la détection d'un chat sur une image en utilisant une architecture classique de réseau de neurones artificiels nécessiterait beaucoup trop de calculs et de ressources rendant l'opération impossible à réaliser en pratique. Les CNN constituent un bon point de départ dans le développement d'algorithmes de détection d'attaques. En effet, les CNN ont connu un succès dans de nombreuses applications telles que la reconnaissance faciale, la reconnaissance vocale ou encore le traitement automatisé du langage.

ARCHITECTURE

Dans le domaine de la sécurité des réseaux, deux types de réseaux neuronaux sont majoritairement utilisés, comme en attestent les recherches menées sur le sujet [5] : l'architecture convolutive (CNN) et le perceptron multicouche (MLP - *MultiLayer Perceptron*). L'avantage majeur des réseaux convolutifs par rapport au MLP est l'utilisation d'un seul poids associé aux entrées de tous les neurones d'une même couche de convolution. Ce qui entraîne une performance globale du système accrue et le traitement par translation reste invariant tout au long du processus [6].

Le modèle CNN que nous avons choisi d'implémenter est composé de plusieurs couches convolutionnelles avec trois étapes de mise en commun (*Pooling*) – où les neurones d'entrée sont réduits à un nombre moindre de neurones en sortie ; classiquement, nous avons retenu la valeur maximale des neurones en entrée (maximisation) – puis suivi par trois couches entièrement connectées comme dans un réseau neuronal multicouche classique. Nous avons construit une architecture CNN pour extraire finalement 10 catégories correspondant aux classes du jeu de données : soit 9 types d'attaques et une catégorie pour les flux sans danger pour le réseau. Les données pré-traitées sont transmises à travers la couche d'entrée comprenant 16 neurones correspondant aux 16 features décrites dans la section 3 (Jeux de données).



Fig. 2 : Architecture de notre réseau de neurones convolutionnels permettant de traiter 16 features en entrée via une série de couches de neurones convolutionnels cachés, entrecoupés de couches de mise en commun par maximisation, et d'obtenir en sortie une classification parmi 10 catégories (9 types d'attaques et 1 catégorie bénigne).

Le modèle (voir figure 2) utilise des filtres de taille 64, 128 et 256 (respectivement 12x64, 4x128 et 2x256 sorties). Les couches de convolution reçoivent donc en entrée plusieurs flux, et calculent la convolution de chacun d'entre eux avec chaque filtre. Les filtres correspondent exactement aux features que l'on souhaite retrouver dans les flux.

Chacun de ces groupes convolutifs est entièrement connecté par l'intermédiaire d'une couche de maximisation (*MaxPooling*). Ces étapes permettent de réduire la taille des données en entrée de ces couches pour extraire une maximisation de chaque feature en entrée. Cela augmente sa fiabilité en réduisant l'impact d'un écart de valeur des features. La couche dite d'aplatissement (*Flatten*) permet de réduire le réseau à une seule dimension précédemment créée par les couches convolutives. À la suite de la couche d'aplatissement, nous insérons un réseau neuronal dense connecté à 3 couches avec respectivement 100, 20 et 10 unités. La couche la plus élevée du groupe est entièrement connectée avec 100 sorties vers la couche intermédiaire ayant 20 sorties entièrement connectées à la couche de sortie possédant 10 neurones correspondant aux 10 catégories que nous considérons dans notre étude. Voir pour plus d'explications [3,21].

JEUX DE DONNÉES

Actuellement, il existe peu de jeux de données en accès libre suffisamment récents et complets (diversité et nouveauté des exemples d'attaques) pour permettre un apprentissage performant sur notre réseau de neurones, eu égard à la diversité et à la



réalité des flux réseau. Le jeu de données UNSW-NB15 [7] fourni par l'université de Nouvelle-Galles du Sud à Sydney (*University of New South Wales*, UNSW) fait usage des outils Argus [8] et Bro [9] pour générer 35 features. Une autre partie des features (12) a été calculée manuellement en prenant en compte la séquence chronologique des flux, ce qui permet ainsi d'introduire une notion de temporalité des attaques (par exemple, une attaque par déni de service sera détectée par un ensemble de flux et non par un unique flux anormal). Argus et Bro IDS pouvant fonctionner en temps réel, il est donc plus simple de mettre en place la collecte et le pré-traitement des données dans notre prototype. Nous avons donc préféré tester et réaliser l'apprentissage de notre modèle sur ce jeu de données.

Les données de l'UNSW-NB15 ont été créées par l'outil IXIA PerfectStorm fourni par le Centre Australien de Cybersécurité (*Australian Centre for Cyber Security*,

ACCS). Le jeu de données ainsi créé représente 100 Go de trafic et 2 540 044 enregistrements. Il comprend neuf types d'attaques détaillées dans le tableau ci-dessous.

Une partie de ce jeu de données a été configuré en tant que jeu d'apprentissage et d'essai pour le rendre plus efficace pour la phase d'apprentissage en réduisant par exemple la surreprésentation de certaines classes comme la classe *Normal* (2 218 761 enregistrements). Le jeu de données d'entraînement contient 175 341 enregistrements et le jeu de données de test contient 82 332 enregistrements. Le tableau ci-dessous contient la distribution pour chaque classe du jeu de données d'entraînement (voir [22]) :

Type	Nb d'enregistrements	Description
Normal	56 000	Données normales
Fuzzers	18 184	Tentative de suspendre un programme ou un réseau en lui fournissant des données générées au hasard.
Analysis	2 000	Contient différentes attaques de scan de ports, spam et fichiers html des pénétrations.
Backdoors	1 746	Porte dérobée : dans un logiciel, fonctionnalité inconnue de l'utilisateur légitime, qui donne un accès secret au logiciel.
DoS	12 264	Déni de service : tentative malveillante de rendre un serveur ou une ressource réseau inaccessible aux utilisateurs, généralement en interrompant ou en suspendant temporairement les services d'un hôte connecté à Internet.
Exploits	33 393	L'attaquant connaît un problème de sécurité au sein d'un système d'exploitation ou d'un logiciel et exploite cette connaissance en exploitant la vulnérabilité.
Generic	40 000	Une technique qui fonctionne contre tous les block-ciphers (avec un bloc et une taille de clé donnés), sans considération de la structure du block-cipher.
Reconnaissance	10 491	Contient tous les flux qui peuvent simuler des attaques qui recueillent de l'information.
Shellcode	1 133	Un petit morceau de code utilisé comme charge utile dans l'exploitation de la vulnérabilité des logiciels.
Worms	130	Attaquant qui se réplique afin de se propager à d'autres ordinateurs. Souvent, il utilise un réseau informatique pour se répandre, en s'appuyant sur les défaillances de sécurité de l'ordinateur cible pour y accéder.

Afin de réduire la complexité de notre prototype, nous avons suivi une implémentation proposée par A. Valero dans son IDS INsIDES [11]. Sa proposition est d'utiliser uniquement les features provenant d'Argus afin de réduire le temps de traitement (mise en corrélation des données des 2 logiciels) et de prouver que l'on peut créer une solution qui fonctionne uniquement avec des flux de données en temps réel. Ensuite, en utilisant la méthode **InfoGainAttributeEval** proposée dans le logiciel Weka [11], nous pouvons réduire notre espace de départ en sélectionnant uniquement les features les plus pertinentes et trouver un compromis acceptable entre précision et coût de calcul. Cela nous permet d'extraire 16 features du jeu de données que notre prototype utilisera. La liste des features sélectionnées est détaillée dans le tableau ci-dessous. Les 5 dernières features doivent être implémentées manuellement.

IMPLÉMENTATION

L'implémentation du système se décompose en 2 parties distinctes (voir figure 3). La première permet d'effectuer la phase d'apprentissage et la seconde permet de réaliser la catégorisation des flux réels sur un réseau local.

Nous avons utilisé Python pour mettre en place notre système de Machine Learning, associé au framework TensorFlow [12] et à la bibliothèque Keras [13]. TensorFlow est un framework open source développé par Google pour le calcul

numérique haute performance. Il facilite le déploiement de tâches de calcul sur une variété de plateformes (CPU, GPU, ou le Tensor Processing Unit, TPU de Google), allant des ordinateurs de bureau aux clusters de serveurs, en passant par les périphériques mobiles et terminaux. À l'origine, un framework pour la programmation de flux de données, il est aussi couramment utilisé pour des applications de Machine Learning, telles que des réseaux de neurones. Keras est une API de haut niveau pour la conception et la programmation de réseaux de neurones, écrite en Python et capable de fonctionner au-dessus de TensorFlow, CNTK ou Theano. Elle a été développée dans le but de permettre une expérimentation rapide par un prototypage facile des réseaux de neurones.

Caractéristique (nom)	Description de la caractéristique
dur	Durée totale de l'enregistrement
dpkts	Nombre de paquets envoyés de la destination à la source
sbytes	Nombre d'octets envoyés de la source à la destination
dbytes	Nombre d'octets envoyés de la destination à la source
rate	Nombre de paquets par seconde
sttl	TTL de la source à la destination
dttl	TTL de la destination à la source
sload	Nombre de bits par seconde pour la source
dload	Nombre de bits par seconde pour la destination
smean	Moyenne de la taille des paquets de flux transmis par l'adresse IP source
dmean	Moyenne de la taille des paquets de flux transmis par l'adresse IP destination
ct_dst_ltm	Ratio de connexions vers une même adresse IP destination parmi 100 connexions
ct_src_dport_ltm	Ratio de connexions depuis une même adresse IP source vers un même port destination parmi 100 connexions
ct_dst_sport_ltm	Ratio de connexions vers une même adresse IP destination depuis un même port source parmi 100 connexions
ct_dst_src_ltm	Ratio de connexion depuis une même adresse IP source vers une même adresse IP destination parmi 100 connexions
ct_state_ttl	Index fonction de l'état de la connexion et du TTL à la source et/ou à la destination

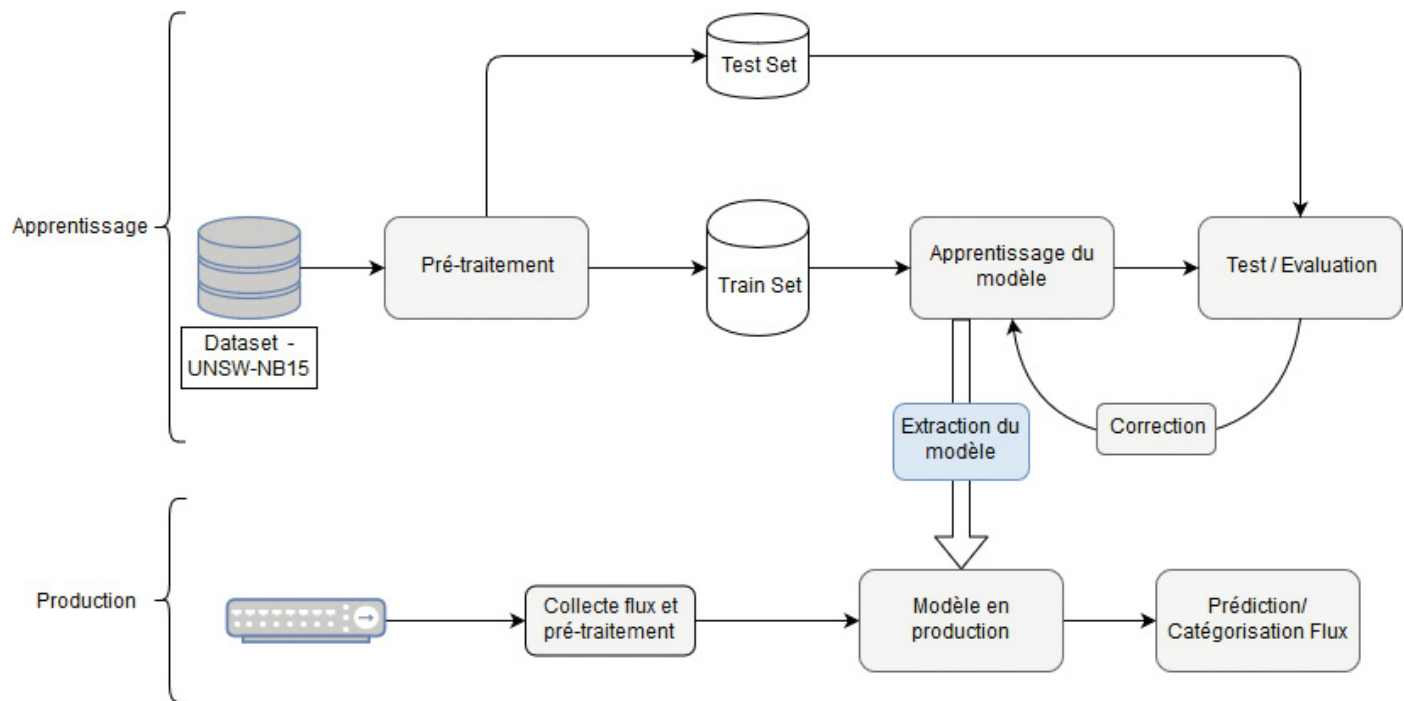


Fig. 3 : Implémentation du système comprenant une phase d'apprentissage à partir du jeu de données UNSW-NB15, et une phase de test en production à partir des flux capturés dans un réseau local.

Partie apprentissage

Dans cette partie, la bibliothèque Keras simplifie grandement la mise en place d'un réseau convolutionnel et la réalisation de la phase d'apprentissage. Pour créer un réseau, il faut dans un premier temps déclarer sa forme (quelles sont les différentes couches, leur taille, leur fonction d'activation, etc.). Le CNN est déclaré de la manière suivante :

```

model = Sequential()
model.add(Conv2D(64, (3, 1), activation='relu', input_shape=(shape, 1, 1))) # entrée
model.add(Conv2D(64, (3, 1), activation='relu')) # type de couche / taille / fonction
model.add(MaxPooling2D(pool_size=(2, 1)))
model.add(Conv2D(128, (3, 1), activation='relu'))
model.add(Conv2D(128, (3, 1), activation='relu', padding="same"))
model.add(Conv2D(128, (3, 1), activation='relu', padding="same"))
model.add(MaxPooling2D(pool_size=(2, 1)))
model.add(Conv2D(256, (3, 1), activation='relu', padding="same"))
model.add(Conv2D(256, (3, 1), activation='relu', padding="same"))
model.add(Conv2D(256, (3, 1), activation='relu', padding="same"))
model.add(MaxPooling2D(pool_size=(2, 1)))
model.add(Flatten())
model.add(Dense(100, kernel_initializer='normal', activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(20, kernel_initializer='normal', activation='relu', name='output'))
return model

```

Chaque ligne permet d'ajouter une nouvelle couche (layer) à notre réseau. Les 11 premières lignes constituent les niveaux convolutionnels (suite de couches **Conv2D** et **MaxPooling2D**) qui permet au réseau d'extraire ses propres features. Nous utilisons la fonction d'activation **ReLU** (Rectified Linear Unit) dans notre modèle. Cette fonction d'activation est la plus utilisée dans les modèles CNN et Deep Learning à l'heure actuelle, car ses propriétés mathématiques permettent d'accélérer

la phase d'apprentissage et d'obtenir de meilleurs résultats [14]. Les couches suivantes sont plus classiques et suivent le modèle du perceptron, il s'agit de la partie du réseau agissant comme classifieur.

Ensuite, il suffit de compiler le modèle déclaré puis de réaliser l'apprentissage à partir du jeu de données que vous lui fournissez. Cela s'effectue en 3 lignes de code Python :

```
model.compile(loss='categorical_crossentropy', optimizer=optimizers.Adam(lr=0.001),
metrics=['categorical_accuracy'])
stopper = EarlyStopping(monitor='categorical_accuracy', patience=2, mode='auto')
model.fit(x_train, y_train, epochs=4, batch_size=50, validation_data=(x_validation,
y_validation), callbacks=[stopper])
```

Quand nous compilons le modèle, nous précisons dans un premier temps la fonction de perte que nous allons utiliser, ici **categorical_crossentropy** qui est efficace pour les problèmes de classification multiple [15]. Ensuite, nous déclarons l'optimiseur que nous utilisons : Adam [16] ainsi que la valeur du paramètre de taux d'apprentissage (**lr** pour *learning rate*). Enfin, nous précisons la métrique que nous souhaitons utiliser pour observer les résultats de notre modèle. L'apprentissage est réalisé à l'aide de la fonction **fit**, à laquelle nous donnons en paramètres les données d'apprentissage et de validation, ainsi que les valeurs de répétition de l'apprentissage (**epochs**) et de taille d'échantillon (**batch_size**). L'utilisation de la bibliothèque scikit-learn [17] permet aussi d'automatiser la recherche des paramètres optimaux pour le modèle, mais nous ne rentrerons pas dans les détails de son utilisation dans cet article.

Cela semble pour l'instant déconcertant de simplicité (si on connaît scikit-learn), mais la partie la plus complexe dans le Machine Learning est généralement la phase de pré-traitement des données pour générer les tableaux **x_train**, **y_train**, **x_validation** et **y_validation**, qui correspondent aux données d'entrée et de sortie des phases d'apprentissage et de validation. Notre application n'échappe pas à cette règle !

Dans notre jeu de données, les données brutes sont au format CSV et sont composées de chaînes de caractères (*string*), d'entiers (*integer*) et de réels (*float*). Or, le réseau utilise obligatoirement des réels en entrée.

Les pré-traitements à effectuer sont les suivants :

- isoler les features souhaitées ;
- encoder les strings à l'aide d'un dictionnaire ;
- convertir le tableau en floats ;
- réadapter l'échelle de chaque feature pour ne pas biaiser le système (**MinMaxScaler**) ;
- découper de manière aléatoire notre jeu de données pour le séparer en jeux d'apprentissage (*train*), de validation et de test.

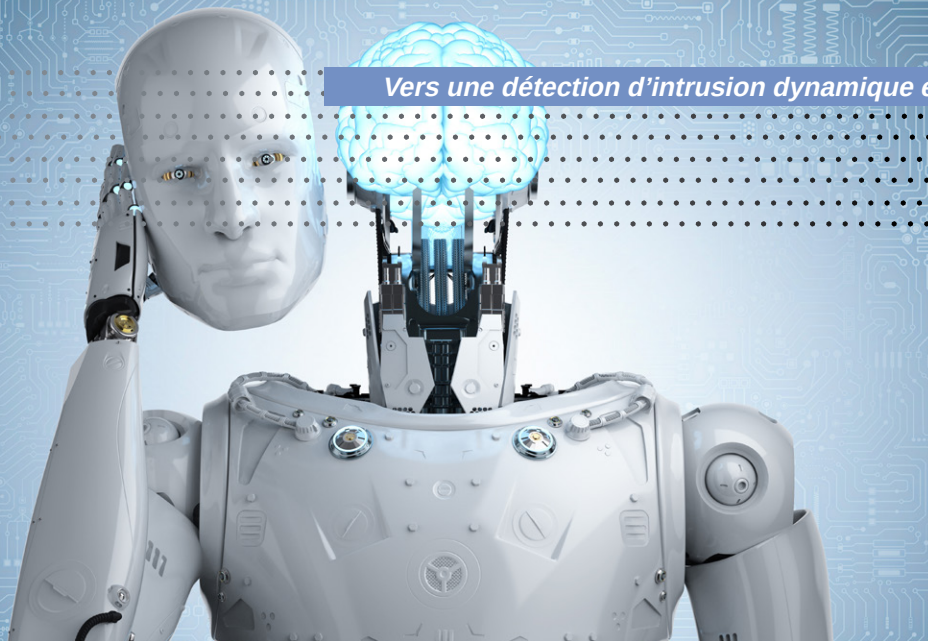
La phase de pré-traitement est généralement propre à chaque jeu de données et par conséquent cette partie du programme doit être ajustée à chaque fois que vous souhaitez tester votre système avec un nouveau jeu de données. Le pré-traitement s'effectue en utilisant la bibliothèque scikit-learn qui contient tout ce qu'il nous faut.

Une fois la phase d'apprentissage réalisée, nous pouvons enregistrer notre modèle ainsi que les poids associés pour le réutiliser dans la seconde phase de notre système : la prédiction/classification.

```
model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
model.save_weights("model.h5")
```

Partie prédiction

La catégorisation d'un flux de données s'effectue simplement à l'aide de Keras en utilisant le modèle extrait lors de la phase d'apprentissage. Cela s'effectue en quelques lignes de code Python, vous pouvez ensuite réécrire la sortie de la prédiction dans un fichier de log pour la réutiliser dans un autre système.



```

json_file = open('model.json', 'r')
loaded_model_json = json_file.read()
loaded_model = keras.models.model_from_json(loaded_model_json) # chargement du modèle
loaded_model.load_weights("model.h5") # chargement des poids
# compilation
loaded_model.compile(loss='binary_crossentropy',
optimizer='rmsprop', metrics=['accuracy'])
res = loaded_model.predict(data_log) # prédiction à partir de data_log
# écriture dans un fichier de log

```

Cependant, nous devons maintenant construire correctement notre tableau **data_log** sur lequel notre réseau va effectuer une prédiction. Pour cela, il faut arriver à extraire les mêmes features depuis le réseau que celles constituant notre jeu de données d'apprentissage.

Pour l'extraction des features depuis les flux collectés par les sondes, nous avons utilisé Argus, qui est un outil open source d'audit de réseau et qui permet de générer des fichiers de log avec des features pré-calculés utiles à la détection de comportements malveillants. Cela nous permet d'extraire directement 11 features que nous utilisons dans notre modèle. Pour les 5 features restantes qui calculent des statistiques basées sur les 100 derniers flux, il est nécessaire de les recalculer à la volée à l'aide d'un script en Python à partir des logs générés par Argus. En utilisant des principes de files d'attente FIFO et de recherche sur des condensats (hashs), nous pouvons exécuter de manière efficace la recherche d'éléments dans les 100 derniers flux.

Ci-dessous, nous présentons un exemple de calcul de la caractéristique **ct_dst_sport_ltm**.

```

# pour chaque nouvelle ligne de log
ct_dst_sport_ltm_hash = hash(logline[1] +
logline[2]) # IP destination + port source
ct_dst_sport_ltm_list.pop()
ct_dst_sport_ltm_list.insert(0, ct_dst_sport_ltm_hash) # insertion nouveau condensat
# calcul des 4 autres features
...
data_log = log_line + str(ct_dst_sport_ltm_list.
count(ct_dst_sport_ltm_hash))] + ...
...
# pré-traitement puis prédiction sur le flux

```

RÉSULTATS

Notre modèle CNN nous permet de classer les flux parmi 9 types d'attaques aux caractéristiques significatives. Il y a donc 9 classes d'attaques et une classe pour les flux dits normaux. L'histogramme (voir figure 4) représente les taux de précision, de rappel et de mesure F_1 calculés lors de nos tests.



NOTE

- Précision = nombre de flux correctement classés dans la catégorie i / nombre de flux classés dans la catégorie i
- Rappel = nombre de flux correctement classés dans la catégorie i / nombre de flux de la catégorie i
- Mesure $F_1 = 2 * (\text{Rappel} * \text{Précision}) / (\text{Rappel} + \text{Précision})$

Certains types d'attaques comme les attaques par déni de service (DoS) ou les attaques par porte dérobée (backdoor) se sont avérées compliquées à classer par le modèle. L'une des raisons est que pour obtenir un modèle bien entraîné et donc, a priori performant, il faut des jeux de données avec un grand nombre d'éléments de tous types afin de catégoriser la majorité des cas de figure d'attaque. Or la table du nombre d'enregistrements montre que certaines classes ne sont pas très bien représentées dans le jeu de données, car d'autres sont surreprésentées.

Nous pouvons aussi remarquer que le taux de précision de certaines classes est très faible. Ceci est dû au fait que dans ces classes nous trouvons beaucoup de faux positifs

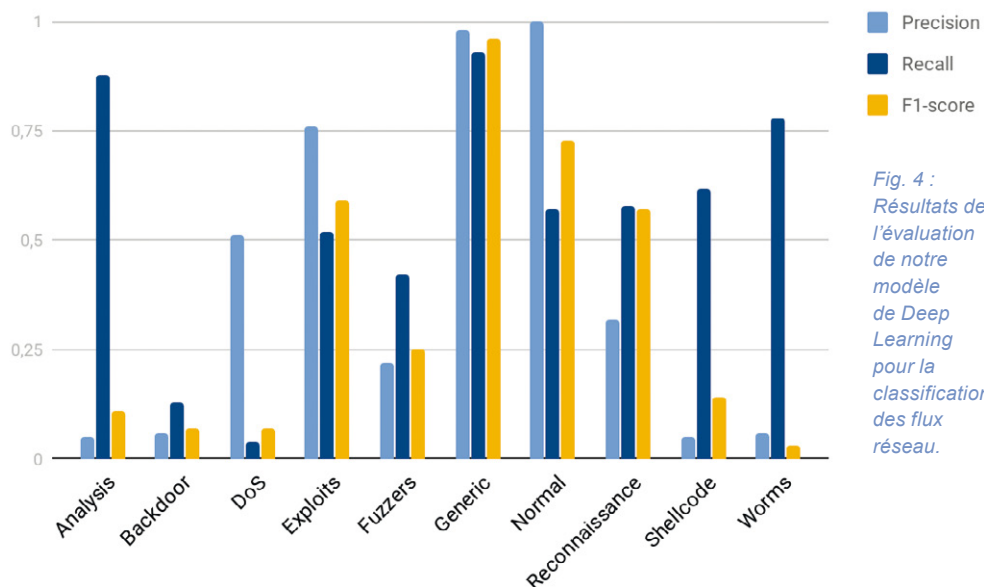


Fig. 4 : Résultats de l'évaluation de notre modèle de Deep Learning pour la classification des flux réseau.

au regard des vrais positifs découverts. Les faux positifs proviennent pour la plupart des erreurs de catégorisation. Par exemple, certains flux d'un scan (*Reconnaissance*) peuvent être catégorisés, par erreur, comme une tentative d'analyse (*Analysis*). Cela va donc induire des faux positifs dans la catégorie *Analysis* et réduire sa précision.

De manière générale, nous observons que notre modèle permet de détecter une très grande partie des vrais positifs de chaque classe, à l'exception des classes **Backdoor** et **DoS**. En effet, pour que l'IDS soit performant dans la catégorisation des flux de ces classes, il serait peut-être utile de réaliser une inspection profonde des paquets (*Deep Packet Inspection* ou DPI) ou d'analyser uniquement les en-têtes des paquets, ce qui donnerait des informations sur le type d'application visé.

Cependant, pour mieux percevoir l'intérêt du Machine Learning dans

un IDS, nous pouvons effectuer une comparaison avec un IDS plus classique. Nous avons choisi de comparer notre modèle à l'IDS open source Snort [18]. Pour cela, nous avons réalisé la phase d'apprentissage de notre modèle sur 175 341 flux, puis effectué la phase de test sur l'ensemble du jeu de données UNSW-NB15, c'est-à-dire sur les 2 540 047 flux. Concernant le test de Snort, nous nous appuyons sur le résultat d'une précédente étude [10] sur le même jeu de données UNSW-NB15, nous avons utilisé le même jeu de signatures, les matrices de confusion nous confirment que le nombre total de flux est identique. Ce test a été réalisé à l'aide des données brutes (*pcap*) et de l'outil *tcpreplay* pour rejouer l'ensemble des échanges sur le réseau. Nous avons montré dans la partie précédente qu'il nous était possible d'extraire les features d'un flux réseau, nous pouvons donc utiliser indifféremment les **.pcap** ou les **.csv** pour notre test.

Prediction par Snort (précision : 60,87%)

		2 540 047	Attaque	Légitime
Catégorie réelle	Attaque	7808	313 475	
	Légitime	680 365	1 538 399	

Prediction par notre modèle Deep Learning (précision : 97,99%)

		2 540 047	Attaque	Légitime
Catégorie réelle	Attaque	315 326	5957	
	Légitime	45 174	2 173 590	

Le Machine Learning et plus spécifiquement le Deep Learning est donc capable de surpasser les méthodes plus classiques dans la détection d'intrusion et semble extrêmement prometteur pour développer les prochaines générations d'IDS.

DISCUSSION

Néanmoins, nos résultats expérimentaux nous montrent que ces méthodes sont encore perfectibles. Les articles traitant de ce sujet sont aussi très critiquables de part les métriques qu'ils utilisent, mais aussi par le fait qu'ils utilisent des jeux de données peu représentatifs de l'état actuel des réseaux. En effet, bien que le jeu de données NSL-KDD [19] soit correctement construit pour tester un modèle, il est trop éloigné du trafic actuel d'un réseau et représente mal la complexité des attaques actuelles qui génèrent de moins en moins de signaux forts. Notre étude nous a aussi permis de faire le constat que trop peu de jeux de données sont disponibles pour pouvoir entraîner et mettre en place des modèles suffisamment précis. À l'inverse d'un système de détection d'anomalies, un modèle basé sur le Deep Learning nécessite d'avoir une répartition équilibrée entre les différentes classes : 130 flux d'attaques par vers et 56 000 flux de trafic normal ne permettent pas de construire un modèle de classification correct. Cela a pour conséquence de biaiser le système dans la phase d'apprentissage, car les flux normaux auront plus d'impact que les flux d'attaques de vers.

De plus, les jeux de données nous limitent à un nombre réduit de features qui nous empêche d'exploiter d'autres indicateurs dans la détec-

tion d'attaques. Par exemple, nous souhaitons mieux prendre en compte la notion de temporalité des attaques dans notre modèle en utilisant des systèmes proches de l'analyse de texte (modèle *Long Short Term Memory* [20]) qui sont actuellement mis en place dans les chatbots ou encore la prédiction de cours boursier. En effet, les attaques complexes peuvent être réparties entre plusieurs flux et s'étaler sur la durée (attaques APT), ou encore un trafic important la nuit ou le week-end dans une entreprise doit être considéré comme suspect. Les jeux de données n'étant pas horodatés, nous n'avons pas pu exploiter cette notion dans notre modèle.

CONCLUSION

Nous avons montré qu'il est possible d'utiliser des méthodes de classification basées sur le Deep Learning pour la détection d'intrusion sur un réseau et celles-ci montrent un véritable potentiel par rapport aux méthodes de détection d'anomalies ou bien par rapport aux méthodes basées sur les signatures. Elles permettent de détecter des attaques inconnues et comblent la principale lacune de la détection d'anomalies qui est le manque d'information sur l'attaque (savoir que le trafic est anormal permet difficilement de choisir l'action à mener pour contrer l'attaque potentielle).

Cependant, le modèle proposé ne possède pas une précision suffisamment élevée dans certaines classes pour être utilisé tel quel dans un système en entreprise, mais il est possible de construire un IDS hybride qui reposerait sur un système de détection par signature et par Machine Learning.

Pour que la détection d'intrusion à l'aide du Deep Learning se développe, nous avons besoin d'un grand volume de données catégorisées et d'exemples d'attaques. Or, la plupart des jeux de données disponibles publiquement sont soit trop vieux, soit trop restreints. À l'heure actuelle, la disponibilité de données d'apprentissage reste le plus grand frein au développement de cette technologie en open source.

Enfin, nous avons pu constater qu'il ne s'agit pas d'une simple application du Machine Learning à la sécurité informatique, mais que ce domaine nécessite une compréhension approfondie des diverses attaques d'un point de vue technique (pour définir les features des jeux de données) mais aussi des modèles de ML. Une forte coopération entre les experts en sécurité et en science des données est donc nécessaire pour développer le potentiel de ces méthodes de détection. ■

REMERCIEMENTS

Les auteurs souhaitent remercier Télécom SudParis, une école de l'IMT, pour leur avoir permis de conduire ces travaux. L'ensemble du code et des résultats peut être consulté sur le dépôt suivant : <https://github.com/CharlesMure/cassiope-NIDS>.



Les références de cet article sont disponibles sur : <https://www.miscmag.com/>.



MACHINE LEARNING EN PYTHON : CLUSTERISATION À LA RESCOUSSE DES HUNTERS (DE MALWARE)

Sébastien LARINIER

Cet article présente des techniques de clusterisation (classification automatique) de malware pour se faciliter la vie dans l'écriture de règles Yara. Cela permet de diminuer fortement le taux de faux positifs. On commence par clusteriser notre ensemble de malware, et, pour chaque cluster nous utilisons un générateur automatique de règles Yara. Nous donnons des exemples avec le jeu de données « theZoo ».

Le Machine Learning est une expression utilisée à tort et à travers dans beaucoup de domaines et l'analyse de malware n'est pas épargnée. Mais correctement utilisés, les algorithmes du Machine Learning peuvent générer un gain de temps non négligeable, par exemple pour les analystes de malware, pour comprendre une grande masse de données, sélectionner des fichiers ayant un profil particulier, et surtout : calculer des ensembles/clusters de fichiers ayant les mêmes propriétés. Pourquoi clusteriser un ensemble de malware ? Par exemple, c'est ce que nous allons montrer dans cet article, pour générer des règles Yara sur ces familles ! Cela va permettre de faire du *hunting* en réponse sur incident, mais aussi en source ouverte pour découvrir de nouvelles souches d'une même famille. Dans cet article, nous présentons les différentes étapes pour générer des règles Yara sur des malware, tous des exécutables Windows faisant partie du jeu de données TheZoo.

GÉNÉRALITÉS SUR LES EXÉCUTABLES WINDOWS

Les fichiers exécutables sous Windows respectent tous le même format appelé PE (*Portable Executable*) qui est décrit par Microsoft. On peut le schématiser sous la forme suivante comme l'a fait Ange Albertini [**Corkami**] dans l'image 1.

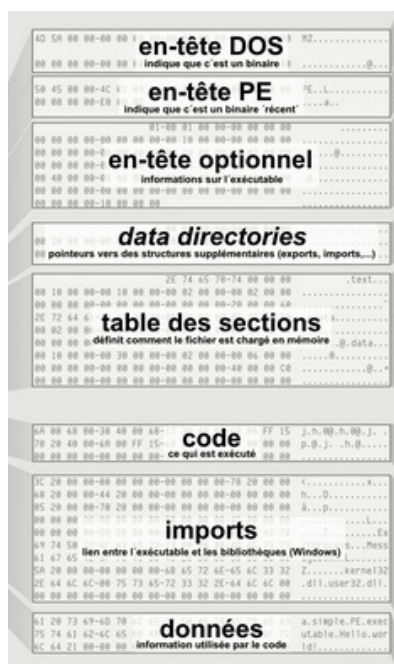


Fig. 1 : Le format PE.

Chaque en-tête, chaque table et chaque section a des informations qui lui sont propres. L'en-tête DOS par exemple est reconnaissable par son *magic number* MZ. L'en-tête PE (commençant par PE\0\0) contient l'en-tête de fichier et l'en-tête optionnelle.

L'en-tête de fichier spécifie plusieurs propriétés dont le type d'architecture sur laquelle peut s'exécuter le binaire, le nombre de sections, le timestamp de la compilation et si le binaire est un driver, une dll ou un exécutable.

L'en-tête optionnelle spécifie l'adresse de l'*entrypoint*, c'est-à-dire à quelle adresse se trouve la première instruction du binaire. Cette adresse doit correspondre à une section exécutable dans le fichier ; si ce n'est pas le cas, c'est que le fichier a été volontairement modifié (protection, packing) ou qu'il est corrompu.

Les autres informations correspondent à des informations système que le système d'exploitation va utiliser pour charger chaque section en mémoire et les *data directories*.

Les *data directories* sont constituées de deux parties :

- une taille ;
- et une adresse relative.

Il y en a 16 et toujours ordonnés dans ce sens dans le binaire, voir la figure 2.

Position	Name	Description
0	IMAGE_DIRECTORY_ENTRY_EXPORT	Export table
1	IMAGE_DIRECTORY_ENTRY_IMPORT	Import table
2	IMAGE_DIRECTORY_ENTRY_RESOURCE	Ressources table
3	IMAGE_DIRECTORY_ENTRY_EXCEPTION	Exceptions table
4	IMAGE_DIRECTORY_ENTRY_SECURITY	Certificats table
5	IMAGE_DIRECTORY_ENTRY_BASERELOC	Relocalisations table
6	IMAGE_DIRECTORY_ENTRY_DEBUG	debug
7	IMAGE_DIRECTORY_ENTRY_COPYRIGHT / IMAGE_DIRECTORY_ENTRY_ARCHITECTURE	copyright
8	IMAGE_DIRECTORY_ENTRY_GLOBALPTR	Global pointer
9	IMAGE_DIRECTORY_ENTRY_TLS	Threads table
10	IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG	Configuration table
11	IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT	Bound import
12	IMAGE_DIRECTORY_ENTRY_IAT	...
13	IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT	...
14	IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR	...
15	-	vide

Fig. 2 : Les *data directories*.

Détaillons que ce qui est important pour la suite de l'article :

- La table d'export est la table des fonctions qui pourront être utilisées par d'autres exécutables.
- La table d'import est l'emplacement des noms de fonctions du système d'exploitation utiles pour l'exécutable. À partir de ces noms, au chargement du binaire, les adresses de ses fonctions chargées seront résolues.
- La table des ressources est l'ensemble des ressources dont a besoin le binaire : chaînes de caractères, icônes, autre exécutable, etc.

Nous n'avons utilisé que ces données, mais il est évident que l'on peut étendre à d'autres caractéristiques : aux certificats, au *Thread Local Storage*, etc. Le but est de donner une approche générale.

Ensuite, structure importante puisque c'est le contenu de l'exécutable : les sections.

Chaque section est décrite dans une table avec :

- le nom de la section ;
- la taille ;
- l'adresse relative où elle démarre dans le binaire ;
- les permissions lorsqu'elle sera chargée en mémoire (lecture, écriture, exécution) ;
- et les informations utiles pour le chargement en mémoire. Il va donc s'agir du code, des ressources, des données, les imports et l'ensemble des informations utiles à l'exécution du binaire.

Il n'y a pas de règle sur le nommage des sections ainsi que sur le nombre de sections que peut compter un binaire.

Il existe plusieurs *parsers* de ce format, le plus utilisé codé entièrement en python est **Pefile** [**Pefile**], et un codé en C++ de plus en plus utilisé est **lief** [**LIEF**].

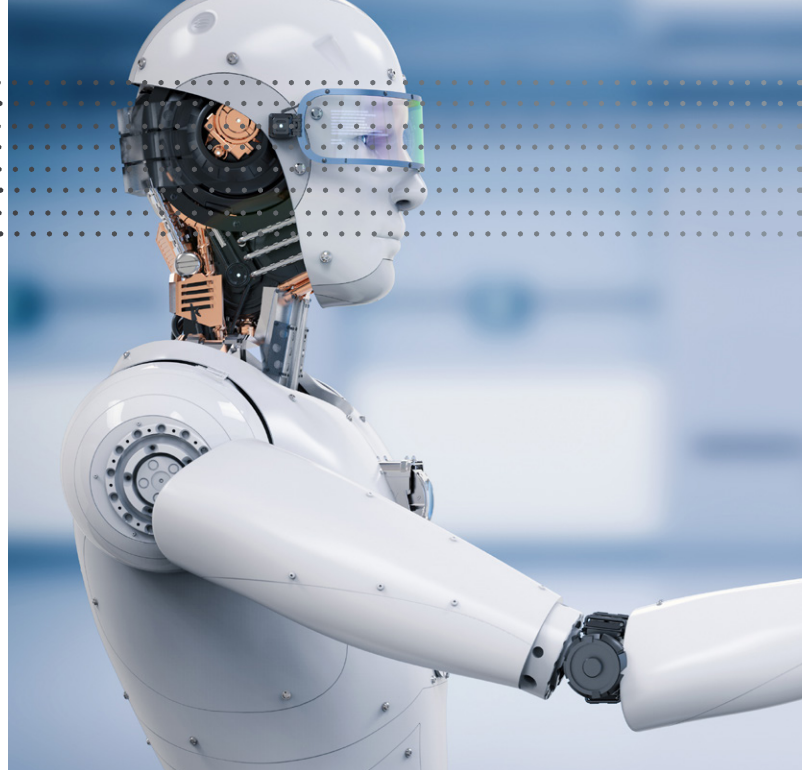
La lecture des deux projets est recommandée pour bien comprendre le format PE. Vous pouvez aussi tout simplement coder votre propre parser (même en OCAML oui oui oui :)).

MALWARE ET CLUSTERISATION

Le clusterisation de malware est une méthode pour rassembler sous forme de familles (clusters) des malware qui sont similaires, i.e. qui se « ressemblent ». Cela dépend évidemment des critères que l'on va choisir pour le faire.

Une des premières tentatives de clusterisation est l'utilisation d'algorithmes de *fuzzy hash* (dit aussi « hash mou »). Une des implémentations les plus connues dans le monde de l'analyse de malware et du forensic est **Ssdeep** [**ssdeep**]. Le principe de **Ssdeep** est basé sur ce qu'on appelle les *Context Triggered Piecewise Hashes* (CTPH), appelés aussi « fuzzy hash », l'objectif est de calculer une « signature » qui reste la même ou similaire s'il y a peu de modifications dans le fichier.

Le format de la signature est le suivant :
Chunksize:chunk:double_chunk.



Comme l'indique le papier présentant l'algorithme en 2006 [**ssdeep_for**], il est possible de calculer une similarité entre les fichiers, car l'algorithme calcule en fait une distance d'édition entre les signatures. Mais il y a un bémol : cela n'a de sens que si le **chunk_size** est identique dans les deux signatures, sinon ce n'est pas pertinent, voire trompeur.

Prenons un exemple de deux malware PlugX [**plugx**]. Leurs hashes sont complètement différents : **5b183e99e-d532cb143c55ded6a46f5b0545c575f6b-54b25afbb04b0bbf40170c302b1196dbe7d90effe72de832e0696016ea69a7a6915217081f013c1db7bc37**

Leur **Ssdeep** correspondants sont :

6144:z4lRkAehaKuqT+FXErGKhVPBD3A29ELNLnqpC+:zkWAehJuqTvbV50LIpD6144:z4lRkAehaKuqT+FXErGKhVPBD3A2LL0fY2axu/oI4R6ZDIB:zkWAehJuqTvbV5D2dsRHB

En lisant ces deux signatures, le lecteur se rend vite compte qu'elles sont comparables.

L'algorithme de vérification nous donne un score de 75%. Autre exemple : les deux malware **tinynuke** [**tinynuke**], dont les hashes sont :

496359ac995aaca927b13f341ed5d00025d06cb83fff1525b|5814f2630019f73 et ab166c-c565183257fa7e7607940bf45d0676853bc-9cee59e3b831f1e9684dd7b

Ils sont clairement différents, mais leurs **Ssdeep** sont identiques :

1536:ySF0RvzKlDowI25JhhDG3HTuPwh1Zcm-NyQknppa cYeXdmngAQ1WIm06/R3xhH9:FF0R79w75zI3HaPm1tXkpyYAmmgAGmF 40



Le problème majeur de cet algorithme, comme nous l'avons mentionné ci-dessus, est la taille du blob de données (**chunk_size**). Ainsi, il est très facile d'ajouter une section au fichier avec de la donnée aléatoire pour faire augmenter cette taille et rendre caduque la comparaison.

Une méthode alternative existe, qui consiste à ne calculer la signature que sur la section de code (section dont les permissions sont en lecture + exécution).

Alors les plus tatillons d'entre vous nous rétorqueront que cette technique ne passe pas la « packersisation ». Et nous vous dirons que vous avez raison ! Petit rappel, un packer est un programme qui va modifier la structure binaire d'un fichier le rendant toujours fonctionnel. Mais si nous regardons la structure binaire, elle est souvent plus complexe et ne correspond en rien au malware que l'on veut analyser, car ce dernier va être chargé en mémoire directement après souvent un passage par une fonction soit de décompression, soit de déchiffrement, soit les deux.

Le problème du packing est inhérent finalement à toute analyse statique, mais nous verrons par la suite que ce problème n'est pas toujours sans solution.

Une autre tentative de clusterisation sur les malware Windows est PEHash, outil présenté à Usenix en 2008 [**pehash**]. L'algorithme va calculer une signature basée sur des informations qui sont propres au format PE et le pondérer avec une approximation de la complexité de Kolmogorov.

```
hash[0]:=characteristics[0..7] ⊕ characteristics[8..15]
hash[1]:=subsystem[0..7] ⊕ subsystem[8..15]
hash[2]:=stackcommit[8..15] ⊕ stackcommit[16..23]
⊕ stackcommit[24..31] hash[3]:=heapcommit[8..15] ⊕
heapcommit[16..23] ⊕ heapcommit[24..31]

shash[0]:=virtaddress[9..31] shash[2]:=rawsize[8..31]
shash[4]:=characteristics[16..23] ⊕ characteristics[24..31]
shash[5]:=kolmogorov ∈ [0..7] ⊂ N
```

Le but est de constituer une base avec ces signatures. Et dans le cas d'une nouvelle variante (pas juste le *packer* qui aurait modifié le binaire), une nouvelle signature est trouvée. Mais il n'existe pas de notion de distance pour comparer deux signatures entre elles. En 2014, Mandiant (racheté par FireEye) propose une nouvelle métrique : l'*imphash* [**imphash**].

L'*imphash* est le md5 de l'ensemble des fonctions de la table d'imports. Le concept principal est que deux malware de la même famille avec peu de modifications utilisent les mêmes fonctionnalités, donc appellent les mêmes fonctions système et ainsi ont le même *imphash*.

Par exemple deux malware tyninuke [**tyninuke**] :

fa4449f3d8fe9b0bb0955b5d1e9da62b1ad7118aec052fdb-57367f1611ef0cca
da15730be7c4b4414ce555f81b06cca41c2a43d672582e1d5dcb6ac-5b1e0c15b

ont le même *imphash* : **bf60559d4831b8d874424ee2ddd0c057**.

Un souci majeur avec cette approche est que si le *linker* change l'ordre des fonctions, ou qu'il y a une fonction qui est ajoutée ou enlevée, les hashes seront complètement différents.

Le Cert national japonais a ainsi publié le concept de *ImpFuzzy* [**impfuzzy**]. Au lieu de calculer le *Ssdeep* sur le fichier complet, les auteurs proposent de le faire uniquement sur la table des imports. On peut ainsi calculer une distance entre les tables d'import [**impfuzzy_clust**].

Depuis 2015, plusieurs indicateurs ont vu le jour : il s'agit de travailler sur les basic blocks des fonctions du binaire. On peut noter les outils suivants : First de Talos, Polichombr [**polichombr**] de l'ANSSI et son implémentation via Radare2 ou encore machoke [**machoke**] de Conix Security. Le but est de reconnaître des similarités dans les fonctions du binaire en utilisant le principe de fuzzy hash pour ces deux derniers. Il est donc possible de calculer une distance, ou au moins une similarité, entre les objets. Le lecteur peut aussi lire l'article sur YaDiff dans ce hors-série.

L'ensemble de ces techniques est très utilisé au niveau opérationnel. Que celui qui ne fait pas de *hunting* dans VirusTotal avec l'*imphash* jette la première pierre à l'auteur.

Mais un problème de taille pour le calcul de ses distances est le passage à l'échelle. Si on a n signatures, on a $n*(n-1)/2$ comparaisons, soit une complexité dans l'ordre de $O(n^2)$: ce qui est vite difficile, voire impossible dès lors que n dépasse le million. Mais certains ont déjà résolu le problème dans les algorithmes que nous allons présenter.

LES VECTEURS DE CARACTÉRISTIQUES ET LES ALGORITHMES UTILISÉS

L'ensemble des algorithmes que nous allons présenter est issu de l'algèbre linéaire, donc (le mathématicien en vous va hurler) : ce sont des matrices et des vecteurs.

Chaque malware va avoir une représentation sous forme de vecteur : le vecteur de caractéristiques. Le but est de décrire le malware avec l'ensemble de ses caractéristiques (ou en tout cas celles choisies et retenues par l'analyste) :

- Sections : nom, taille, entropie, permission.
- Table d'imports : nombre de modules, nombre de symboles, fonctionnalités.
- La table d'exports : nombre de modules, nombre de symboles, fonctionnalités.
- Taille du fichier.

Ce sont des informations obtenues par analyse statique, et il n'y a évidemment pas l'ensemble des informations de ce type qui pourrait décrire un malware, et il n'y a aucune information de type comportemental. Mais essayons déjà de voir si ces dernières suffisent à notre problème, car finalement elles recouvrent l'ensemble des indicateurs décrits ci-dessus.

Un vecteur de caractéristiques n'est rien d'autre qu'un tableau. En python, c'est un **array** dans la librairie numpy de Scikit [**Scikit**] (qui est la librairie de référence d'algèbre linéaire pour le Machine Learning).

Les algorithmes que nous avons utilisés sont DBSCAN et K-means (voir le papier de A. Letois dans ce hors-série), et la distance que nous allons utiliser entre les vecteurs de caractéristiques est la distance euclidienne.

L'idée principale est que deux vecteurs ayant une faible distance représentent finalement deux objets qui sont similaires, voir la figure 3.

Nous n'allons pas évidemment calculer chaque distance entre chaque vecteur, sinon nous tomberions dans la même problématique que nous avons citée ci-dessus, mais nous allons utiliser deux algorithmes de clusterisation : DBSCAN et K-means, tous deux disponibles dans Scikit-Learn.

EXPÉRIMENTATIONS SUR LE DATASET THEZOO

Pour la suite de cet article, nous utiliserons le jeu de données (*dataset*) theZoo [**theZoo**], ainsi le lecteur curieux pourra retrouver les mêmes résultats. Il n'y a pas que des malware pour Windows, mais aussi pour Android et OS X. Il y a même le code source de certains.

La première étape consiste avec votre parseur préféré à extraire l'ensemble des métadatas dans des fichiers au format json ou dans la base de données que vous souhaitez.



NOTE

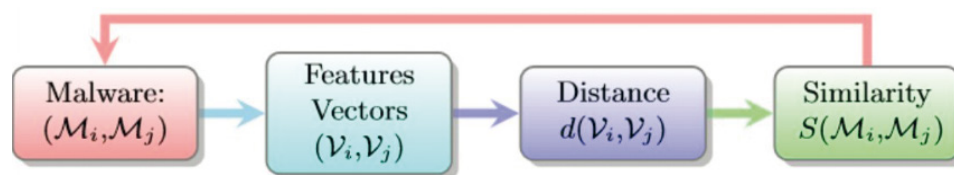
« theZoo » est un (petit) jeu de données (de malware étiquetés) disponible sur [theZoo]. Les étiquettes vont nous permettre de connaître les véritables clusters et donc de pouvoir comparer nos résultats avec les valeurs réelles. Il est à noter que le jeu n'est pas homogène, il est composé majoritairement de malware EquationGroup.

Le premier vecteur de caractéristiques que nous allons utiliser est le suivant :

[Taille du fichier, Nombre de sections, Moyenne de l'entropie des sections, Nombre de symboles de la table d'Import, Nombres de symboles de la table d'Export]

Pour la suite, nous allons travailler sur 724 malware (il se peut que des malware aient été ajoutés au *dataset* si vous le téléchargez à la lecture). Notre vecteur de caractéristiques est donc de dimension 5 :

Fig. 3 : Une distance sur les vecteurs de caractéristiques donne seulement une similarité sur les malware.



```
features = ['size of file', 'number of sections', 'median of entropy', 'number of imports', 'number of exports']
```

L'ensemble des données dont nous avons besoin est chargé dans une base de données Redis.

```
import glob
from redis_management import RedisManagement
from collections import OrderedDict
from ml_helpers import ml_helpers

rmgt = RedisManagement('malwares')

all_json_files = [ json.load(open(path)) for path in glob.glob('jsonfiles/*')]

for j in all_json_files:
    malware = OrderedDict()
    malware[features[0]] = j['size']
    malware[features[1]] = j['number_sections']
    malware[features[2]] = sum( sec['entropy'] for sec in j['sections']) / j['number_sections']
    malware[features[3]] = sum(len(v) for k,v in j['imports'].items())
    malware[features[4]] = sum(len(v) for k,v in j['exports'].items())
    rmgt.record_vector(j['hashes']['sha256'], 'first_vector', malware)
    rmgt.record_malware_label(j['hashes']['sha256'], j['path'], path_repo)
```

Un petit *helper* pour construire la matrice dont le but est de charger toutes les données que nous avons dans une même liste (ici de taille de 724 *5) :

```
def create_matrix(self, vector_name):

    self.all_malwares = [list(eval(self.redis_client.hget(k, vector_name)).values()) for k in self.get_all_malwares]

    return np.array(self.all_malwares)

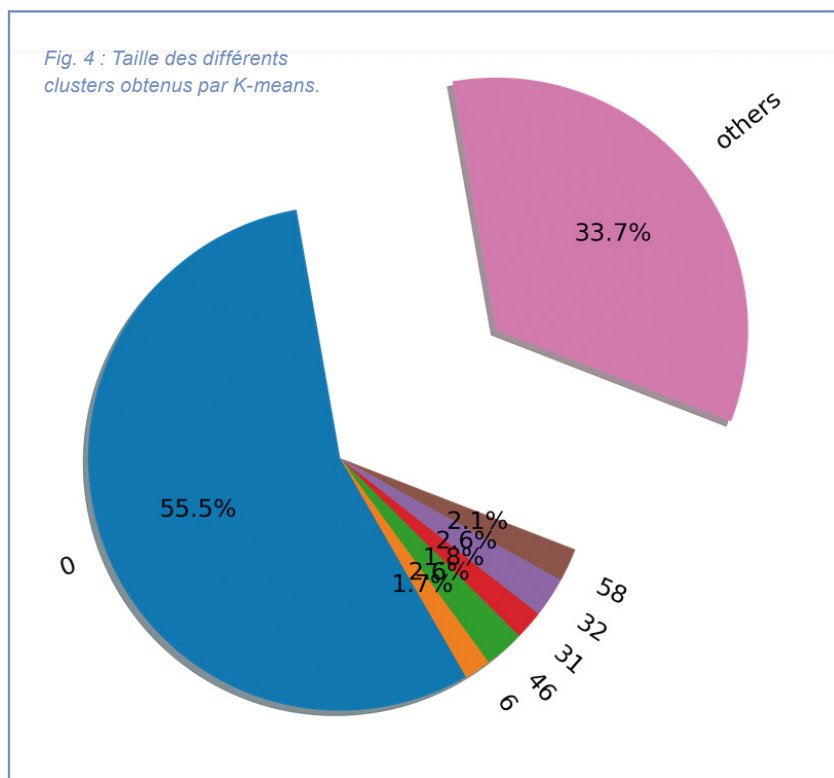
ml_h = ml_helpers(rmgt.redis_client)
mat = ml_h.create_matrix('first_vector')
```

mat est donc notre matrice contenant l'ensemble des vecteurs de caractéristiques.

Maintenant que nous avons en mémoire notre matrice, il ne reste plus qu'à la passer à K-means et DBSCAN. Pour l'algorithme K-means, le code est (avec un choix de k=90 un peu arbitraire) :


```
k_means = KMeans(n_clusters=90,n_jobs=8,precompute_distances=False)
kmeans_result = k_means.fit(mat)
```

Nous avons obtenu 90 clusters étiquetés de 0 à 89. Penchons-nous sur la distribution. Le graphique de la figure 4 a été réalisé en suivant la règle suivante : si une famille n'a pas plus de 10 éléments, elle est mise dans « other ».



La figure 4 montre que la famille 0 a 55.5% de représentativité. Si l'on regarde le type de malware classés :

Famille 0

EquationGroup.Fanny : 1
EquationGroup : 388
Backdoor.MSIL.Tyupkin : 1
Ransomware.Locky : 1

Famille 58

PotaoExpress : 5
EquationGroup : 8
EquationGroup.EquationLaser : 1
Win32.Avatar : 1

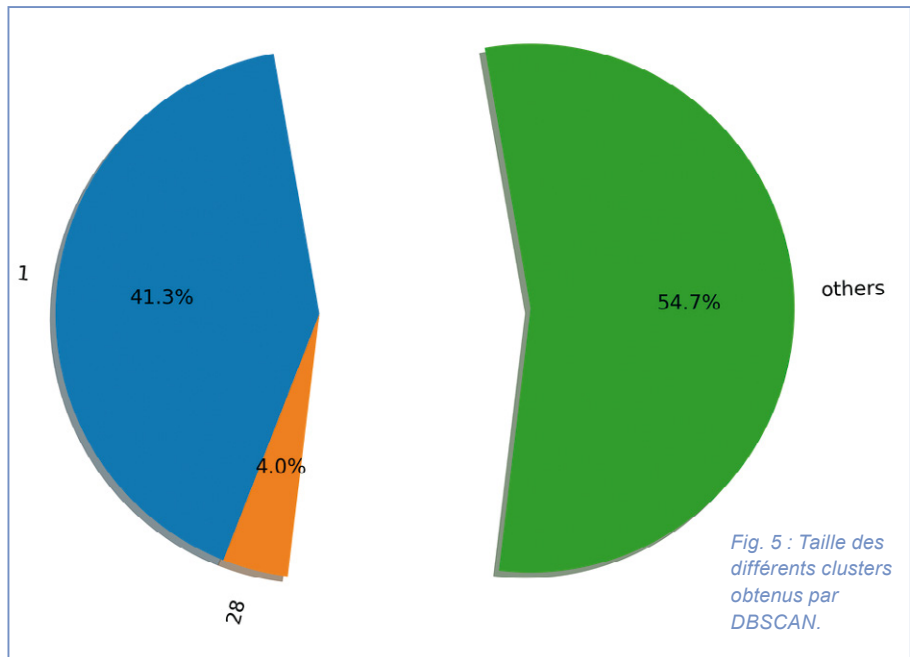
Famille 32

Duqu2 : 3
Trojan.Regin : 11
njRAT-v0.6.4 : 2
Neurevt.1.7.0.1 : 1
Win32.Cutwail : 1

On a un partitionnement qui est acceptable, mais on voit bien qu'il n'est pas parfait.

Regardons avec l'algorithme DBSCAN, qu'on utilise de la manière suivante :

```
metric="euclidean",n_jobs=8)
dbscan.fit(mat)
labels=dbscan.labels_.tolist()
len(set(labels))
dbscan = DBSCAN(eps=0.001,min_samples=1,
```



DBSCAN nous trouve 346 familles. Si l'on regarde l'hétérogénéité du jeu de données, cela est cohérent.

L'avantage de DBSCAN est que nous n'avons pas besoin de lui donner le nombre de familles/clusters comme c'est le cas pour K-means. DBSCAN calcule le nombre de clusters au cours de l'exécution de l'algorithme. Ce qu'il n'arrive pas à classer, il considère cela comme du bruit. L'image 5 montre la distribution basée sur les mêmes critères que DBSCAN. Il est à noter que **eps** est ici un hyperparamètre et que l'utilisateur doit donc le choisir, à défaut il doit en essayer plusieurs. Le lecteur curieux pourra aussi essayer Automeans, décrit dans ce hors-série.

Famille 1

EquationGroup.Fanny: 1
EquationGroup: 298

Famille 28

EquationGroup: 29

DBSCAN est meilleur que K-means sur l'homogénéité de chaque famille, mais K-means donne un meilleur classement (car ici nous connaissons le nombre de clusters que nous désirons étudier !).

L'ensemble ne reste pas totalement satisfaisant. Si l'on regarde de plus près chaque vecteur de caractéristiques et si on calcule sa norme (qui correspond à la longueur dans un monde à deux dimensions), on se rend compte $\text{norm}(\mathbf{v}) \sim \text{taille de fichier}$. Ce qui veut dire que les autres propriétés sont anodines dans le classement. Nous allons donc normaliser les vecteurs. C'est-à-dire que nous allons diviser chaque propriété, chaque composante du

tableau, par sa valeur maximale rencontrée dans le jeu de données. L'ensemble des composantes se trouve alors dans [0,1]. Le vecteur de caractéristiques devient :

[taille du fichier/ max(taille du fichier de tous les fichiers), nombres de sections/ max(nombres de sections de tous les fichiers), Moyenne de l'entropie / max(la moyenne de l'entropie de tous les fichiers), nombre de symboles de la table d'imports / max(nombre de symboles de la table d'imports de tous les fichiers), nombre de symboles de la table d'export / max(nombre de symboles de la table d'exports de tous les fichiers)]

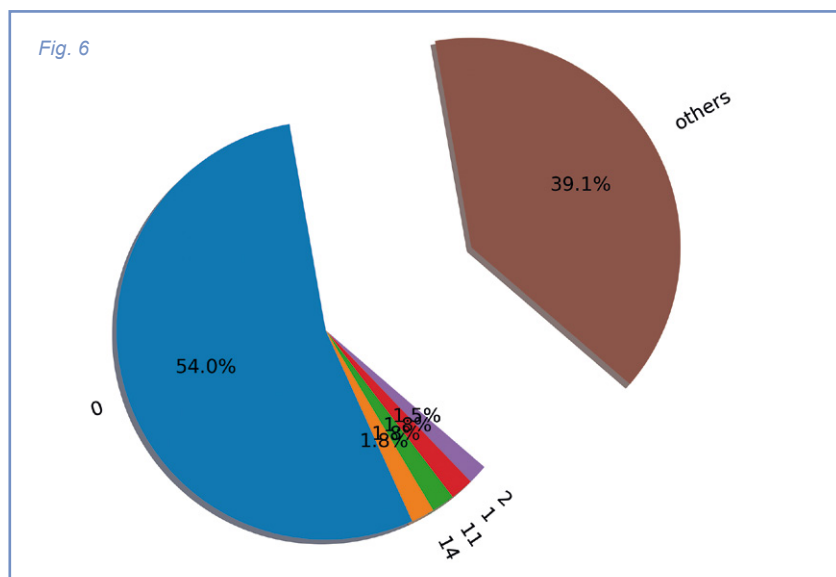
```
max_values = ml_h.get_max_values(all_vectors, features)

OrderedDict([('size of file', 41988096),
             ('number of sections', 10),
             ('median of entropy', 6.6367328295474355),
             ('number of imports', 577),
             ('number of exports', 61)])
```

Les vecteurs sont calculés ainsi :

```
list_second_vector = [OrderedDict((k, v / max_values[k]) for k,v in
m.items()) for m in all_vectors]
```

Relançons K-Means : la figure 6 montre les résultats.



Famille 0

EquationGroup.Fanny : 1
EquationGroup : 388
Shamoon : 2

Famille 2

Waski.Upatre : 1
Trojan.Sinowal : 8
WMIGhost : 1
Rustock : 1

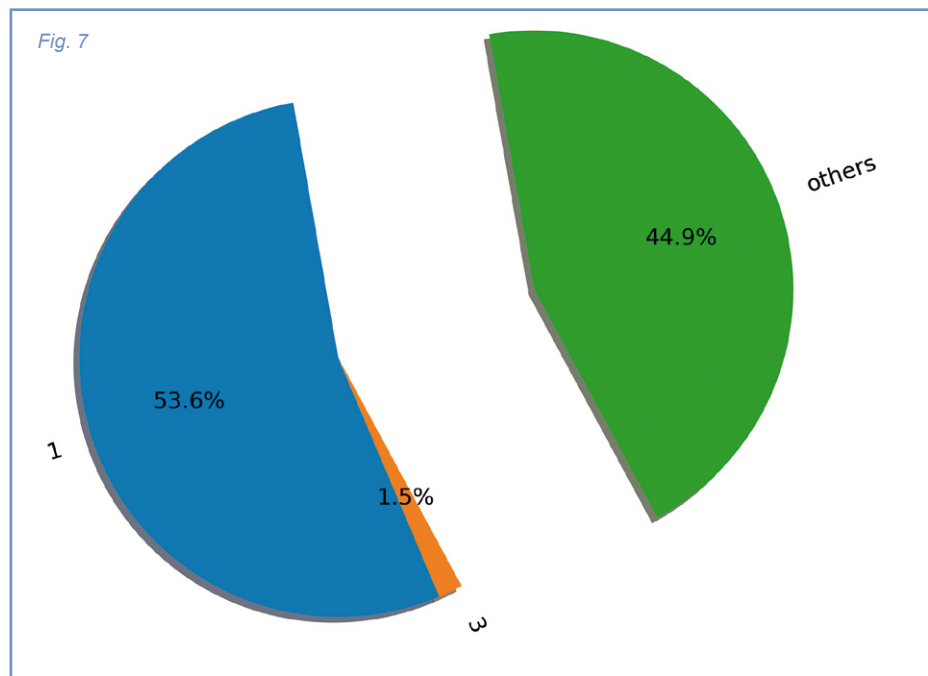
Famille 11

WMIGhost : 6
Dyre : 1
KRBanker : 1
Kelihos : 1
VolatileCedar.Explosion : 1
PotaoExpress : 1
Keylogger.Ardamax : 1
PlugX : 1

Famille 14

VolatileCedar.Explosion : 13

Fig. 7



Il y a un peu moins de mélange que pour le tour précédent.

Si on regarde du côté de DBSCAN, nous avons maintenant 212 familles. La distribution globale n'a pas vraiment évolué, comme le montre l'image 7.

Mais la qualité est meilleure et nous avons les versions des différentes familles. Le lecteur peut faire d'autres tests avec une valeur de k plus grande pour se faire sa propre opinion.

Famille 1	Famille 3	Famille 88	Famille 38	Famille 39
EquationGroup. Fanny : 1 EquationGroup : 387	VolatileCedar. Explosion : 11	EquationGroup : 8 EquationGroup.EquationLaser : 1	Trojan.Sinowal : 8	EquationGroup. TripleFantasy : 1 EquationGroup : 6 Win32.Avatar : 1

Nous retrouvons notre famille de 388 malware de la famille *Equation group* du dropper Fanny.

GÉNÉRATION DES RÈGLES YARA ET HUNTING

Yara peut être vue comme une grammaire pour faire des signatures **[Yar]**. Ce format tiré du logiciel open source du même nom est beaucoup utilisé chez VirusTotal et dans la communauté de chercheurs.

Une idée commune utilisée notamment chez un éditeur comme Intezer ou dans l'open source est qu'une même famille de malware va partager une partie de ses chaînes de caractères.

Pourquoi ? Car **strings -a** reste effectivement l'outil de threat intelligence le plus évolué, jamais égalé.

Côté Yara, un outil open source YaraGenerator **[YaraGen]** reprend cette idée, mais il commence à dater et quelques modifications devraient être apportées pour le rendre plus performant.

Si nous appliquons à la famille dite *Equation Group* qui comporte pour notre jeu de données 388 malware, nous obtenons la règle Yara suivante générée par l'outil :

```

rule EquationGroup
{
meta:
    author = "Anonymous"
    date = "2017-10-10"
strings:
    $string0 = "BINARY"
    $string1 = "5_5m586"
    $string2 = "MmBuildMdlForNonPagedPool"
    $string3 = "Hs 7({"
    $string4 = ";/;I;h;"
    $string5 = "L$(PQW"
    $string6 = "\\shelldoc.dll"
    $string7 = "5:5F5P5"
    $string8 = "Software\\Microsoft\\MSNetMng"
    $string9 = "msvcrt"
    $string10 = "8(8Q8m8"
    $string11 = "RedrawWindow"
    $string12 = "Global\\DirectMarketing"
    $string13 = "WinStaObj" wide
    $string14 = "PVVVVV"
    $string15 = "fseek(SEEK_SET) failed"
    $string16 = "lNSDEyR]"
    $string17 = ":1:a:f:"
condition:
    17 of them
}

```

Si le lecteur attentif et sérieux que vous êtes refait l'expérience, il se peut que vous n'obteniez pas la même chose, car si on regarde un peu le code :

```

def buildYara(options, strings, hashes):
    date = datetime.now().strftime("%Y-%m-%d")
    randStrings = []
    #Ensure we have shared attributes and select twenty
    try:
        for i in range(1,20):
            randStrings.append(random.choice(strings))
    except IndexError:
        print '[!] No Common Attributes Found For All
Samples, Please Be More Selective'
        sys.exit(1)

    #Prioritize based on specific filetype
    if options.FileType == 'email':
        for string in strings:
            if "@" in string:
                randStrings.append(string)
            if "." in string:
                randStrings.append(string)

    #Remove Duplicates
    randStrings = list(set(randStrings))

```

On se rend compte qu'il y a un tirage aléatoire des chaînes de caractères retenues pour construire la règle. Cela peut paraître surprenant, mais finalement se trouve être une bonne idée (pensez-y !).

Il ne reste plus qu'à passer la règle dans VirusTotal pour faire du *hunting* et du *retrohunt*. Nous avons obtenu ainsi 42 nouveaux samples qui n'étaient pas dans le jeu de données initial. Une remarque : nous avons eu 0 faux positif.

Il est évident que nous nous passons sûrement à côté de nouvelles familles, mais rappelons que le but premier n'est pas d'être exhaustif, mais d'avoir une règle Yara représentative de la famille qui nous intéresse.

CONCLUSION

La similarité des malware nous permet de générer des règles Yara plus performantes et mieux ciblées à nos recherches. Il faut évidemment bien connaître son jeu de données et avoir en tête certaines limites (*packers*, *false flags*...). Cela va permettre de définir des vecteurs de caractéristiques décrivant au mieux ce jeu de données. Une fois cela fait, les algorithmes font le reste (ajoutons une précision de taille : une fois choisis les « bons » hyperparamètres!), car ils ont été pensés pour et reprennent l'ensemble des techniques que nous avons détaillées. Dans la plupart des cas, il est suffisant d'en rester aux propriétés des fichiers sans descendre jusqu'aux similarités de code. La similarité de code va être inintéressante pour déterminer les recoupements entre les différentes familles ou des bibliothèques utilisées (cas des bibliothèques de cryptographie), mais peut être intéressante dans d'autres problématiques comme la phylogénie de malware. ■

REMERCIEMENTS

Merci à la fine équipe de Zone de confort et au LSE de l'EPITA.



Les références de cet article sont disponibles sur : <https://www.miscmag.com/>.

CONSULTEZ



MISC

EN NUMÉRIQUE !



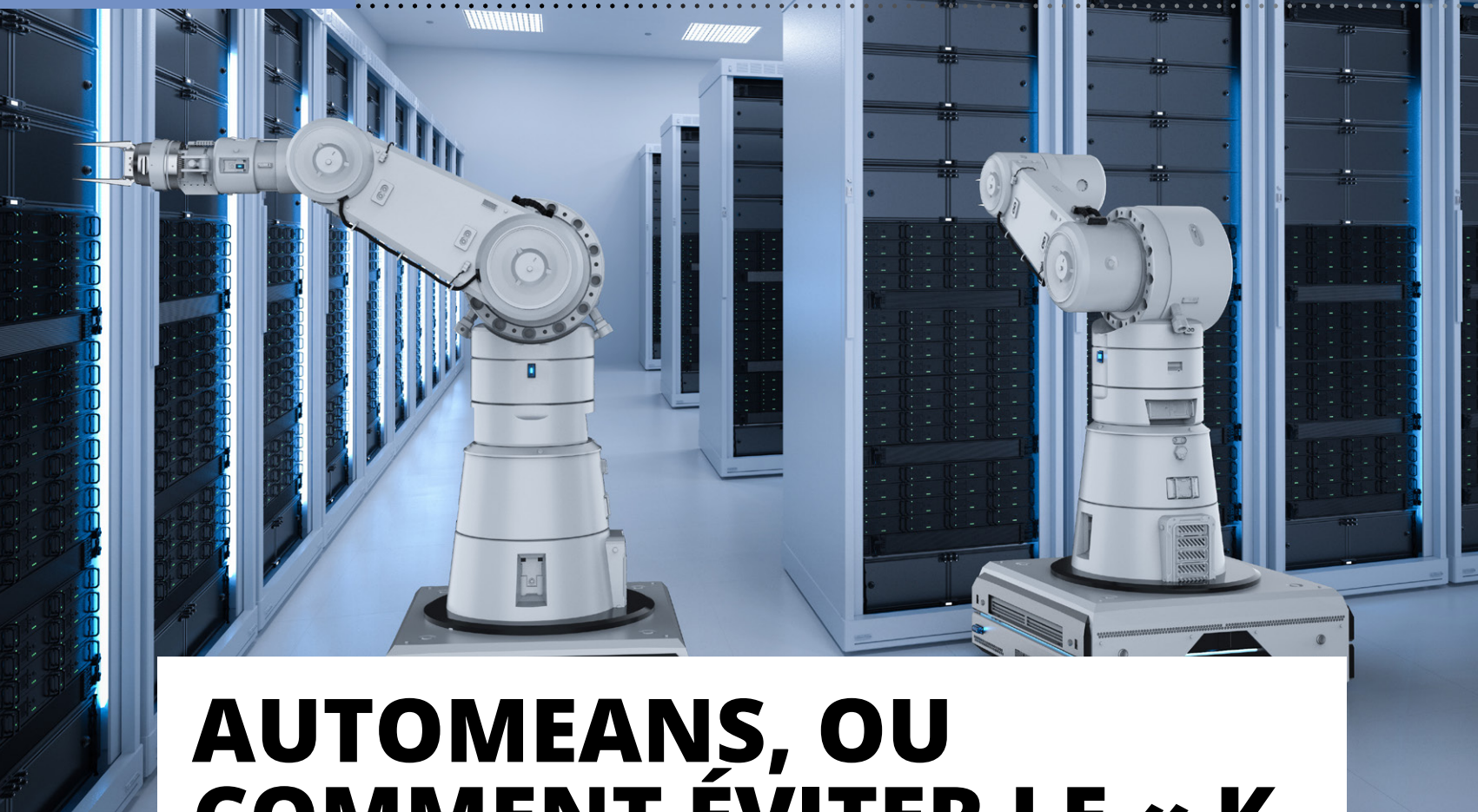
...CELUI D'AUJOURD'HUI ET CEUX D'HIER...

...LE BIMESTRIEL ET LES HORS-SÉRIES !

RENDEZ-VOUS SUR :

connect.ed-diamond.com

À PARTIR DE 239 € TTC (TARIF FRANCE MÉTRO.)



AUTOMEANS, OU COMMENT ÉVITER LE « K PAR K » AVEC K-MEANS

Alexandre LETOIS

Nous voulons (par exemple) classer un jeu de données très grand, disons quelques millions de données, voire quelques milliards, mais non labellisées. Face au problème, l'algorithme K-means semblait un bon candidat. Il s'agit d'un des grands classiques des algorithmes du Machine Learning qui comporte cependant un défaut : il est nécessaire de lui donner k , le nombre de clusters (c'est ce qu'on appelle un « hyperparamètre »). Quand cette valeur optimale est inconnue, il est coûteux de trouver une bonne approximation, surtout sur un très grand jeu de données. Pour pallier à ce problème, nous avons développé un nouvel algorithme : Automeans, cet algorithme calcule le nombre « optimal » de clusters.

L'utilisation du Machine Learning dans le cadre de la sécurité n'est pas nouvelle, l'article [1] date de 2011. Désormais, ces techniques sont étudiées pour repérer des attaques, détecter des malware, identifier des familles de malware, etc. Voir l'article de S. Larinier dans ce hors-série.

Notre étude s'est faite dans le cadre de la classification de malware. De nombreux travaux ont déjà été effectués dans ce domaine, voir [8] pour des références. Pour notre étude, nous voulons nous placer dans un cadre le plus proche possible d'une étude réelle. Nous nous plaçons donc à large échelle (millions de

fichiers) : nous sommes face à un grand ensemble de malware inconnus et non labellisés.

L'objectif est de faire un prétraitement de ce large ensemble de malware pour avoir une première partition rapidement sans toutefois négliger la qualité. De cette partition, on peut récupérer des grandes familles ou approfondir notre analyse. Des grandes familles peuvent facilement être déterminées en analysant en profondeur les éléments les plus représentatifs de chaque cluster (le centre du cluster, par exemple, va contenir les éléments les plus typiques). Sinon, en prenant à part chaque cluster trouvé et en les soumettant à une étude plus approfondie (éventuellement personnalisée en fonction de chaque cluster), on peut obtenir au final une partition précise en contournant le problème d'un jeu de données trop grand.

N'ayant pas connaissance des malware, nous ne pouvons pas nous baser sur de l'apprentissage supervisé, ce qui nous force à utiliser des techniques non supervisées. Dans la suite de cet article, nous allons expliquer notre méthode, ainsi que nos motivations pour procéder ainsi.

CHOIX DES ARMES

Algorithmes non supervisés

Nécessité fait loi : nous utiliserons un algorithme non supervisé. Cela nous permet de ne pas dépendre d'un jeu de données labellisées d'apprentissage qui ne sera pas forcément représentatif du jeu que nous allons traiter (et que nous n'avons pas en général). Nous évitons ainsi le *surapprentissage* ou le risque qu'un nouveau type de malware soit intégré « de force » par l'algorithme dans un cluster de la famille entraînée la plus proche.

Ce choix, nécessaire, nous limite forcément dans les outils qui sont à notre disposition.

Nous avons expérimenté différents algorithmes disponibles dans la littérature afin de voir quel outil serait le plus adéquat. L'autre contrainte qui se dresse sur notre chemin est le *passage à l'échelle*. En effet, il faut que notre algorithme soit très rapide pour qu'il puisse analyser rapidement le grand ensemble que nous lui donnons. Nous avons aussi testé une autre stratégie, présentée à la figure 1, qui sera la base sur laquelle reposeront les éventuelles analyses plus en profondeur. C'est une stratégie classique de type « Diviser Pour Régner » : on fait une 1ère passe grossière pour obtenir des clusters grands, avec un algorithme qui est capable de gérer un très grand

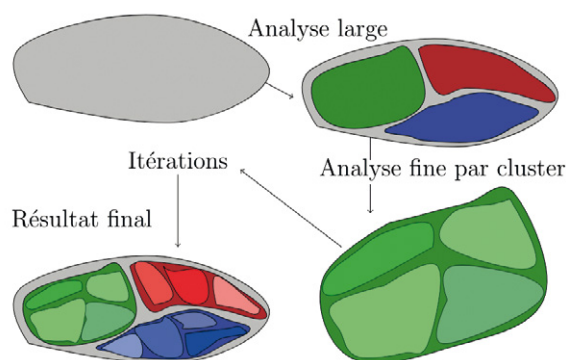


Fig. 1 : Schéma de notre stratégie pour récupérer une partition de notre grand jeu de données.

nombre de données comme K-means et ensuite, chaque cluster est « redécoupé » à l'aide d'un autre algorithme qui lui ne passe pas à l'échelle, mais est capable de traiter ces clusters, comme DBSCAN par exemple.

De nombreux algorithmes ont été mis à l'épreuve : K-means, clustering spectral, DBSCAN, mélange gaussien, clustering hiérarchique de Ward, etc. Pour ces algorithmes, nous avons utilisé l'implémentation donnée par la bibliothèque Scikit-Learn [6]. Nous nous sommes concentrés sur les plus rapides pour s'assurer qu'elles puissent être utilisées sur notre grand jeu de données.

K-means, DBSCAN et Ward

Nous nous sommes penchés plus précisément sur K-means [2], DBSCAN et Ward. Ces algorithmes sont très utilisés et méritent que nous nous attardions sur eux. Nous avons fait des expériences sur des jeux de données factices pour vérifier leurs performances, ainsi que sur un petit jeu de données, « theZoo », pour nous assurer de leur efficacité face à des malware.

Pour représenter ces jeux de données, on utilise des vecteurs de « features », ou vecteurs de caractéristiques. Chaque malware de notre jeu de données va être représenté par un de ces vecteurs qui va contenir des valeurs spécifiques permettant (si possible) d'identifier de manière unique ce malware. L'extraction de ces caractéristiques peut se faire de multiples façons et de nombreux paramètres peuvent être exploités ou non. Cependant, ceci est un autre problème et nous allons considérer pour les algorithmes à suivre que notre jeu de données est représenté par une matrice X, contenant les n malware sous forme de vecteur de caractéristiques, voir par exemple le vecteur calculé par le parser [7]. Chaque algorithme reçoit cette matrice X en valeur d'entrée, plus d'éventuels paramètres en fonction de l'algorithme.

» K-means

K-means est un grand classique de la clusterisation (*clustering*) qui se base sur une fonction coût, l'inertie, qu'on cherche à minimiser. Il nécessite que l'utilisateur ne donne qu'un seul paramètre : k , le nombre de clusters désiré. Le concept est simple : l'algorithme place au départ k points aléatoirement (des techniques permettent de choisir des bons points de départ et non se référer uniquement au hasard, cela donne la variante k-means++ de Scikit) dans l'espace qui vont être considérés comme le centre de chacun des k clusters. Chaque point est ensuite affilié au cluster du centre le plus proche. Ensuite, chaque centre de cluster est recalculé en fonction des points appartenant au cluster. Ces opérations sont répétées jusqu'à arriver à un état stable (la fonction qualité ou fonction coût ne change plus). C'est un problème d'optimisation numérique, voir l'article sur le tour d'horizon.

Cette solution est assez rapide (par exemple par rapport à Ward), d'autant plus quand la valeur de k est petite, car chaque point du jeu de données se compare uniquement à chaque centre pour voir celui qui lui est le plus proche. Cependant, une mauvaise valeur de k peut réduire drastiquement son efficacité.

» DBSCAN

DBSCAN est un autre type d'algorithme qui va se baser sur la densité des clusters. Il faut lui donner deux (hyper)paramètres : la taille du voisinage et le nombre minimum de voisins pour créer un cluster. Avec DBSCAN, pas besoin de donner la valeur de k , les clusters vont se trouver d'eux-mêmes. Le concept est également simple : un point

du jeu de données est choisi et on regarde les points de son entourage, appelés « voisins ». Si un point a un nombre de voisins supérieur ou égal à la valeur entrée en paramètre, on va considérer qu'il fait partie d'un cluster ; dans le cas contraire, il s'agit d'un élément de bruit et il est étiqueté comme tel (il y a un cluster de bruit). Si le point a assez de voisins, alors ceux-ci sont considérés comme faisant partie du même cluster que lui. La même opération va se faire sur les voisins qui vont permettre au cluster de se propager de proche en proche. Quand le cluster ne se propage plus, on recommence l'opération initiale avec un nouveau point non assigné à un cluster,

Cette technique est efficace avec de bonnes valeurs pour la taille et le rayon du voisinage. Elle est également assez rapide, bien que moins que K-means.

» Ward

La méthode de clustering hiérarchique de Ward est un algorithme hiérarchique. C'est-à-dire qu'au départ de l'algorithme, chaque point est considéré comme un cluster mono-élément. De ce point de départ, les clusters vont être fusionnés pour minimiser une fonction de qualité (basée sur l'inertie comme K-means). Les clusters vont continuer à fusionner jusqu'à atteindre le nombre de clusters entrés en paramètre par l'utilisateur.

Cette technique supporte moins bien les grandes échelles que les algorithmes précédents et partage le handicap de K-means qui nécessite une valeur pour k .

LES LIMITATIONS DE K-MEANS

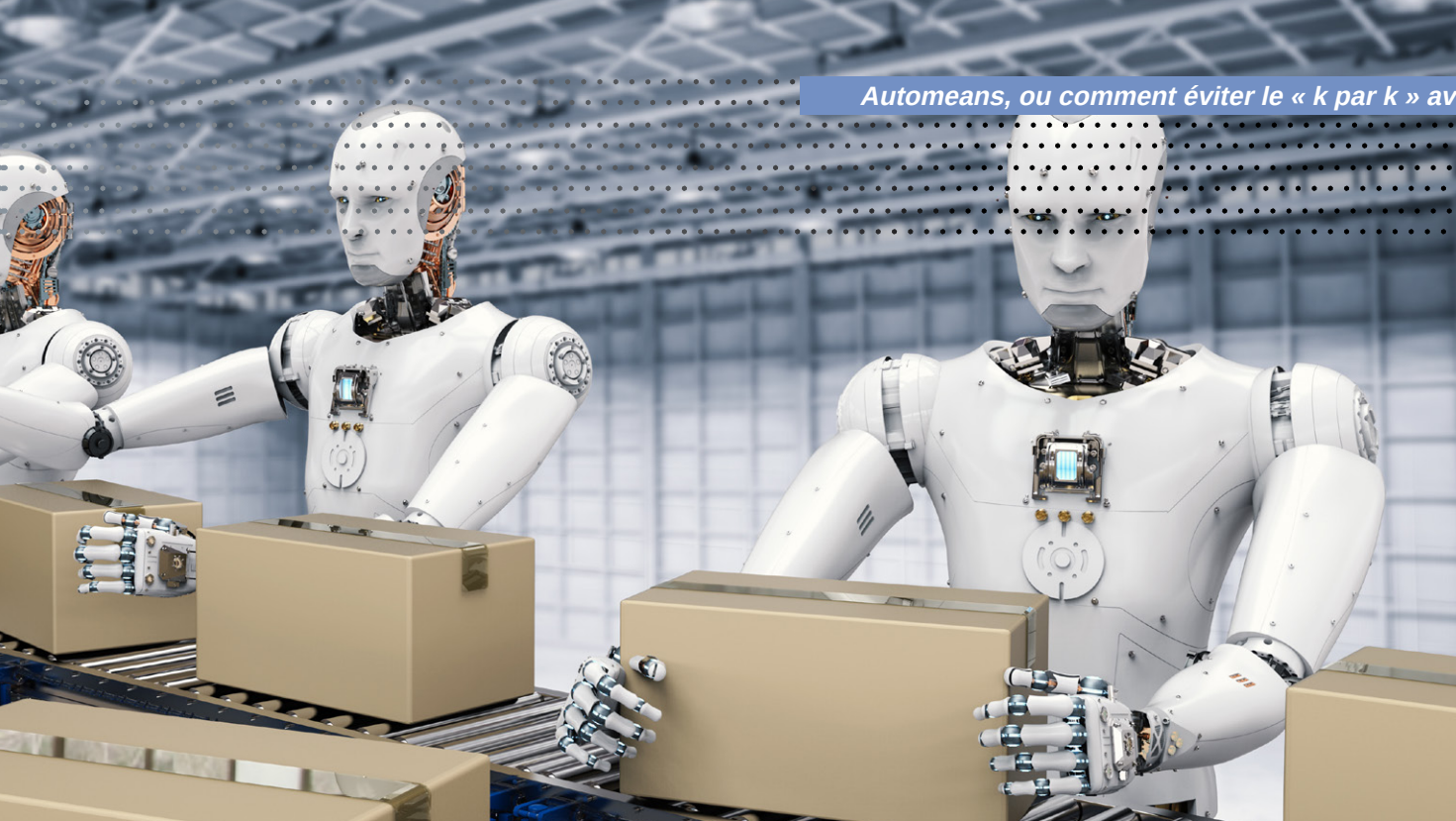
Nous sommes prêts à utiliser K-means en tant que premier algorithme sur notre grand jeu de données. Il passe à l'échelle, mais un problème persiste : quelle valeur doit être choisie pour k ?

En effet, n'ayant aucune idée de la composition de cet ensemble de malware, impossible d'avoir ne serait-ce qu'une estimation de la valeur de k . Et nous avons déjà dit dans la section 1.2.1 qu'une mauvaise valeur pour k va mener à de mauvais résultats. Il est donc nécessaire de trouver un moyen d'estimer k .

La méthode du coude

Il existe différentes méthodes permettant d'estimer une bonne valeur de k en comparant les résultats d'un algorithme lancé en changeant le paramètre. Une étude statistique est alors utilisée avec les différents k pour déterminer la valeur optimale.





La méthode du coude (*Elbow Method*), par exemple, va se baser sur la variance pour trouver la valeur de k qui sera placée au niveau du « coude » de la courbe. Comme on peut le voir sur la figure 2, il y a un moment où l'inclinaison de la courbe cesse brutalement, créant le fameux coude.

Le souci principal de ce type de méthode est que l'étude va prendre du temps. En effet, si on veut avoir une étude complète, il va falloir lancer n fois un K-means pour pouvoir comparer le résultat obtenu avec chaque valeur de k .

Il est bien entendu possible d'accélérer la méthode en rognant les petites et les grandes valeurs (en prenant le risque de passer à côté de l'optimum si ce dernier est placé près des bords) ou en réutilisant les valeurs des précédentes itérations de K-means pour accélérer les suivantes.

Ces méthodes ne sont pas entièrement fiables et peuvent tout simplement échouer. Il arrive que la méthode utilisée n'arrive pas à démarquer une valeur optimale de k ou, pire, qu'elle trouve une valeur fautive (c'est le cas si la courbe de la méthode du coude est très régulière).

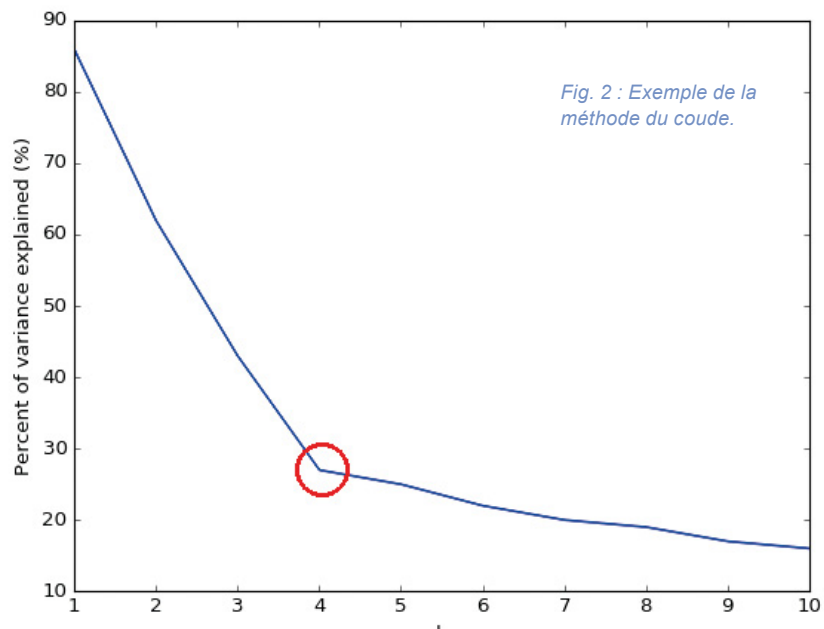


Fig. 2 : Exemple de la méthode du coude.

Ces méthodes ne semblent donc pas optimales. Il va être nécessaire de trouver une solution différente pour gérer le problème de k .

X-means

X-means [3] est un autre algorithme connu du domaine. Il s'agit d'un algorithme inspiré de K-means qui n'a pas besoin d'une valeur pour k , mais d'un intervalle de valeurs où k va pouvoir se situer.

La littérature suggère que l'algorithme est très rapide et semble avoir d'excellents résultats.

Nous décidons donc de tester cet algorithme pour notre problème. N'ayant toujours aucune idée d'une valeur pour k , nous choisissons comme intervalle une valeur minimale à $k=1$ et une valeur maximale à $k=n$.

Malheureusement, les résultats ne sont pas à la hauteur avec nos expériences. X-means n'a pas l'air de réussir à manier un intervalle aussi grand et retourne toujours une partition composée de n clusters mono-éléments.

X-means étant également non fonctionnel pour nos besoins, mais utile dans d'autres situations, il devient nécessaire de trouver une autre solution.

UN K-MEANS « AUTOMATISÉ » : AUTOMEANS

N'ayant à disposition que des méthodes statistiques du style de la méthode du coude qui va nécessiter de lancer de nombreux K-means, k par k , nous avons tenté de trouver une meilleure alternative.

Ayant un bagage sur la recherche de communautés dans un graphe, nous connaissons l'algorithme de Louvain [4]. C'est un algorithme qui permet de trouver des communautés dans un graphe, très rapidement, sans avoir besoin de connaître à l'avance le nombre de communautés, se basant sur le voisinage de chaque point pour éviter d'avoir à faire des comparaisons nœud à nœud avec tout le graphe. C'est un algorithme de type vorace (*greedy*).

Avec ce concept en tête, nous avons décidé de créer notre propre algorithme qui va lui aussi trouver des clusters sans avoir besoin de lui donner une valeur pour k à l'avance.

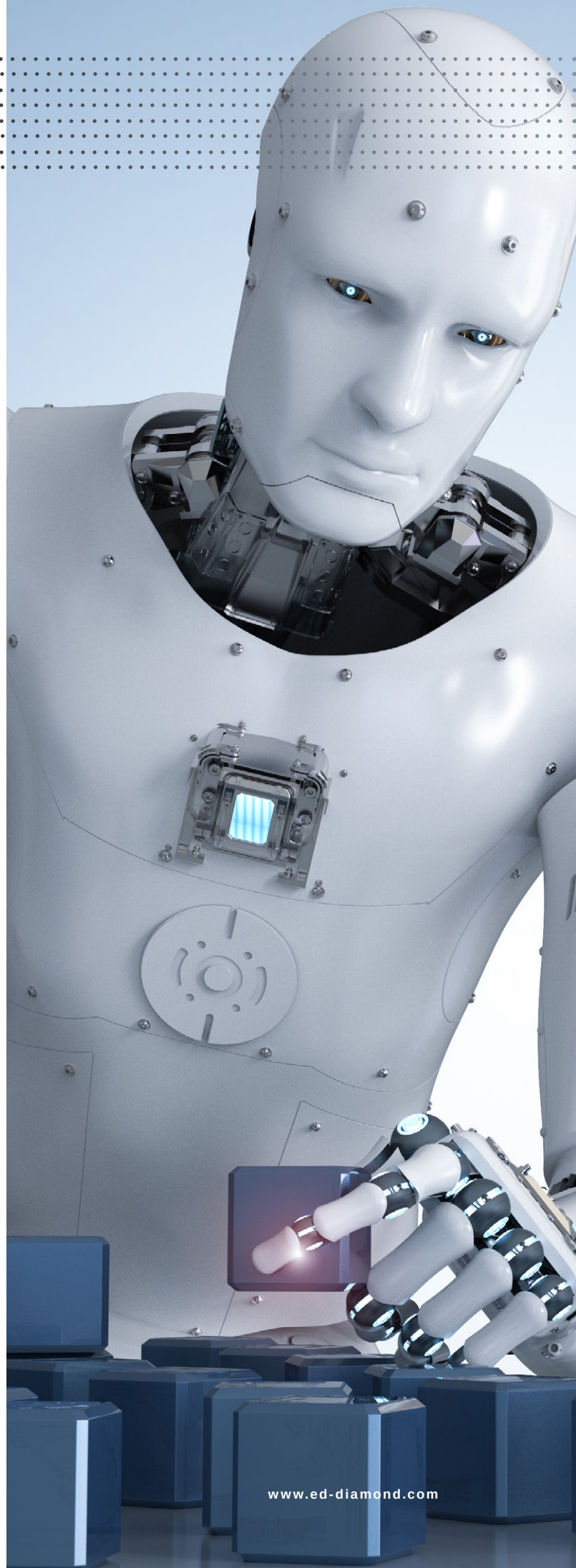
L'idée est de faire un prétraitement de nos données permettant d'accélérer la suite.

En utilisant des techniques de *nearest-neighbors* pour trouver le voisinage de nos points (ou même des valeurs approchées pour aller plus vite) on va pouvoir éviter d'avoir à faire des comparaisons de points 2 à 2 sur tout le jeu de données.

On part sur une base d'algorithme de clustering hiérarchique avec un cluster mono-élément pour chacun de nos points. Ensuite, on va chercher à minimiser une fonction de qualité en fusionnant les clusters voisins ensemble.

La fonction de qualité est inspirée de l'inertie, tout comme K-means, pour pouvoir obtenir des résultats semblables. Cependant, certaines modifications sont nécessaires pour pouvoir avoir une fonction dynamique qui puisse adapter la valeur de k au fur et à mesure que l'algorithme tourne.

Après nos expériences nous avons pu constater que Automeans obtient de bons résultats, rapidement. La valeur de k estimée est proche de la valeur réelle, ainsi que la partition obtenue.



Automeans nous permet donc de récupérer, rapidement, une bonne partition sur un grand jeu de données. L'objectif que nous visions est donc atteint et nous pouvons entrer plus dans les détails. Lors de la création d'Automeans, plusieurs choix ont été faits qui pourraient être modifiés, comme nous le verrons plus tard dans cette section. Certains éléments de l'algorithme peuvent tout à fait être modifiés en fonction des préférences de l'utilisateur, de son type de données ou si ce dernier a des informations sur le jeu de données qui peuvent lui permettre de préférer une option par rapport à une autre.

Voisinage

Automeans fonctionne, car le voisinage est calculé au début de l'algorithme pour permettre d'éviter d'avoir à faire des comparaisons avec les points trop éloignés de chaque point. N'importe quel algorithme ou structure permettant de récupérer un voisinage peut être utilisé pour cette étape. L'important est de récupérer rapidement ce voisinage pour diminuer au maximum le temps que l'algorithme va prendre. Pour nos expériences, nous avons utilisé une structure de BallTree qui donne un voisinage en $O(n \log(n))$.



NOTE

Le temps de calcul du voisinage est compté lors du calcul du temps d'exécution d'Automeans.

Il faut choisir une taille de voisinage, que nous avons fixée, après expérimentations, à 3. C'est une valeur petite, ce qui permet de diminuer le temps nécessaire à cette étape de calcul de voisinage. Comme on peut le voir dans la section 3.3, au fur et à mesure que l'algorithme va s'exécuter, les voisinages vont s'étendre petit à petit, donc le 4e ou 5e plus proche voisin vont naturellement finir par être rencontrés de toute façon. On s'évite ainsi des calculs inutiles.

Il est également possible d'utiliser des approximations de voisinage plutôt qu'un voisinage exact. Cela permettrait d'utiliser des algorithmes plus rapides et le fait d'avoir une approximation plutôt qu'un voisinage exact. Pour la même raison que celle citée au-dessus, ne pas avoir le bon voisinage avec certitude ne devrait pas être un problème, car les véritables voisins finiront de toute façon par être rencontrés pendant la phase d'agrégation de l'algorithme.

Fonction objectif : une heuristique pour adapter l'inertie

L'inertie est la fonction de qualité sur laquelle K-means se base. Nous voulons rester proches de K-means, c'est pourquoi nous avons décidé de garder une fonction de qualité proche de l'inertie.

L'inertie est une formule qui prend en compte de la distance entre les points faisant partie d'un même cluster (nommé le « *Within-Cluster Sum of Square* » ou WCSS) ainsi que de la distance entre les clusters (nous la

nommons « *Between-Cluster Sum of Square* » ou BCSS). L'inertie est la somme de ces deux éléments.

Pour les formules de la figure 3, on considère une partition P d'un jeu de données X , composée de k clusters appelés p_1, p_2 , etc. L'inertie est notée I .

Pour avoir une bonne partition, on désire avoir des clusters bien séparés du reste (haute valeur pour le BCSS) ainsi que des clusters ayant des éléments très proches (faible valeur pour le WCSS).

K-means cherche à minimiser le WCSS. Seulement, la minimisation du WCSS est triviale, il suffit de prendre $k=n$, on se retrouve donc avec n clusters mono-éléments et une valeur nulle pour le WCSS étant donné que chaque point est à une distance nulle de tous les autres points de son cluster. C'est pourquoi il est nécessaire dans le cadre de K-means de fixer la valeur de k . Ainsi, on évite de retomber sur une partition triviale de mauvaise qualité. L'algorithme va donc chercher à minimiser WCSS, pour la valeur de k donnée.

Pour pouvoir avoir un algorithme qui va adapter dynamiquement k , nous devons donc modifier la formule de l'inertie. Nous avons gardé la formule du WCSS et avons modifié le BCSS, dépendant de k , par une heuristique. L'objectif est que l'heuristique donne une bonne valeur approchée du BCSS dans le cas optimal.

$$\begin{aligned}
 W_{CSS}(P) &= \sum_{p \in P} \sum_{x \in p} \|x - \bar{p}\|_2^2 \\
 B_{CSS}(P) &= \sum_{p \in P} |p| \|\bar{p} - \bar{X}\|_2^2 \\
 \forall P, I(X) &= W_{CSS}(P) + B_{CSS}(P)
 \end{aligned}$$

Fig. 3 : Formules de l'inertie.

Nous avons expérimenté de nombreuses formules différentes pour le BCSS que nous avons ensuite mis à l'épreuve dans de nombreux tests. Ces formules ont d'abord été théorisées en se basant sur le BCSS, afin de s'assurer d'une cohérence, avant de lancer les calculs.

L'ensemble de ces expériences a été fait sur des jeux de données factices qui ont été créés grâce à la fonction `make_blob` de scikit-learn. Différents types de jeux ont été créés afin de s'assurer que les exemples soient pertinents dans un maximum de cas possibles. Quelques exemples de créations de jeux de données sont visibles avec le code ci-dessous.

```
def create_uniform_dataset(k, n, d, seed):
    actual_centers = random_centers(k, d, seed)
    return make_blobs(n_samples=n, centers=actual_centers,
n_features=d, random_state=seed)
#
def create_power_dataset(k, n, d, seed):
    np.random.seed(seed=seed)
    sizes = np.random.power(5., size=(k,))
    sizes *= n/sum(sizes)
    sizes = list(map(lambda f: int(np.round(f)), sizes))
    return create_custom_sized_dataset(k, n, d, sizes, seed)
```

Un des résultats obtenus est montré en figure 4, on y voit la différence de résultats obtenus avec les multiples heuristiques testées.

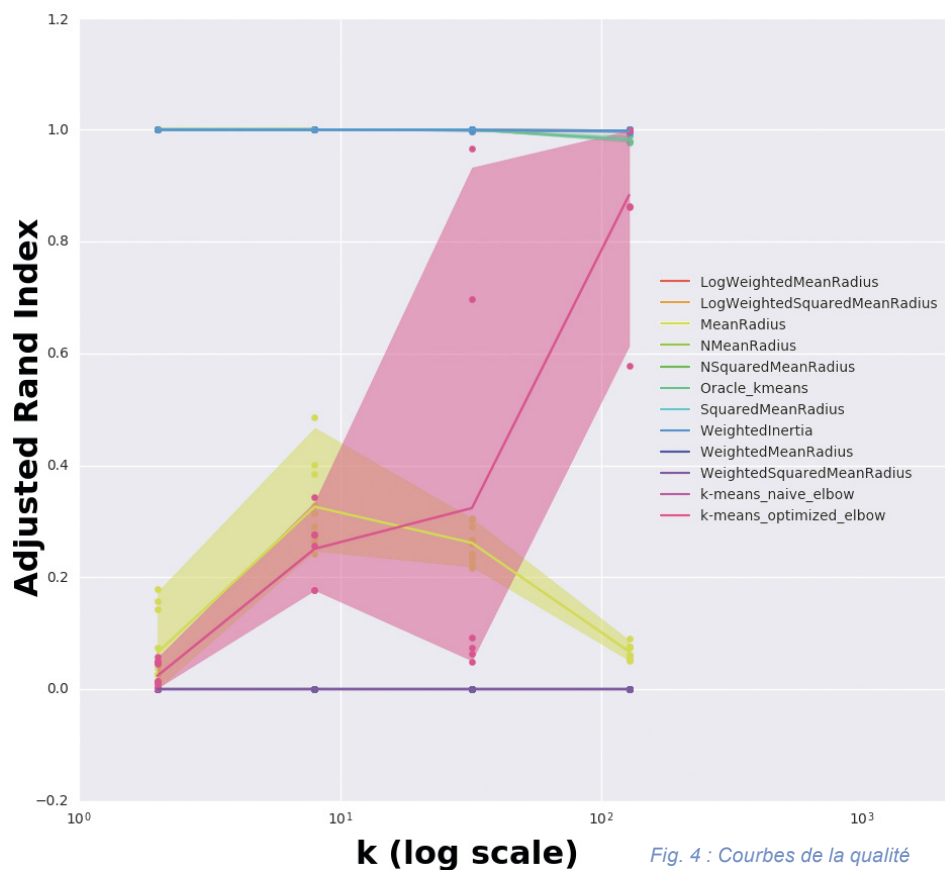


Fig. 4 : Courbes de la qualité de différentes heuristiques en fonction de k .



De ces expériences, une heuristique s'est démarquée du reste, visible sur la figure 3, où k est le nombre de clusters dans la partition P au moment du calcul et $\bar{R}$ est le « rayon moyen » du jeu de données. Le rayon moyen est la distance moyenne, sur tout le jeu de données, entre un point de l'ensemble et le centre. Ce rayon est calculé une seule fois au début de l'algorithme et ne varie plus ensuite.

On peut voir qu'elle est assez proche de la valeur du BCSS et les bornes sont d'ailleurs identiques entre les deux formules : minimum quand $k = 1$ et maximum quand $k = n$.

Notre fonction de qualité retenue, que nous avons nommée « Automated WCSS » (AWCSS) se calcule donc grâce à la formule de la figure 5.

$$AW_{CSS}(P) = W_{CSS}(P) + k\bar{R}(X)^2$$

Fig. 5 : Formule de la fonction de qualité AWCSS.

Fonctionnement d'Automeans

L'algorithme est composé de deux grandes étapes :

- Une phase d'agrégation.
- Une phase de réduction.

La phase d'agrégation consiste, pour chaque point, à regarder ses voisins et de tenter de les agréger dans leur cluster. Le choix d'incorporer un point ou non dans le nouveau cluster est décidé par rapport à une augmentation ou une diminution de la fonction de qualité. Comme nous désirons trouver la valeur minimale pour notre fonction, une diminution de la valeur entraînera l'incorporation au cluster. Pour éviter d'avoir à recalculer les valeurs d'AWCSS à chaque fois, on compare uniquement la différence entre la valeur d'AWCSS avant et après la modification : $\Delta AWCSS$. Cette valeur est beaucoup plus rapide à calculer, car seuls les deux clusters concernés sont pris en compte, le reste du jeu de données ne bougeant pas pendant cette opération. Ainsi, si on a un $\Delta AWCSS$ positif, on ne va pas garder la modification, et réciproquement. Si plusieurs voisins diminuent la fonction de qualité, le voisin offrant la plus grande différence est intégré. Cette phase continue jusqu'à arriver à un état stable où les clusters ne bougent plus. On lance alors la phase de réduction.

La phase de réduction consiste à réduire chaque cluster en un unique point, placé au centre du cluster. On

se retrouve alors avec un nouveau jeu de données, plus petit.

On réitère classiquement ces deux phases jusqu'à

l'obtention d'un état stable, même après réduction, c'est-

à-dire que la fonction coût reste « stable » sur plusieurs itérations ou change de manière minime. Le voisinage est recalculé après chaque phase de réduction, mais seul le premier calcul est vraiment conséquent, car sur un grand jeu de données.

Une fois obtenu une dernière partition stable, on regarde les clusters attribués pour chaque point et on réaffecte ce même cluster à tous les points du cluster dont ce point représente le centre (après la phase de réduction). On continue cette opération jusqu'à revenir au jeu de données de base, dont tous les points sont répartis en une nouvelle partition.

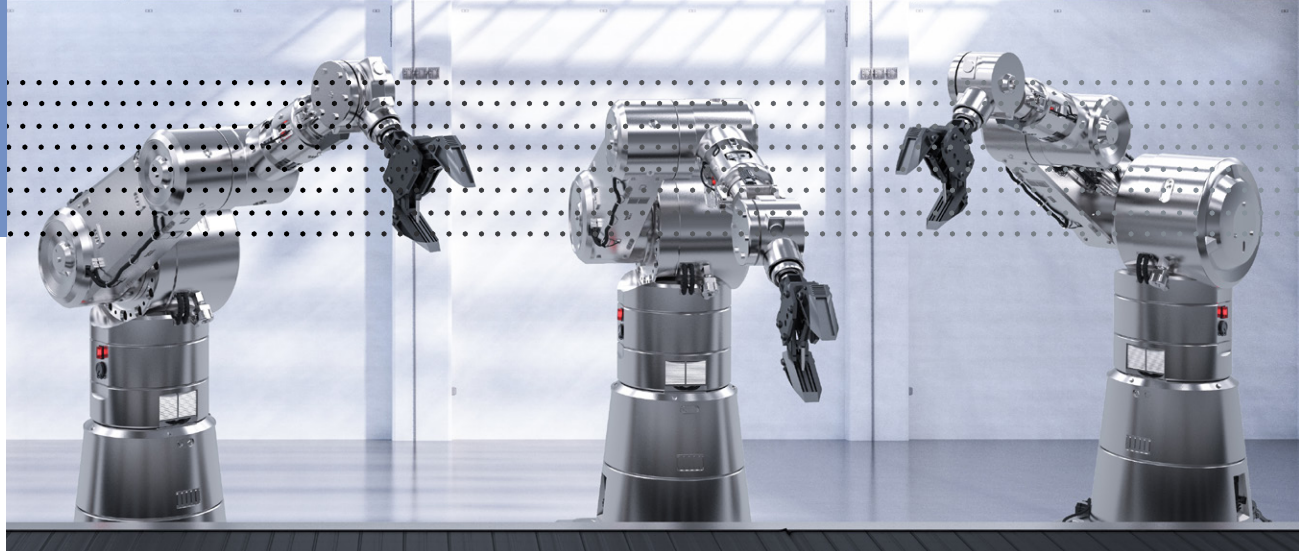
Forces d'Automeans

Automeans est un nouvel algorithme non supervisé. Sa principale force est qu'il peut être lancé sur un jeu de données sans avoir la moindre information à son sujet. Aucun hyperparamètre n'est nécessaire par l'utilisateur qui peut uniquement le lancer en boîte noire et regarder ensuite les résultats. Il passe également à l'échelle (*scalability*), lui permettant d'être utilisé en premier algorithme de « défrichage » sur un gros ensemble avant de passer à des algorithmes plus précis pour avoir une analyse plus fine.

Automeans a été comparé à une version « oracle » de K-means, à qui on donnait la valeur exacte de k ainsi qu'à une méthode du coude optimisée. Automeans est plus lent que K-means (ce qui est normal), mais plus rapide que la méthode du coude. De plus, les résultats sont nettement meilleurs avec Automeans qu'avec la méthode du coude. Les partitions obtenues sont proches de celles obtenues par le K-means « oracle » et la valeur de k est également proche de la valeur attendue.

Un autre point fort d'Automeans est que son temps de calcul n'augmente pas quand k augmente, contrairement à K-means. En effet, plus k est grand, moins il sera long de trouver sa valeur avec la technique d'Automeans. Cette propriété rend Automeans d'autant plus appréciable à très grande échelle où le nombre de clusters peut devenir également très grand. Précisons que le problème à l'origine de ce travail était de clusteriser un ensemble inconnu de près de 2 millions de malware, avec un vecteur de caractéristiques de plus d'un million de malware, calculés avec PetoJson, développé par S. Larinier [7].

Une autre force de cet algorithme est qu'il peut être modifié pour s'adapter aux besoins de l'utilisateur. Par exemple, la possibilité de modifier la structure permettant le calcul de voisinage a déjà été énoncée auparavant. Il est également possible de modifier la taille du voisinage ou l'heuristique que nous utilisons si nécessaire, le concept de l'algorithme reste le même. Cependant, il vaut mieux éviter de faire trop de modifications sans avoir une bonne raison, soit au moins une certaine connaissance sur son jeu de données pour s'assurer que les résultats finaux auront un sens.



Nous avons remarqué qu'Automeans n'est pas très efficace à très petites échelles. Cela n'est néanmoins pas un problème, car Automeans n'a d'intérêt qu'à une échelle suffisante. En effet, dans le cas contraire, lancer quelques itérations de K-means n'est pas un problème.

CONCLUSION

Nous avons montré une méthode permettant de faire une analyse d'un grand ensemble de malware.

Notre technique peut d'ailleurs s'adapter à tous types de jeux de données du même genre : c'est-à-dire un grand ensemble d'éléments sur lesquels nous n'avons pas ou peu d'informations.

L'utilisation d'Automeans permet de récupérer rapidement une partition de bonne qualité sans avoir à connaître des paramètres souvent cruciaux comme le nombre de clusters ou la taille idéale du voisinage des clusters.

Nous avons trouvé une fonction de qualité qui donne des résultats proches de ceux d'un K-means idéal en un temps raisonnable. Cette technique est plus rapide que des techniques comme la méthode du coude et trouve également des partitions plus proches de la vérité que celles estimées par ces méthodes.

Automeans apporte une plus-value en comparaison des algorithmes de la littérature. Néanmoins, l'utilisation

d'Automeans ne va pas forcément pour autant donner une partition parfaite et il peut être très intéressant de continuer d'étudier chaque cluster séparément pour avoir une étude plus fine de notre ensemble.

Le code en Python de l'algorithme sera mis en ligne asap, il devrait être disponible quand vous lirez cet article. Le lecteur curieux pourra sans problème l'essayer sur le jeu de données « theZoo » pour comparer avec les résultats présentés dans ce hors-série l'article de S. Larinier, avec les vecteurs de caractéristiques de malware calculés avec le parser PetoJson [7]. ■

REMERCIEMENTS

L'auteur souhaite remercier la DGA/MI qui a financé une partie de ces recherches dans le cadre de ViralStudio, un projet RAPID en collaboration avec SEKOIA. Remerciements également à Marc Espie pour son aide, ainsi qu'à Sébastien Larinier pour son parser de PE PetoJson [7] et Paul Rascagnères qui a mis des malware à notre disposition.

RÉFÉRENCES

- [1] K. Rieck, P. Trinius, C. Willems, T. Holz, *Automatic analysis of malware behavior using Machine Learning*, Journal of Computer Security, Volume 19 Issue 4, décembre, p. 639-668, 2011.
- [2] A. K. Jain, *Data clustering: 50 years beyond K-means*, Pattern Recognition Letters 31, p. 651-666, 2010.
- [3] D. Pelleg et A. W. Moore, *X-means : Extending k-means with efficient estimation of the number of clusters*, ICML, p. 727-734, 2000.
- [4] V. D Blondel, J.-L. Guillaume, R. Lambiotte, E. Lefebvre, *Fast unfolding of communities in large networks*, Journal of Statistical Mechanics: Theory and Experiment (10), 2008.
- [5] <https://github.com/ytisf/theZoo/>
- [6] <http://scikit-learn.org/stable/>
- [7] <https://github.com/sebdraven/petojson>
- [8] <https://github.com/lse/>
- [9] C. Chio et D. Freeman, *Machine Learning & Security, protecting systems with data and algorithms*, O'Reilly, 2018.

ACTUELLEMENT DISPONIBLE

LINUX PRATIQUE N°110



ÉCHAPPEZ À LA CENSURE DU NET !

NE LE MANQUEZ PAS
CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :
<https://www.ed-diamond.com>





ANALYSE DES SENTIMENTS AVEC LE DEEP LEARNING

Abdessalam BOUCHEKIF

Les progrès réalisés cette dernière décennie en Deep Learning ont profité à plusieurs domaines, ce qui lui a permis de s'imposer comme une approche incontournable en Machine Learning. La classification automatique des documents textuels est l'une des tâches dans laquelle le Deep Learning s'avère particulièrement utile et performant. Dans cet article, nous allons présenter en détail les aspects théoriques et pratiques liés à l'implémentation d'un système d'analyse des sentiments. Il s'agit de la classification automatique de tweets en trois classes : positive, négative ou neutre.

INTRODUCTION

Plusieurs entreprises s'intéressent aux avis des clients sur leurs produits et services. L'analyse des sentiments est une approche permettant de détecter automatiquement les émotions des clients à partir des commentaires postés sur les réseaux sociaux. Le but du travail présenté dans cet article est d'implémenter un réseau de neurones profond pour détecter automatiquement la polarité d'un tweet. À partir des données étiquetées/labellisées (c'est-à-dire que chaque tweet est accompagné d'une étiquette : positive, négative ou neutre), l'apprentissage d'un réseau de neurones permet de construire un modèle (une fonction mathématique) qui servira par la suite à prédire de nouvelles données (i.e. des tweets non vus lors de la phase d'apprentissage). Nous utilisons le corpus US Twitter Airline disponible sur le lien [1].

La figure 1 donne un aperçu global du système d'analyse des sentiments que nous allons implémenter avec la librairie Keras [2]. Nous commençons par une étape de préparation de données qui consiste à :

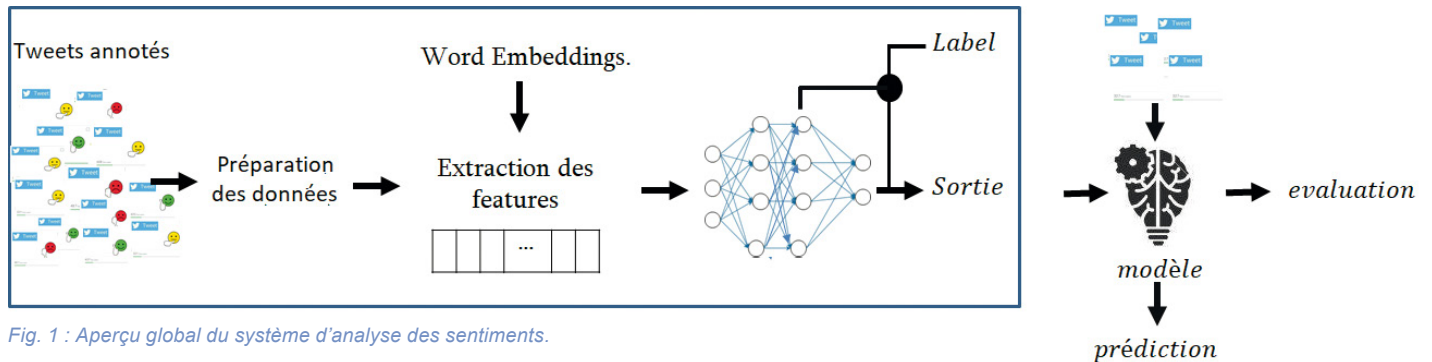


Fig. 1 : Aperçu global du système d'analyse des sentiments.

- Supprimer les informations non importantes.
- Normaliser les données.

Ensuite, nous transformons les tweets en vecteurs numériques représentant au mieux le contenu des messages (étape d'extraction des features). Puis, nous lançons l'apprentissage automatique qui a comme but de minimiser l'erreur commise sur l'ensemble des données. Pour cela, nous utilisons un algorithme d'optimisation comme la descente de gradient stochastique ou Adam. Enfin, nous mesurons la performance du système en utilisant un corpus de test (c'est-à-dire les tweets non vus durant la phase d'apprentissage). L'évaluation consiste à comparer les étiquettes retournées par le système avec les étiquettes de référence et à compter le nombre de fois où le système a réussi à affecter la bonne classe aux tweets.

PRÉPARATION DE DONNÉES

À partir des mots présents dans le tweet, le système décidera s'il s'agit d'un avis positif, négatif ou neutre. Deux mots n'ayant pas la même forme écrite sont considérés comme différents. C'est le cas par exemple des mots 'épita' et 'Epita'.

La préparation des données textuelles (prétraitement) consiste à segmenter les tweets en mots (tokenisation), convertir tous les mots en minuscule et supprimer les informations parasites telles les URLs, les dates, les pseudos et les mots non porteurs de sens (e.g. le, de, ce etc.). Néanmoins, les mots de négation comme 'ne' et 'pas' ont été retenus, car la négation dans une phrase peut inverser le sentiment du tweet. Les smileys donnent aussi des informations supplémentaires sur les émotions de l'internaute. De ce fait, nous regroupons les smileys ayant une signification commune (tristesse, joie, colère, etc.). Les mots d'une langue donnée sont accordés en genre, en nombre et en mode (indicatif, impératif...). Le regroupement des différentes formes d'un mot est connu sous le nom de « lemmatisation », il permet de retourner la forme canonique des mots (i.e. chaque verbe sera ramené à son infinitif et les autres mots au masculin singulier). Nous utilisons l'outil NLTK [3] et les expressions régulières (via la bibliothèque 're' du python) pour effectuer l'ensemble des prétraitements. Par exemple dans la phrase « je déteste ce film » on lemmatise les mots, ce qui donne ['je', 'détester', 'ce', 'film'], puis on supprime les mots 'je' et 'ce', ce qui donne ['détester', 'film'].

Pour installer NLTK :

```
$ sudo pip install nltk
```

Lancez python et tapez en lignes de commandes :

```
>>> import nltk
>>> nltk.download()
```

L'outil NLTK contient une liste de stopwords (non porteurs de sens) par langue, le code ci-dessous permet de faire le chargement des stopwords :

```
from nltk.corpus import stopwords
stopwords.words('english')
```

Il faut ensuite garder les mots utiles pour la tâche d'analyse des sentiments (comme les adverbes de négation). Nous écartons la ponctuation (à l'exception du '!' et '?'), les e-mails et les noms des utilisateurs :

```
from string import punctuation
import re
notstopwords = set(('not', 'no'))
```



```
stopwords = set(nltk.corpus.stopwords.words('english')) - notstopwords
punctuation = set(punctuation) - set(('?', '!'))
tweet = re.sub(r"(http|https)?://[a-zA-Z0-9\.-]+\.[a-zA-Z]{2,4}(\S*)?", " ", tweet)
tweet = re.sub(r"@w+", r' ', tweet)
```

Le corpus contient plusieurs attributs (colonnes), nous avons besoin uniquement des mots du tweet (la colonne **text**) et le sentiment exprimé (la colonne **airline_sentiment**). Nous utilisons la fonction **read_csv** de la bibliothèque pandas pour lire le corpus et sélectionner uniquement que les informations dont nous avons besoin.

```
tweets = pd.read_csv(path_tweets, encoding='utf-8', sep=',')
```

Pour découper les tweets en mots, nous utilisons la classe TweetTokenizer de NLTK.

```
from nltk.tokenize import TweetTokenizer
tknzs = TweetTokenizer()
tknzs.tokenize(tweet)
```

Avant de procéder à la phase de lemmatisation, il est intéressant de connaître la catégorie grammaticale du mot. Par exemple, le mot « meeting » peut être un nom ou un verbe, cela dépend de son contexte. Sans spécifier la catégorie grammaticale du mot, l'outil NLTK lemmatise le mot comme étant un nom.

```
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()
>>> lemmatizer.lemmatize("meeting")
'meeting'
>>> lemmatizer.lemmatize("meeting", pos='v')
'meet'
>>> lemmatizer.lemmatize("meeting", pos='n')
'meeting'
```

Le code ci-dessous effectue l'ensemble des prétraitements.

```
from nltkcorpus import stopwords
from nltk.tokenize import TweetTokenizer
from nltk import word_tokenize, pos_tag
from nltk.stem import WordNetLemmatizer
from string import punctuation
import nltk
import pandas as pd
import re

lemmatizer = WordNetLemmatizer()
tknzs = TweetTokenizer()
punctuation = set(punctuation) - set(('?', '!'))
notstopwords = set(('not', 'no'))
stopwords = set(nltk.corpus.stopwords.words('english')) - notstopwords

def data_preprocessing(path_tweets):
    tweets = pd.read_csv(path_tweets, encoding='utf-8', sep=',')
    tweets['text'] = tweets['text'].apply(lambda x: standardization(x))
    tweets['sentiment'] = tweets['airline_sentiment']
    apply(lambda x: 0 if x=='negative' else (1 if x=='neutral' else 2))
    return tweets['text'], tweets['sentiment']

def standardization(tweet):
    print(tweet)
    tweet = tweet.lower()
    # emoticons -> meaning https://en.wikipedia.org/wiki/List\_of\_emoticons
    tweet = re.sub(r"\sd(-)?(')?(:|;|=|x|<)?", ' sadness ', tweet)
    tweet = re.sub(r">?:-?(\(|@|c|<|\[|\{)", ' angry ', tweet)
```

```

tweet = re.sub(r'(:|8|x|b|=) (-|\^)?((d)|(\|))| (=3) ', ' happy ', tweet)
tweet = re.sub(r"\s(:|-|o|c|\^)?(\|)|3|>|\})|(8-?)| (=|\|))", ' happy ', tweet)
tweet = re.sub(r"(http|https)?:\|/[a-zA-Z0-9\.-]+\.[a-zA-Z]{2,4}(/\S*)?", " ", tweet)
tweet = re.sub(r"'ve", " have", tweet)
tweet = re.sub(r"can't", " cannot", tweet)
tweet = re.sub(r"n't", " not", tweet)
tweet = re.sub(r"'re", " are", tweet)
tweet = re.sub(r"'d", " would", tweet)
tweet = re.sub(r"'ll", " will", tweet)
tweet = re.sub(r"'s", "", tweet)
tweet = re.sub(r"'n", "", tweet)
tweet = re.sub(r"'m", " am", tweet)
tweet = re.sub(r"@w+", r' ', tweet)
tweet = re.sub(r"#w+", r' ', tweet)
tweet = re.sub(r"[0-9]+ ", " ", tweet)
tweet = [lemmatizer.lemmatize(i,j[0].lower()) if j[0].lower() in ['a','n','v']
          else lemmatizer.lemmatize(i) for i,j in pos_tag(tknzr.tokenize(tweet))]
tweet = [i for i in tweet if ((i not in stopwords) and (i not in punctuation))]
tweet = ' '.join(tweet)
return tweet
data_preprocessing("USAirlineSentiment.csv" )

```

42

EXTRACTION DES « FEATURES »

Chaque mot est représenté par un vecteur de valeurs numériques. Supposons que V soit la taille du vocabulaire (i.e. le nombre de mots différents figurant dans le corpus). Avec la représentation *One-hot*, chaque mot est un vecteur de taille V ne contenant que des 0, à l'exception de la position du mot dans le vocabulaire où la valeur est égale à 1. Exemple : soient les t_1 , t_2 et t_3 trois tweets constituant notre corpus.

t_1 : je déteste ce film ; t_2 : très bon film ; t_3 : j'ai bien aimé ce film ;

- Vocabulaire après les prétraitements : ['détester', 'film', 'bon', 'aimer'] ($V=4$)
- Représentation One-hot des mots : détester = [1,0,0,0] film = [0,1,0,0] bon = [0,0,1,0] aimer = [0,0,0,1]

Cette représentation a deux inconvénients majeurs :

- La dimension des vecteurs peut devenir très large, car une langue peut contenir beaucoup de mots.
- La similarité entre les mots s'écrivant différemment est toujours égale à 0.

La représentation *Word Embedding* est une méthode d'extraction automatique, elle permet de retourner des vecteurs denses (i.e. moins de zéro) et capturant la sémantique des mots avec une faible dimensionnalité par rapport à One-hot. Deux mots sont considérés comme similaires si leurs représentations vectorielles sont proches dans le même espace continu. Plusieurs outils sont disponibles pour calculer les *Word Embeddings*. Parmi les outils plus utilisés, on peut citer Glove [4] et Word2vec [5]. Ce dernier utilise un corpus volumineux de données textuelles pour apprendre un réseau de neurones à une seule couche cachée.

Nous utilisons des vecteurs de mots de 300 dimensions entraînés à partir d'articles de presse ; le fichier des vecteurs est disponible sur le site [6].

Nous utilisons la fonction `load_word2vec_format` de la librairie Gensim [7] pour lire le *Word2vec* :

```
from gensim.models import KeyedVectors
model = KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative300.bin', binary=True)
print (len(model.vocab)) # Afficher la taille du vocabulaire
print (model['france']) # Afficher le vecteur du mot France
```

La fonction `most_similar` permet alors de retourner les n mots les plus proches sémantiquement, le code :

```
model.most_similar(['france'], topn = 6)
```

renvoie

```
[('spain', 0.6375303864479065), ('french', 0.6326056122779846), ('germany',
0.6314355134963989), ('europe', 0.626425564289093), ('italy',
0.6257959008216858), ('england', 0.6120776534080505)]
```

Tandis que le code :

```
model.most_similar(['love'], topn = 6)
```

renverra :

```
[('loved', 0.6907792091369629), ('adore', 0.6816873550415039), ('loves', 0.661863386631012),
('passion', 0.6100709438323975), ('hate', 0.600395679473877), ('loving', 0.5886635780334473)]
```

Nous utilisons la classe `Tokenizer` de Keras pour transformer nos tweets en des séquences d'entiers où chaque entier est l'indice d'un token dans le vocabulaire :

```
tokenizer = Tokenizer(filters='')
tokenizer.fit_on_texts(tweets)
sequences = tokenizer.texts_to_sequences(tweets)
```

Le nombre de neurones de la couche d'entrée correspond au nombre de mots présents dans les tweets. En revanche, les tweets n'ont pas forcément la même taille. Nous utilisons la fonction `pad_sequences` pour donner aux tweets la même taille en remplissant 0 au début de chaque séquence jusqu'à ce que la longueur de chaque séquence corresponde à la longueur de la séquence la plus longue :

```
from collections import Counter
from preprocessing import standardization, data_preprocessing
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np
from gensim.models import KeyedVectors
from keras.layers.embeddings import Embedding
```



```

from keras.models import Sequential, Model
from keras.layers import LSTM, Dropout, Dense, Bidirectional, Dropout
np.random.seed(10)
EMBEDDING_FILE="GoogleNews-vectors-negative300.bin"
EMBEDDING_DIM=300
corpora="data/USAirlineSentiment.csv"
tweets, sentiments=data_preprocessing(corpora)
tokenizer = Tokenizer(filters='')
tokenizer.fit_on_texts(tweets)
sequences = tokenizer.texts_to_sequences(tweets)
word_index = tokenizer.word_index
MAX_SEQUENCE_LENGTH = 0
for elt in sequences:
    if len(elt) > MAX_SEQUENCE_LENGTH:
        MAX_SEQUENCE_LENGTH = len(elt)
print(MAX_SEQUENCE_LENGTH)
data = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH)
indices = np.arange(data.shape[0])
data = tweets[indices]
labels = sentiments[indices]
nb_words=len(word_index)+1
embedding_matrix = np.zeros((nb_words, EMBEDDING_DIM))
word2vec = KeyedVectors.load_word2vec_format(EMBEDDING_FILE, binary=True)
embedding_matrix = np.zeros((nb_words, EMBEDDING_DIM))
for word, i in word_index.items():
    if word in word2vec.vocab:
        embedding_matrix[i] = word2vec.word_vec(word)
embedding_layer = Embedding(len(word_index) + 1, EMBEDDING_DIM, weights=[embedding_matrix],
                             input_length=MAX_SEQUENCE_LENGTH, trainable=False)

```

APPRENTISSAGE AUTOMATIQUE

Il existe plusieurs types de réseaux de neurones suivant la façon dont les neurones sont connectés entre eux. Nous avons choisi d'implémenter deux modèles basés sur des concepts différents. Le premier modèle est basé sur une architecture multi-couches et le deuxième un réseau de neurones récurrents de type LSTM (*Long Short-Term Memory*) [8].

Réseaux de neurones multi-couches

Un réseau de neurones multi-couches (voir la figure 2) comporte une couche d'entrée, une ou plusieurs couches cachées et une couche de sortie. Chaque neurone est lié à tous les autres neurones de la couche précédente.

Chaque neurone (sauf ceux de la couche d'entrée) calcule la somme pondérée des entrées :

$$z = b + \sum_{i=1}^n x_i w_i$$

où b est le biais (cercle noir) qui peut être vu comme un neurone qui a toujours +1 comme valeur de sortie. Une fonction d'activation f sera appliquée à la somme

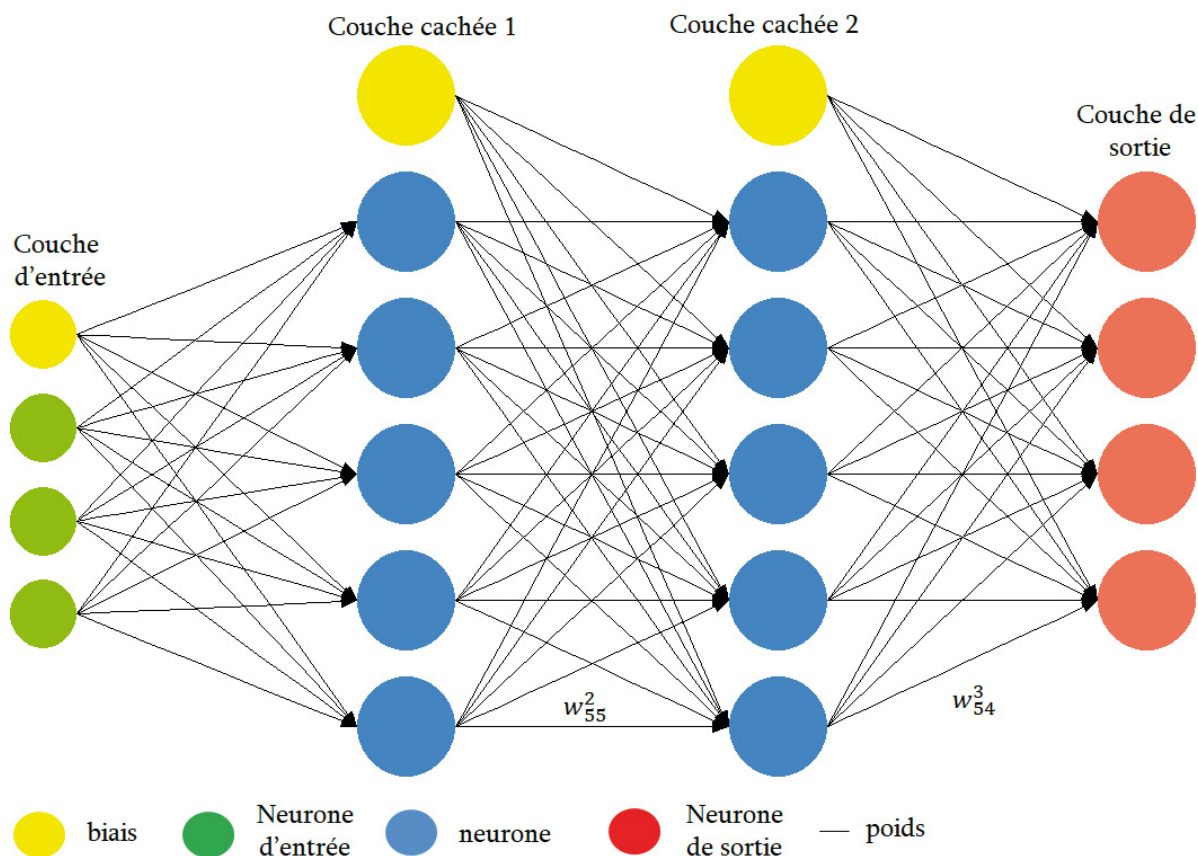


Fig. 2 : Un exemple de réseau de neurones multi-couches.

pondérée z afin de produire la sortie du neurone, voir [9] pour plus d'explications. Les fonctions d'activation les plus connues sont définies par :

- ReLU (pour Rectified Linear Unit) : $\text{ReLU}(x) = \max(0, x)$
- Sigmoide (ou logistique ou encore « marche douce ») :

$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

- et Tanh :

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

Un seul neurone n'est pas en mesure d'effectuer des tâches complexes. En revanche, l'utilisation de plusieurs

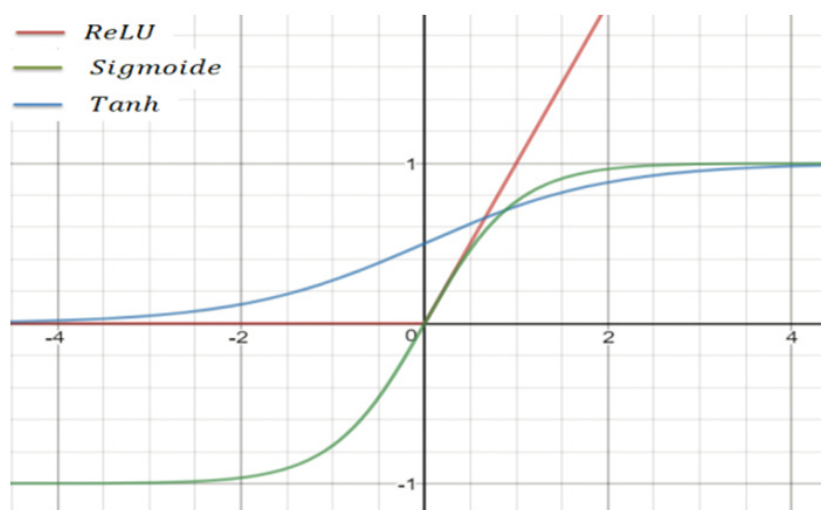


Fig. 3 : Principales fonctions d'activation

neurones interconnectés donne souvent lieu à de bonnes performances. Le comportement du réseau de neurones est déterminé par son architecture (i.e. le nombre de couches cachées, le nombre de neurones dans les couches cachées

et la manière d'interconnexion entre les neurones) et les poids affectés à chacune des connexions. On parle de Deep Learning lorsque le nombre de couches cachées est supérieur à deux. Le poids w_{ij}^l est la connexion entre le i ème neurone de la couche l et le j ème neurone de la couche $l+1$, le b^l est le biais de la couche cachée l .

Le réseau apprend

L'apprentissage d'un réseau de neurones consiste à trouver les meilleurs paramètres w_{ij}^l et les biais b^l qui minimisent le taux d'erreur sur les données de validation. L'apprentissage d'un réseau de neurones s'effectue via l'algorithme de la rétropropagation du gradient. Il consiste à modifier les poids de telle sorte qu'on trouve pour chaque entrée (un tweet dans notre cas d'application) sa sortie correspondante (positive, négative ou neutre). D'abord on propage vers l'avant les entrées jusqu'à obtenir une sortie (prédiction) calculée par le réseau. Ensuite, on calcule l'erreur commise entre la sortie réelle et la sortie désirée. Pour cela, nous utilisons une fonction d'erreur tout au long de l'apprentissage pour calculer l'erreur commise par le réseau de neurones. Enfin, on rétropropage l'erreur de la couche de sortie vers l'arrière jusqu'à la couche d'entrée tout en modifiant les poids des neurones. Une grande valeur de l'erreur implique un ajustement plus important des poids et des biais lors de la rétropropagation. L'apprentissage peut s'effectuer en présentant au réseau de neurones soit un seul exemple (i.e. un tweet) soit un ensemble d'exemples (batch) à la fois. Avec l'apprentissage par batch, l'erreur est calculée pour tous les exemples sans mettre à jour les poids. La rétropropagation est lancée

une fois que tous les exemples du batch ont effectué un passage de la couche d'entrée vers la couche de sortie. Présenter une seule fois tous les exemples à notre réseau ne suffit pas pour apprendre un réseau de neurones profond. Une propagation et une rétropropagation sur l'ensemble d'exemples d'apprentissage s'appellent une époque (*epoch*). Nous allons alors lancer l'apprentissage pour plusieurs époques.

L'apprentissage automatique a comme but de construire un modèle capable de faire le moins d'erreurs possible sur de nouvelles données. En effet, il peut arriver que notre réseau fasse très peu d'erreurs sur les données d'apprentissage, alors qu'il est mauvais sur d'autres données. Ce phénomène est connu sous le nom de sur-apprentissage (*overfitting*). Pour détecter ce dernier, on fait de la validation croisée, nous séparons les données en deux sous-ensembles : l'ensemble d'apprentissage et l'ensemble de validation. Le premier sert à apprendre (i.e. faire évoluer les poids du réseau de neurones). Le plus souvent on lance l'apprentissage plusieurs fois en faisant varier les valeurs des hyperparamètres (i.e. le nombre de couches cachées, les fonctions d'activation, etc.). Après chaque apprentissage, nous vérifions la pertinence du réseau avec des données non utilisées durant l'apprentissage, c'est justement le rôle de l'ensemble de validation.

Sur-apprentissage et régularisation Dropout

Le sur-apprentissage survient quand l'erreur de prédiction du réseau diminue sur l'ensemble d'apprentissage alors que l'erreur sur les données de validation augmente. Cela signifie que le réseau est en train de se spécialiser sur l'ensemble d'apprentissage et de perdre sa capacité de généralisation sur de nouvelles données (l'ensemble de validation). L'arrêt prématuré (*early stopping*) est une technique permettant de surveiller le problème de sur-apprentissage en interrompant l'apprentissage du réseau lorsque l'erreur sur l'ensemble d'apprentissage continue à diminuer et l'erreur sur l'ensemble de validation atteint le minimum (voir la figure 6). Un ensemble de tests est aussi réservé pour estimer l'erreur de généralisation du modèle sur de nouvelles données avant que le modèle ne soit mis en production. Nous séparons nos données en trois ensembles : 80% pour l'apprentissage, 10% pour la validation et 10% pour le test), ce qui donne comme code :

```
split_idx = int(len(data)*0.8)
x_train, x_val = data[:split_idx], data[split_idx:]
y_train, y_val = labels[:split_idx], labels[split_idx:]
test_idx = int(len(x_val)*0.5)
x_val, x_test = x_val[:test_idx], x_val[test_idx:]
y_val, y_test = y_val[:test_idx], y_val[test_idx:]
```

Le *Dropout* est l'une des techniques de régularisation utilisées pour éviter le sur-apprentissage des réseaux de neurones profonds en intégrant une ou plusieurs couches *dropout*, voir [9]. Une couche *dropout* ne contient pas de neurones et n'ajoute aucun paramètre au réseau, son rôle est d'éteindre temporairement (i.e. pour chaque entraînement) certains neurones de la couche précédente. En appliquant le *Dropout*, chaque neurone de la couche précédente a une probabilité p (indépendante des autres neurones) d'être temporairement éteint. La technique du *Dropout* est applicable uniquement pendant la phase

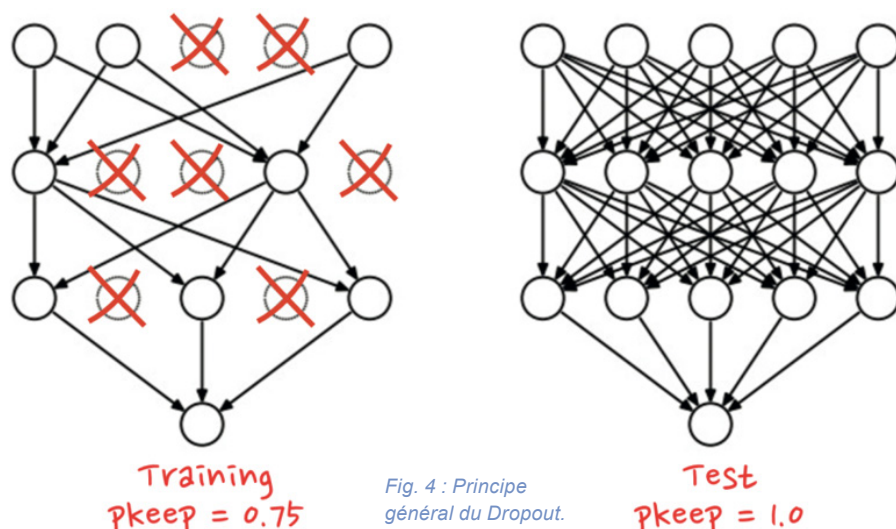


Fig. 4 : Principe général du Dropout.

d'apprentissage. Autrement dit, p vaut 0 pendant la phase test. Seuls les neurones des couches cachées peuvent être « éteints » et donc, puisque chaque neurone peut être présent ou absent, il existe 2^n réseaux possibles (où n est le nombre de neurones « éteignables »). Le *dropout* permet d'avoir des poids associés aux neurones qui sont moins corrélés entre eux et dont la pertinence individuelle est renforcée, car le réseau a été entraîné sur plusieurs variations présentant des neurones éteints (forçant ainsi les neurones restés allumés à être pertinents).

Le code Keras

Keras permet de considérer un réseau de neurones comme une pile linéaire de couches de neurones (la *Functional API* de Keras permet de définir des architectures de réseaux librement). La classe **Sequential** permet de créer une structure vide à partir de laquelle nous pouvons ajouter des couches denses en faisant appel à méthode **add()**.

```
sequence_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')
embedded_sequences = embedding_layer(sequence_input)
model1 = Sequential()
model1.add(embedding_layer)
model1.add(Flatten())
model1.add(Dense(3, activation='softmax'))
```

Le modèle a besoin de savoir la forme des données d'entrée à laquelle il doit s'attendre. Cette information est transmise à la première couche du modèle séquentiel. Softmax est une fonction d'activation qui retourne les probabilités que l'exemple appartienne à chaque classe.

Une fois que nous avons défini le modèle, nous devons le compiler pour qu'il puisse être exécuté.

```
model.compile(loss='categorical_crossentropy', optimizer='Adam', metrics=['acc'])
```



Layer (type)	Output Shape	Param #
embedding_layer (Embedding)	(None, 26, 300)	3090900
lstm_1 (LSTM)	(None, 64)	93440
dropout_1 (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 32)	2080
dropout_2 (Dropout)	(None, 32)	0
dense_2 (Dense)	(None, 3)	99
Total params: 3,186,519		
Trainable params: 95,619		
Non-trainable params: 3,090,900		

Train on 11712 samples, validate on 1464 samples

```
Epoch 1/8
11712/11712 [=====] - 8s 672us/step - loss: 0.7465 - acc: 0.6867 - val_loss: 0.5435 - val_acc: 0.7937
Epoch 2/8
11712/11712 [=====] - 6s 536us/step - loss: 0.5670 - acc: 0.7792 - val_loss: 0.5409 - val_acc: 0.7978
Epoch 3/8
11712/11712 [=====] - 6s 538us/step - loss: 0.5377 - acc: 0.7866 - val_loss: 0.5064 - val_acc: 0.8074
Epoch 4/8
11712/11712 [=====] - 6s 539us/step - loss: 0.5166 - acc: 0.7958 - val_loss: 0.5099 - val_acc: 0.8074
Epoch 5/8
11712/11712 [=====] - 6s 542us/step - loss: 0.5006 - acc: 0.8008 - val_loss: 0.5045 - val_acc: 0.8122
Epoch 6/8
11712/11712 [=====] - 6s 544us/step - loss: 0.4864 - acc: 0.8072 - val_loss: 0.5126 - val_acc: 0.8101
Epoch 7/8
11712/11712 [=====] - 6s 539us/step - loss: 0.4673 - acc: 0.8159 - val_loss: 0.5103 - val_acc: 0.8142
Epoch 8/8
11712/11712 [=====] - 6s 543us/step - loss: 0.4509 - acc: 0.8206 - val_loss: 0.5116 - val_acc: 0.8204
1464/1464 [=====] - 0s 258us/step
Test accuracy: 0.849043718941
```

Fig. 5 : Architecture du réseau de neurones multi-couches, obtenue avec `model.summary()` suivie des performances du modèle LSTM.

La méthode `compile()` est utilisée pour spécifier les hyperparamètres de l'apprentissage (la fonction de perte, l'algorithme d'optimisation) ainsi que la métrique d'évaluation. Nous devons spécifier :

- La fonction de coût (*loss function*) qu'on cherche à minimiser. Nous utilisons la fonction entropie croisée (`categorical_crossentropy`) adaptée à la classification multi-classes.
- L'algorithme d'optimisation permettant de mettre à jour les poids durant la phase d'apprentissage. Les optimiseurs les plus répandus sont SGD, RMSProp et Adam.
- La métrique qui est utilisée pour calculer le taux de précision (*accuracy*).

Pour afficher l'architecture du réseau de neurones, il suffit d'ajouter la ligne de code suivante : `model.summary()` qui renvoie ce qu'on peut voir en haut de la figure 5.

Une fois le modèle compilé, il peut être exécuté en faisant appel à la fonction `fit()` qui prend comme paramètres :

- les données d'apprentissage,

- les données de validation,
- le nombre d'époques,
- et la taille du batch.

À la fin de l'apprentissage, nous traçons les courbes du taux d'erreur en apprentissage et en validation (voir la figure 6). Pour cela, nous utilisons la bibliothèque matplotlib. Les meilleures performances sur le corpus de validation ont été enregistrées à la 10e époque (avec une précision égale à 80.0%), au-delà le réseau commence à sur-apprendre. Ce qui donne le code :

```
history=model1.fit(x_train,y_train,
                  validation_data=(x_val, y_val),
                  epochs=20,batch_size=100)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

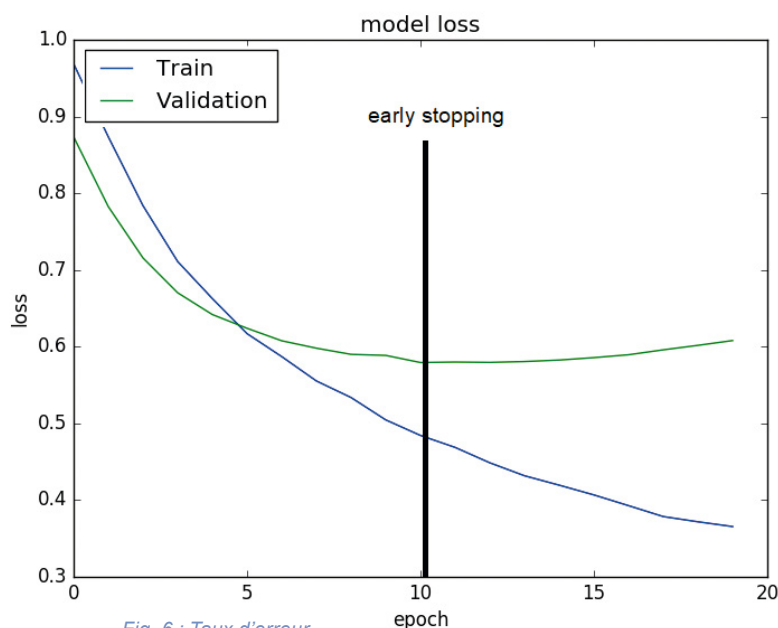


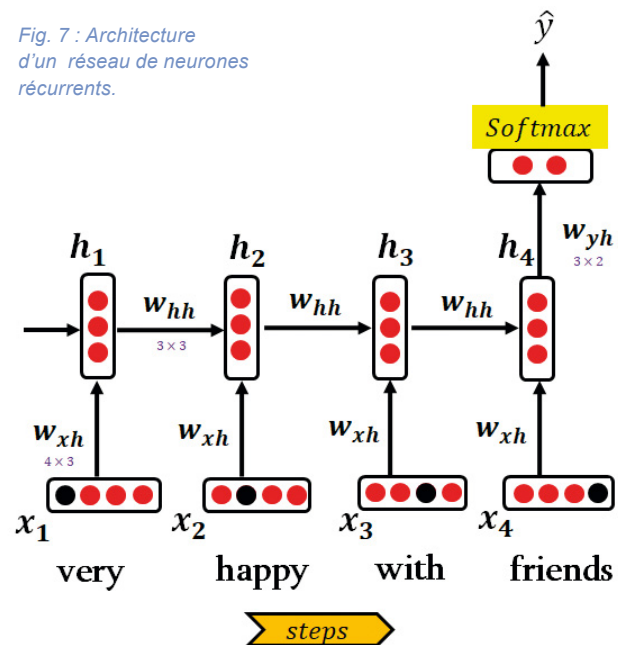
Fig. 6 : Taux d'erreur en apprentissage et en validation.

Réseaux de neurones récurrents

Les réseaux de neurones multi-couches supposent que toutes les entrées sont indépendantes les unes des autres, c'est-à-dire qu'ils ne prennent pas en compte



Fig. 7 : Architecture d'un réseau de neurones récurrents.



l'ordre des entrées (dans notre cas, les mots). En revanche, les réseaux de neurones récurrents (RNN, *Recurrent Neural Network*) traitent les données au fur et à mesure, tout en respectant l'ordre des entrées. Les RNNs ont été principalement conçus pour modéliser des séquences et sont capables de mémoriser des informations passées.

Un RNN simple (voir la figure 7) à l'instant t observe l'entrée $x^{(t)}$ (un mot dans un tweet) et met à jour la sortie de la couche cachée $h^{(t)}$ en prenant en compte le vecteur $h^{(t-1)}$ calculé à l'étape précédente. Du fait que chaque neurone conserve l'information des étapes précédentes, il est qualifié de cellule de mémoire.

La sortie de la couche à l'étape temporelle t est donnée dans l'équation suivante : $h^{(t)} = f(W_{xh} x^{(t)} + W_{hh} h^{(t-1)} + b_h)$

où :

- f est la fonction d'activation choisie.
- $x^{(t)} \in \mathbb{R}^d$ est le vecteur du mot à l'instant t et b_h le biais.
- $W_{xh} \in \mathbb{R}^{d, n_h}$ est la matrice des poids entre l'entrée $x^{(t)}$ et la couche cachée h sachant que n_h est le nombre de neurones dans la couche cachée.
- $W_{hh} \in \mathbb{R}^{n_h, n_h}$ est la matrice des poids entre la couche cachée et la sortie sachant que n_h est le nombre de neurones dans la couche de sortie.
- $W_{hh} \in \mathbb{R}^{n_h, n_h}$ est la matrice des poids entre la sortie des couches cachées courante et précédente.

Le RNN simple souffre du problème de la disparition des premières entrées en mémoire, qui vient principalement de la taille de la séquence. Prenons le tweet « j'ai adoré les trains de la ligne L, il manque juste un peu de nettoyage de temps en temps ». Si le RNN perd de l'information sur les premiers mots, le système pourra considérer le message comme ayant une polarité négative. Un LSTM est un réseau de neurones récurrent capable d'apprendre des dépendances à long terme. Chaque neurone LSTM est liée, non seulement à $h^{(t-1)}$, mais également à un état de la cellule c de l'étape précédente qui joue le rôle de mémoire. En d'autres termes, en plus de $x^{(t)}$, une cellule LSTM possède deux entrées : l'ancien état de la cellule $c^{(t-1)}$ et le vecteur $h^{(t-1)}$. Pour aller plus loin, nous vous conseillons de voir l'ouvrage [9].

```
Model2.add(embedding_layer)
Model2.add(LSTM(64))
Model2.add(Dropout(0.2))
Model2.add(Dense(64, activation='relu'))
Model2.add(Dropout(0.2))
Model2.add(Dense(3, activation='softmax'))
Model2.compile(loss='categorical_crossentropy',
optimizer='Adam', metrics=['acc'])
Model2.fit(x_train,y_train,validation_data=(x_val, y_val),epochs=13,batch_
size=100)
```

Avec le réseau LSTM choisi, le système arrive à obtenir 82% d'*accuracy*, ce qui est meilleur que ce nous avons eu avec le précédent réseau de neurones multi-couches. Cela s'explique par le fait que le LSTM prend en compte le contexte des mots et ainsi de mieux comprendre la sémantique du tweet.

Enfin, nous évaluons notre système sur le corpus de test. Pour cela, nous utilisons la méthode **evaluate** qui prend comme paramètres l'ensemble des tweets et labels constituant le corpus du test ainsi que la taille du batch. Notre modèle arrive à classer correctement 84.9% des tweets, voir le bas de la figure 5, ce qu'on peut vérifier avec le code suivant :

```
score = model1.evaluate(x_test, y_test, batch_size=100, verbose=1)
print('Test accuracy:', score[1])
```

REMERCIEMENTS

Merci à mes collègues du LSE pour leurs conseils et leur relecture.

RÉFÉRENCES

- [1] <https://drive.google.com/file/d/1YinXLJDxxaXXZB03SeeV28xJNpl5wbbJ/view?usp=sharing>
- [2] <https://keras.io/>
- [3] <https://www.nltk.org/install.html>
- [4] <https://nlp.stanford.edu/projects/glove/>
- [5] <https://code.google.com/archive/p/word2vec/>
- [6] <https://github.com/mmihaltz/word2vec-GoogleNews-vectors>
- [7] <https://radimrehurek.com/gensim/index.html>
- [8] https://www.researchgate.net/publication/13853244_Long_Short-term_Memory
- [9] F. Chollet, *Deep learning with Python*, Manning Publications Co., 2018.



YADIFF : DE L'INTELLIGENCE ARTIFICIELLE POUR COMPARER LES CODES BINAIRES

Eric RENAULT et Martin TOURNEBOEUF

YaDiff est un outil permettant la propagation d'informations d'une base IDA vers une autre pour assister l'analyse de codes binaires. Il utilise un réseau de neurones entraîné à identifier les routines similaires.

Le projet YaDiff a vu le jour suite au problème récurrent du suivi de version lors de la rétro-ingénierie de binaires sous IDA. En effet, quel rétro-concepteur n'a jamais été confronté, après de longues heures d'analyse d'un binaire, à la difficulté de devoir transposer le fruit de son travail sur la toute dernière version dudit binaire ? Pour pouvoir porter les symboles de documentation d'une base IDA vers une autre, YaDiff doit être capable d'identifier les parties de code semblables de deux binaires différents.



NOTE

La documentation qui accompagne les binaires désassemblés peut être issue de l'analyste ou du code source. Elle peut être présente sous forme de « symboles », c'est-à-dire de noms de fonctions, de données globales, de labels, commentaires, prototypes de fonctions ou données, renommage (variables, registres...), typage de structures, etc. Ces symboles, le nom des objets sont des informations essentielles pour l'analyse de binaires complexes.

Fréquemment, les références aux chaînes de caractères (« toto »), valeurs immédiates (**0x42**), instructions folkloriques (**fabs**), appels système (**call WriteFile**) permettent d'identifier des parties de code semblables. En effet, il s'agit de caractéristiques facilement identifiables qui agissent comme des marqueurs naturels, des points d'accroches pour une comparaison. Il existe plusieurs scripts qui identifient ces marqueurs et permettent parfois de reconnaître quelques fonctions. En fait, presque chaque analyste a le sien. Les plus téméraires propagent même les symboles entièrement à la main !

Cependant, afin de pouvoir exploiter intelligemment l'ensemble vertigineux de ces marqueurs, une des solutions les moins complexes aujourd'hui est d'utiliser un réseau de neurones. Ce réseau est le cœur d'un système qui prend en entrée deux fonctions (routines) et donne, en sortie, un score de similarité.

À la fin de ce bref article, non seulement vous comprendrez comment reconnaître les fonctions semblables au sein de binaires compilés, mais aussi, puisque la méthode employée est générique, vous serez capables d'utiliser un réseau de neurones afin d'identifier les éléments semblables de n'importe quelle collection à condition que :

- suffisamment de caractéristiques puissent être extraites de ces éléments ;
- suffisamment d'éléments connus (i.e. : étiquetés) existent pour constituer le corpus d'apprentissage.

La méthode décrite ci-après pourrait être adaptée pour cataloguer des logiciels, bâtiments, pages web, molécules, galaxies...

DES VECTEURS DE CARACTÉRISTIQUES

Les informations de documentation d'un binaire accompagnent le code. Elles sont donc, à son image, encapsulées dans des fonctions. Le nom d'une fonction et son prototype sont des éléments cruciaux pour un analyste. D'autant qu'IDA propage les symboles de prototypages. Une fois une fonction reconnue, il n'est pas trop difficile d'identifier ses blocs basiques : la commande **diff** sur du code désassemblé puis standardisé donnera de bons résultats.



Les fonctions en langage assembleur peuvent être des objets complexes. Leur comparaison n'est alors pas une tâche évidente : si elle repose sur des marqueurs trop stricts tels que les hashes, l'association ne sera pas robuste aux changements, mais si elle repose sur des marqueurs trop simples, elle pourrait produire de fausses associations (i.e. : faux-positifs).

Puisque les marqueurs complexes (tels que des strings, n-grams, plus longue sous-chaîne commune) sont difficiles à exploiter, il est prudent de n'utiliser que des nombres entiers ou réels pour représenter les fonctions. Une fois mis côte à côte, ces scalaires forment des vecteurs représentant les fonctions, lesdits vecteurs de caractéristiques. Aux dépens de la perte de certaines informations, parfois marquantes, il est plus facile de travailler avec des vecteurs de taille fixe qu'avec une collection (i.e. : tuple) de taille variable d'objets différents : (ex : séquences, arbres).

Une comparaison tolérante

Une même fonction peut différer de multiples façons :

Événement perturbateur	Conséquence pour une fonction
Nouvelle Compilation	Changement d'offset Réagencement d'instructions
Changement d'option de compilation	Inclusion d'instruction ou données dans le code Aplatissage du graphe de flot de contrôle Inlining d'une fonction
Changement de compilateur	Changement de convention d'appel, changement des options d'optimisation Traitement différent des variables : des registres et de la mémoire (pile)
Changement d'OS cible	Changement d'API, d'appels systèmes Changement d'implémentation des bibliothèques statiques Différentes implémentations pour une même fonctionnalité, nom et prototype (exemple dans Vim [6] : <code>mch_dirname</code> implémenté dans <code>os_unix.c</code> ou <code>os_win32.c</code>)
Mise à jour du code	Ajout ou suppression d'instructions Changement de données référencées (tableaux, chaînes de caractères) Modification du graphe d'appels

Au vu des changements que peut subir une fonction, pour la reconnaître au sein de binaires inconnus, il faut utiliser des signatures tolérantes aux changements. Mais de plus, ce n'est pas parce que deux fonctions partagent une signature rare qu'elles sont semblables. Par exemple, deux fonctions différentes pourraient référencer la même chaîne de caractères. Ce qui empêche l'usage exclusif de signatures complexes.

Recherche de scalaires

Une fois l'objectif fixé de représenter les fonctions par des vecteurs, il faut établir les coordonnées de ces vecteurs en cherchant les caractéristiques scalaires qui donnent un sens à une fonction. Une fonction est un ensemble d'instructions (1) formant son graphe de flot de contrôle (2). Une fonction peut en appeler d'autres ou être appelée par d'autres. Le graphe des appels est appelé à juste titre graphe d'appels (3). Il est possible de cueillir des valeurs scalaires de ces trois champs. La liste complète et à jour est présente dans le code de YaDiff [3].

» Les instructions en langage assembleur

Une instruction est l'atome de tout code assembleur et donc des fonctions qui le constituent. Le logiciel de désassemblage en source libre Capstone [9] permet de les décoder. Les instructions reçoivent une ou plusieurs étiquettes selon leur type (arithmétique, lecture mémoire, saut conditionnel...). Chaque type d'instruction est compté. Par exemple, dans la fonction de la figure 1 il y a un appel (`call`). Mais de plus, la distribution de chaque type d'instructions au sein de la fonction est prise en compte. Puisqu'il est difficile de représenter une dispersion au sein des nœuds d'un arbre, ce dernier est aplati et puisque ces arbres aplatis sont de taille variable, la position d'une instruction est normalisée de zéro à un. Les positions moyennes, variances, kurtosis, etc. de chaque type d'instruction sont sauvegardés. Ce qui permet, toujours dans l'exemple de la figure 1, de conserver l'information que 4 sauts conditionnels (`jc`) sont plutôt vers le haut et séparés en moyenne d'une instruction.

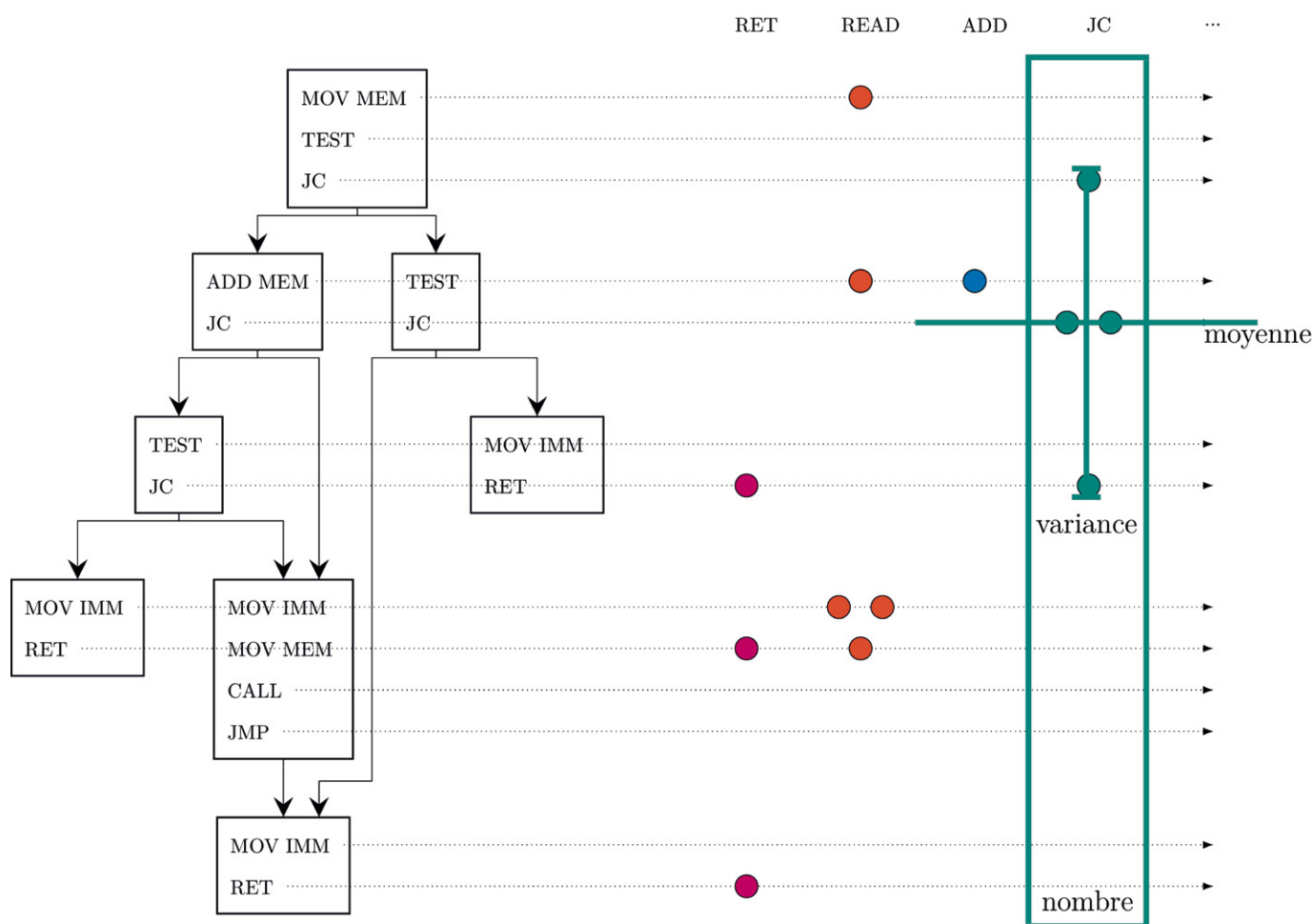


Fig. 1 : Étiquetage des instructions, aplatissement du graphe de flot de contrôle puis extraction des moments centrés de leur distribution : moyenne, variance, kurtosis ...

» La forme du graphe de flot de contrôle

Le graphe de flot de contrôle représente l'imbrication et les relations causales des instructions. Formellement, c'est un arbre binaire ordonné, orienté et enraciné. Il existe diverses manières de comparer les arbres : parcours de sous-arbres [1], distance d'édition, expressions régulières [2]... C'est toute une science de comparer des arbres : la dendrologie. Mais n'oublions pas le principe de tolérance aux changements : des nœuds peuvent être interchangeables sans que cela change le flot d'exécution. Par exemple, des conditions d'optimisation pour sortir d'une boucle plus tôt peuvent apparaître. Quelques informations pourtant donnent rapidement un aperçu de l'arbre : sa hauteur, largeur, son nombre de feuilles, de nœuds, de branches, d'intersections en diamant (if ... then ... else), etc. Par exemple, le graphe de flot de contrôle de la fonction figure 1 possède 8 nœuds, 9 branches et 3 feuilles. Ces informations sont précieusement sauvegardées dans le vecteur de caractéristiques de chaque fonction.

» Le graphe d'appels

À l'image du graphe de flot de contrôle, le graphe d'appels ne peut être exploité directement sous forme de graphe. Réduisons-le ! Pour chaque fonction, ce graphe apporte l'information de qui l'appelle (les appelants) et qui est appelé par elle (les appelés). Par exemple, **vim_snprintf** appelle **vim_vsnprintf** ainsi que deux petites fonctions et est appelée par une grande diversité de fonctions (198). Des objets référençant des objets

semblables sont potentiellement semblables. Ici encore la probabilité de similarité dépendra d'autres facteurs venant des instructions ou du graphe de flot de contrôle.

Les fonctions sont représentées par un vecteur, nous en bénéficions pour la première fois. Chaque fonction s'est déjà vu attribuer un vecteur interne représentant ses instructions et son graphe de flot de contrôle. La moyenne et la médiane des vecteurs internes de ses appelants et de ses appelés seront sauvegardées dans le vecteur de caractéristiques de chaque fonction.

Concaténation : 1000 scalaires pour 1 vecteur

Le vecteur de caractéristiques est issu, comme prévu, d'une longue concaténation :

- Les scalaires indépendants (nombre d'appelés, nombre de références aux données ...) : 10 nombres.
- La forme du graphe de flot de contrôle : on extrait 20 nombres.
- Les paramètres de distributions (au nombre de 5) de chaque type d'instruction (au nombre de 20) : $20 \times 10 = 200$ nombres.
- La moyenne et la médiane (2) des appelants et des appelés (2) du vecteur précédent : $2 \times 2 \times (10 + 20 + 200) = 920$ nombres.

Il résulte donc de mise en commun de toutes ces caractéristiques, un vecteur de dimension $10 + 20 + 200 + 920 = 1150$.

Les deux fonctions C++ suivantes implémentent le processus de concaténation : après avoir accumulé pour chaque fonction ses caractéristiques dans une grande structure **FunctionData_t**, la fonction **CreateConcatenatedVector** crée les vecteurs de caractéristiques de toutes les fonctions.

```
#include<vector>

std::vector<double> FunctionData2Vector(const FunctionData_t& function_data);

/* Crée le vecteur de caractéristique pour toutes les fonctions
 * - function_signature_map : Table de hachage qui associe à chaque fonction ses données
 * Retourne : Le vecteur de caractéristique dans la function_signature_map
 */
void CreateConcatenatedVector(FunctionSignatureMap_t& function_signature_map)
{
    // Crée le vecteur interne de chaque fonction
```

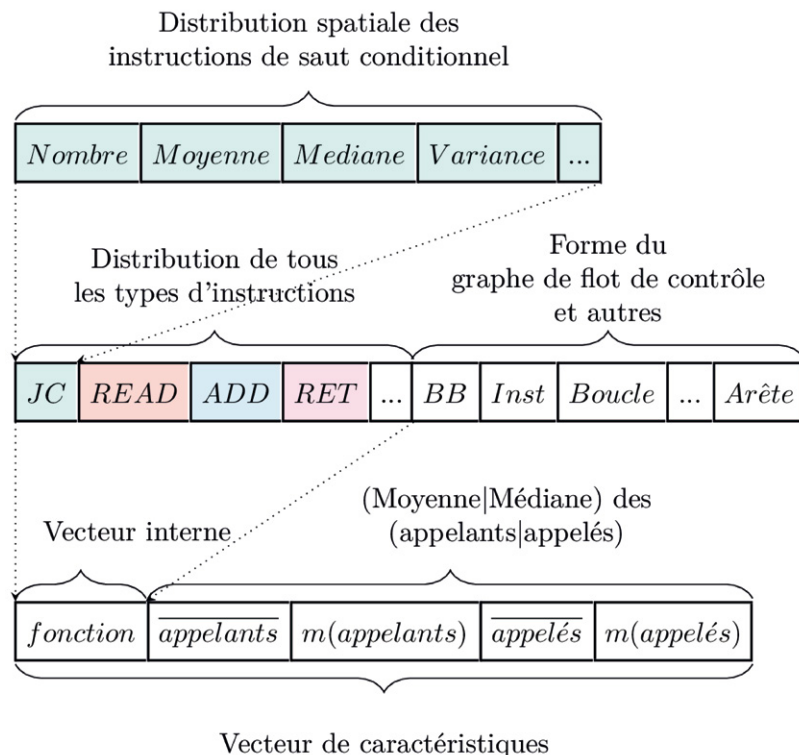


Fig. 2 : Concaténation des caractéristiques émanant du graphe de flot de contrôle, des différents types d'instructions d'une fonction et de ses appelants.

```

for (auto& it : function_signature_map)
{
    FunctionSignature_t& function_signature = it.second;
    function_signature.vector = FunctionData2Vector(function_signature.function_data);
}

// Pour chaque fonction
for (auto& it: function_signature_map)
{
    FunctionSignature_t& function_signature = it.second;

    // 1/ Interne : Initie le vecteur de caractéristique comme copie du vecteur interne
    function_signature.concatenated_vector = function_signature.vector;

    // 2/ Appelants
    // Réunis les vecteurs internes des appelants dans une matrice
    typedef std::vector<std::vector<double>> Matrix;
    Matrix matrix = Matrix();
    for (auto& parent_id : function_signature.parents)
    {
        std::vector<double> parent_vector = function_signature_map[parent_id].vector;
        matrix.push_back(parent_vector);
    }
    // Concatène la moyenne et médiane des lignes (appelants) dans le vecteur de
    // caractéristiques de la fonction courante
    ConcatenateFamilly(function_signature.concatenated_vector, matrix);

    // 3/ Appelés (même travail que les appelants)
    ....
}
}

/* Aplatie dans un vecteur les caractéristiques internes d'une routine
 * - function_data: structure contenant l'ensemble des caractéristiques
 *   internes d'une fonction (nombre de blocs basiques, d'instructions de saut ...)
 * Retourne: Vecteur interne de la fonction
 */

std::vector<double> FunctionData2Vector(
    const FunctionData_t& function_data
)
{
    std::vector<double> res ;

    // Concatène les caractéristiques du graphe de flot de contrôle
    res.push_back(static_cast<double>(function_data.cfg.bb_nb));
    .....
    res.push_back(static_cast<double>(function_data.cfg.flat_len));

    // Concatène les moments centrés de la distribution de chaque type d'instruction
    for (int i = 0; i < INST_TYPE_COUNT; i++)
    {
        InstructionData_t instruction_data = function_data.insts[(InstructionType_e)i];
        res.push_back(static_cast<double>(instruction_data.total));
        .....
        res.push_back(static_cast<double>(instruction_data.offset_kurt_per_inst));
    }

    return res;
}

```

Diversité des caractéristiques

La démarche jusqu'ici consistait à rassembler un maximum de features (dans ce travail ce sont des nombres entiers ou réels) pour chaque objet. Puisqu'ils viennent de champs différents, ces caractéristiques ne sont pas faciles à exploiter ensemble. Elles sont nombreuses, parfois liées, suivent une dispersion variée, certaines sont plus importantes que d'autres. Pourtant, ce qui complique le plus la comparaison de ces vecteurs est la définition du semblable. Il serait possible de définir des limites du type : « si une fonction diffère d'une autre de plus de 3 instructions alors elles sont différentes », mais une instruction **mov** dans une boucle est potentiellement équivalente à 4 instructions **mov** à la suite (*inlining*). C'est ici que la solution des réseaux de neurones paraît la plus prometteuse : premièrement, les transformations ne sont pas limitées aux transformations linéaires des entrées, et deuxièmement, l'algorithme peut apprendre la logique derrière les données et être capable de plus de généralité.

En choisissant de représenter les fonctions sous forme d'un vecteur de taille fixe (mais tout de même avec beaucoup de valeurs), nous avons choisi de simplifier la problématique. Car analyser les chaînes (non orientées) des graphes devient vite très compliqué, surtout en apprentissage machine. Il reste maintenant à exploiter cela intelligemment afin que cela fonctionne.

UN RÉSEAU DE NEURONES COMME OPÉRATEUR DE PROJECTION

L'intérêt d'un réseau de neurones est qu'il peut être considéré comme un « approximateur universel » de fonctions très complexes qui prennent des entrées (routines) et donnent une sortie (décision). Cela marche bien aussi avec les fonctions simples, mais pour ces dernières, il y a des méthodes plus directes. C'est en gros apprendre à faire « $y = f(x)$ » connaissant les « x » et « y », mais pas « f ». Plus il y a de couples (x, y) à donner en exemple, meilleur sera le système. Dans notre cas, les « x » sont les vecteurs de caractéristiques des routines (avec ~1000 valeurs par fonction). Reste à déterminer « y », ce qui revient à se demander : que cherche-t-on exactement à trouver ? En reprenant l'exemple « $y = f(x)$ », cela transforme une routine en langue assembleur en quelque chose de différent. Mais puisque le but est de comparer des routines entre elles, nous sommes plutôt dans le cas où nous essayons de trouver « $y = f(x_1, x_2)$ » avec « x_1 » et « x_2 » deux routines et « y » la similarité (ou dissimilarité).

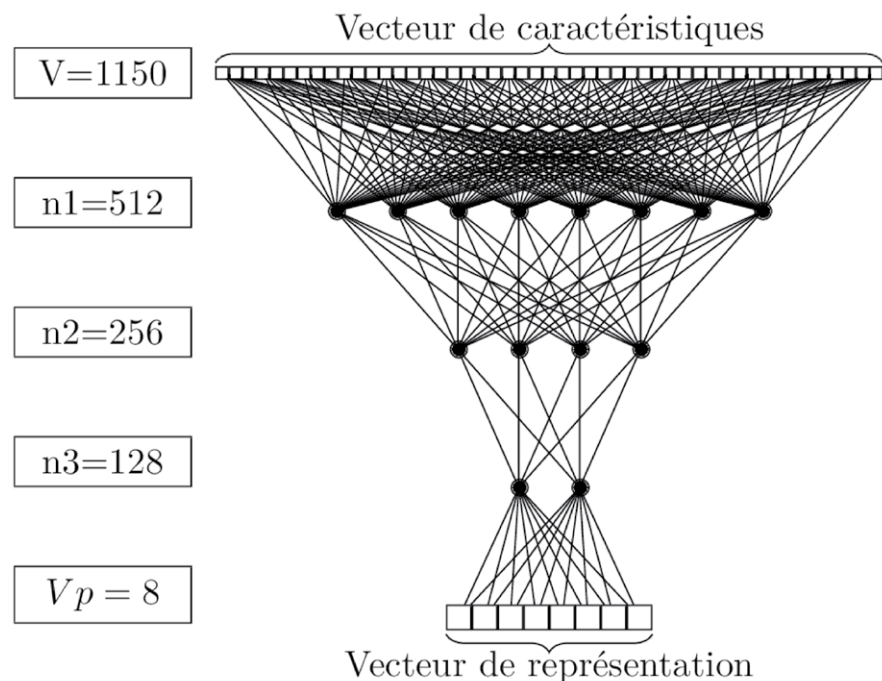


Fig. 3 : Une fois entraîné, ce réseau de neurones projette les vecteurs de caractéristiques dans un espace de dimension 8 (i.e. un vecteur réel de dimension 8) tel que les routines qui étaient considérées semblables lors de l'entraînement soient proches.

Les corpus

» Le corpus d'apprentissage : les dépôts Debian

Comme dit précédemment, plus le système aura d'exemples pour apprendre plus il sera pertinent, un peu à la manière du cerveau humain. Par exemple, quelqu'un qui aurait vu l'ensemble des épisodes des 35 premières saisons des Feux de l'Amour (soit 8799 épisodes au total [7]), aurait alors une bien meilleure estimation des chances de succès du mariage entre Ashley et Tucker tout juste sorti de son coma [8], contrairement à une autre personne avec « seulement » 5 saisons à son actif.

Dans notre cas (qui est au moins aussi intéressant), il faut des binaires, plein de binaires. Et c'est encore mieux de savoir ce qu'il y a dedans (étiquetage). Les binaires des dépôts Debian (généralement compilés avec des options semblables) sont d'excellents candidats pour cela. Leurs étiquettes sont conservées (noms de binaires, noms de fonctions, architecture, etc.). Les vecteurs de caractéristiques de chaque fonction sont calculés comme vu précédemment. Cela constitue le corpus d'apprentissage (i.e. le dataset).

Dernière étape, l'ensemble du corpus (les valeurs flottantes) est normalisé pour avoir des valeurs aux mêmes échelles : cela améliore l'apprentissage. Pour les valeurs très importantes, une normalisation logarithmique est effectuée. Puis une simple normalisation linéaire borne les valeurs entre -1 et 1. À cette étape, il ne reste plus qu'à fournir ces données au réseau de neurones pour qu'il apprenne à notre place ce qui définit deux fonctions semblables (outre leur nom).

» Le corpus de validation : nfsd.ko, cc1plus, libxhtml

La validation est cruciale pour ... valider le modèle appris (d'où son nom). En effet, si on repose toujours les mêmes questions à un enfant et qu'on le corrige quand il se trompe, il saura donner la bonne réponse à la 3ème ou 4ème question, même s'il ne comprend pas la question.

Pour valider le système, une partie des binaires est donc mise de côté et tout vecteur de fonction qui serait similaire dans l'entraînement est écarté afin d'être sûr de demander au système d'évaluer des fonctions qu'il n'aura jamais vues. Le plus simple est d'isoler 3 ensembles de binaires avec des usages éloignés, pour limiter le biais d'une validation reposant uniquement sur un sous-ensemble spécifique de fonctions écrites pour un même objectif.

L'apprentissage

» Un réseau triplet

L'algorithme utilisé est relativement simple. Basé sur un réseau de neurones, il emploie le principe des réseaux siamois, voir [10] par exemple, car il a pour but de comparer deux fonctions. Mais en fait, il en compare trois à chaque fois. Pourquoi trois ?

Trois vecteurs de caractéristiques sont fournis en entrée. Le premier vecteur est un vecteur dit de référence, le deuxième dit positif est un vecteur appartenant au même groupe de fonctions, et pour terminer un troisième vecteur dit négatif



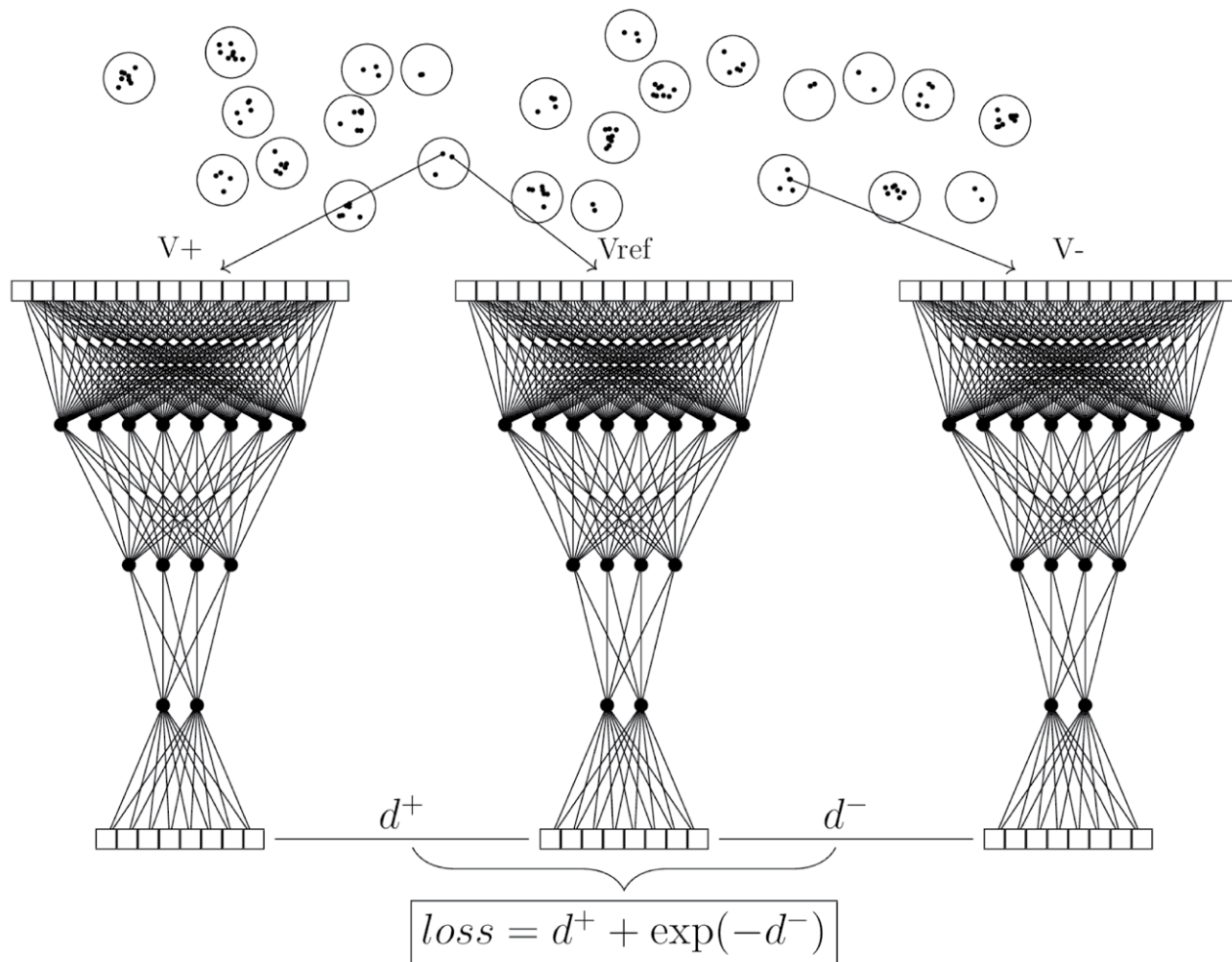


Fig. 4 : Un réseau triplet qui simultanément s'entraîne à obtenir un espace de sortie tel qu'une fonction semblable ($V+$) soit proche de la référence (V_{ref}) et une fonction différente ($V-$) en soit loin.

appartenant à un autre groupe de fonctions. Cela permet de fournir aux réseaux en même temps des fonctions similaires et dissimilaires. A posteriori, un espace de sortie à 8 dimensions suffit pour que le réseau de neurones représente correctement les fonctions en regroupant celles qui sont similaires. Passer de 1000 caractéristiques à 8 c'est plutôt sympa !

Le réseau qui est un réseau de type perceptron multicouche fait une projection non linéaire du vecteur d'entrée dans un espace de plus petite dimension. Le résultat de cette projection est appelé vecteur de représentation (*embedding*). L'opérateur de projection est défini dans la fonction Python **projectron** et la fonction de coût du réseau triplet dans la fonction **triplet**.

```
import tensorflow as tf

def projectron(vecteur_entrée, assertion_réutilise):
    """ Crée un réseau de projection : 1150 -> 8
        - vecteur_entrée: vecteur d'entrée, s'obtient avec
          `tf.placeholder(tf.float32, [None, ts.vecsize], name='ref')`
        - assertion_réutilise: booléen ce réseau est-il le clone d'un autre ?
        Retourne: le réseau de neurones
    """
```

```

# La première couche du réseau est le vecteur de caractéristiques d'entrée
reseau = vecteur_entrée

with tf.variable_scope("réseau_de_décision"):
    # Ajoute les couches densément connectées
    for index, taille in enumerate([512, 256, 128]):
        # Connecte la couche courante avec le réseau amont
        reseau = tf.layers.dense(
            inputs = reseau,
            units = taille,
            reuse = assertion_réutilise,
            activation = tf.nn.leaky_relu,
            name = 'dense_%d' % index,
        )

    # Ajoute la dernière couche qui décide avec une tangente hyperbolique
    reseau = tf.layers.dense(
        inputs = reseau,
        units = 8,
        reuse = assertion_réutilise,
        activation = tf.nn.tanh,
        name = 'output',
    )
    return reseau

def triplet(self):
    """ Crée un réseau triplet pour l'apprentissage
    - self: vecteurs de référence, positif et négatif dans leurs
    placeholders
    Retourne (dans self): les trois réseaux qui pourront prendre en entrée
    les vecteurs de référence.
    Le coût à minimiser pour ce réseau

    """
    # Définition des caractères de remplissage (placeholders)
    self.vecteur_reférence = tf.placeholder(tf.float32, [None, 1150], name='ref')
    self.vecteur_positif = tf.placeholder(tf.float32, [None, 1150], name='pos')
    self.vecteur_négatif = tf.placeholder(tf.float32, [None, 1150], name='neg')

    # Création du réseau triplet
    self.réseau_reférence = projectron(self.vecteur_reférence, False)
    self.réseau_positif = projectron(self.vecteur_positif, True)
    self.réseau_négatif = projectron(self.vecteur_négatif, True)

    # Calcul de la métrique (ploss = positive loss)
    ploss = tf.reduce_mean(tf.square(net_ref - net_pos), 1)
    nloss = tf.exp(-tf.reduce_mean(tf.square(net_ref - net_neg), 1))
    self.loss = ploss + nloss

    return self

```

» La Descente de gradient

L'entraînement consiste à donner au réseau de neurones des vecteurs en entrée et à évaluer ce qu'il en sort. Et s'il ne donne pas la bonne réponse (par exemple des vecteurs de représentation de fonctions différentes sont trop proches), il reçoit comme une décharge électrique au sens mathématique, qui va remonter tout le long du réseau de neurones pour qu'il se corrige. Et plus un neurone participe à la décision erronée, plus la puissance de la décharge et donc sa correction seront importantes.

En apprentissage machine, les termes suivants sont fréquemment employés :

- Taux d'apprentissage (*learning rate*) : c'est tout simplement l'amplitude de la châtaigne en cas d'erreur.
- La fonction de coût (*loss function*) : la « bonne » réponse attendue.
- Le surapprentissage (*overfitting*) : quand le système a appris par cœur les bonnes réponses, sans comprendre le pourquoi du comment. Et ça se voit en lui demandant de résoudre des exercices nouveaux, car il se loupe bien. Il existe des techniques pour éviter ou tout du moins limiter ces effets, voir [11] et [12].

Exploitation du réseau

L'extraction de caractéristiques puis le réseau de neurones entraîné projettent les fonctions dans un espace de dimension 8 où les distances euclidiennes ont le sens qu'on a voulu leur donner. Enfin, nous retrouvons notre sacré théorème de Pythagore ! Maintenant, il suffit de faire appel à des algorithmes anciens, simples, rapides, génériques, déjà implémentés de recherche de plus proches voisins afin de retrouver les fonctions semblables au sein de deux binaires différents. Dans un espace vectoriel euclidien, un analyste peut également mettre en base ces objets. Cette démarche permet principalement de donner un nom aux objets étudiés, ici des fonctions en langage assembleur.

Un article plus détaillé accompagné d'une vidéo, pour chacun des outils YaDiff et YaCo est disponible sur le site de la conférence SSTIC [4,5].

CONCLUSION

Le lecteur attentif est maintenant capable d'expliquer comment :

1. Différencier des fonctions en langage assembleur.
2. Différencier tout type d'objets représentables sous forme de vecteurs (et il en existe un paquet) !
3. Collaborer avec une intelligence artificielle. Dans les deux sens, l'humain donne à la machine les données jugées pertinentes. En échange, en plus de cataloguer ces objets, la machine retourne à l'humain l'information des données qui étaient effectivement pertinentes lui suggérant par là même de décrire plus précisément ces données, d'approfondir ce secteur d'études.
4. Déterminer lorsqu'un réseau de neurones est une méthode adaptée (*i.e.* : possible) ou non pour résoudre un problème. ■

REMERCIEMENTS

Benoît Amiaux, Jérémy Bouetard, Léonard Caquot, Valérian Comiti, Frédéric Grelot et Maxime Pinard pour avoir participé au développement de YaDiff pour la Direction Générale de l'Armement sur le site de Maîtrise de l'Information (Rennes).

RÉFÉRENCES

- [1] Gorille : G Bonfante, M Kaczmarek, Jean-Yves Marion : *Architecture of a Morphological Malware Detector*, Journal in Computer Virology. 5(3):263–270, 2009
- [2] Tregexp : <https://nlp.stanford.edu/software/tregexp.shtml>
- [3] YaDiff: <https://github.com/DGA-MI-SSI/YaCo>
- [4] YaCo: <https://www.sstic.org/2017/presentation/YaCo>
- [5] YaDiff: <https://www.sstic.org/2018/presentation/yadiff>
- [6] Vim 8.1.0000 : <https://github.com/vim/vim/commit/b1c9198afb7ff902588b45f4e44f0760a9f48375>
- [7] Les Feux de l'Amour – Wikipédia : https://fr.wikipedia.org/wiki/Les_Feux_de_l%27amour
- [8] Description épisode 9682 : <https://www.programme-tv.net/news/series-tv/59986-les-feux-de-l-amour-ashley-a-epouse-tucker-mais-qui-sont-ses-anciens-epoux/>
- [9] <https://www.capstone-engine.org/>
- [10] <http://www.cs.utoronto.ca/~gkoch/files/msc-thesis.pdf>
- [11] A. Géron : *Machine Learning avec Scikit-Learn, Mise en Œuvre et cas concrets*, Dunod, 2017
- [12] A. Géron : *Deep Learning avec TensorFlow, Mise en Œuvre et cas concrets*, Dunod, 2017

VOYAGE EN C++IE : UN DIAMANT D'INNOCENCE

Serge GUELTON

Cet article est le troisième d'une mini-série sur le C++, ou plutôt sur les binaires compilés depuis C++, leurs particularités, comment les concepts du langage se retrouvent parfois dans le binaire final.

Cet opus corrige un oublié lors de l'épisode précédent (*Voyage en C++ie : les objets*, publié dans *MISC n°96*), et présente le boss final : le fameux « deadly diamond of death ». Manquement d'autant plus intolérable dans une revue des Éditions Diamond...

LE PROBLÈME DU DIAMANT

Imaginons la (mauvaise) hiérarchie de classe suivante, qui n'est pas sans rappeler certains jeux très ancrés dans l'imaginaire médiéval fantastique.

```
struct Fighter {  
    int hp = 10;  
    int chop() const;  
};
```

```
struct Warrior : Fighter {
};

struct Monk : Fighter {
};
```

Osons maintenant créer un type de personnage bi-classé (et d'une humilité sans faille) :

```
struct Me : Warrior, Monk {
    void strike(Fighter& other) const {
        if (hp > 0)
            other.hp -= chop();
    };
};
```

Ce code ne compile pas, et pour cause : **Warrior** et **Monk** ont une base commune, **Fighter** et l'accès à **hp** est ambigu. Quand on veut que les membres communs ne soient présents qu'une seule fois, et c'est le cas ici, on utilise l'héritage virtuel, le fameux héritage en diamant.

```
struct Warrior : virtual Fighter {
};

struct Monk : virtual Fighter {
};
```

Il n'y aura alors qu'une seule instance de **Fighter** instanciée par instance de **Me** (notez l'élégante référence à une célèbre licence de jeux de baston édités par SEGA).

INITIALISATION D'UN OBJET AVEC HÉRITAGE VIRTUEL

Si l'on tente d'instancier un objet de la classe **Me** :

```
Me spontaneous_me() {
    return {};
}
```

On obtient le bitcode LLVM suivant (d'après `clang++ -S -emit-llvm -O0 -o - diamond.cpp -std=c++11 | c++filt`, on regardera le code optimisé une fois qu'on a bien tout compris) que nous allons décortiquer. Notons le filtrage par `c++filt` pour appliquer le *demangling* (décodage des symboles, cf. épisodes précédents disponibles chez tous les bons libraires) sur le bitcode LLVM, cette commande est bien pensée ! J'ai enlevé quelques attributs (genre `align 8` sur les store) pour ne garder que l'essentiel.

```
define linkonce_odr void @Me::Me()(%struct.Me*) {
; copy parameter to a register through the stack,
; let's call this register `this'
%2 = alloca %struct.Me*
store %struct.Me* %0, %struct.Me** %2
%3 = load %struct.Me*, %struct.Me** %2

; access the 16th byte of `this' and
; cast it to Fighter* to initialize it
%4 = bitcast %struct.Me* %3 to i8*
%5 = getelementptr inbounds i8, i8* %4, i64 16
%6 = bitcast i8* %5 to %struct.Fighter*
call void @Fighter::Fighter()(%struct.Fighter* %6)
```



```

; cast this as warrior and call a specific constructor
; that uses the 1st element of the VTT
%7 = bitcast %struct.Me* %3 to %struct.Warrior*
call void @Warrior::Warrior(%struct.Warrior* %7, i8** getelementptr inbounds ([4 x i8*],
[4 x i8*]* @VTT for Me, i64 0, i64 1))

; access the 8th byte of `this` and
; cast it to Monk* to initialize it
%8 = bitcast %struct.Me* %3 to i8*
%9 = getelementptr inbounds i8, i8* %8, i64 8
%10 = bitcast i8* %9 to %struct.Monk*
call void @Monk::Monk(%struct.Monk* %10, i8** getelementptr inbounds ([4 x i8*], [4 x i8*]*
@VTT for Me, i64 0, i64 2))

;;;;;;;;;;;;; PAUSE ;;;;;;;;;;;;;;

; store values from the vtable in the 0st bytes of `this`
%11 = bitcast %struct.Me* %3 to i32 (...)**
store i32 (...)** bitcast (i8** getelementptr inbounds ({ [3 x i8*], [3 x i8*] }, { [3 x i8*],
[3 x i8*] }* @vtable for Me, i32 0, inrange i32 0, i32 3) to i32 (...)**), i32 (...)** %11,

; store other values from the vtable in the 8th bytes of `this`
%12 = bitcast %struct.Me* %3 to i8*
%13 = getelementptr inbounds i8, i8* %12, i64 8
%14 = bitcast i8* %13 to i32 (...)**
store i32 (...)** bitcast (i8** getelementptr inbounds ({ [3 x i8*], [3 x i8*] }, { [3 x i8*],
[3 x i8*] }* @vtable for Me, i32 0, inrange i32 1, i32 3) to i32 (...)**), i32 (...)** %14
ret void
}

```

Mmmmh, notre diamant a de multiples facettes qui reflètent autant de subtilités du langage ! On retrouve bien les constructeurs de chaque classe parente **@Fighter::Fighter**, **@Warrior::Warrior** et **@Monk::Monk()**, mais les deux derniers constructeurs prennent un nouveau paramètre, la VTT, la *Virtual Table Table*... Qu'est-ce donc ? Regardons son initialisation, en s'arrêtant à la **; PAUSE ;** :

```

@VTT for Me = constant [4 x i8*] [
  i8* bitcast (i8** getelementptr inbounds ({ [3 x i8*], [3 x i8*] }, { [3 x i8*],
[3 x i8*] }* @vtable for Me, i32 0, inrange i32 0, i32 3) to i8*),
  i8* bitcast (i8** getelementptr inbounds ({ [3 x i8*] }, { [3 x i8*] }* @
construction vtable for Warrior-in-Me, i32 0, inrange i32 0, i32 3) to i8*),
  i8* bitcast (i8** getelementptr inbounds ({ [3 x i8*] }, { [3 x i8*] }* @
construction vtable for Monk-in-Me, i32 0, inrange i32 0, i32 3) to i8*),
  i8* bitcast (i8** getelementptr inbounds ({ [3 x i8*], [3 x i8*] }, { [3 x i8*],
[3 x i8*] }* @vtable for Me, i32 0, inrange i32 1, i32 3) to i8*)
]

```

Okay... On parle de *construction vtable for Warrior-in-Me* et de *construction vtable for Monk-in-Me*, inspectons cela :

```

@construction vtable for Warrior-in-Me = constant { [3 x i8*] } { [3 x i8*] [
  i8* inttoptr (i64 16 to i8*),
  i8* null,
  i8* bitcast ({ i8*, i8*, i32, i32, i8*, i64 }* @typeinfo for Warrior to i8*)
] }
@construction vtable for Monk-in-Me = constant { [3 x i8*] } { [3 x i8*] [
  i8* inttoptr (i64 8 to i8*),
  i8* null,
  i8* bitcast ({ i8*, i8*, i32, i32, i8*, i64 }* @typeinfo for Monk to i8*)
] }

```

Soit, un pointeur à **null**, des *typeinfo* et un entier qui ressemble à un offset... regardons comment cette VTT est utilisée par les constructeurs :

```
define void @Warrior::Warrior()(%struct.Warrior*, i8**) {
: typical copy-arg-to-stack-then-load
%3 = alloca %struct.Warrior*
%4 = alloca i8**
store %struct.Warrior* %0, %struct.Warrior** %3
store i8** %1, i8*** %4
%5 = load %struct.Warrior*, %struct.Warrior** %3
%6 = load i8**, i8*** %4

; %5 = `this', %6 = `construction vtable for Warrior-in-Me'
%7 = load i8*, i8** %6
%8 = bitcast %struct.Warrior* %5 to i32 (...)**
%9 = bitcast i8* %7 to i32 (...)**

; the VTT is stored in `this'
store i32 (...)** %9, i32 (...)** %8
ret void
}
```

Le même schéma se retrouve pour l'appel au constructeur de **Monk**, on a donc maintenant une idée un peu plus précise de l'organisation mémoire du bouzin à la ; **PAUSE** ; :

```
-----
construction vtable for Warrior-in-Me
-----
construction vtable for Monk-in-Me
-----
actual attributes
-----
```

Et si on continue après la ; **PAUSE** ;, on voit que les deux premiers champs sont écrasés par les *vtables* des deux parents ! En fait, les *construction vtable for Warrior-in-Me* et *construction vtable for Monk-in-Me* sont des tables temporaires qui permettent de construire l'objet alors qu'il n'est pas encore complètement initialisé. Effectivement, si on appelle une méthode virtuelle dans le constructeur de **Warrior**, l'ordre de résolution sera différent que si on l'appelle depuis le constructeur de **Me**, d'où cette rustine temporaire de la *vtable*. Extra !

Comme notre exemple est trop simple, ces tables ne sont pas vraiment utiles, comme le montre le code craché par **clang** en -O2 :

```
define void @spontaneous_me()(%struct.Me* noalias nocapture sret) {
; first set everything to zero
%2 = bitcast %struct.Me* %0 to i8*
tail call void @llvm.memset.p0i8.i64(i8* %2, i8 0, i64 24, i32 8, i1 false)

; store the actual attribute in the 2nd slot
%3 = getelementptr inbounds %struct.Me, %struct.Me* %0, i64 0, i32 2, i32 0
store i32 10, i32* %3

; store the two vtables in the 0th and 1st slot
%4 = bitcast %struct.Me* %0 to <2 x i32 (...)**>*

store <2 x i32 (...)**> <i32 (...)** bitcast (i8** getelementptr inbounds ({ [3 x i8*], [3
x i8*] }, { [3 x i8*], [3 x i8*] }* @vtable for Me, i64 0, i32 1, i64 0) to i32 (...)**>,
i32 (...)** bitcast (i8** getelementptr inbounds ({ [3 x i8*], [3 x i8*] }, { [3 x i8*], [3
x i8*] }* @vtable for Me, i64 1, i32 0, i64 0) to i32 (...)**>, <2 x i32 (...)**>* %4
ret void
}
```

Comme les *vtable* temporaires ne sont pas utilisées, l'optimiseur les a proprement dégagées. On remarquera combien l'optimisation faite par **clang** est raccord avec le discours :-)

Vérification par le code :

```
int main() {
    Me me = spontaneous_me();
    void* raw = &me;
    void** as_ptr = (void**)raw;
    printf("%p - %p - %p\n", as_ptr[0], as_ptr[1], as_ptr[2]);
    return 0;
}
```

Qui à l'exécution affiche le tant attendu résultat :

```
$ ./diamond
0x4008a0 - 0x4008b8 - 0xa
```

Et un petit **objdump -C -t diamond | grep 4008** nous confirme que la *vtable* de **Me** prend 48 octets chargés de 0x400888 à 0x4008b8, on est bon !

POUSSIÈRE DE DIAMANT

Continuons avec un scénario d'apparence banale :

```
int chop(Me const& ref) {
    return ref.chop();
}
```

Une fois compilé sans optimisation, on obtient ce gros pavé, commenté au fil de l'eau :

```
define i32 @chop(Me&)(%struct.Me* dereferenceable(24)) #4 {
; copy-arg-to-stack-then-load-to-ref
%2 = alloca %struct.Me*
store %struct.Me* %0, %struct.Me** %2
%3 = load %struct.Me*, %struct.Me** %2

; load the first pointer in this, i.e. the vtable for Warrior
%4 = bitcast %struct.Me* %3 to i8** %5 = load i8*, i8** %4

; access an element before this vtable, taking advantage of the vtable layout for Me
%6 = getelementptr i8, i8* %5, i64 -24
%7 = bitcast i8* %6 to i64*
%8 = load i64, i64* %7

; perform a small correction of the this ptr using the value from the vtable
%9 = bitcast %struct.Me* %3 to i8*
%10 = getelementptr inbounds i8, i8* %9, i64 %8
%11 = bitcast i8* %10 to %struct.Fighter*
%12 = call i32 @Fighter::chop() const(%struct.Fighter* %11)
ret i32 %12
}
```

L'offset de -24 peut paraître surprenant ! Mais si on se souvient de l'initialisation du premier champ de **this**, c'était à travers :

```
store i32 (...)** bitcast (i8** getelementptr inbounds ({ [3 x i8*], [3 x i8*] }, { [3 x i8*],
[3 x i8*] }* @vtable for Me, i32 0, inrange i32 0, i32 3) to i32 (...)**), i32 (...)** %11
```


Soit le 4ème élément du premier élément du tableau **@vtable for Me** qui n'en contient que 3... Donc le premier du deuxième élément. Et en reculant de 24 octets, on tombe (ouille) sur le premier champ du premier item, à savoir :

```
@vtable for Me = constant { [3 x i8*], [3 x i8*] } {
    [3 x i8*] [i8* inttoptr (i64 16 to i8*), i8* null, i8* bitcast ({ i8*, i8*,
    i32, i32, i8*, i64, i8*, i64 }* @typeinfo for Me to i8*)],
    [3 x i8*] [i8* inttoptr (i64 8 to i8*), i8* inttoptr (i64 -8 to i8*), i8*
    bitcast ({ i8*, i8*, i32, i32, i8*, i64, i8*, i64 }* @typeinfo for Me to i8*)]
}
```

Soit la valeur **0x10**. Dans le doute, le code suivant permet de le confirmer :

```
int main() {
    Me me = spontaneous_me();
    void* raw = &me;
    void** as_ptr = (void**)raw;
    void* vtable = as_ptr[0];
    printf("%p", (void*)(intptr_t)((char*)vtable -24));
    return 0;
}
```

À l'exécution, on obtient :

```
$ ./diamond
0x10
```

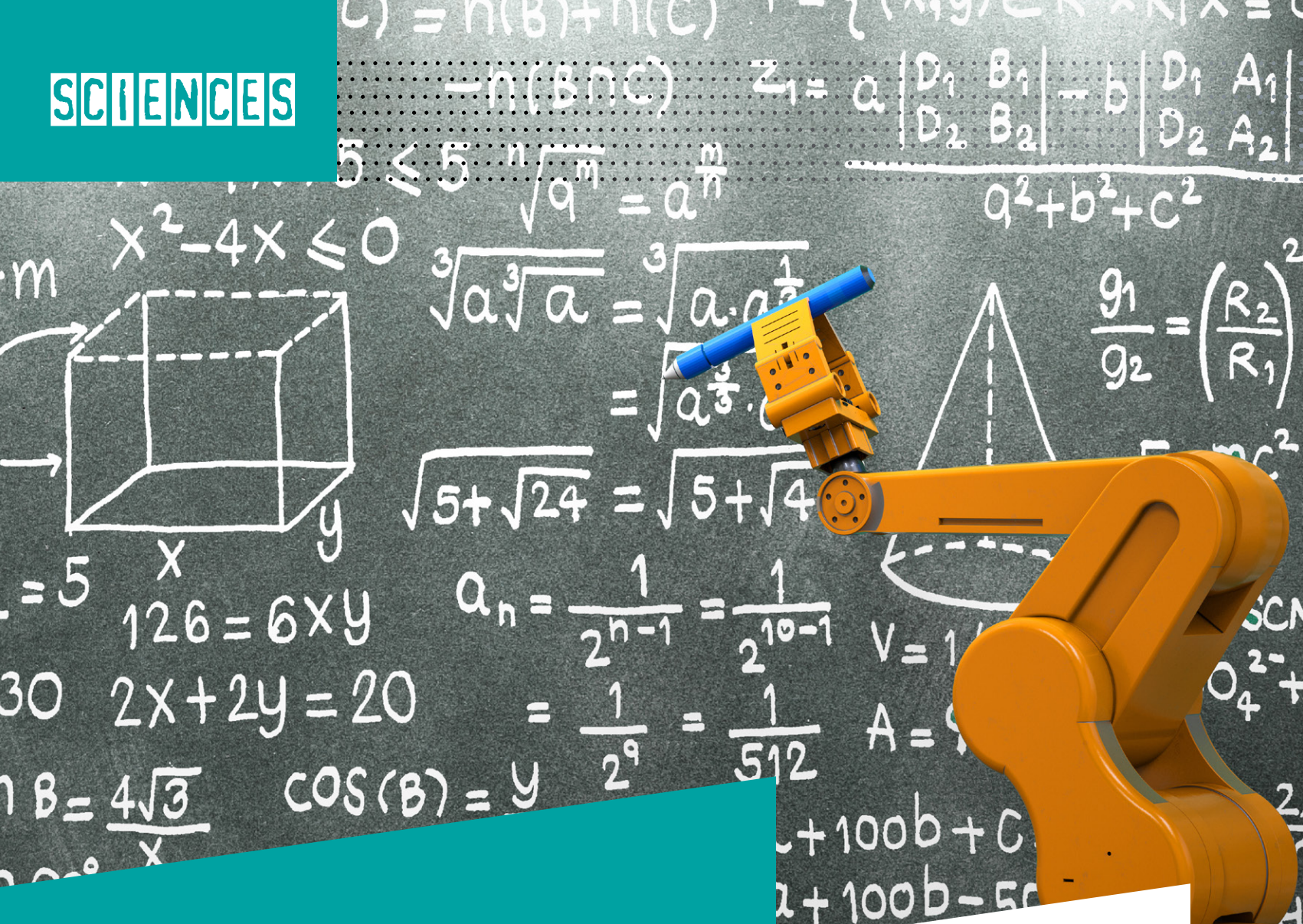
Ouf ! Ce petit ajustement de +16 permet de repositionner **this** après les deux **vtable** des classes filles, de façon à ce que **this** ressemble à une instance de **Fighter** normale :-)

CONCLUSION

Cette exploration de code devrait avoir un peu démystifié le diagramme en diamant à la sauce C++, avec pour effet de bord une piqure de rappel sur les **vtables**. Comme quoi il y avait de quoi creuser !

REMERCIEMENTS

Cet article a bénéficié des conseils et relectures de neitsa, papa zours et lancelet. Un grand merci à eux !



INTRODUCTION À L'APPRENTISSAGE PAR RENFORCEMENT, SANS CODE ET PRESQUE SANS MATHS

Édouard KLEIN

L'apprentissage par renforcement est un domaine de l'apprentissage statistique qui cherche à résoudre le problème de la prise de décision séquentielle dans l'incertitude.

L'apprentissage par renforcement est l'héritier du domaine du contrôle optimal, remontant aux années 50, cherchant à donner à des systèmes dynamiques la consigne qui permet d'optimiser un critère donné.

Tout cela est très abstrait, nous allons nous intéresser ici à des exemples concrets, qui présentent pour le domaine une difficulté croissante.



NOTE

Les lecteurs curieux des notions mathématiques sous-jacentes, parfois un peu pointues, pourront utilement se référer aux ouvrages de référence cités en fin d'article, tandis que les lecteurs souhaitant simplement une explication succincte du domaine pourront se contenter de mes approximations, dont je m'excuse par avance auprès des puristes.

Comme la plupart des méthodes de ML, les algorithmes de RL sont agnostiques au sens des données qu'ils traitent. Ils ne demandent :

- qu'une description numérique de l'état du système à contrôler ;
- la liste des actions possibles ;
- et une indication de la qualité du contrôle (c.f. section 1.1).

On peut utilement comprendre cette indication de qualité comme les variations du score d'un jeu vidéo, ou des entrées-sorties d'argent.

Si l'on est capable d'abstraire le problème réel auquel on est confronté en ces trois composantes numériques, alors les algorithmes de RL permettent de trouver l'action optimale à chaque instant, quel que soit le problème. Les subtilités qui viennent assombrir cette ambitieuse déclaration sont nombreuses, et nous allons les explorer au travers de l'optimisation de plans de production (section 1), de l'étude de règles dynamiques de sécurité dans un environnement virtuel (section 2), du A/B Testing (section 3), de l'analyse de logs (section 4). Nous verrons aussi comment il est possible de gérer un modèle adversarial où l'incertitude n'est pas seulement liée à la stochasticité du monde (par exemple le jet de dé dans un jeu de plateau), mais aux choix délibérés d'un autre agent qui cherche à minimiser notre critère de réussite (section 5).

ALLOCATION OPTIMALE DES RESSOURCES

États, actions, critère

Nous sommes en 1937 et Leonid Kantorovitch vient de recevoir une lettre d'une usine de contreplaqué qui lui demande comment allouer les matières premières à ses 8 machines pour produire les 5 différentes sortes de

contreplaqué, dans les quantités demandées par le plan quinquennal.

C'est ainsi que commence la passionnante et tragique histoire de la programmation linéaire [Spufford2012]. La solution de Kantorovitch, qui lui vaudra le prix Nobel d'économie, ne sera jamais appliquée à l'échelle de l'Union. La résolution du problème implique des quantités qui ressemblent trop à des prix...

Le RL, ou plus précisément ici la programmation dynamique (*Dynamic programming*, D.P.) permet également de résoudre ce problème d'allocation des ressources. Il faut commencer par modéliser l'état du monde par un vecteur s_t indiquant pour la semaine t la quantité q_p de matière première disponible, le nombre d de semaines avant la fin du plan quinquennal et les quantités q_1 à q_5 des différentes sortes de contreplaqué déjà fabriquées :

$$s_t = (q_p, d, q_1, q_2, q_3, q_4, q_5)$$

Le camarade contremaître peut alors décider quelle machine $a_t \in \{1, \dots, 8\}$ doit être mise en route pour la semaine. En fonction de la machine qu'il a choisi de mettre en route, la quantité de matière première q_p va diminuer, et certaines quantités $q_1, \dots, q_5$ vont augmenter. Le temps disponible d va diminuer d'une semaine, amenant ainsi le système dans l'état s_{t+1} .

Cette évolution, à chaque pas de temps, d'un état s_t vers un autre s_{t+1} en fonction de l'action a_t choisie est appelée la *dynamique* du système.

Ce que l'on cherche, c'est une fonction π , appelée la politique, qui à chaque état possible s associe une action $a = \pi(s)$. Si la politique du contremaître est fixe, on se trouve alors dans ce que l'on appelle en mathématiques une chaîne de Markov (*Markov Chain*), dont la dynamique est celle du système, contrainte par la politique du contremaître.

Le contremaître étant libre de choisir sa politique de contrôle de l'usine, on va se placer dans le cadre plus général des processus décisionnels de Markov (*Markov Decision Process*, M.D.P). Ce cadre va l'aider à choisir la politique qui satisfait un critère d'optimalité. Ici, ce critère d'optimalité se mesure grâce à la différence, à la fin du Plan, entre les quantités produites par l'usine ($q_1, \dots, q_5$) et les quantités demandées par le Госплан ($q'_1, \dots, q'_5$) :

$$d \neq 0 \Rightarrow R(s_t) = 0$$

$$d = 0 \Rightarrow R(s_t) = -|q'_1 - q_1| - \dots - |q'_5 - q_5|$$

Avec ce critère, la sanction n'arrive qu'à la fin du Plan. Le contremaître peut optimiser localement : fixer fausse-

ment d à 0 et évaluer pour chaque semaine t la quantité d'ennuis $R(s_t)$ qu'il aurait si le plan finissait aujourd'hui. Il n'a plus qu'à choisir la machine a_t qui fera en sorte que la quantité d'ennuis $R(s_{t+1})$ qu'il aura la semaine prochaine si le plan finissait la semaine prochaine soit plus petite que $R(s_t)$.



NOTE

Par convention, l'on cherche à *maximiser* la somme des récompenses $R(s_t)$. Pour *minimiser* un critère (comme les ennuis du contremaître), il suffit d'en prendre l'opposé. Ici une différence entre la production et le plan entraînera une récompense d'autant plus négative que la différence est grande. La récompense maximale atteignable par le contremaître est 0.

Cette méthode correspond à l'intuition derrière certains algorithmes de programmation linéaire, qui ne s'embêtent pas à calculer les ennuis pour chaque état possible, mais suivent un chemin qui amène à la solution optimale sans explorer tout l'espace.

Les algorithmes de programmation dynamique arriveraient au même résultat, mais les calculs sont un peu plus laborieux :

- il faut résoudre la chaîne de Markov en calculant pour chaque état s la somme des ennuis $V^\pi(s)$ que l'on va avoir en écoutant, à partir de cette semaine et jusqu'à la fin du Plan le contremaître et sa politique π . Cette somme d'ennuis est appelée la valeur d'un état.
- Il faut ensuite encourager le contremaître à améliorer sa politique π en

une politique π' en choisissant à chaque état s_t l'action a_t qui nous fait aller dans l'état s_{t+1} dont la valeur $V^\pi(s_{t+1})$ est supérieure à celles des autres états qu'il est possible d'atteindre à partir de s_t .

- La politique étant modifiée, la chaîne de Markov a changé. Il faut maintenant calculer la nouvelle valeur V^π de tous les états.

Le cycle recommence jusqu'à ce que la politique ne varie plus : un des résultats centraux de la programmation dynamique et que ces itérations successives d'amélioration de la politique et de mise à jour de la valeur des états mène forcément vers un point fixe où la politique améliorée est la même que celle qui a été évaluée : c'est l'une des politiques optimales.

La programmation dynamique peut se résoudre avec cette famille d'algorithmes qu'on appelle itération généralisée de la politique (*Generalized Policy Iteration*, GPI) [Sutton1998], où les étapes d'amélioration et d'évaluation de la politique se succèdent plus ou moins rapidement. Dans les cas extrêmes, soit l'on met à jour la valeur de tous les états avant de changer la politique (c'est l'algorithme « d'itération de la politique », *Policy Iteration*), soit l'on change la politique dès qu'on réévalue la valeur d'un état (itération de la valeur, *Value Iteration*).

Ici, le problème pouvant s'exprimer par des contraintes linéaires sur l'espace d'état, la programmation linéaire nous permet d'arriver plus rapidement à la solution optimale.

Stochasticité et non linéarité

Mais le contremaître n'est pas un bleu. Il sait qu'il ne faut pas trop en demander aux machines : à chaque fois qu'on en utilise une sans l'avoir arrêtée une semaine pour la réviser, la probabilité de la voir bloquée par une panne augmente.

Avec un programme linéaire, on ne pourrait prendre ces contraintes en compte qu'en réduisant la capacité de production estimée des machines. Ce chiffre restant forcément une sur- ou sous-estimation du vrai chiffre selon que le contremaître organise correctement la maintenance, la solution trouvée n'est plus optimale.

Avec la programmation dynamique, en revanche, on peut facilement rajouter 8 composantes $m_1, \dots, m_8$ à l'espace d'état, qui indiquent pour chaque machine le nombre de semaines où elles ont fonctionné depuis la dernière maintenance, ou si une machine est en panne (avec une composante négative indiquant la gravité de la panne).

Il suffit alors de coder dans la dynamique l'évolution de la probabilité des pannes (qui n'est vraisemblablement pas linéaire), et dans le critère R le coût de réparation (qui n'est sans doute lui non plus pas linéaire, les pannes dues à une absence totale de maintenance sont souvent beaucoup plus graves que les pannes de routine) :

$$d \neq 0 \text{ et } m_1 \geq 0, \dots, m_8 \geq 0 \Rightarrow R(s_t) = 0$$

$$\exists i, m_i < 0 \Rightarrow R(s_t) = -m_i^2$$

$$d=0 \Rightarrow R(s_t) = -|q'_1 - q_1| - \dots - |q'_8 - q_8|$$

La dynamique indique, pour chaque état s_t et chaque action a_t non pas l'état s_{t+1} , comme je l'ai fait entendre jusqu'à présent, mais la distribution de probabilité de

l'état s_{t+1} sur l'espace d'état : utiliser la machine 3 peut nous faire arriver dans un état s_{t+1} où les quantités q_2 , q_4 et q_7 augmentent, ou dans un état où elles restent identiques et la machine 3 est en panne. Les mécanismes stochastiques sont ainsi pris en compte au cœur de la méthode.

Utiliser une machine n'ayant pas bénéficié d'une maintenance récente devient ainsi un pari, et le calcul de la valeur d'un état revient au calcul du compromis entre assumer le coût de la panne ou désobéir au planificateur.

La programmation dynamique permet au contremaître de remplir le Plan au mieux, en alternant les périodes de travail et de maintenance de ses machines.

Résumé

Pour résoudre un problème avec la programmation dynamique, il faut le modéliser sous la forme :

- d'un espace d'état qui encode toutes les données pertinentes sous la forme d'un nombre fini de vecteurs s ;
- d'un espace d'action qui présente, à chaque pas de temps, les choix a disponibles à l'opérateur ;
- d'un critère R que l'on appelle fonction de récompense, et qui est le coût instantané d'une action dans un état.

La famille d'algorithmes d'itération généralisée de la politique fournit alors une politique π^* qui permet de maximiser la somme des récompenses (et donc d'accepter une perte à un moment si elle permet un gain futur, ou d'éviter une perte plus grosse encore).

POLITIQUE AUTOMATIQUE DE SÉCURITÉ

Faites vos jeux

Fort de ces enseignements, vous décidez d'utiliser les indicateurs de votre système de détection d'intrusion (*Intrusion Detection System*, IDS) comme espace d'état :

- nombre d'utilisateurs connectés,
- nombre de CVE ces dernières 24 heures,
- taux de warnings et d'erreurs,
- *load average*, mémoire utilisée,
- débit entrants et sortants,
- nombre de flux simultanés,
- âge moyen des quelques plus anciennes connexions,
- etc.

Vous rencontrez là un des premiers pièges de la modélisation pour le RL : le vecteur d'état s doit contenir toute l'information nécessaire pour prendre la décision a . La politique π , pour trouver $a=\pi(s)$, ne doit pas avoir besoin de regarder la suite d'états qui nous a menés jusqu'à s .

Cela implique qu'il ne faut pas se contenter de placer les indicateurs instantanés de l'IDS, mais aussi quelques valeurs historiques plutôt récentes.

Par exemple, une diminution subite du nombre de login ratés peut indiquer que l'attaquant s'est lassé, qu'il a été banni par un dispositif du style *fail2ban*, ou plus embêtant, qu'il a réussi. Il peut être alors judicieux de savoir si le nombre d'utilisateurs connectés a augmenté de 1. Cela implique d'avoir les valeurs instantanées et historiques dans l'état s , ce qui a pour effet de faire gonfler spectaculairement la taille de l'espace d'état.

Afin d'endiguer un peu cette augmentation exponentielle de la taille, vous êtes amené à discrétiser certaines caractéristiques : ainsi par exemple les débits ne vont pas se mesurer en kbps, mais être divisés en trois valeurs : basse, moyenne, haute.

Les actions possibles sont assez simples :

- alerter l'équipe de réponse à incidents,
- éteindre telle ou telle machine,
- durcir ou adoucir les règles du firewall selon des configurations préétablies,
- déclencher une sauvegarde,
- etc.

C'est en cherchant à définir les probabilités de transition que les limites de la programmation dynamique vous frappent de front. Comment parvenir à chiffrer exactement, par exemple, les probabilités des variations relatives des flux entrants et sortants et du taux de warnings et d'erreurs ?

Un problème réel, même modérément complexe, est impossible à

modéliser dans le cadre de la programmation dynamique, sauf au prix d'approximations qui rendent très théorique l'optimalité de la solution trouvée.

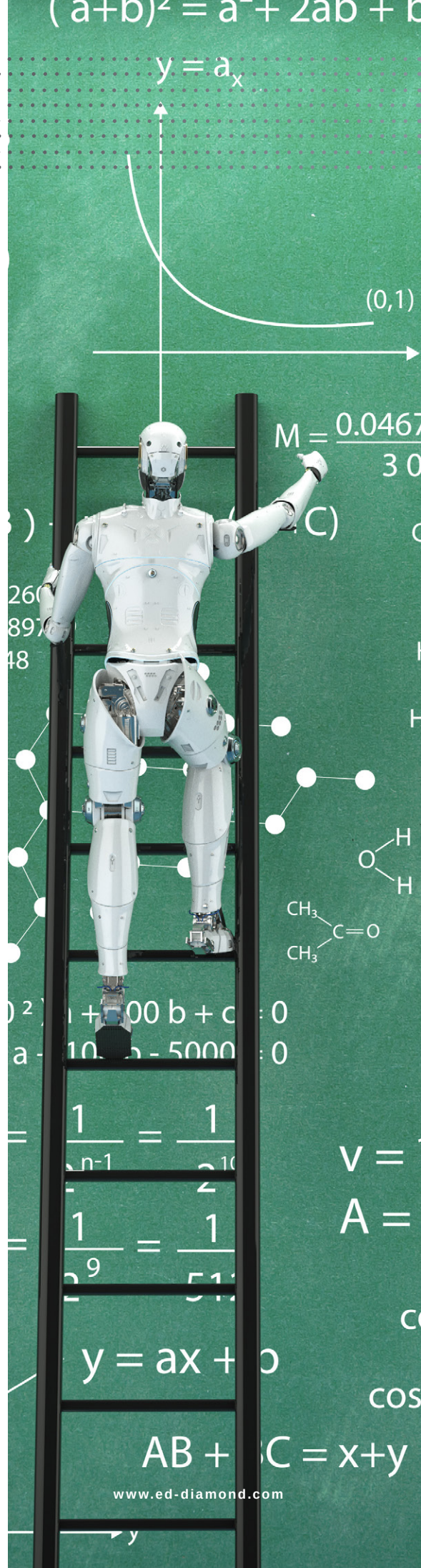
C'est là que le cadre devient celui de l'apprentissage par renforcement. Il va falloir bâtir un simulateur du système afin de pouvoir évaluer la valeur des différents états.

La méthode, le MCMC pour *Markov Chain Monte-Carlo* est conceptuellement simple (mais il existe de nombreuses subtilités [Andrieu2003]). Il s'agit de placer le simulateur dans un état s et de laisser la politique π réagir. Plusieurs fois. Pour tous les états s . À la fin, on a une très bonne approximation de la valeur V^* . La politique sera mise à jour à certains intervalles, dont le choix dépend du compromis entre itération de la politique et itération de la valeur.

Un premier souci est que les garanties d'optimalité de la politique qui existent pour les algorithmes de programmation dynamique tombent dès que l'on utilise des approximations. Le nombre d'échantillons qu'il est nécessaire de recueillir pour satisfaire les bornes théoriques est bien trop grand. En pratique cependant le MCMC fonctionne tant que la dynamique n'est pas complètement aléatoire, et les dynamiques réelles le sont rarement.

Le deuxième souci est la construction du simulateur. Bâtir un simulateur pour un jeu vidéo ou un jeu de plateau est comparativement assez trivial. Dans notre exemple, il faudrait :

- faire un parc de machines virtuelles pour simuler toutes les machines du réseau ;
- user et abuser des snapshots de ces machines pour revenir à un



état s après un run de la simulation, afin de lancer une nouvelle branche du Monte-Carlo ;

- demander au CERT et à la Red Team d'élaborer des scénarios d'attaque scriptés qui doivent permettre entre autres de définir la récompense R ;
- être capable de placer le simulateur dans chacun des états s .

La récompense R est calculée en fonction des actions de la politique de sécurité (par exemple : pénalité pour chaque utilisateur connecté à une machine que l'on choisit d'éteindre, pénalité pour avoir réveillé l'astreinte pour rien), mais aussi en fonction des attaques scriptées : pénalité pour chaque instant où l'attaque parvient à extraire des données, ou à miner des cryptomonnaies, ou à chiffrer les fichiers, etc., mais bonus pour un réveil de l'astreinte alors que l'attaque est en cours ou l'extinction d'une machine où l'attaquant se trouve.

Créer un tel simulateur nécessite une rigueur de développement considérable, mais cela reste encore dans le domaine du possible. Cela présente en outre l'avantage de pouvoir tester des politiques de sécurité définies à la main, et d'entraîner la Red Team et la Blue Team.

Il n'est pas nécessaire pour les besoins de la simulation d'utiliser de vrais exploits (sauf si ceux-ci se voient dans les indicateurs de l'IDS). Il suffit de les simuler en donnant aux scripts de la Red Team les informations nécessaires à se logger sur les machines cibles. Cela simplifie grandement le développement de ces scripts. Certains scripts peuvent être aussi simples que de déposer une charge virale sur une machine.

La seule difficulté insurmontable dès que le réseau dépasse quelques machines est de pouvoir placer le simulateur dans des points de départ qui correspondent à chacun des états s . Cette tâche est rendue ardue d'une part par le simple nombre d'états, qui même dans les problèmes triviaux dépasse allègrement le millier, et d'autre part par le fait que certains états sont très très improbables et correspondent à des configurations complètement tordues, qu'il sera difficile, voire impossible, de reproduire dans le simulateur.

Approximations

Tout n'est pas perdu cependant. Il n'est pas nécessaire, comme nous l'avons fait jusqu'à présent, de maintenir une représentation tabulaire de la valeur $V^\pi(s)$ des états s . C'est au contraire le meilleur moyen de faire exploser le temps de calcul.

En pratique, les valeurs de deux états proches ont toutes les chances d'être proches. Il existe bien sûr des discontinuités, qu'il faut prendre en compte, car ce sont elles qui permettent au RL de résoudre des problèmes non linéaires.

Il faut non pas garder un dictionnaire qui à chaque s associe sa valeur $V^\pi(s)$, mais utiliser une méthode de régression (faire du ML. supervisé, donc) afin de généraliser à tout l'espace d'état la valeur trouvée par la méthode MCMC sur quelques points de départ dans lesquels il est possible de placer le simulateur. Il n'est plus non plus nécessaire de discrétiser les données fournies par l'IDS (on peut à nouveau compter les débits en kpbs).

D'un point de vue pratique, on cumule les avantages : le simulateur est plus simple à coder et les calculs deviennent moins gourmands.

D'un point de vue théorique, c'est une catastrophe : on approxime une fonction V^π avec une méthode entraînée sur quelques points eux-mêmes approximés (en effet, MCMC ne renvoie qu'une approximation de $V^\pi(s)$).

Pour contourner cette catastrophe théorique, il faut s'assurer que la méthode d'approximation de la fonction de valeur travaille dans un espace d'hypothèses qui permet de représenter la vraie fonction de valeur. Il faut donc lui représenter l'espace d'état par le biais d'une fonction d'attribut (*feature*) qui met en avant les relations pertinentes.

Ce travail, mêlant à la fois connaissance théorique des méthodes de ML et connaissance métier du domaine (ici la détection et réponse à incidents de sécurité) est appelé le *feature engineering* et représente en pratique la part la plus importante de la mise en place d'une solution de RL.

Il n'existe pas de méthode générale. Le *feature engineering* est vu comme de « la triche » dans le milieu académique, car il permet à n'importe quelle méthode de briller. C'est pourquoi depuis quelques années la recherche se porte sur la détection automatique de features. Les réseaux de neurones s'en sortent très bien, mais il faut pouvoir les entraîner, il est difficile de comprendre les features qu'ils fournissent [Samek2017] et ils sont sujets à des attaques [Papernot2016].

A/B TESTING

Découragé par le travail de titan qu'est la programmation du simulateur, vous décidez plutôt de vous consacrer à un problème plus simple, dont la formulation date des années 50 et qui peut avoir un effet rapide sur votre entreprise : le *A/B Testing*.

Il s'agit de présenter aux visiteurs d'un site web deux versions du site, et de choisir automatiquement la version qui déclenche le comportement souhaité. Les considérations éthiques dépassent le cadre de cet article, mais sachez que ce type d'optimisation amène par exemple Facebook à enfermer ses utilisateurs dans une bulle d'opinion politique (ce qui ne les empêche pas de blâmer les utilisateurs [Bakshy2015]), ce qui peut amener à une radicalisation.

Ce problème est connu sous le nom du problème des bandits manchots, et est étudié pour tenter de résoudre le dilemme exploration exploitation : faut-il continuer à jouer à une machine à sous dont l'espérance est la meilleure connue (la version A du site), ou doit-on essayer la machine d'à côté dont l'espérance est peut-être supérieure (la version B du site) ? Quand doit-on s'arrêter ? Après tout, on a peut-être joué de malchance, si l'espérance de la machine B est inférieure, c'est dû au hasard, en jouant plus dessus on en aurait le cœur net.

Le RL permet de résoudre ce problème, avec des stratégies qui vont du très simple (<http://stevehanov.ca/blog/index.php?id=132>) au très complexe (<http://downloads.tor-lattimore.com/banditbook/book.pdf>). Néanmoins, il est important de noter que ce problème en apparence plus simple que le problème précédent présente en réalité une caractéristique qui le rend plus complexe : il est impossible de bâtir un simulateur.

La seule solution pour évaluer une politique π consiste donc à la mettre aux commandes et à la laisser choisir la version A ou la version B. Pour le A/B Testing, il n'est pas forcément très grave de présenter une version sous-optimale du site à quelques visiteurs, surtout si les deux versions ont été approuvées par les collègues du contrôle qualité. De toute façon, l'algorithme de RL convergera très vite vers la meilleure version.

Dans les tests cliniques, en revanche...

Il n'est pas toujours possible de bâtir un simulateur réaliste, et il n'est parfois pas possible de laisser une politique encore non optimale aux commandes du système. Comment dans ce cas évaluer la politique, étape nécessaire à son amélioration ?

APPRENDRE AVEC LES LOGS

Revenons au cadre de l'apprentissage d'une politique automatique de sécurité.

Si l'on ne donne pas à l'équipe de développement les moyens de développer le simulateur qui permettrait d'apprendre la politique de sécurité optimale, tout espoir n'est pas perdu. Il reste possible d'utiliser une classe d'algorithmes dits *off-policy*. Il existe des algorithmes de RL capables d'apprendre la politique optimale, même s'ils ne peuvent influencer les choix des actions afin d'en apprendre les conséquences.

La convergence des algorithmes *off-policy* est forcément plus lente et, si la politique de contrôle est lacunaire, la politique apprise ne pourra pas miraculeusement être optimale. Mais l'on dispose maintenant d'une solution qui permet d'apprendre à contrôler un système sans le casser pendant l'apprentissage.

Il suffit maintenant de logger les actions des administrateurs réseau pour pouvoir, avec le log de l'IDS, créer les données historiques $s_t, a_t \rightarrow R(s_t), s_{t+1}$

nécessaires à l'entraînement des algorithmes *off-policy*. Plus besoin de simulateur.

APPRENDRE À DÉFAIRE UN ADVERSAIRE

Certains problèmes présentent un élément incertain particulièrement difficile à analyser : un adversaire intelligent.

Une méthode qui a fait ses preuves en intelligence artificielle est la famille d'algorithmes dits *mini-max* : il s'agit de prendre l'action qui *mini*-mise l'espérance de gain de l'adversaire, en supposant que celui-ci joue parfaitement (qu'il *max*-imise son gain).

Pour appliquer la méthode MCMC, cela présuppose que l'on connaît la meilleure action que l'adversaire pourrait effectuer. Ce qui implique d'avoir déjà une stratégie optimale sous la main.

Dans les problèmes symétriques, il existe une astuce qui consiste à utiliser la même politique des deux côtés. La politique s'améliorant, on joue au fur et à mesure de l'entraînement contre un adversaire de plus en plus performant. C'est le principe utilisé par AlphaGo Zero [Silver2017], à la différence de la version originale, qui avait analysé un énorme nombre de parties jouées par les meilleurs joueurs du monde.

Si le problème est asymétrique, comme l'est l'attaque et la défense d'un périmètre informatique, cette astuce ne peut fonctionner.

CONCLUSION

Le RL permet de résoudre des problèmes stochastiques non linéaires. L'effort de modélisation doit nécessairement passer par la présentation du problème sous la forme d'un processus de prise de décisions séquentielles, avec un espace d'état dont les éléments s_t donnent toute l'information nécessaire à la prise de la décision a_t , entraînant le système à transitionner dans l'état s_{t+1} et à recevoir la récompense $R(s_t)$ au passage.

Selon les cas, on peut ensuite exprimer la dynamique sous la forme de probabilités de transitions explicites comme dans l'exemple d'allocation des ressources, sous la forme d'un simulateur comme dans l'exemple de la politique de sécurité, en mettant la machine aux commandes comme pour le *A/B Testing* ou enfin en analysant les données recueillies comme dans le cas des algorithmes *off-policy*.

Dans la plupart des cas, la modélisation mettra en jeu une approximation de la fonction de valeur (la somme des récompenses tout au long du fonctionnement du système). Trouver les bonnes *features* pour que cette approximation ne trahisse pas la théorie derrière les méthodes, là est l'art du Renforcement. ■

RÉFÉRENCES

La bible (régulièrement rééditée) est le livre de Sutton et Barto [Sutton1998], la bible en Français est le « PDMIA » [Sigaud2008].

Il n'y a pas d'implémentation libre de référence. Le cœur des algorithmes est en général assez trivial à implémenter (par exemple pour un petit jeu vidéo, l'auteur a codé Q learning en 30 lignes de Python disponibles sur : <https://github.com/edouardklein/Outsmart/blob/master/rl.py#L34>). La difficulté résidant dans le choix d'un bon schéma d'approximation, qui dépend du domaine d'application, il est difficile de proposer une librairie de référence comme l'est Scikit-Learn, par exemple. On pourra aussi lire [Geron2017] pour avoir des exemples.

[Spufford2012] F. Spufford, *Red Plenty*, Graywolf Press, 2012.

[Sutton1998] R. S. Sutton & A. G. Barto, *Reinforcement learning: An introduction*, MIT Press Cambridge, 1998. Disponible gratuitement en ligne (<http://incompleteideas.net/book/bookdraft2017nov5.pdf>).

[Andrieu2003] C. Andrieu, N. de Freitas, A. Doucet & M.-I. Jordan, *An introduction to MCMC for machine learning*, *Machine learning*, Machine Learning, pp 5-43, 2003.

[Samek2017] W. Samek, T. Wiegand & K.-R. Müller, *Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models*, <https://arxiv.org/abs/1708.08296>

[Papernot2016] Papernot, McDaniel, Jha, Fredrikson, Celik & Swami, *The limitations of Deep Learning in adversarial settings*, IEEE European Symposium on Security and Privacy (EuroS&P), 2016.

[Bakshy2015] Bakshy, Messing & Adamic, *Exposure to ideologically diverse news and opinion on Facebook*, Science, 5 June, VOL 348 Issue 6239, 2015.

[Silver2017] Silver, Schrittwieser, Simonyan, Antonoglou, Huang, Guez, Hubert, Baker, Lai, Bolton & others, *Mastering the game of Go without human knowledge*, Nature, 2017.

[Sigaud2008] O. Sigaud & O. Buffet, *Processus décisionnels de Markov en intelligence artificielle*, Hermes Science Publications, 2008.

[Geron2017] A. Géron : *Deep Learning avec TensorFlow, Mise en Œuvre et cas concrets*, Dunod, 2017.

GRAPHES GÉANTS CREUX : COMMENT DÉFINIR LE CENTRE DU WEB

Robert ERRA

Les graphes, composés de sommets et d'arêtes sont des objets communs en mathématiques (et indispensables) en informatique. Lorsqu'on veut manipuler des graphes de plusieurs centaines de millions de sommets, voire de plusieurs milliards de sommets, comme le graphe du web (ou un sous-ensemble) ou le graphe de certains réseaux sociaux, les choses se compliquent singulièrement : la plupart des algorithmes « académiques » se heurtent au « mur » de la complexité en temps (voire en espace), que nous appellerons le mur du « Big Data ». Tout algorithme dont la complexité est de l'ordre de $O(n^3)$ ou même de l'ordre de $O(n^2)$ est en fait inutilisable en pratique (ou très coûteux) dès lors que n , le nombre de sommets, dépasse (disons) le milliard. Il faut alors suivre d'autres stratégies. Il faut par exemple accepter de ne pouvoir calculer qu'une approximation même si dans certains cas, cette approximation peut en fait être la valeur exacte.

Cet article se propose de présenter la stratégie dite des « sauts multiples » (multiple sweep) et certaines tactiques développées ces dernières années pour résoudre un problème classique sur les graphes géants : le calcul du diamètre, du rayon et du centre d'un graphe non orienté. Nous finirons par discuter de la faisabilité de résoudre le problème du calcul du diamètre, du rayon et du centre d'un célèbre graphe non orienté : le « graphe du web ».

GRAPHES

Dessine-moi un graphe

Un graphe, au sens mathématique, est une structure composée d'objets (les sommets ou les nœuds) dans laquelle certaines paires d'objets sont « liées » par des arcs (on dit arête dans le cas où ce lien est orienté). Ainsi, lorsque vous regardez un plan de métro, que ce soit à Londres ou à Paris, en fait vous regardez un graphe dont les sommets représentent les stations et les arcs représentent les trajets entre les sommets. Les graphes sont très présents en informatique.

Un graphe G est défini par son ensemble de sommets, V , et l'ensemble des arcs, E : on note en général $G=(V,E)$ un graphe de n sommets, on aura par convention $V=\{v_1, v_2, \dots, v_n\}$ et l'ensemble des

arêtes est noté $E=\{e_{i,j} : i,j \text{ dans } \{1,2,\dots,n\}\}$. Le graphe peut être non orienté, i.e. $e_{i,j}=e_{j,i}$, ou orienté, pondéré (les arcs sont valués) ou non. La figure 1 montre trois exemples de graphes : non orienté et non pondéré, orienté et non pondéré et enfin, orienté et pondéré. Une des meilleures références est [3].

Quelques définitions classiques :

- Graphe simple : il ne comporte pas de boucles, i.e. $e_{i,i}=0$;
- Arête : lien qui joint deux sommets consécutifs, distincts ou non, d'un graphe non orienté ;
- Arc : lien qui joint deux sommets consécutifs, distincts ou non, d'un graphe orienté.
- Chaîne : dans un graphe non orienté, suite d'arêtes consécutives, i.e. chaque arête de la suite a une de ses extrémités en commun avec l'arête précédente et l'autre extrémité avec l'arête suivante, sauf pour la première et la dernière.
- Chemin : dans un graphe orienté, c'est une suite d'arcs consécutifs ;
- Distance (entre deux sommets) : c'est le nombre minimum d'arêtes d'une chaîne allant de l'un à l'autre ;
- Un graphe G non orienté est dit connexe si pour tous sommets u et v de G , il existe une chaîne allant de u à v .
- Une composante connexe d'un graphe est un sous-graphe connexe de ce graphe
- Si un graphe G est non connexe on peut considérer en général la plus grande composante connexe, c'est le plus grand sous-graphe connexe de G . Si le graphe est orienté, on peut alors utiliser la composante fortement connectée d'un sommet (*Strongly Connected Component*, SCC) : c'est l'ensemble des sommets qui sont atteignables à partir de ce sommet.

Problèmes sur les graphes

Il existe de nombreux problèmes, et de nombreux algorithmes sur les graphes, chacun résolvant un problème particulier, de manière exacte ou approchée, voir par exemple [3]. La figure contient les définitions de l'excentricité d'un sommet, du diamètre et du rayon d'un graphe G , dont nous allons discuter.

Quelques problèmes concernant les graphes :

- L'excentricité d'un sommet : soit u un sommet, l'excentricité d'un sommet est donnée par la distance qui le sépare du sommet le plus éloigné (en nombre d'arcs traversés). La figure 2 donne une définition mathématique.

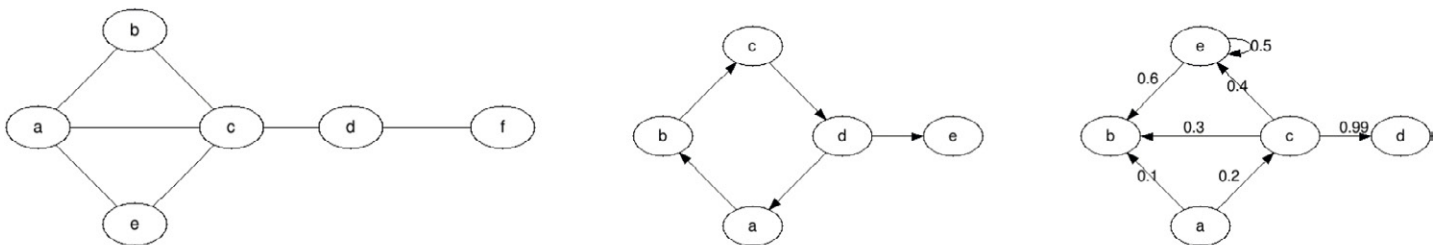


Fig. 1 : Graphe de gauche : non orienté, non pondéré ; au milieu : orienté non, pondéré ; à droite : orienté et pondéré.

Excentricité du sommet u : $Exc(u) = \max_{v \in V, v \neq u} d(u, v)$ (1)

Diamètre: $Diam(G) = \max_{u \in V} Exc(u) = \max_{u, v \in V, u \neq v} d(u, v)$ (2)

Rayon: $R(G) = \min_{u \in V} Exc(u) = \min_{u, v \in V, u \neq v} d(u, v)$ (3)

Fig. 2 : Excentricité, diamètre et rayon d'un graphe G (non orienté et supposé simple) et inégalités utiles.

- Le calcul du diamètre : la figure 2 donne la définition du diamètre à partir de l'excentricité d'un sommet, c'est le maximum des excentricités, en clair : le diamètre dit combien au maximum on doit sauter de sommet en sommet pour à partir de tout sommet u atteindre n'importe quel autre sommet ; si le graphe est non connecté, le diamètre est considéré comme infini, dans la suite nous ne considérerons que des graphes connectés.
- Le calcul du rayon : c'est le minimum des excentricités, il permet de définir le centre d'un graphe.
- Le calcul du centre : un sommet u dont l'excentricité est le rayon, i.e. $exc(u)=R(G)$, est dit faire partie du centre du graphe (qui est un ensemble éventuellement composé d'un seul point).
- Les plus courts chemins d'un sommet u donné à tous les autres : le chemin le plus court entre deux nœuds donnés u et v d'un graphe non orienté et non pondéré est un chemin qui minimise le nombre d'arcs traversés pour aller de u à v , l'algorithme de Dijkstra résout ce problème en $O(n^2)$, voir [3].
- Les plus courts chemins de tous les points entre eux : l'algorithme de Floyd-Marshall résout ce problème de manière exacte en $O(n^3)$, voir [3].
- Les parcours : parcours en largeur d'abord (BFS, qui nous sera très utile dans la suite) et parcours en profondeur d'abord (DFS), voir [3].
- Le calcul de communautés ; une communauté peut être définie de manière informelle comme étant un sous-ensemble des sommets fortement connectés entre eux, mais faiblement connectés aux membres des autres communautés ; calculer une décomposition en communautés est un problème NP-complet, quelle que soit la définition retenue, il n'existe donc que des algorithmes approchés, voir par exemple l'algorithme de Louvain, très efficace [4]. Ce problème est très fréquent dans l'analyse des graphes de réseaux sociaux.

nombre d'arcs : avec $G=(V,E)$, $|V|=n$ (nombre de nœuds) et $|E|=m$ (nombre d'arêtes). G est dit « creux » (*sparse*) si $m=O(n)$.

On pourrait y ajouter la contrainte : G provient du monde réel : graphe du web, graphe d'un réseau social, graphe du réseau BGP, d'où l'expression « graphe de terrain », en Anglais on trouve aussi bien *large graph*, *huge graph* que *very large graph*.

Une conséquence de cette définition informelle c'est qu'un graphe géant peut se stocker, du moment que le nombre d'arcs m est raisonnable, c'est à dire $m=O(n)$. On utilise pour ce faire une matrice d'adjacence stockée dans une structure creuse, méthode issue des éléments finis et qui date des années 1960 !

L'idée est simple (et vraiment très ancienne): on stocke les éléments de E qui sont non nuls et on ignore ceux qui sont nuls.

Donnons un exemple naïf. Soit A la matrice suivante :

$$A = \begin{bmatrix} 14 & 0 & 19 & 0 \\ 0 & 17 & 0 & 0 \\ 0 & 0 & 18 & 0 \\ 0 & 0 & 0 & 15 \end{bmatrix}$$

GRAPHES GÉANTS (MAIS CREUX) !

Un graphe géant (dit encore graphe de terrain) est un graphe qui a un grand nombre de nœuds (au moins quelques millions et jusqu'à quelques milliards voire dizaines de milliards). On dira qu'il est creux (*sparse*) s'il a un faible

Le code python permettant de la stocker (et de l'utiliser) est très court :

```
01 # csr_matrix(arg1[, shape, dtype, copy])
02 # Compressed Sparse Row matrix
03 from scipy import sparse
04 from numpy import array
05 I = array([0,0,1,2,3])      # indices de lignes
06 J = array([0,2,1,2,3])      # indices de colonnes
07 V = array([14,19,17,18,15]) # liste des valeurs des éléments non nuls
08 A = sparse.csr_matrix(I,J,V, shape=(4,4))
```

Nous dirons que A est une matrice creuse. Et (hourra !), de nombreux algorithmes de Scikit-Learn permettent (heureusement) de passer en arguments une matrice creuse, c'est le cas de Kmeans.

Un exemple de graphe géant creux ? En 1929, F. Karinyth, un auteur Hongrois, dans une courte nouvelle intitulée « *Láncszemek* » (soit « *Chaînes* ») suggéra que deux personnes prises au hasard sont en général séparées par au plus six degrés de séparation. C'est le véritable auteur de ce concept même si l'expérience réalisée par S. Milgram en 1967 est plus connue : à partir d'environ 250 volontaires et à l'aide de lettres postales, le nombre moyen de liens oscillait entre 4,4 et 5,7 (mais dépendait du sous-ensemble de volontaires choisis). En clair, suivant Wikipédia : sur toute la planète, chaque individu peut être connecté à un autre individu en suivant une chaîne d'amitié (ou de connaissances) ne contenant pas plus de cinq intermédiaires. Le graphe est colossal, plus de 7 milliards de sommets, mais très creux ! Théoriquement, on pourrait le calculer.

LE MUR DU BIG DATA : SMALL POWERS WIN !

Presque tous les cours d'algorithmique contiennent des algorithmes sur les graphes : calcul du diamètre, du rayon, calcul des plus courts chemins, parcours BFS ou DFS, etc. Mais, lorsqu'on veut manipuler des graphes de plusieurs centaines de millions de sommets, voire de plusieurs milliards de sommets, comme le graphe du web (ou un sous-ensemble) ou le graphe de certains réseaux sociaux, la plupart de ces algorithmes « académiques » se heurtent au « mur » de la complexité en temps (et espace), que nous appellerons le « mur du Big Data » : tout algorithme dont la complexité est de l'ordre de $O(n^2)$ ou même de l'ordre de $O(n^3)$ est en fait impraticable (ou très coûteux) dès lors que n , le nombre de sommets, dépasse (disons) le milliard. Même un algorithme de l'ordre de $O(n^{1+e})$ avec par exemple $e=0,5$ pose des problèmes dès que n devient assez grand. On ne peut accepter que des algorithmes dont la complexité en temps (et en espace) est de l'ordre de $O(n)$ ou éventuellement $O(n \log(n))$. Alors : Big Data versus Graphes Géants Creux ? Well, sorry but : *Small powers win !*

Il faut alors suivre d'autres stratégies. Il faut par exemple accepter de ne pouvoir calculer qu'une approximation même si dans certains cas, cette approximation peut en fait pour certains graphes être la valeur exacte. Nous y reviendrons dans la suite.

UNE STRATÉGIE POUR LE DIAMÈTRE : LES SAUTS MULTIPLES

Dans tout cours d'algorithmique un peu avancé traitant des algorithmes sur les graphes, pour le calcul diamètre, on trouve à coup sûr une remarque du style : il suffit de calculer toutes les distances entre tous les couples de sommets et le max est le diamètre, ce que fait l'algorithme de Floyd-Marshall de manière exacte en $O(n^3)$.



Mais est-ce si sûr ? En fait, dans la pratique à cause du « mur du Big Data » cette affirmation est fausse ou disons exagérée. Dès 1984, Lesk [5] aborde ce problème et fait remarquer qu'il faut distinguer :

- le cas où on désire le diamètre exact,
- du cas où une (bonne) approximation suffit.

Et il donne deux algorithmes approchés, l'un étant raisonnable en moyenne, mais dans le pire des cas en $O(n^3)$, mais ne nécessitant que $O(n^2)$ tandis que l'autre demande une mémoire dans l'ordre de $O(n^3)$ et a donc une complexité en temps de l'ordre de $O(n^3)$. Il a fallu attendre la publication du travail [1] pour voir apparaître de nombreux travaux comme [10,11,12,13,14,15].

Stratégie de base

La plupart des algorithmes de calcul d'une approximation du diamètre utilise la même stratégie : le *multiple sweep*, qu'on pourrait traduire par la stratégie des sauts multiples. Cette stratégie a été présentée pour la première fois et utilisée sur des vrais graphes géants dans la thèse de Clémence Magnien et publiée dans [1]. On trouvera le code en C et des exemples de « vrais » graphes géants dans le site [2].

Depuis de nombreux papiers ont été publiés pour proposer des variantes (dans le choix).

Voyons sur un exemple ce que cela donne avec deux « sauts » en considérant l'exemple du graphe de la figure 2 :

1. On part d'un sommet, par exemple le sommet u du graphe G (voir figure 3), on calcule son excentricité, par exemple par un BFS et on obtient le sommet v , ils sont plusieurs à avoir cette propriété, mais un seul suffit. On a une distance de 4 entre u et v : $\text{dist}(u,v)=4$.
2. On (re)part du sommet v dont on calcule là aussi l'excentricité, toujours avec un BFS et on obtient le sommet w . On a une distance de 6 entre v et w : $\text{dist}(v,w)=6$.

Dans la pratique, chaque BFS donne une valeur de plus court chemin, l'idée des algorithmes de *multiple sweep* est de combiner les informations données par chaque « saut ». Mais comment utiliser ces informations ? Donnons un pseudo-code :

```

00 Choisir NBMAX                # Nombre d'itérations max
01 Initialiser k=0              # k : Nombre itérations
02 Choisir u0 dans V            # au hasard ou selon une autre tactique(sommet de degré maximal,
sommet de degré minimal)
02 Calculer, via un BFS le sommet u1 tel que : Exc(u0)=d(u0,u1)=x0
03 BG=x0 ; BD=2*x0              # Initialisation de la borne gauche (BG) et de la borne droite (BD)
                                # on aura toujours BG ≤ D(G) ≤ BD
#
04 Tant que (BG différent de BD) & (k <= NBMAX) Faire
05     k=k+1
06     Choisir un sommet u0     # on peut prendre le précédent ou en choisir un autre
07     Calculer, via un BFS le sommet u1 tel que : Exc(u1)=d(u0,u1)=duv
08     Mettre à jour BG=max(BG,duv) et BD=min(BD,duv)
09     Si BG=BD retourner BG    # on a trouvé le diamètre sinon on continu
10 Fin du Tant Que
11 Retourner (BG,BD)            # ou retourner le milieu de [BG,BD] : (BD+BG)/2 ou sa partie entière,
                                # meilleure approximation possible

```

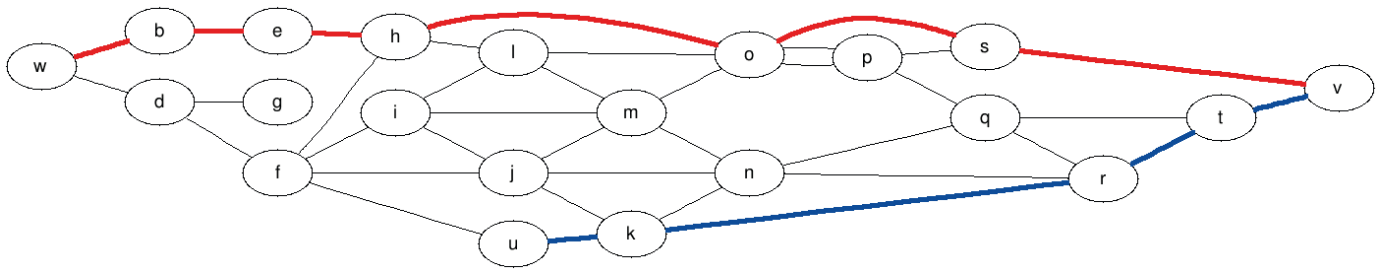


Fig. 3 : Une illustration de l'algorithme de sauts multiples (ici le double-sweep) : on part du sommet u , on veut calculer l'excentricité de u , on fait un BFS et un des points les plus éloignés (ils sont plusieurs, mais un seul suffit) est le sommet v ; un des chemins les plus courts est représenté en bleu. On (re)part ensuite du sommet v , dont on calcule l'excentricité, on (re)fait donc un BFS et le point le plus éloigné est le sommet w ; un des chemins les plus courts est représenté en rouge.

Ainsi, cet algorithme soit retourne une valeur unique (ligne 09) qui est la valeur exacte du diamètre, soit retourne un couple (BG, BD) tel que : $BG \leq D(G) \leq BD$. Et on peut encore évidemment améliorer l'approximation en choisissant comme valeur approchée de $D(G)$ la valeur milieu de l'intervalle $[BG, BD]$ (ligne 11).

La figure 3 résume ce qu'on a obtenu sur le graphe avec juste un double saut : 6 est une bonne approximation du diamètre, le lecteur se convaincra que c'est en fait le diamètre exact : repartir du sommet w donne à nouveau le sommet v , ce qui est un très bon signe.

Auto-vérification

L'intérêt de la stratégie précédente est qu'elle donne un algorithme auto-vérifiant (*self-checking*). En effet, il est facile de prouver que pour tout sommet u , on a une 1ère inégalité : $Ex(u) \leq D(G) \leq 2 \cdot Ex(u)$ qui entraîne une seconde inégalité $R(G) \leq D(G) \leq 2 \cdot R(G)$: la 1ère inégalité découle de l'inégalité triangulaire : pour tous sommets x, y , et z : $d(x, y) \leq d(x, z) + d(z, y)$.

TACTIQUES D'OPTIMISATION

Choisir les sommets pour mieux « sauter » ?

Comment choisir les sommets dans un algorithme de sauts multiples ? Lesk [5] donne quelques pistes, mais de nombreux autres auteurs en ont proposé d'autres (voir [2,10,11,12,13,14,15]) :

1. choisir un sommet de degré maximal
2. choisir le centre d'un arbre couvrant.

On peut y ajouter (ce qu'on a vu auparavant dans le pseudo code) :

- faire un BFS à partir d'un sommet, on arrive au sommet u ;
- repartir de u et arriver à v par un BFS.

Cela donne l'algorithme dit de *double-sweep* de [1]. Il est amusant de constater que dans les années 60 à 90 de nombreux algorithmes de réordonnancement pour les matrices creuses de l'analyse numérique utilisaient cet algorithme, on appelait les points (u, v) des points pseudo diamétraux, mais c'est la même idée, voir l'algorithme de Sloan [6] par exemple.

On peut aussi utiliser l'idée suivante, testée de nombreuses fois par les étudiants d'EPITA dans le projet du module « *Very Large Graphs* » :

- Calculer une valeur approchée du diamètre (avec les doubles bornes) et du rayon (borne supérieure).
- Pour le diamètre, procéder de manière classique.
- Pour le rayon, il faut approcher le centre ; pour ce faire : à chaque calcul de BFS, on obtient un arbre, on calcule donc le centre de cet arbre BFS (on laisse au lecteur le soin de vérifier que c'est rapide : en deux BFS, on calcule le centre d'un arbre !).
- Et on repart de ce sommet pour faire un BFS, etc.

L'idée c'est que les sommets diamétraux sont ceux qui sont les plus loin des sommets du centre. On peut ainsi alterner une mise à jour des bornes pour le diamètre et approcher le centre, ce qui donne une borne inférieure du rayon.

Notons que pour un cycle, il suffit de deux sauts pour avoir le diamètre exact et de même pour un arbre : deux BFS donnent le diamètre exact.

Un petit coup d'accélérateur ? Réordonner !

L'auteur ne résiste pas à décrire ici une tactique d'accélération qui donne de bons résultats, peu coûteuse et qui devient d'autant plus intéressante que le graphe est vraiment géant (de quelques centaines de millions à quelques milliards de sommets). Considérons l'algorithme en pseudo-code suivant :

```
# Algorithme de réordonnancement simple mais rapide
01 Choisir un sommet u, faire un BFS qui renvoie le sommet v
02 Faire un BFS à partir de v pour obtenir le sommet w
# On a juste fait un double-sweep, on peut faire plusieurs sauts de plus
03 Partir de w et faire un BFS,
04 Utiliser l'arbre du BFS pour obtenir une permutation P # plus facile qu'il n'y paraît
# Cette permutation P permet de réordonner les sommets
05 Stocker le graphe réordonné à partir de la permutation P
```

Théoriquement, nous n'avons fait que réordonner (*reordering* ou *renumbering*), donc en fait nous avons juste permuté les sommets (et les arcs !), et on obtient donc un graphe G' isomorphe au graphe G , en théorie cela ne change donc rien. En particulier, le diamètre de G' est identique à celui de G , de même pour le rayon. Disons le même si c'est clair pour le lecteur attentif : ceci ne sert à rien si le graphe est « petit », soit de quelques milliers à quelque centaines de milliers. Par contre, si votre graphe fait 10 ou 100 millions de sommets, alors normalement vous devriez voir le temps CPU nécessaire à certains calculs diminuer. Dans [7], les auteurs donnent des résultats intéressants, ainsi : pour le calcul du diamètre, on peut avoir un gain de temps de 50 % à 70 %, et même le chargement est légèrement plus rapide. Pour le calcul de communautés, on a facilement un gain de 30 %. Pourquoi ? L'explication la plus simple est qu'on réduit le nombre de défauts de cache mémoire (*caches misses*). Il suffit d'utiliser la commande Linux de profilage **perf**, les résultats dépendant de l'architecture mémoire de votre processeur, et de la taille des caches mémoire L1, L2 et L3.

Le coût du réordonnancement ? Cela dépend de l'algorithme utilisé, les algorithmes utilisés pour les matrices des éléments finis, Cuthill McKee ou Sloan [6] étant trop coûteux, un simple réordonnancement via un BFS, comme présenté dans le pseudo-code est préférable.

ET LES GRAPHS ORIENTÉS ? LE « GRAPHE DU WEB »

Considérons le graphe réel probablement le plus gros qu'on puisse calculer chez soi (avec beaucoup de CPU et beaucoup de mémoire) : le « graphe du web ». C'est le graphe utilisé par Google au tout début de l'algorithme *PageRank* qui permettait de calculer la popularité d'un page web et permettait au moteur de recherche de classer les réponses à une requête par mots-clés. Il est défini de la manière suivante :

- un sommet représente une page web ;
- un lien est créé entre le sommet u et le sommet v si la page web « u » contient un lien vers la page web « v ».

Il est gigantesque, on pense que Google n'a jamais exploité (i.e. stocké) plus de 20 % de ce graphe, il comporte aujourd'hui probablement plus de 70000 milliards de sommets ($7 * 10^{13}$).

De plus, ce graphe est dynamique, à chaque seconde des pages apparaissent et d'autres disparaissent, il est très instable en fait. Admettons qu'on puisse obtenir une copie de tout (ou plutôt d'une partie, restons raisonnable) de ce graphe, avec un bon crawler web (scrapy, Pyspider ou MechanicalSoup par exemple) une bonne bande passante et un gros paquets de Go de disque durs. Que peut-on faire avec ? On pourrait se poser comme questions (on rappelle que ce graphe est orienté) :

- Quel est son diamètre orienté ?
- Y a-t-il plusieurs sommets diamétraux ?
- Quel est son centre ? Plus exactement, y a-t-il un sommet ou plusieurs dans le centre de la plus grande composante fortement connexe ?

Curieusement, à la connaissance de l'auteur, la question du diamètre n'a pas été abordée depuis plus de 15 ans, et on ne recense que 4 publications à ce sujet. La dernière publication qui donne un résultat, calculé sur un graphe de plus de 200 millions de sommets et 1,5 milliards de liens est [8], avec des données d'Altavista, elle date de presque 15 ans ! Les auteurs donnent des résultats intéressants : ils écrivent « ... *the directed diameter of the SCC is at least 28* » qu'on peut traduire par : le diamètre orienté de la composante [géante] fortement connectée (*[giant] strongly connected component, SCC*) est « au moins » de 28, donc ce n'est pas le diamètre exact ! En effet, en 2003 les algorithmes à passes multiples, auto-vérifiants, n'étaient pas encore sur le marché. Ils montrent aussi que si un chemin existe entre deux pages, sa longueur moyenne est de 16. Mais ce travail n'aborde ni le calcul du rayon ni du centre, dommage.

Alors, est-ce faisable de calculer une approximation du diamètre et du rayon actuels ? Et de trouver au moins un sommet du centre ? L'auteur pense que oui, au moins sur un sous-ensemble du graphe et en disposant quand même d'assez de RAM. Le fait que le graphe soit non connexe et orienté oblige à modifier (un peu) les algorithmes auto-vérifiants développés pour les graphes non orientés, mais ce n'est pas très compliqué. Le lecteur intéressé pourra consulter le site de P. Crescenzi (voir la section « pour aller plus loin ») et lire l'article [14] (code disponible en ligne). On peut aussi citer un travail remarquable sur le « graphe de Facebook » (sic) : [10].

CONCLUSION

Nous n'avons fait qu'effleurer le sujet des graphes géants creux. Nul doute que ce domaine va croître. Donnons un exemple : le graphe de « tous » les malware connus. On pourrait créer de manière collaborative un graphe géant, mais creux a priori qui contiendrait toutes les informations connues sur les malware répertoriés. Un tel graphe permettrait par exemple de calculer des clusters (des communautés du graphe) suivant divers paramètres à l'aide de l'algorithme de Louvain (qui permet de gérer des graphes de plusieurs milliards de sommets et d'arcs sans difficulté). On pourrait évidemment aussi calculer le diamètre et le centre de ce graphe ! On peut même imaginer aller plus loin : calculer des arcs qui seraient des vecteurs de caractéristiques communes à deux malware (genre, nombre de sections communes, nombre d'imports et d'exports communs, nombre de ressources communes, etc.), on obtiendrait ce qu'on appelle un hypergraphe, mais du coup on peut clusteriser ces arcs et récupérer (éventuellement) des communautés de malware similaires ! ■

REMERCIEMENTS

Merci à tous mes collègues du LSE, en particulier à Marc Espie pour sa relecture, et à tous les (courageux) étudiants de la majeure SCIA d'EPITA pour leurs projets du module « Very Large Graphs » pour leurs idées et les nombreuses expérimentations qu'ils ont permis à l'auteur de faire. Merci à mon ex collègue Marwan Burelle pour son intérêt constant pour cette problématique, pour ses idées et pour ses codes. Merci à Clémence Magnien pour avoir mis son code en ligne ce qui a (re)déclenché l'intérêt de l'auteur pour cette problématique.

POUR ALLER PLUS LOIN

Si vous voulez des graphes géants creux, de vrais graphes de terrain, vous en trouverez :

- Sur le site [2] : essayez le code, un peu ancien, mais encore tout à fait pertinent, il contient plusieurs stratégies et 3 graphes (dont un très grand, prévoir assez de RAM !).
- Sur le site du projet SNAP : <https://snap.stanford.edu/data/>, un grand nombre de graphes.
- Sur le site <https://www.pilucrescenzi.it/wp/software/> : des graphes et des codes !

Pour celles et ceux qui voudraient utiliser un graphe géant creux sur un graphe du monde réel, lié à la sécurité ou pas, vous pouvez essayer (avec vos données) :

- Neo4j : code source libre, langage de requêtes Cypher, très utilisé dans l'affaire des Panama Papers [16].
- JanusGraph : fork de TITAN DB, si vous êtes intéressés, lisez donc la page <http://janusgraph.org/> où on trouve : « JanusGraph is a scalable graph database optimized for storing and querying graphs containing hundreds of billions of vertices and edges distributed across a multi-machine cluster. JanusGraph is a transactional database that can support thousands of concurrent users executing complex graph traversals in real time ». À noter que les algorithmes du calcul du diamètre n'aiment pas trop être « distribués » et qu'il vaut donc mieux avoir assez de RAM !
- Et enfin, une bonne librairie, disponible en C, en Python, en R : igraph [17].



Les références de cet article sont disponibles sur : <https://www.miscmag.com/>.

AccesSecurity

LE SALON EURO-MÉDITERRANÉEN
DE LA **SÉCURITÉ GLOBALE**

MARSEILLE CHANOT ■ 6 - 7 MARS 2019

SALON / COLLOQUE / RENDEZ-VOUS D'AFFAIRES



SÛRETÉ / SÉCURITÉ • CYBERSÉCURITÉ

MISC
PARTENAIRE

accessecurity.fr



#AccesSecurity



SECURING EVERY BIT OF YOUR DATA

CHAQUE ENJEU DE SÉCURITÉ EST DIFFÉRENT,
CHAQUE SOLUTION DOIT L'ÊTRE.

PRODUITS



IRMA[®] Enterprise

Une plateforme flexible d'analyse de fichiers, utilisant plusieurs moteurs d'analyse pour améliorer la détection des menaces.

Epona[®]

Une protection logicielle pour vos applications prévenant les attaquants de voler vos actifs et menacer vos utilisateurs.

SERVICES



Nos recherches de pointe en sécurité conduisent les organisations à une nouvelle posture de sécurité : **obliger l'attaquant, et non le défenseur, à s'adapter.**

- **Recherche de vulnérabilités : évaluer la sécurité CSPN**

Évaluation sécuritaire des produits, attaques logicielles et matérielles.

- **Reverse engineering : comprendre la sécurité**

Tests de protections (jeux, paiement, DRM,...), développement de patches, attaques hardware.

- **Vulnerability intelligence : trier les menaces**

Développement d'exploits pour tester des équipements (ex.: blueborne), analyse de patches, attaques par canaux auxiliaires.

- **Sécurité logicielle : construire la sécurité.**

Cryptographie (conception et optimisation), design sécurisé, développement de composants de sécurité.

FORMATIONS



Apprenez la sécurité avec ceux qui la pratiquent quotidiennement

- Reverse engineering comme un pro
- Applications Android du point de vue d'un reverse engineer
- iOS : sécurité des applications et de l'OS

LA FORMATION
REVERSE ENGINEERING
DÉBUTERA LE
26 NOVEMBRE
INSCRIPTION
ENCORE
POSSIBLE

Plus d'informations sur www.quarkslab.com

