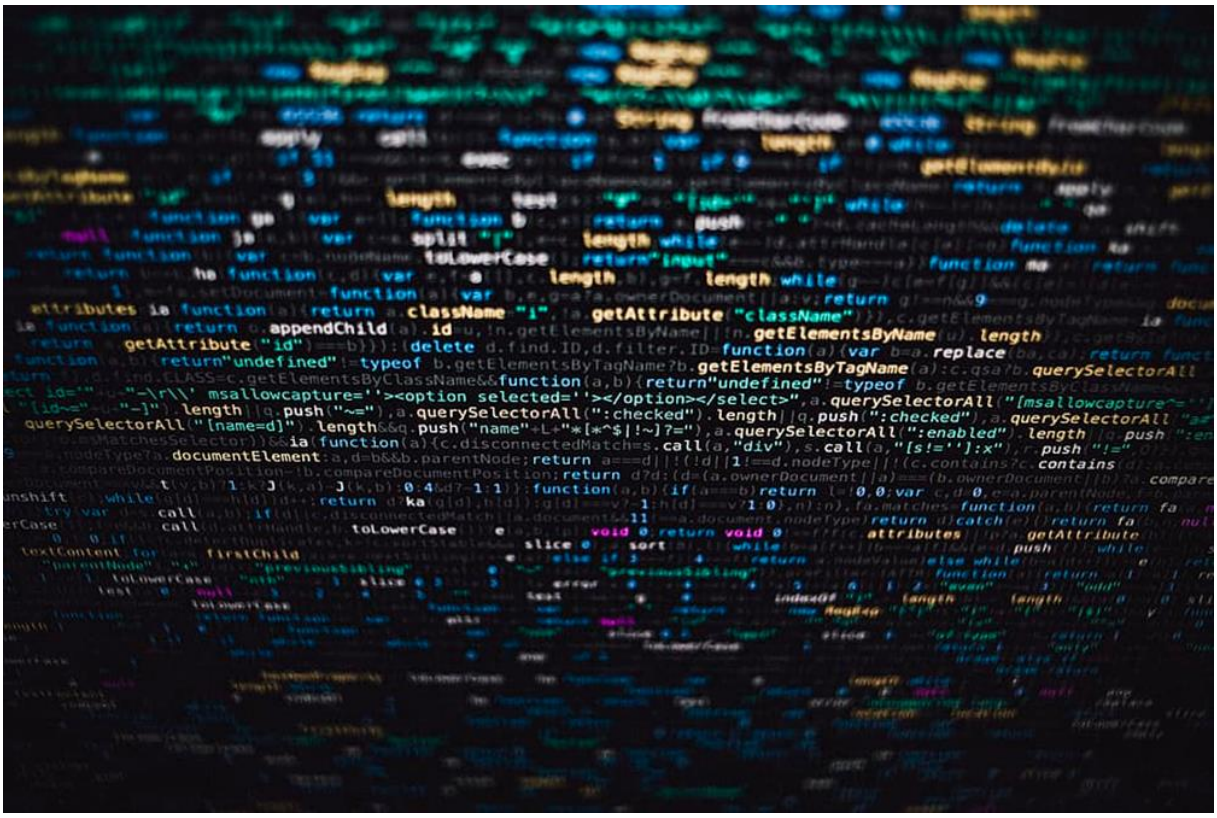


Mémoire de fin d'étude

L'identification de programmes malveillants par modèle d'apprentissage automatique

Comment des modèles d'apprentissage automatique identifient un programme malveillant d'un programme légitime ?

Alexis Delée	Architecture Logicielle
Marcel Adani	Architecture Logicielle

Maître de mémoire	Michel Oval
Date de soutenance	10 mars 2020

Table des matières

Introduction générale 1

CHAPITRE I : Apprentissage automatique et analyse de maliciels 5

I.1. Applications de l'IA dans la sécurité informatique 6

I.1.a. Sondes de détection 6

I.1.b. SecuML 7

I.1.c. Exemple d'un projet : identification d'URLs malveillantes 8

I.2. Ember 11

I.2.a. Présentation 11

I.2.b. Déploiement 12

I.2.c. Classification d'un échantillon 16

I.3. Format PE 18

I.3.a. Schéma général 18

I.3.b. Rôles des sections 19

I.3.c. Table des imports 21

CHAPITRE II : Les modèles de classification 22

II.1. Régression logistique 23

II.1.a. Classificateur binaire 23

II.1.b. Fonction à coût 26

II.1.c. Algorithmes de minimisation d'erreurs 27

II.2. Machine à support de vecteur 29

II.2.a. Classification à marge souple 29

II.2.b. Classification non-linéaire 31

II.3. Arbre de décision 33

II.3.a. Classification par arbre 33

II.3.b. Algorithme CART 36

II.4. Apprentissage d'ensemble 37

II.4.a. Système de vote 37

II.4.b. Entraînement des classificateurs 39

II.4.c. Boosting 39

CHAPITRE III : Comparaison des modèles 40

III.1. Indicateurs de performance 41
III.1.a. Matrice de confusion 41
III.1.b. Précision et rappel 42
III.1.c. Score F1 43
III.1.d. Courbe ROC 43
III.2. Entraînement des modèles 44
III.2.a. Validation croisée 44
III.2.b. Échantillons croisés 45
III.2.c. Architecture 45
III.3. Résultats 47
III.3.a. Hyperparamètres utilisés 47
III.3.a. Comparaison des performances 48
Conclusion générale 50
Notations 51
Annexes 52
Bibliographie 53

Table des figures

Figure I.1 Workflow pour un modèle de détection de PDFs malveillants 7

Figure I.2 Interface SecuML 8

Figure I.3 Mindmap de la classification d'URL 9

Figure I.4 Résumé de l'installation d'EMBER 13

Figure I.5 Architecture du package EMBER 15

Figure II.1 Régression logistique 23

Figure II.2 Représentation graphique de LOGIT 25

Figure II.3 Descente de gradient pour un vecteur à un coefficient 27

Figure II.4 Exemple d'un chemin 30

Figure II.5 Exemple d'un jeu de données 31

Figure II.6 Exemple d'une projection 32

Figure II.7 Arbre de classification du jeu de données IRIS 33

Figure II.8 Entraînement des modèles 37

Figure II.9 Exemple d'un modèle d'apprentissage d'ensemble 38

Figure III.1 Exemple d'une courbe ROC 44

Figure III.2 Exemple d'une validation croisée 44

Figure III.3 Schéma de notre architecture d'entraînement 46

Introduction générale

Les bénéfices entre *machine-learning* et sécurité informatique ont été prouvés économiquement par les diverses solutions d'antivirus adoptant des analyseurs faisant de la classification malveillante / non malveillante ou par clusterisation de familles de programmes malveillants.

La popularisation dans les entreprises d'outils de cybersécurité, comme les *pares-feux* intelligents, les antivirus ou les *sandboxs*, a augmenté le volume de la télémétrie disponible, ce qui a poussé des acteurs de la cybersécurité vers l'intelligence artificielle afin d'en faciliter l'analyse.

Cette approche s'inscrit dans un mouvement plus général d'automatisation de la cybersécurité, autant sur les moyens offensifs que défensifs. Ce sujet étant trop vaste pour le développer dans ce mémoire, nous focaliserons notre étude sur l'analyse de programmes malveillants.

L'analyse de programmes malveillants repose sur plusieurs activités :

- **L'identification** : un programme est-il malveillant ou non ?
- **La catégorisation / labélisation** : ce programme est-il un *trojan*, *ransomware*… ?
- **L'attribution** : à quelle famille ce programme malveillant appartient-il ?
- **La signature** : quelles règles écrire pour détecter ce programme malveillant ?

Les ressources humaines faisant ce type d'analyse sont limitées. On peut donc utiliser des techniques d'apprentissage automatique pour « écrémer » la télémétrie. Ces techniques sont :

- **La clusterisation** : utile pour l'attribution, elle permet de rapprocher de nouveaux échantillons à d'autres déjà labellisés. Cela peut permettre de détecter de nouveaux acteurs malveillants, ou de voir si leurs programmes se rapprochent de celui d'autres acteurs.

- **La classification** : faire de la rétro-ingénierie de programmes est une tâche longue et complexe. Un modèle d'apprentissage automatique peut rapidement indiquer sur un grand nombre d'échantillons ceux qui ont le plus de chances d'être malveillants.

Dans notre démarche d'étude, nous allons créer des modèles d'apprentissage automatique afin de classifier des programmes en programme légitime ou malveillant. Cela nous demande de récolter un grand nombre d'échantillons et de les analyser. Notre périmètre de recherche présente donc les contraintes suivantes :

- **Le temps** : l'analyse de programmes malveillants est une discipline où l'innovation est cyclique à cause de l'antagonisme des techniques de détection et de leurs parades. Face à l'apparition des *désassembleurs*, permettant de lire et d'analyser du code machine, des maliciels programmes malveillants basés sur des machines virtuelles logicielles sont apparus. Des outils de *sandboxing* ont alors permis de simplement exécuter un programme malveillant et d'avoir la liste de ses interactions avec le système d'exploitation et le réseau, afin de ne plus être ralentis par ces machines virtuelles. Et là encore, des programmes malveillants avec des codes de détection d'environnement en *sandboxing* ont vu le jour. Code qui en réponse a aussitôt été utilisé pour écrire des règles de signatures sur le postulat qu'un code qui n'a rien à cacher, n'a besoin de détecter son exécution en *sandbox*. Postulat qui s'avère faux car les éditeurs de jeux-vidéos, première victime du *cracking*, sont à la pointe du renforcement logiciel. Ainsi après avoir collecté des échantillons et entraîné un modèle, ce dernier pourrait être obsolète le temps de sa mise en production.

- **Le caractère limité de l'information** : l'information étant le nerf de la guerre, les derniers échantillons sont rarement publiés, ou leurs collectes se font par l'intermédiaire d'une API payante ou semi-public. Tous les échantillons ne sont pas non plus diffusés pour plusieurs raisons : certaines attaques s'inscrivent dans des contextes de secret-défense, ou bien des entreprises préfèrent éviter de communiquer sur leurs infections. Néanmoins, la récente prise de conscience des problèmes liés à la cybersécurité, et aux avantages de la transparence, ont conduit de nombreux chercheurs à publier leurs travaux,

notamment sur Twitter. D'autres projets tels que MISP (*Malware Information Security Plateform*) permettent aux différentes équipes d'analystes de partager leurs données pour faciliter la corrélation des informations.

- **Les technologies propriétaires** : les outils les plus performants en matière d'analyse sont généralement propriétaires, ce qui rend la constitution de jeux de données d'analyse dynamiques compliquée. Les collections d'échantillons récoltées au travers d'un service payant ne peuvent être diffusés, ce qui dans le monde de l'intelligence artificielle est un indispensable, tout résultat doit être publié avec son jeu de données afin de permettre la reproduction de l'expérience.

L'analyse d'un programme malveillant est un processus complexe se découpant en deux parties distinctes :

- L'analyse statique : étude d'un programme sans son exécution
- L'analyse dynamique : exécution du programme afin de détecter ses interactions avec le système d'exploitation

Ce découpage fait consensus dans le milieu, notamment par sa diffusion au travers du livre *Pratical Malware Analysis*. Les outils et les techniques d'analyse varient selon l'environnement ciblé par le maliciel.

Comme évoqué précédemment, la constitution d'un jeu de données d'analyse dynamique est complexe car de plus en plus de programmes malveillants tentent de détecter leur analyse en sandbox. Il nous faudrait alors vérifier chaque résultat avant de le joindre au jeu de données. Nous nous limiterons donc uniquement à l'analyse statique.

De plus, Windows étant le système d'exploitation le plus répandu, c'est aussi le plus ciblé par les attaquants. Ainsi les plus grandes collections d'échantillons sont des programmes au format PE (*Portable Executable*). Ce format définit la façon d'assembler et de compiler un binaire sous Windows. Il existe le format ELF (*Executable and Linkable Format*) pour Linux qui lui est très rassemblant. Les

informations et les métadonnées de ce format peuvent être extraites avec un programme appelé LIEF. Il existe ainsi une collection d'analyse LIEF qui sert de référence dans l'application du *machine-learning* à la détection de programmes malveillants, appelé EMBER (https://github.com/endgameinc/ember).

Ce mode d'apprentissage appelé PE-Feature n'a pour l'instant qu'un seul concurrent appelé N-Gram qui base sa modélisation sur les enchaînements d'OP-code (instructions assembleurs).

Notre objectif de départ était de faire de la classification de programmes à partir d'analyse dynamique, ce qui offrait l'avantage d'être plus agnostique aux formats des fichiers ou de la plateforme ciblée que le modèle N-Gram. Mais face aux contraintes de temps et de moyens, nous avons choisi d'étudier la création de modèles de classification binaire entraînés avec le jeu de données EMBER, pour faire de l'identification de programmes malveillants par classification binaire.

CHAPITRE I : Apprentissage automatique et analyse de maliciels

I.1. Applications de l'IA dans la sécurité informatique

I.1.a. Sondes de détection

Avec l'application de la LPM (Loi de Programmation Militaire) de 2013, un certain nombre de secteurs, et donc d'entreprises sont définies comme d'importance vitale. Elles doivent donc respecter une réglementation en matière de pratique de sécurité, et ainsi s'équiper en diverses solutions de sécurité, comme des sondes détection.

La détection réseau ou applicatif repose sur des signatures (Yara, Snort ou Suricata). Ces signatures doivent être écrites puis envoyées aux sondes de détection. Le délai entre l'apparition d'une menace, l'écriture des signatures et leurs déploiements sur les sondes rend la détection inopérante sur cette menace pendant ce temps. L'apprentissage automatique peut permettre de détecter les menaces pendant ce délai, même si l'entraînement de modèles et leurs mises en production posent d'autres problèmes :

- Le crédit a apporté aux résultats des modèles
- Le choix de l'apprentissage supervisé qui pose les mêmes problèmes que la détection par signature : à savoir le délai entre le réentraînement d'un modèle et son déploiement
- Le choix de l'apprentissage non-supervisé qui est sensible à de nouveaux types d'attaque : à savoir la soumission de données proche de la frontière de décision afin de la faire évoluer

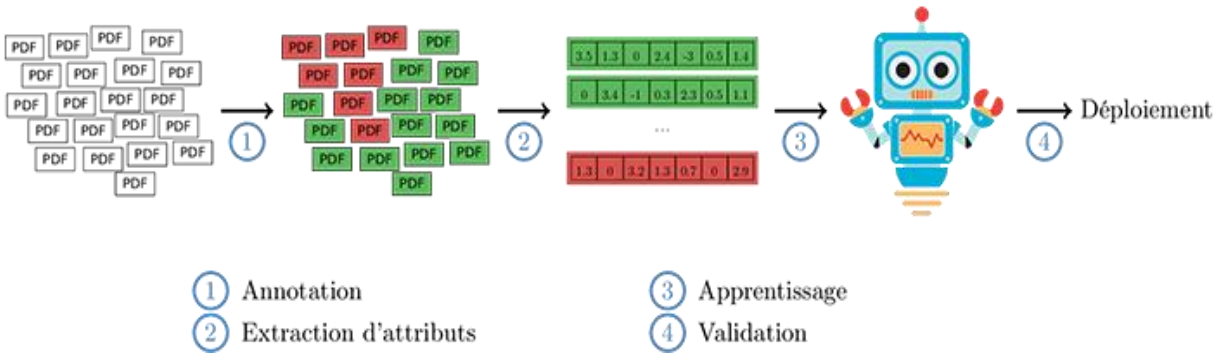


Figure I.1 Workflow pour un modèle de détection de PDFs malveillants

I.1.b. SecuML

L'application de l'apprentissage automatique à la sécurité informatique demande l'adéquation entre deux champs de compétences vastes et complexes, sujet l'un et l'autre à de fréquentes innovations. De plus la demande sur ces deux marchés de l'emploi est supérieure à l'offre. Les nombreux frameworks d'apprentissage automatique, tels que Numpy, Scikit, voire Tensorflow, permettent déjà à de nombreux experts de d'autres domaines de mobiliser des modèles pour un coût d'apprentissage moindre.

En effet, ces frameworks, développés en Python, un langage haut niveau, implémentent les techniques de création de jeux de données d'entraînement et de tests, d'optimisation des hyperparamètres, le parallélisme pour les phases d'apprentissage et les algorithmes des différents modèles.

C'est dans cette optique que le projet libre SecuML a été développé par l'ANSSI (https://github.com/ANSSI-FR/SecuML). Il vise à faciliter l'entraînement et la comparaison de différents modèles sur un même problème de classification ou de clusterisation. De plus il fournit une interface visant à rendre transparent la prise de décision par les modèles.

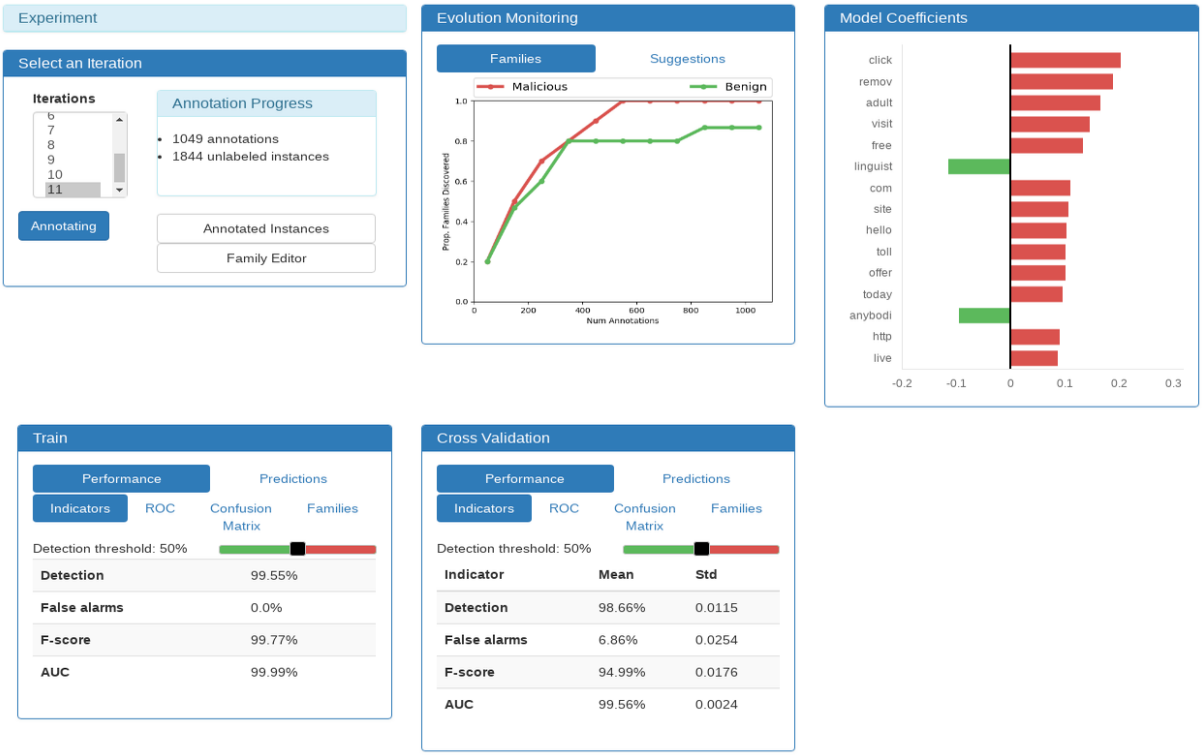


Figure I.2 Interface SecuML

I.1.c. Exemple d'un projet : identification d'URLs malveillantes

La détection et le blocage des liens malveillants permettent d'éviter de nombreuses attaques. Les adresses malveillantes peuvent être de plusieurs sortes :

- Lien de téléchargement de programmes malveillants
- Adresse de paiement de rançon dans le cadre d'une attaque par *ransomware*
- Url frauduleuse pour de l'hameçonnage, telle que faceebook.com
- Adresse d'un serveur de commande ou d'exfiltration de données dans le cadre d'une infection par un *cheval de Troie*

La détection de ces URL (*Uniform Resource Locator*) a été le premier point d'entrée du *machine-learning* dans la sécurité. Grâce à la technique de classification, l'URL est-elle « légitime » ou non ?

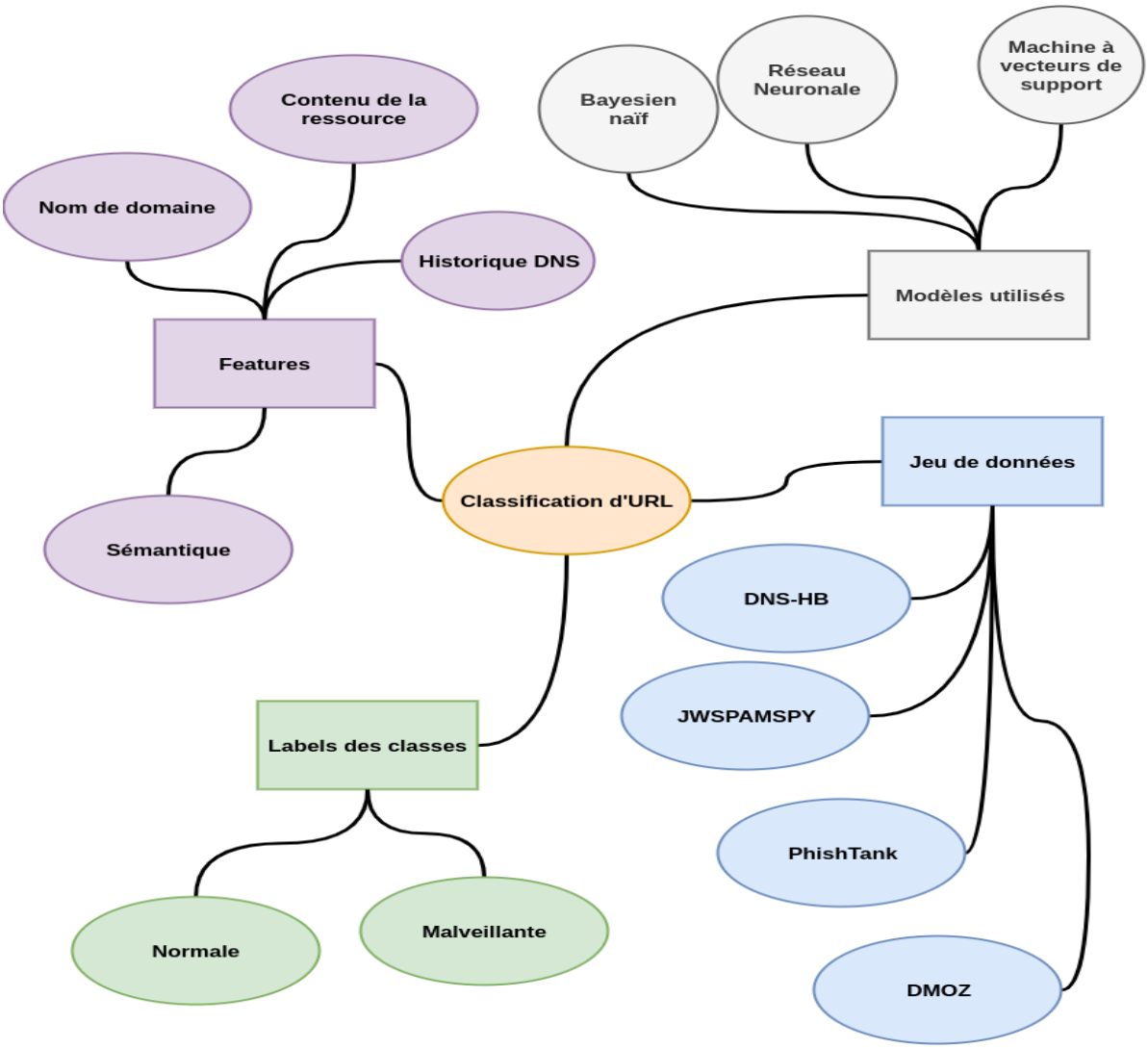


Figure I.3 Mindmap de la classification d'URL

Un classificateur *bayésien* traite chaque propriété ou *feature* d'un échantillon indépendamment des autres. Par exemple un fruit sera reconnu comme une pomme s'il est rouge, rond et de taille moyenne. Ne pas prendre en compte l'indépendance des propriétés entre elles, ici dans l'exemple elles découlent toutes du code génétique du fruit, permet d'entrainer un modèle avec une grande quantité de données. L'important étant leur diversité, ou *entropie*.

La modélisation du problème est une étape importante avant la collecte et l'entraînement d'un modèle d'apprentissage automatique. Dans notre exemple, les variables sont de natures diverses :

- Nom de domaine : les attaquants, notamment dans le phishing, jouent sur l'inattention et la force de l'habitude chez les utilisateurs en ajoutant des *tokens* aux URLs légitimes (exemple : '.', '/', '?', '=', '-' avec facebook-fr-com.biz)
- Le contenu du site web où les métadonnées de celui-ci peuvent entrer en compte : ratio de code CSS / code HTML, nombre de liens entrants / sortants, taille totale de la page
- La sémantique : les URLs légitimes sont généralement *human-readable*, une analyse fréquentielle des lettres peut fournir une information supplémentaire
- Historique DNS : peut prendre plusieurs aspects tels que le nombre d'adresse IP différentes vers lesquelles le nombre de domaine renvoie, ses périodes d'inactivité

C'est ensuite pendant l'entraînement que le modèle définira l'importance des variables dans la classification. Cette étape repose sur le modèle suivant :

1. La collecte de données labélisées : une URL est ajoutée au jeu de données en précisant sa catégorie
2. Le pré-entraînement des données : les données sont nettoyées. Ici on peut enlever par exemple les paramètres GET associés à l'URL, le nom du protocole (https://), normaliser l'encodage (%20 devient un espace)
3. La création d'un jeu d'entraînement : un des moyens de prévenir le surentraînement est d'entraîner un modèle avec une partie du jeu de données et de le tester avec le reste
4. L'entraînement du modèle : itération de classification avec le jeu d'entraînement afin de corriger les coefficients associés aux features tant que le score de prédiction du modèle augmente significativement

I.2. Ember

I.2.a. Présentation

Le projet EMBER (*Endgame Malware Benchmark for Research*) est une collection de plus de 1,1 millions d'entrées dans sa version 2018. Chaque entrée est une analyse statique d'un fichier PE réalisée avec LIEF (https://github.com/lief-project/LIEF).

Ce jeu de données a été constitué pour permettre la comparaison de différents modèles ou approches sur la classification et la clusterisation de maliciels, parmi lesquelles on retrouve :

- Les modèles bayésiens
- L'analyse des chaînes de caractères
- L'entropie des bytes

Le jeu de données est divisé en :

- Un jeu d'entraînement de 900'000 échantillons
    - 300'000 malicieux
    - 300'000 bénins
    - 300'000 non labelisés
- Un jeu d'entraînement de 200'000 échantillons
    - 100'000 malicieux
    - 100'000 bénins

Les chercheurs Hyrum S. Anderson et Phil Roth, à l'origine de ce projet, déclarent avoir rencontrés les problèmes suivants dans la réalisation de ce dernier :

- **Restrictions légales** : les échantillons ne sont pas fournis, seulement leurs analyses. Car hormis les bases de données gratuites comme CirusShare et VX Heaven, les échantillons récupérés depuis des services payants comme VirusTotal ou les échantillons bénins protégés par copyright ne peuvent être distribués librement.
- **Problème de labélisation** : contrairement aux images d'animaux, la labélisation d'un échantillon est subjective. Les noms attribués peuvent dépendre de normes propres aux entreprises, être faux, ou modifiés avec le temps.
- **Précautions de sécurité** : distribuer un aussi gros volume de maliciels peut être dangereux s'il se retrouve entre de mauvaises mains

I.2.b. Déploiement

Actuellement, le projet EMBER est en train d'être porté en python 3. Une des dépendances du projet, LIEF, a déjà été portée dans cette dernière version. Il s'agit de la bibliothèque s'assurant de l'extraction des features sur un PE. Il est donc nécessaire d'assurer une rétrocompatibilité avec les précédentes versions afin de garantir l'homogénéité du jeu de données d'analyse.

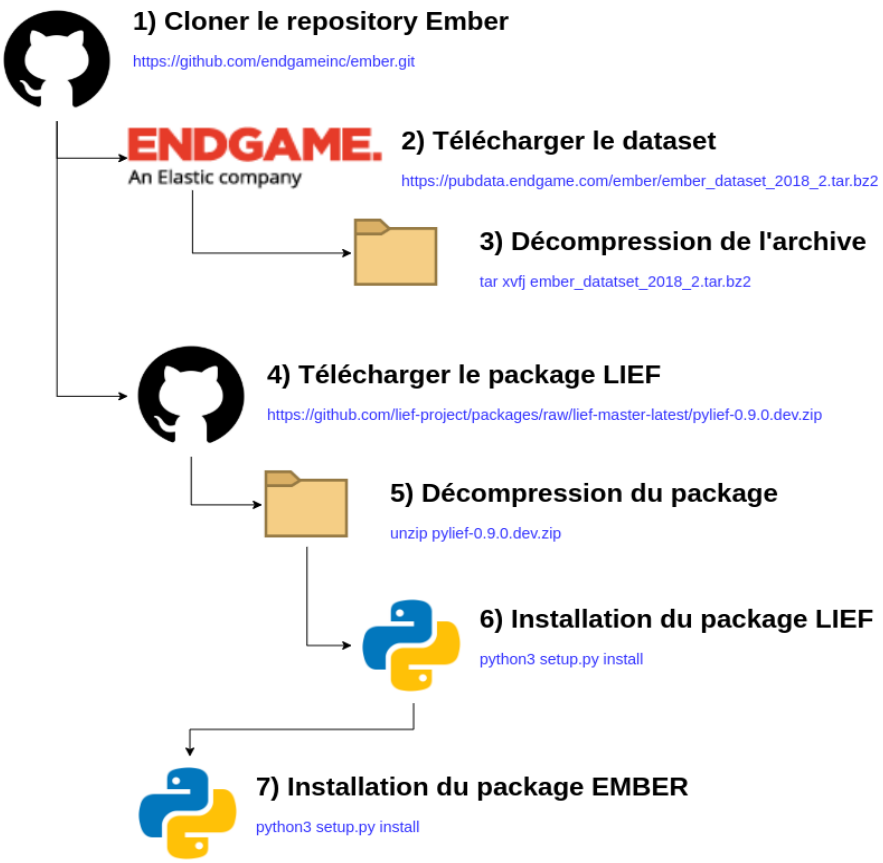


Figure I.4 Résumé de l'installation d'EMBER

L'installation du package nécessite la version 0.9.0 de LIEF en passant par l'archive présente sur le repository Github (*https://github.com/lief-project/packages/raw/lief-master-latest/pylief-0.9.0.dev.zip*).

Le dataset d'analyse LIEF est à télécharger sur les serveurs d'Endgame (*https://pubdata.endgame.com/ember/ember_dataset_2018_2.tar.bz2*).
Il contient le fichier *ember_model_2018.txt*, un modèle LGBM (expliqué plus tard dans ce mémoire) sérialisé et généré par le script *train_ember.py*.

LIEF est un package python développé par QuarksLab destiné entre autres à parser des fichiers binaires exécutables.

EMBER propose un modèle LGBM entraîné avec les paramètres par défaut. Le package contient deux fonctions liées à l'entraînement :

- train_model() : la fonction contient les paramètres suivants :

```
params = {
    "boosting": "gbdt",
    "objective": "binary",
    "num_iterations": 1000,
    "learning_rate": 0.05,
    "num_leaves": 2048,
    "max_depth": 15,
    "min_data_in_leaf": 50,
    "feature_fraction": 0.5
}
```

Durant nos tests, nous avons pu remarquer qu'il est préférable d'ajouter le paramètre d'early-stopping ou d'augmenter le learning-rate.

- optimize_model() : une fonction implémentant la recherche par grille pour optimiser les hyperparamètres :

```
param_grid = {
    'boosting_type': ['gbdt'],
    'objective': ['binary'],
    'num_iterations': [500, 1000],
    'learning_rate': [0.005, 0.05],
    'num_leaves': [512, 1024, 2048],
    'feature_fraction': [0.5, 0.8, 1.0],
    'bagging_fraction': [0.5, 0.8, 1.0]
}
```

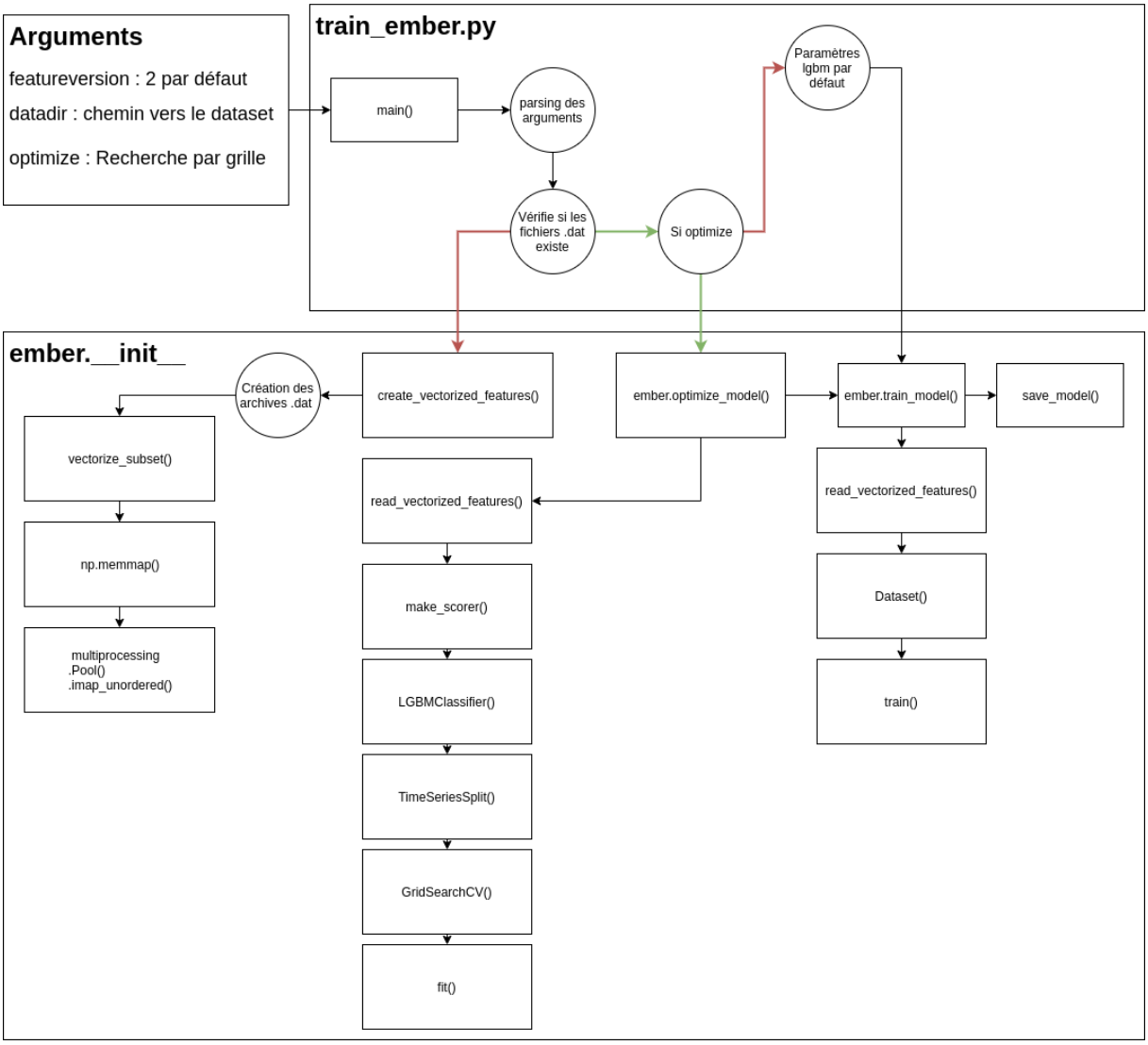


Figure I.5 Architecture du package EMBER

L'argument *featureversion* renvoie à la version de LIEF utilisée pour extraire les features des échantillons. Plusieurs datasets étant disponibles, certains ont été réalisés avec une version antérieure de LIEF.

I.2.c. Classification d’un échantillon

Afin d’évaluer rapidement le modèle fourni dans le dataset, nous avons téléchargé quelques programmes labélisés comme malveillants sur le site app.any.run, et extrait des programmes légitimes du système d’exploitation Windows. Nous avons ensuite demandé au modèle de les classifier.

Voici les résultats obtenus :

Nom	Label	Résultat
agenttesla1.bin	Malveillant	0.9999980425610477
masad1.bin	Malveillant	0.9971127990042151
agenttesla2.bin	Malveillant	0.9451598376937728
njrat1.bin	Malveillant	0.999998620691217
lokibot1.exe	Malveillant	0.002277468267835311
bfsvc.exe	Sain	1.1069421001873718e-08
explorer.exe	Sain	1.5571337146861159e-09
HelpPane.exe	Sain	2.2130217967818003e-08
notepad.exe	Sain	6.567443727007072e-08
regedit.exe	Sain	2.1704916075521367e-07
RtkBtManServ.exe	Sain	2.5760130876208956e-08

On constate que le modèle a correctement classifié 10 échantillons sur 11. Cette erreur (lokibot1.exe) peut s’expliquer par la nature du onzième échantillon. Il s’agit d’un *loader*. Son but est de télécharger un autre programme malveillant une fois exécuté, d’où sa dénomination concaténant deux extensions de fichier .png.exe (voir l’analyse sur app.any.run).

Afin d’éviter d’être identifié par les antivirus, ce type de programme malveillant tente au maximum de ressembler à un programme légitime. De plus, la charge utile est *packé*. On peut néanmoins remarquer un écart significatif entre la prédiction de lokibot1.exe avec les programmes légitimes.

Pour classifier un programme avec EMBER, il faut passer en argument à la fonction ember.predict_sample() le code binaire du fichier. Ce code sera ensuite analysé avec LIEF pour renvoyer un JSON contenant les features. Ce JSON est accompagné de métadonnées sur l'échantillon :


```
{
   "sha256":
"ca65e1c387a4cc9e7d8a8ce12bf1bcf9f534c9032b9d951d5d3aed7a12f8d8ed",
        "md5": "e93049e2df82ab26f35ad0049173cb14",
        "appeared": "2007-02",
        "label": 1,
        "avclass": "gepys"
}
```


La définition des labels est la suivante :

- 0 : programme sain
- 1 : programme malveillant
- -1 : programme non identifié

Les créateurs d'EMBER ont identifié deux groupes de features :

- Les parsed features
    1. General file information : contient des informations telles que la taille physique (sur le disque) et la taille virtuelle (taille dans la mémoire). Si le fichier a une section debug, le nombre de fonctions importées y sera également.
    2. Header information : cette partie détaille les métadonnées du header et certaines fonctionnalités liées à la compilation
    3. Imported functions : liste des fonctions importées et des DLL associées
    4. Section information : liste les sections, leurs permissions (lecture / exécution notamment). Si elle contient du code ou de la donnée, alors

leurs noms, leurs tailles virtuelles et leurs entropies y seront également. Cela permet de détecter un éventuel packing.

5. Exported functions : propre aux DLLs, contient la liste des fonctions accessibles
6. Strings information : contient le nombre de chaîne de caractères, leurs tailles moyennes, une liste contenant la taille de chaque chaîne, le nombre de chaînes affichables, l'entropie et le nombre de celles représentant un chemin, une URL ou une clé de registre

- Les Format agnostic features
    1. Byte histogram : découpe le programme en 256 morceaux et additionne les bits sur chacune des parties
    2. Byte-entropy histogram : découpe le programme en 256 morceaux et calcule l'entropie de chacune des parties

I.3. Format PE

I.3.a. Schéma général

Un *Portable Executable* est un format de fichier binaire. Il a été conçu par Microsoft comme standard pour les fichiers exécutables et les bibliothèques pour le système d'exploitation Windows. De plus, les questions de portabilité entre les plateformes 32 bits et 64 bits sur Windows sont résolues lors de la compilation, et donc le format PE est indépendant de ces questions.

Le format PE se décompose en plusieurs parties :

- Un header MZ-DOS : contient le nombre magique « MZ »
- Le segment DOS : contient l'adresse vers le header PE

Ces deux parties servent à assurer la rétrocompatibilité entre les différents loaders, notamment ceux présents sur les systèmes Windows NT

Un header PE contient plusieurs informations permettant aux lanceurs d'initialiser correctement le PE en mémoire, tel que :

- Une **signature** identifiant le programme comme un PE
- Le **point d'entrée** du programme indiquant l'adresse à partir de laquelle démarre le code à charger en mémoire
- Les **adresses physiques** des sections pour correctement les charger en mémoire avec les bonnes permissions
- L'**ImageBase** permettant de calculer les positions relatives des adresses virtuelles une fois le programme chargée en mémoire

Le format PE est complexe et le détailler ici ne servirait pas notre propos. Nous nous attarderons sur les éléments les plus pertinents dans l'analyse statique de programmes malveillants.

I.3.b. Rôles des sections

Un programme PE est chargé en mémoire dans des pages possédant des permissions différentes. On peut créer ses propres sections mais la norme est basée sur les sections suivantes :

- **.text** : contient les instructions assembleurs du programme
- **.rdata** : contient les données en lecture seule, comme les variables déclarées en static dans un code C
- **.bss** : contient les adresses vers des variables non-initialisées, comme les pointeurs NULL
- .data : contient les variables initialisées
- **.reloc** : contient les adresses virtuelles des adresses mémoires (sert à convertir ces adresses lorsque le programme n'a pu être chargé à l'adresse contenue dans l'ImageBase)

- **.idata** : contient soit la table des imports, soit les adresses mémoires des fonctions provenant des librairies externes
- **.rsrc** : contient le code binaire des ressources du programme comme les images, les sons ou les icônes
- **.edata** : uniquement pour les librairies, contient la liste des fonctions pouvant être appelées

La présence de toutes ces sections n'est pas obligatoire pour le bon fonctionnement d'un programme.

L'étude des sections peut permettre à l'analyste de programmes malveillants de savoir si le programme est *packé* ou non, voire même de savoir de quel packer il s'agit. Si les routines de chiffrement du packer sont connues, l'analyste pourra ainsi « depacker » statiquement le programme.

En effet, certains packers utilisent toujours les mêmes noms de sections contenant le code binaire chiffré. On peut aussi identifier la présence d'un packer en comparant la différence entre la taille virtuelle d'une section, le nombre d'octets occupés dans la RAM avec la taille, et le nombre d'octets occupés sur le disque. SI l'écart entre ces deux valeurs est trop grand, cela signifie que la donnée a été compressée.

Pour extraire le code binaire packé, un analyste devra exécuter le programme, poser un point d'arrêt logiciel à la fin de la routine de dépackage, extraire le code binaire de la RAM dans un nouveau fichier et reprendre son analyse depuis le début avec le nouveau programme.

I.3.c. Table des imports

Afin de diminuer la taille des programmes et pour éviter que le même code soit dupliqué, un système d'exploitation offre des fonctions dites natives, mais aussi des librairies, appelées DLL (*Dynamic Load Librairies*) dans l'écosystème Windows.

Ces librairies pouvant avoir des versions différentes et donc les fonctions qu'elles contiennent des adresses différentes, l'interopérabilité des imports de ces fonctions est assurée par l'IAT (*Import Address Table*).

Les entrées dans cette table se font par nom de DLL, et pour chacune on peut trouver la liste des noms de fonctions importées et/ou leurs numéros ordinaux.

Si les standards ne sont pas respectés, les noms des fonctions peuvent ne pas être présents, ou bien on peut avoir directement l'adresse virtuelle de la fonction dans la DLL, ce qui ne permet pas d'assurer la rétrocompatibilité de sa librairie.

Cette table peut permettre de rapidement identifier ce que fait un programme. Un binaire de petite taille appelant uniquement des fonctions pour communiquer avec Internet et pour exécuter des commandes a de forte chance d'être malveillant. Afin de camoufler l'activité des programmes malveillants, leurs développeurs importent des fonctions bégnines et utilisent de l'import dynamique dans leurs codes afin d'utiliser des fonctions sans qu'elles n'apparaissent dans cette table.

La table des exports, ou l'EAT, n'est présente que dans les librairies et expose aussi la version actuelle de celle-ci. Pour faciliter leurs importations dans d'autres programmes, la liste des fonctions est également mise à disposition avec les valeurs ordinales des fonctions.

CHAPITRE II : Les modèles de classification

II.1. Régression logistique

II.1.a. Classificateur binaire

Le modèle de régression logistique est le pendant de la régression linéaire dans le domaine de la classification.

Un classificateur binaire est un modèle ne renvoyant que deux valeurs, 0 ou 1. Il permet de classifier un échantillon entre deux catégories : ici programme sain ou programme malveillant.

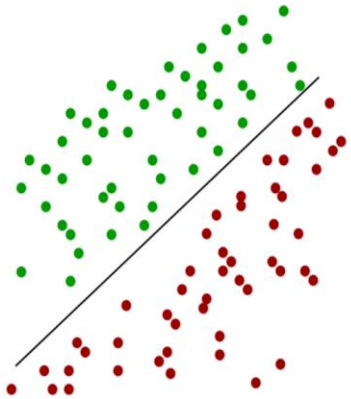


Figure II.1 Régression logistique

$Y \in \{0,1\}$ avec Y le résultat de la classification du modèle.

Le modèle de régression logistique peut être utilisé pour prédire plus de classes en testant tour à tour l'appartenance à chaque classe. Mais nous ne détaillerons pas cet aspect ici.

Le but du modèle est de définir une frontière de décision, ou *Boundary Decision*. La classification d'un échantillon se fera ensuite par rapport à sa position vis-à-vis de cette frontière.

Beaucoup d'exemples illustrent la régression logistique avec des graphiques et une frontière de décision sous la forme d'une droite, ou de plan si l'exemple à trois features. Néanmoins, un modèle de régression logistique peut prendre en argument des échantillons à N features, ce qui n'est plus représentable graphiquement. Dans ce cas, seules les mathématiques nous permettent de modéliser le problème.

La frontière de décision est donc d'une dimension N - 1 pour un modèle à N dimensions. Dans le cas d'un modèle à une feature, une dimension, les échantillons sont placés sur une ligne, la frontière de décision est un point. Dans le cas d'un modèle à deux dimensions, les points des échantillons sont placés dans un plan, le séparateur est une droite. Et dans le cas d'un modèle à trois dimensions, les échantillons sont placés dans un espace et le séparateur est un plan. Au-delà, le séparateur est appelé un hyperplan.

La classification se fait en deux étapes :

- Calcul de la probabilité avec la fonction Score $S(X^i)$ avec X le vecteur de feature associé à un échantillon
- Normalisation de la probabilité avec une fonction logistique de type sigmoïde

La fonction Score est le produit vectoriel du vecteur des poids, ou coefficients, avec celui des features :

$$S(X^i) = Coefficient_i * Feature_i$$

$$S(X^i) = Biais + Coefficient_1 * Feature_1 + Coefficient_2 * Feature_2 + Coefficient_3 * Feature_3$$

$$S(X^i) = \sum_{i=0}^{n+1} Coefficient_i * Feature_i$$

Le biais correspondant au $Coefficient_0$ multiplié par 1 afin de permettre la formulation algébrique linéaire de type $f(x) = a * x + b$

Une fois le score calculé, représenté par y, cette valeur est envoyée dans la fonction logistique, ou *Logit* :

$$LOGIT(y) = \frac{1}{(1 + e^{-y})}$$

$LOGIT(10000000) = 1$

$LOGIT(100) = 1$

$LOGIT(10) = 0.99$

$LOGIT(0) = 0.5$

$LOGIT(-0.5) = 0.377$

$LOGIT(-1) = 0.268$

Cette formule permet de border les résultats qui lui sont envoyés dans un intervalle compris entre 0 et 1.

Si le résultat est inférieur à 0.5, il est arrondi à 0 sinon à 1.

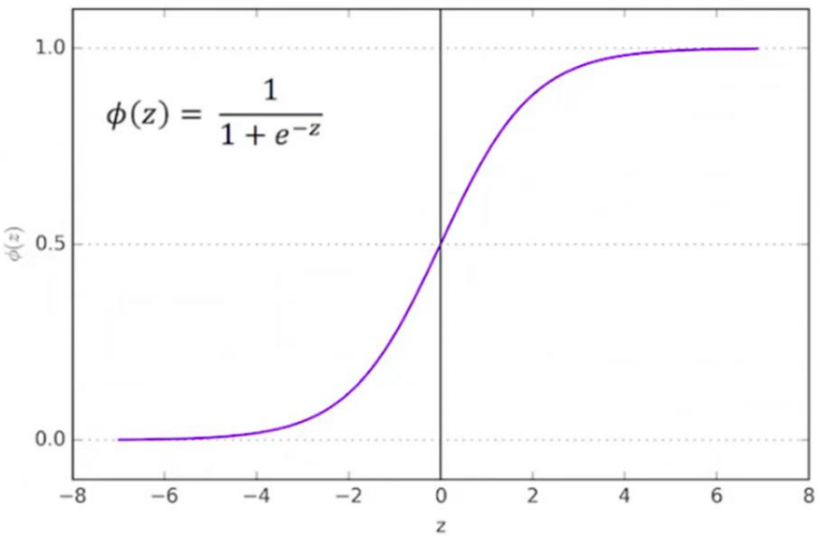


Figure II.2 Représentation graphique de LOGIT

II.1.b. Fonction à coût

L'enjeu de l'entraînement d'un modèle de régression logistique est d'identifier les meilleurs coefficients associés aux features. Le modèle utilise pour cela une fonction de coût dont l'objectif sera sa minimisation au travers d'itérations successives.

Cette fonction intervient après la fonction Score, à la place de la fonction logistique, pendant la phase d'entraînement. Son but est de pondérer les probabilités élevées si l'échantillon testé est du label 1, $-\log(y)$, et les probabilités basses si l'échantillon est du label 0, $-\log(1-y)$.

Soit x un échantillon, et y, la probabilité du modèle sur cet échantillon :

$Si\ Label_x = 1\ et\ que\ y = 0{,}9\ alors\ Co\hat{u}t(y) = 0.105$

$Si\ Label_x = 1\ et\ que\ y = 0{,}45\ alors\ Co\hat{u}t(y) = 0.798$

$Si\ Label_x = 0\ et\ que\ y = 0{,}9\ alors\ Co\hat{u}t(y) = 2.302$

$Si\ Label_x = 1\ et\ que\ y = 0{,}05\ alors\ Co\hat{u}t(y) = 0.05$

On calcule ensuite la moyenne du coût de classification pour l'ensemble du jeu d'entraînement afin d'évaluer la performance du vecteur de coefficient choisi.

Le but de l'entraînement est donc de trouver le vecteur de coefficient minimisant le coût des classifications. Cette recherche peut se faire de plusieurs façons selon le contexte.

II.1.c. Algorithmes de minimisation d'erreurs

Il existe plusieurs algorithmes pour minimiser la fonction de coût et donc de trouver les valeurs optimales des coefficients appliqués au vecteur de feature.

La descente de gradient est l'un des plus courants. Le gradient se calcule par la dérivée partielle de chaque coefficient, ce qui dans le cas des fonctions linéaires donnent des fonctions convexes ou concaves. La dérivée partielle nous donne la variation d'une fonction selon la variation du paramètre dérivé.

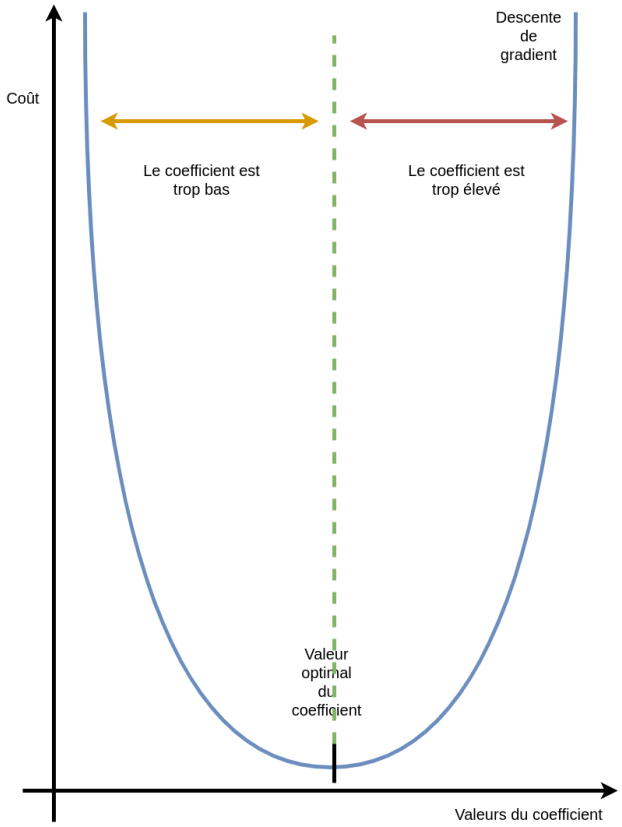


Figure II.3 Descente de gradient pour un vecteur à un coefficient

Le gradient se base sur le vecteur des fonctions dérivées de chaque coefficient. Le but est donc de trouver la valeur minimale du gradient, valeur à partir de laquelle le coût ne fait que d'augmenter. La classification se faisant entre 0 et 1, le gradient est une fonction convexe et d'où l'expression : la descente de gradient.

La convergence vers la valeur optimale se fait au travers d'itérations successives en se déplaçant sur la courbe du gradient selon une valeur définie par un paramètre appelé taux d'apprentissage.

Cette valeur est cruciale car un taux d'apprentissage trop élevé pose le risque de « rater » le minimum. Tandis qu'un trop faible rend l'optimisation plus longue.

Calculer à chaque itération, le gradient serait très couteux, on utilise alors deux méthodes de descente : la descente stochastique, qui sélectionne au hasard un échantillon et son coût de classification, pour en calculer le gradient. Cette méthode permet de se rapprocher plus rapidement de l'optimum, mais aura des difficultés à l'atteindre. L'autre façon est appelée la descente par mini-lot. Elle prend plusieurs échantillons pour réaliser l'optimisation des coefficients.

Dans certains cas, la descente de gradient peut être contre-productif si le gradient présente des minimums dit locaux. D'autres algorithmes permettent de réaliser cette optimisation.

Nom	Détails	Inconvénients
Méthode de Newton	Utilise des dérivés partiels de plus hauts niveaux	Coût de calcul plus élevé. Plus sensible aux minimums locaux
LBFGS (*Limited-memory Broydeen-Fletcher-Goldfarb-Shanno Algorithm*)	Estime les matrices et valeurs utilisées dans la méthode Newton	Peu efficace avec un jeu de données de grande taille
Liblinear	Se base sur des résolutions géométriques pour approcher l'optimum	Ne peux être parallélisé. Ne fonctionne pas dans une classification avec plusieurs catégories
SAG (*Stochastic Average Gradient*)	Découpe la descente de gradient en sous-fonction	Ne supporte que les pénalités de type L2. Complexité algorithmique élevée O(n)
SAGA (*Stochastic Average Gradient Algorithm*)	Se base sur SAG mais en intégrant le support des pénalités L1	

Afin de limiter la complexité d'un modèle et donc les temps de calcul à chaque itération, on peut appliquer deux types de pénalités :

- La **pénalité L1** qui permet de limiter la taille et le nombre de coefficients. Les couples coefficients-features les moins influant sur les résultats de la classification sont éliminés.
- La **pénalité L2** qui est semblable à la pénalité L1 mais qui ne fixe aucun coefficient à 0.

II.2. Machine à support de vecteur

II.2.a. Classification à marge souple

Les modèles SVM (*Machine à Support de Vecteur* ou séparateur à vaste marge) sont aussi des classificateurs linéaires mais présente deux atouts majeurs :

- La recherche de la meilleure frontière de décision
- La possibilité d'avoir des frontières de décision non-linéaire

Alors que la classification par perceptron est dite probabiliste, elle cherche à avoir les meilleures probabilités de classification, les modèles SVM cherchent à obtenir les meilleures classifications, et sont donc dit déterministes.

La méthode par perceptron consiste à utiliser une frontière de décision aléatoire, puis de sélectionner un point au hasard. Si ce point est mal classé, le séparateur est rapproché du point selon le taux d'entraînement défini. Le but est de répéter le processus pour améliorer le séparateur défini par la frontière de décision.

SVM utilise de la classification à marge souple afin d'être moins sensible aux données aberrantes (données mal observées ou dépendante de facteurs non pris en compte dans la modélisation). Ces données ont un impact plus fort sur la frontière de décision dans un modèle probabiliste.

Il est important de préciser que les calculs du modèle SVM, étant de nature géométrique, il devient crucial de réduire l'échelle des valeurs entre les features afin de ne pas pénaliser celle avec des valeurs plus basses. Par exemple si on prend un modèle basé sur le PIB en euros et le nombre de villes de plus d'un million de personnes, le PIB sera alors exprimé en millions ou milliards d'euros.

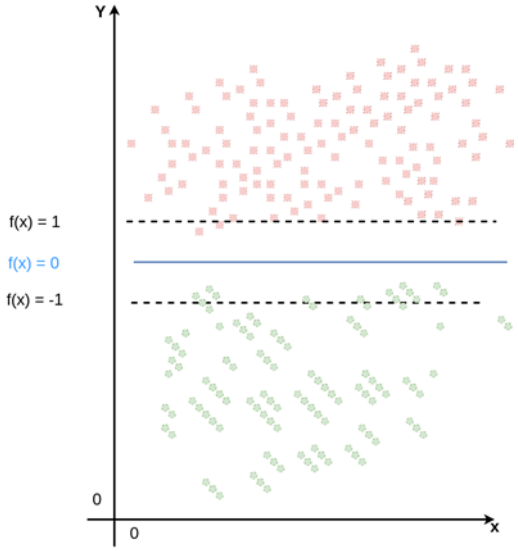


Figure II.4 Exemple d'un chemin

La première étape de SVM consiste à définir une frontière de décision séparant correctement les échantillons en deux groupes distincts.

On définit une frontière de décision aléatoire $f(x) = Vecteur_{coefficient}.Vecteur_{feature} + Biais$ tel que $f(x) = 0$.

Cette frontière aura deux chemins $f(x) = 1$ et $f(x) = -1$.

Ainsi les labels des classes devront être -1 ou 1, et non 0 ou 1 comme dans la régression logistique.

Les vecteurs de support sont les points situés sur les chemins. Un point à l'intérieur du chemin est appelé *empiètement de marge*.

Le but est d'obtenir un compris entre la largeur du chemin et le nombre *d'empiètement de marge*, ce qui revient à arbitrer entre la précision et le surentraînement du modèle.

Il existe plusieurs types de SVM avec des optimisations mathématiques différentes. Une des approches consiste à utiliser la validation croisée pour avoir plusieurs sous-ensembles du jeu d'entraînement, puis de définir pour chaque sous-ensemble des séparateurs avec la méthode du perceptron. Ces différents séparateurs sont ensuite appliqués sur l'ensemble du jeu d'entraînement. Une fois les vecteurs de support identifiés, le séparateur avec le moins de données mal classifiées est sélectionné.

II.2.b. Classification non-linéaire

Il est possible avec SVM d'ajouter une dimension à un jeu de données pour le rendre linéairement séparable. Cette technique est appelée *kernel trick*, et utilise la projection mathématique.

Supposons le jeu de données suivant ne contenant qu'une seule feature :

Figure II.5 Exemple d'un jeu de données

Tel quel, il n'est pas linéairement séparable. On peut lui appliquer un noyau de transformation, ou *kernel*. La projection fera passer le plan R au plan R^2, avec la fonction de projection $Projection(x) = x^2$.

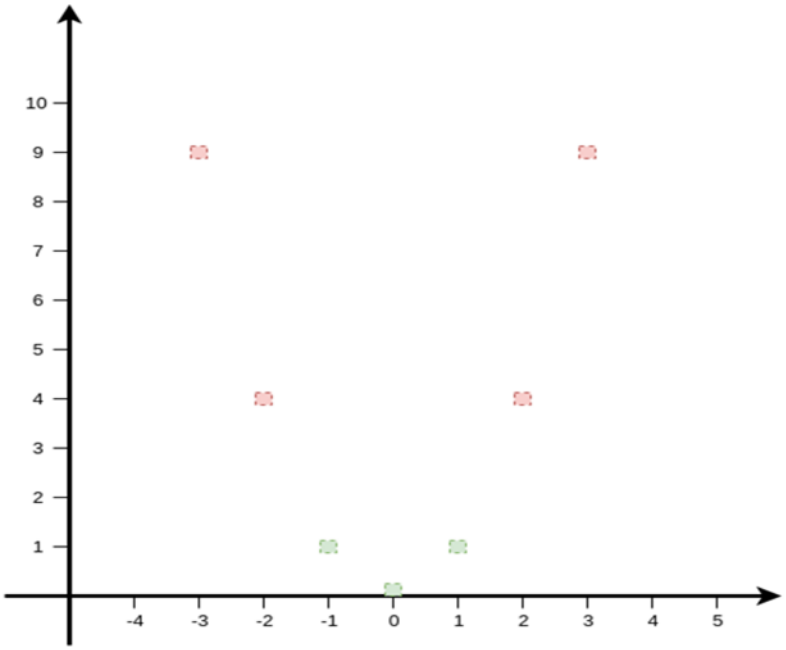


Figure II.6 Exemple d'une projection

Si malgré cette technique le jeu de données n'est toujours pas séparable, il existe d'autres formules d'optimisation SVM avec une variable $Ressort$ qui introduit le coût des erreurs des échantillons mal classifiés dans les formules d'optimisation de la marge.

II.3. Arbre de décision

II.3.a. Classification par arbre

Les arbres de décision ont l'avantage de pouvoir être modélisés afin de comprendre la prise de décision. Nous allons illustrer notre propos avec un arbre de classification sur le jeu de données IRIS.

Ce jeu de données est composé de mesures sur des pétales de fleurs. Il comprend trois classes de fleurs : **setosa**, **versicolor** et **virginica**. Chaque échantillon est composé de quatre features. Chaque catégorie est équitablement représentée avec 50 échantillons.

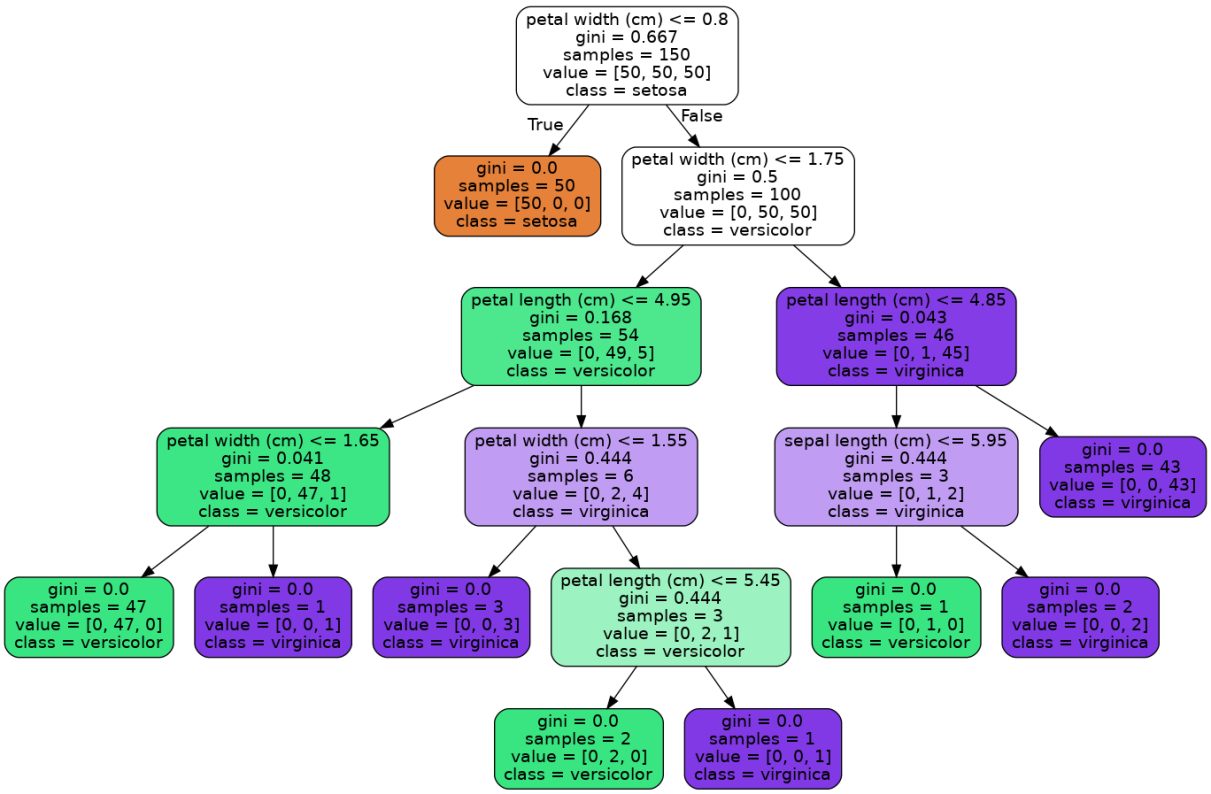


Figure II.7 Arbre de classification du jeu de données IRIS

La procédure de classification d'un échantillon démarre au sommet de l'arbre. Chaque nœud contient une comparaison sur une feature. Selon le résultat de cette comparaison avec la feature de l'échantillon à classifier, on descend dans le nœud gauche si elle est vraie ou à droite dans le cas contraire. Une fois arrivée dans une *feuille*, l'échantillon est classifié.

Un nœud contient aussi le nombre d'échantillons passé à travers lui pendant la phase d'entraînement et le découpage de ce nombre par classe.

L'attribut GINI, aussi appelé *impureté*, mesure la précision de la classification d'une feuille. Elle s'obtient par la formule suivante :

$Gini(i) = 1 - \sum(p_{i,k})^2$, avec i un nœud, p_i, le pourcentage d'échantillons par classe ayant transité par le nœud.

Dans le cas du deuxième nœud en partant de la gauche au niveau 3 de l'arbre, cela nous :

$$Gini(i) = 1 - ((\frac{2}{6})^2 + (\frac{4}{6})^2) = 0{,}444$$

La mesure GINI est préférée à celle de l'entropie qui mesure le même aspect d'un nœud mais qui est toutefois plus rapide à être calculée.

Il est aussi important, afin d'éviter que l'arbre de décision ne s'adapte trop au jeu d'échantillons d'entraînement, de fixer les hyperparamètres de celui-ci. Les différents hyperparamètres non dépréciés sont :

Nom	Valeurs possibles	Détails
criterion	{“gini”, “entropy”} default=”gini”	Mesure de la qualité de classification d’un nœud
splitter	{“best”, “random”} default=”best”	Choix de la séparation à chaque nœud
max_depth	Entier default=None	Nombre maximum de niveaux pour l’arbre. Si aucune valeur n’est donnée, l’arbre construit des niveaux jusqu’à ce que toutes les feuilles aient une valeur GINI nulle ou atteignent la valeur du critère min_samples_split
min_samples_split	Entier ou flottant default=2	Nombre d’échantillons devant transité par le nœud pour justifier la création de deux feuilles. Si la valeur est un flottant, il s’agit du pourcentage d’échantillons par rapport à l’ensemble du jeu de donnée
min_samples_leaf	Entier ou flottant default=1	Minimum d’échantillons nécessaire pour la création d’une feuille. Si la valeur est un flottant, il sera traité comme le pourcentage d’échantillons par rapport à l’ensemble du jeu de données
min_weight_fraction_leaf	Flottant default=0.0	Permet de préciser les fractions de chaque classe devant être présentes dans un nœud pour justifier la création d’une feuille. Si la valeur est égale à 0, ce paramètre n’entre pas en compte dans la construction de l’arbre
max_features	Entier ou flottant {“auto”, “sqrt”, “log2”} default=None	Nombre de features à étudier par l’arbre avant de choisir celle avec laquelle séparer les échantillons
max_leaf_nodes	Entier default=None	Nombre total de feuilles de l’arbre

min_impurity_decrease	Flottant default=0.0	Gain minimum pour justifier la création d'une feuille
class_weight	Dictionnaire, liste de dictionnaires ou "balanced" default=None	Poids à associer aux différentes catégories si l'une d'elle est sous-représentée
ccp_alpha	Flottant positif default=0.0	Paramètres pour réduire la complexité de l'arbre lors de sa construction

II.3.b. Algorithme CART

L'algorithme CART (*Classification And Regression Trees*) est celui utilisé par Scikit Learn pour entraîner les arbres de décision. Il existe d'autres algorithmes, notamment l'IDS (*Iterative Deepening Search*) qui créé des arbres qui peuvent avoir plus de deux feuilles.

Cet algorithme est récursif. Il s'applique à l'ensemble du jeu d'entraînement lors de la création du sommet et sur les sous-ensembles de chaque nœud. L'entraînement s'arrête lorsque que l'un des hyperparamètres est atteint ou qu'il ne peut plus partager les échantillons, ou bien s'il ne diminue plus l'*impureté* des feuilles.

A chaque itération, CART établi l'ensemble des paires (k, t_k), avec k, une feature du modèle, et t_k, la condition appliquée à cette feature. Il calcule ensuite le coût de chaque paire. Le coût est l'impureté générée par la paire (k, t_k). Plus l'impureté est élevée, plus le coût augmente.

Ce coût est estimé avec la fonction suivante $Coût(k, t_k) = (\frac{m_{GAUCHE}}{m}) * GINI(FEUILLE_{GAUCHE}) + (\frac{m_{DROITE}}{m}) * GINI(FEUILLE_{DROITE})$, avec m le nombre d'échantillons présents dans la feuille.

II.4. Apprentissage d’ensemble

II.4.a. Système de vote

Les arbres sont simples à comprendre mais il est difficile d’apprécier les hyperparamètres optimales, et sans eux l’arbre a de forte chance d’être surentraîné. C’est pourquoi on a conçu les forêts aléatoires, et autres modèles d’apprentissage d’ensemble.

Elles reposent sur le concept d’apprentissage d’ensemble. Ce système va nous permettre d’utiliser plusieurs modèles de natures différentes : arbre, régression logistique…, appelées estimateurs.

Il faut dans un premier temps disposer de plusieurs modèles déjà entraînés. Des erreurs interdépendantes pouvant fausser le système de votes, il est préférable d’avoir des modèles de natures différentes et/ou entraînés avec des jeux de données différents.

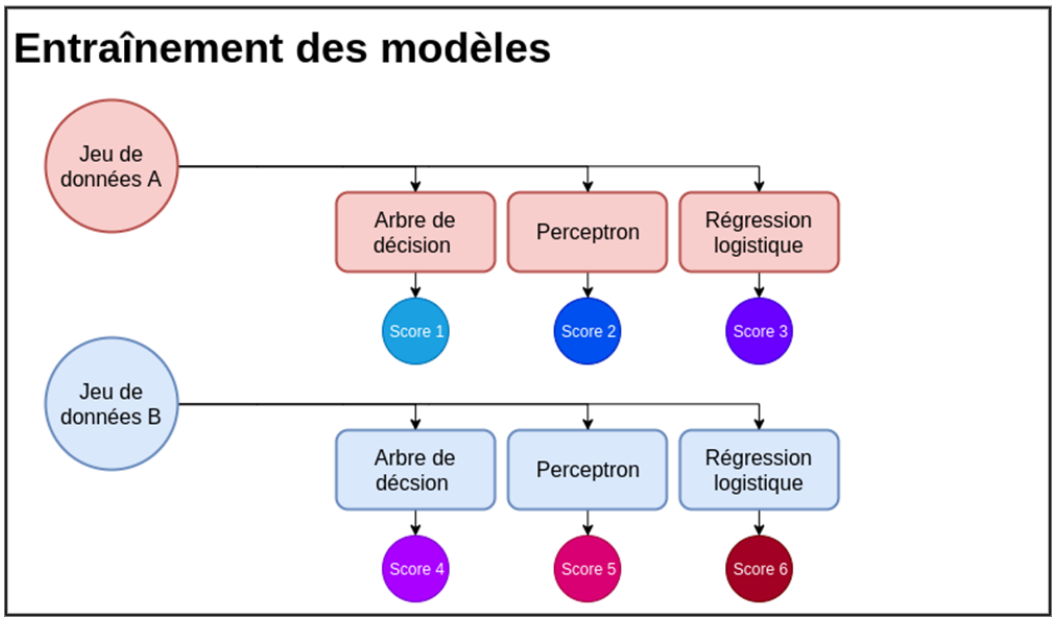


Figure II.8 Entraînement des modèles

Le score de chaque modèle peut servir dans le système de vote pour pondérer les prédictions des modèles les plus performants.

Dans un second temps, lors de la classification d'un échantillon, celui-ci est classifié par chaque modèle de l'ensemble. La catégorie la plus prédite sera retenue comme classification de l'ensemble. Si l'on attribue des coefficients de pondération aux prédictions des modèles, il s'agit d'un vote souple, sinon d'un vote dur.

Avec un vote souple, il est possible d'effectuer un apprentissage supervisé au modèle afin d'obtenir un vecteur de poids optimal. SI l'on ajoute une série de classificateurs après la première, cette méthode s'appelle le *stacking*.

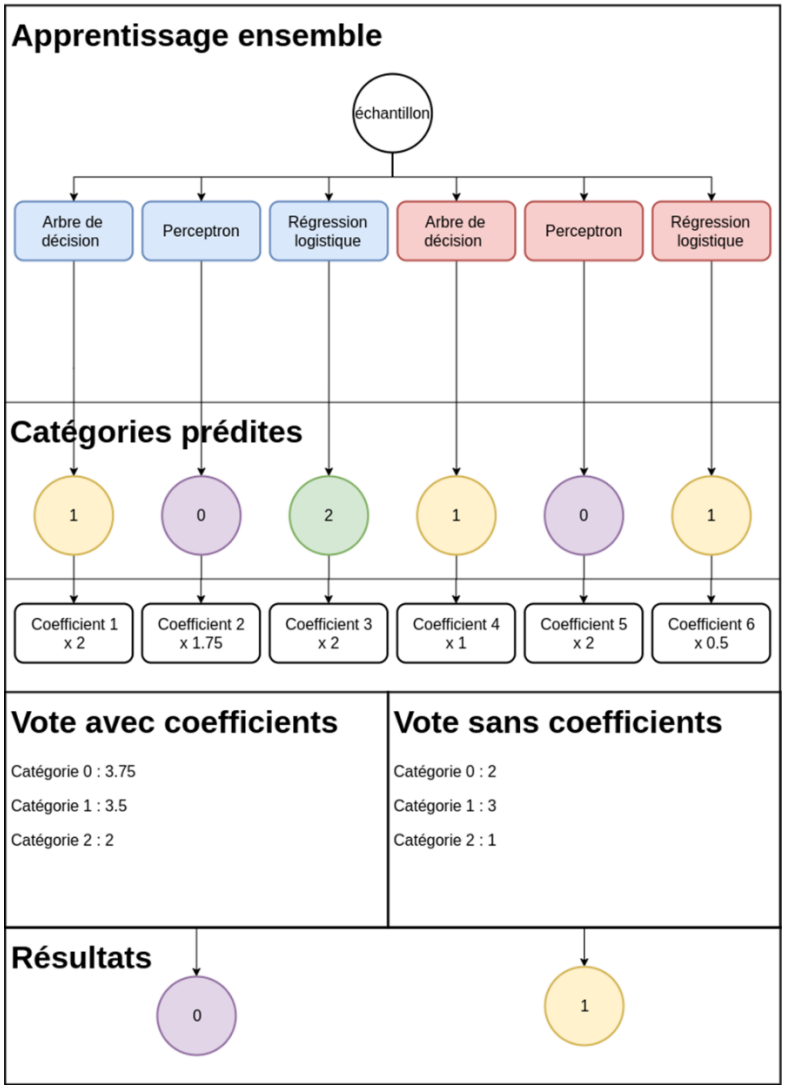


Figure II.9 Exemple d'un modèle d'apprentissage d'ensemble

II.4.b. Entraînement des classificateurs

Si l'on ne dispose pas de plusieurs jeux d'entraînement, il existe plusieurs façons de partager son jeu de données pour procéder à l'entraînement des classificateurs de l'ensemble :

- **Bagging** : les échantillons sont extraits aléatoirement, puis ils sont remis dans le jeu de données
- **Pasting** : les échantillons sont extraits aléatoirement, mais ils ne sont pas remis dans le jeu de données

Une forêt aléatoire est un ensemble composé uniquement d'arbres de décision entraîné à l'aide du *Bagging*.

II.4.c. Boosting

Une autre technique de construction d'un modèle d'apprentissage d'ensemble repose sur le *Boosting*. Il existe plusieurs types de *Boosting*, mais nous ne détaillerons que les deux plus connues à savoir l'*AdaBoost* et le *Gradient Boosting*. Le modèle entraîné dans le projet EMBER est un modèle LGBM (*Light Gradient Boosting Machine*). Le LGBM a été conçu pour diminuer le coût en ressources de l'entraînement du modèle.

Le principe du *Adaboost* se déroule lors de l'entraînement du classificateur. Une fois le premier classificateur entraîné, il effectuera des classifications sur le jeu de données. Les échantillons mal classés sont ensuite réutilisés dans l'entraînement du second classificateur mais avec un poids plus fort afin que ce modèle prenne mieux en compte leurs spécificités. Ce procédé est répété autant de fois qu'il y a de classificateurs.

LGBM se base sur le même procédé mais utilise un modèle de prédiction pour optimiser la descente de gradient.

CHAPITRE III : Comparaison des modèles

III.1. Indicateurs de performance

III.1.a. Matrice de confusion

L'évaluation de la performance d'un modèle dépend de l'objectif de son utilisation. Dans notre contexte de détection de programmes malveillants, le plus important est d'en détecter le plus possible afin d'empêcher leurs téléchargements ou leurs exécutions sur une machine.

Dans ce cas d'utilisation pratique où un modèle analyse les programmes afin de les valider, il est relativement aisé de « *whitelister* » après coup un programme légitime détecté à tort comme malicieux. A contrario si un programme malicieux est validé, c'est l'ensemble de l'usage du modèle qui sera remis en cause.

Il nous faut dans un premier temps obtenir un résumé des performances du modèle pour évaluer et comparer les différents modèles. La matrice de confusion résume les résultats des classifications faites par un modèle sur un jeu d'entraînement.

Dans notre cas, nous effectuerons de la classification binaire : le programme est-il malveillant ou non ? Ce qui nous donne la matrice suivante :

	Réel		
Prédis		Malveillant	Bénin
	Malveillant	Vrai positif [TP]	Faux positif [FP]
	Bénin	Faux négatif [FN]	Vrai négatif [TN]

III.1.b. Précision et rappel

La *précision* et le *rappel* sont deux indicateurs essentiels d'un modèle. La *précision* nous donne un indice de confiance a accordé à la classification d'un échantillon par le modèle. Ce taux nous est donné par la formule $Précision = \frac{TP}{(TP+FP)}$.

Prenons un jeu de données de 50 programmes malveillants et de 50 programmes légitimes. Après entraînement, le modèle nous donne la matrice suivante :

	Réel		
Prédis		Malveillant	Bénin
	Malveillant	45	5
	Bénin	5	45

Sa *décision* est donc de : $Précision = \frac{45}{(45+5)} = 90\%$.

S'il classe correctement moins de programmes malveillants :

	Réel		
Prédis		Malveillant	Bénin
	Malveillant	40	10
	Bénin	10	40

Sa *précision* sera alors de $\frac{40}{(40+5)} = 88\%$.

Le *rappel* représente le taux de programmes malveillants détectés et bloqués, et nous est donnés par la formule $Rappel = \frac{TP}{(TP+FN)}$.

Si on reprend nos deux exemples précédents, dans le premier cas, le *rappel* sera de $Rappel = \frac{45}{(45+5)} = 90\%$ et dans le deuxième cas $Rappel = \frac{40}{(40+10)} = 80\%$.

Ces deux taux permettent de calculer différents scores. Lors de l'entraînement d'un modèle, on affiche en général ces scores pour chaque itération afin d'évaluer le *gain marginal* d'une itération supplémentaire. Ces courbes sont généralement des asymptotes montrant que les itérations excédant un certain nombre ne sont plus efficaces dans l'entraînement d'un modèle.

III.1.c. Score F1

Le *score F1* est une moyenne harmonique du *taux de rappel* et du *taux de précision*. Elle permet d'évaluer ces deux valeurs au travers d'une seule. La formule du score est $F1 = \frac{2*(Précision*Rappel)}{(Précision+Rappel)}$.

La moyenne harmonique a aussi pour effet de pondérer les faibles taux dans le score. Par exemple avec un taux de précision de 100% et un de rappel de 50%, on a le score suivant $F1 = \frac{2*(1*0,5)}{(1+0,5)} = 66\%$ contre 75% dans le cadre d'une moyenne classique.

III.1.d. Courbe ROC

La courbe ROC (*Receiver Operating Characteristic*) se définit dans un plan où l'abscisse est le taux de faux positif $TFP = 1 - \frac{FP}{(FP+TN)}$ et l'ordonnée le taux de vrai positif $TTP = \frac{TP}{(TP+FN)}$.

Cette courbe ROC permet de se représenter l'AUC (*Area Under the Curve*). Si l'AUC est de 1, alors le modèle est parfait, il classifie correctement tous les échantillons. Ça peut aussi signifier une situation de surentraînement

Cette courbe permet aussi de comparer les résultats du modèle avec un modèle théorique faisant des prédictions aléatoires avec une chance sur deux de se tromper.

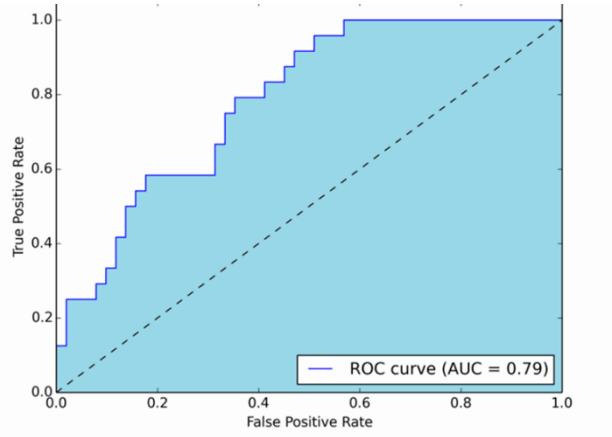


Figure III.1 Exemple d'une courbe ROC

III.2. Entraînement des modèles

III.2.a. Validation croisée

La validation croisée est une des techniques pour éviter les situations de surentraînement. L'idée est de sélectionner les sous-ensembles différents du jeu d'entraînement au cours des itérations.

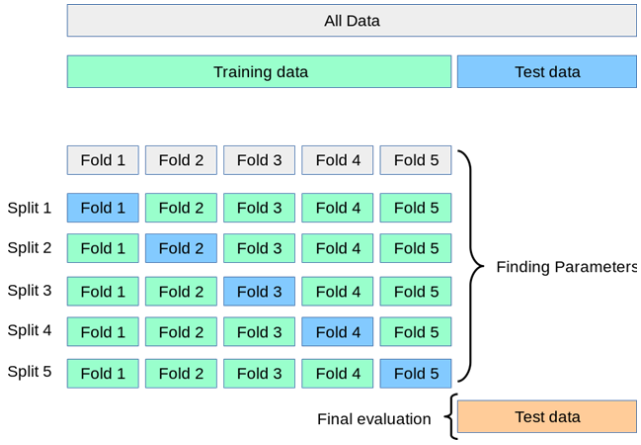


Figure III.2 Exemple d'une validation croisée

III.2.b. Échantillons croisés

Une deuxième mesure pour diminuer le biais de surentraînement dans la comparaison des modèles consistera à évaluer les classifications des modèles sur d'autres jeux de données qu'EMBER, qui pour rappel aura servi lors de l'entraînement.

Les auteurs du livre *Malware Data Science* ont mis à disposition les jeux de données ayant servis à la constitution de l'ouvrage. Ce jeu de données est accessible à l'adresse suivante : https://drive.google.com/open?id=11kvnSB9yQLaIdRI47z0PNjAyhXH6h0UK.

Deux jeux de données ont été extrait de cette archive. Après avoir vérifié si des échantillons étaient présents dans le jeu d'EMBER et s'il s'agissait de programmes au format PE, nous avons gardé deux groupes.

Résumé des jeux de données autres qu'EMBER :

Nom	Description
APT_1	Contient 217 samples attribués au groupe API 1
VT_2017	Contient 2 jeux de données : ▪ Sains : 991 échantillons ▪ Malicieux : 428 échantillons

III.2.c. Architecture

Après avoir sélectionné une série de modèles issues de la librairie scikit-learn, ils seront entraînés avec le jeu de données d'EMBER. Afin d'identifier les meilleurs hyperparamètres, une recherche par grille sera réalisée.

La recherche par grille se fait à l'aide de liste de paramètres. La fonction GridSearchCV testera ensuite les différentes combinaisons des paramètres entre eux en entraînant le modèle et en l'évaluant grâce à la fonction de score (F1, ROC ou algorithmes personnelles). La meilleure combinaison est retournée à l'algorithme.

Une fois les hyperparamètres d'un modèle sont identifiés, on réentraîne le modèle avec ces derniers en mesurant le temps nécessaire. Le modèle est ensuite sérialisé et sauvegardé.

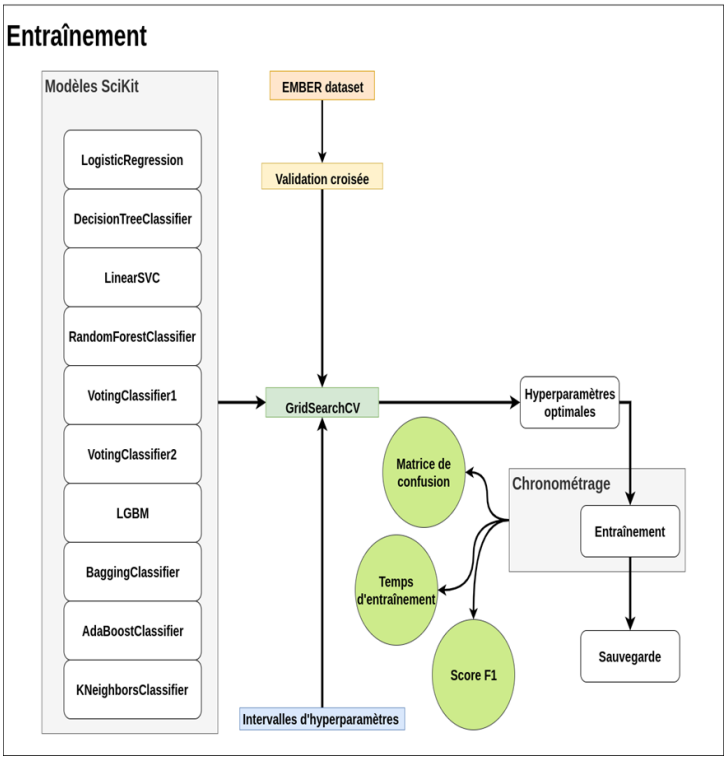


Figure III.3 Schéma de notre architecture d'entraînement

Une fois les modèles enregistrés, ils sont réutilisés pour faire de la classification sur les deux autres jeux de données. Cela permettra de comparer leurs scores avec ceux tirés du jeu d'entraînement afin de détecter un éventuel cas de surentraînement.

Le schéma de l'annexe I. résume l'évaluation d'un modèle.

III.3. Résultats

III.3.a. Hyperparamètres utilisés

A cause des limitations matérielles, nous n'avons pas pu faire de recherche par grille avec le jeu de données complet d'EMBER. Après plusieurs tests, nous avons choisi d'utiliser un jeu d'entraînement contenant 20'000 échantillons :

- 10'371 programmes malveillants
- 9'629 programmes bénins

Le jeu de données EMBER a aussi été au préalable filtré des échantillons non-catégorisés : ceux avec le label -1.

Le modèle est ensuite évalué avec un score F1 sur un jeu de test de 20'000 échantillons composés de :

- 10'035 programmes malveillants
- 9'965 programmes bénins

Nous comparerons ce score avec ceux tirés des deux jeux de données afin d'avoir une valeur plus proche de celle d'une mise en production du modèle.

Nom du modèle	Hyperparamètres	Temps d'entraînement	Précision	Rappel	Score F1
LogisticRegression	C: 1.0 class_weight: None max_iter: 200 penalty: l1 solver: liblinear	18m 08s	0.83	0.74	0.78
DecisionTreeClassifier	criterion: gini max_depth: None max_features: None	36s	0.80	0.72	0.76
LinearSVC	C: 0.5 loss: hinge,	39s	0.57	0.88	0.69

	penalty: l2				
RandomForestClassifier	criterion: gini max_depth: 2000, max_features: None n_estimators: 500	3h 15s	0.88	0.79	0.83
BaggingClassifier	base_estimator: DecisionTreeClassifier() n_estimators: 50	15m 48s	0.88	0.74	0.81
AdaBoostClassifier	algorithm: SAMME, 'base_estimator: DecisionTreeClassifier() n_estimators: 10	40s	0.80	0.69	0.74
KNeighborsClassifier	algorithm: auto n_neighbors: 5 p: 1 weights: distance	11s	0.49	0.42	0.45

III.3.b. Comparaison des performances

Lors du pré-traitement des jeux de données, deux problèmes ont été rencontrés :

- Les erreurs dans le parsing des DLL
- Le manque de parsing des fichiers MS-DOS

Nous avons donc trois jeux de données d'évaluation :

- APT_1 : 187 : programmes malveillants
- VT_2017_MLW : 389 programmes malveillants
- VT_2017_SAFE : 542 programmes légitimes

Les modèles *VotingSystem* utilisent les autres modèles entraînés dans un système de vote.

Nom du modèle	Vrai positif	Faux positif	Vrai négatif	Faux négatif	Précision	Rappel	Score F1 (Décalage)
LGBMEMber	493	14	528	83	0.97	0.85	0.91
LogisticRegression	390	177	365	186	0.68	0.67	0.68 (-0.10)
DecisionTreeClassifier	324	171	371	252	0.65	0.56	0.60 (-0.16)
LinearSVC	526	511	31	50	0.50	0.91	0.65 (-0.04)
RandomForestClassifier	312	147	395	264	0.67	0.54	0.60 (-0.23)

BaggingClassifier	306	68	474	270	0.81	0.53	0.64 (-0.17)
AdaBoostClassifier	364	149	393	212	0.70	0.63	0.66 (-0.08)
KNeighborsClassifier	188	440	102	388	0.29	0.32	0.31(-0.14)

Face aux faiblesses de résultats de nos modèles, en comparaison du LGBM d'EMBER, nous avons construit des classificateurs basés sur des systèmes de vote. Le faible nombre de votants les a conduits à avoir des résultats similaires. De plus, le modèle d'EMBER n'a pas été intégré comme estimateur afin de ne pas fausser la comparaison.

Nom du modèle	Vrai positif	Faux positif	Vrai négatif	Faux négatif	Précision	Rappel	Score F1
VotingSystem_NoCoeff	370	176	366	206	0.67	0.64	0.65
VotingSystem_Coeff_precision	370	176	366	206	0.67	0.64	0.65
VotingSystem_Coeff_scoref1	370	176	366	206	0.67	0.64	0.65

Conclusion générale

Ce mémoire nous aura permis de mieux appréhender certains mécanismes de l'apprentissage automatique, d'en mesurer les coûts d'entrée en termes de savoir. La mise en place de protocoles de test pour évaluer, même de manière partielle, la potentielle efficacité d'une technologie a été un challenge instructif.

Le point de départ du mémoire, à savoir comment l'apprentissage automatique pouvait aider à lutter contre les programmes malveillants, s'est orienté vers de la détection (de la classification binaire dans notre cas). Cependant, nous aurions aussi pu explorer d'autres voies comme la *clusterisation* ; ce qui nous aurait permis de regrouper les programmes malveillants par famille. C'est d'un article traitant de ce sujet que nous est venus l'idée de ce mémoire.

Avec l'apparition de modèles de plus en plus complexes, comme les modèles d'ensemble, rechercher le modèle avec des hyperparamètres optimales est devenu une tâche quasi sans fin dont les résultats doivent être arbitrés avec le temps disponible. Par exemple, on peut pour les modèles d'ensemble (Adaboost, Bagging) chercher les meilleurs hyperparamètres pour les estimateurs les composant.

Nous n'avons malheureusement pas pu obtenir un meilleur modèle que celui fourni par EMBER, sûrement à cause du fait que les modèles n'étaient entraînés qu'avec 4% de l'échantillon total d'EMBER.

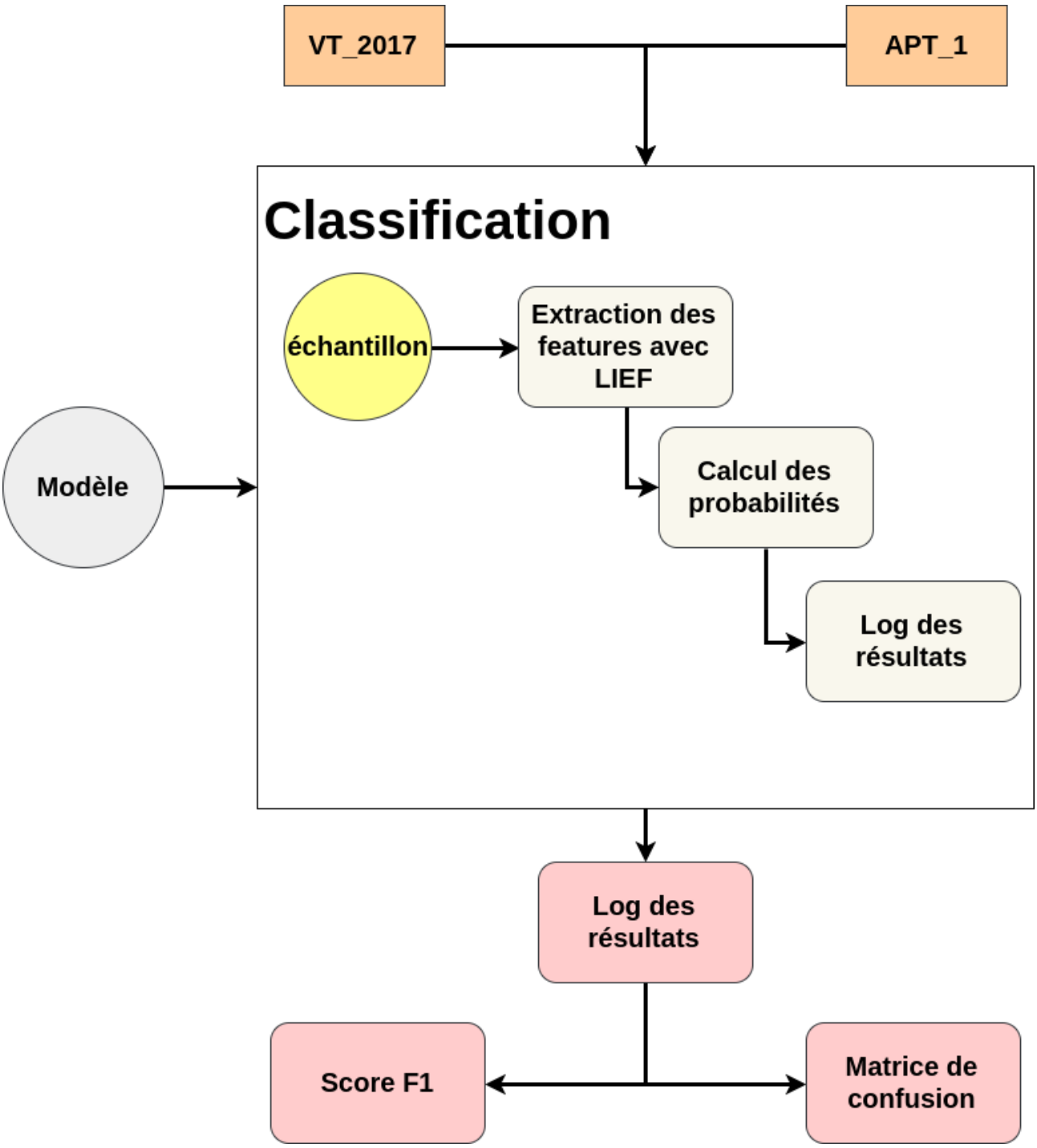
Notations

Acronymes

ANSSI	**A**gence **N**ationale de la **S**écurité des **S**ystèmes d'**I**nformation
AUC	Aire sous la courbe « ***A**rea **U**nder the **C**urve* »
CART	Arbre de classification et de régression « ***C**lassification **A**nd **R**egression **T**rees* »
DLL	Bibliothèque de chargement dynamique « ***D**ynamic **L**oad **L**ibrairies* »
DNS	Système de noms de domaines « ***D**omain **N**ame **S**ystem* »
ELF	Format exécutable et liable « ***E**xecutable and **L**inkable **F**ormat* »
EMBER	« ***E**ndgame **M**alware **BE**nchmark for **R**esearch* »
IAT	Import de la table d'adresses « ***I**mport **A**ddress **T**able* »
IDS	Recherche d'approfondissement itératif « ***I**terative **D**eepening **S**earch* »
LGBM	Machine d'amplification de gradient de lumière « ***L**ight **G**radient **B**oosting **M**achine* »
LPM	**L**oi de **P**rogrammation **M**ilitaire
MISP	Plateforme de sécurité des informations sur les logiciels malveillants « ***M**alware **I**nformation **S**ecurity **P**lateform* »
ROC	Caractéristiques de fonctionnement du récepteur « ***R**eceiver **O**perating **C**haracteristic* »
PE	Exécutable portable « ***P**ortable **E**xecutable* »
SVM	Machine à vecteur de support ou séparateur à vaste marge « ***S**upport **V**ector **M**achine* »
URL	Localisateur de ressources uniformes « ***U**niform **R**esource **L**ocator* »

Annexes

I. Évaluation d'un modèle



Bibliographie

Sikorski, M., Honig, A., 2012. Pratical Malware Analysis.

Saxe, J., Sanders, H., 2018. Malware Data Science.

Géron, A., 2017. Machine Learning avec Scikit-Learn.

Un peu de Machine Learning avec les SVM, URL : https://zestedesavoir.com/tutoriels/1760/un-peu-de-machine-learning-avec-les-svm/

Andeson, H. S., Roth, P. An Open Dataset for Training Static PE Malware Machine Learning Models, URL : https://arxiv.org/pdf/1804.04637.pdf

Decisions Trees, URL : https://scikit-learn.org/stable/modules/tree.html

sklearn.tree.DecisionTreeClassifier, URL : https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier

Cross-validation : evaluating estimator performance, URL : https://scikit-learn.org/stable/modules/cross_validation.html

Portable Executable, URL : https://fr.wikipedia.org/wiki/Portable_Executable

PE Format, URL : https://docs.microsoft.com/en-us/windows/win32/debug/pe-format

Résumé

L’identification de programmes malveillants par modèle d’apprentissage automatique : comment les modèles d’apprentissage automatique identifient un programme malveillant d’un programme légitime ?

Après présentation du projet EMBER, un jeu de données composé de plus d’un million d’analyses LIEF de PE, ce mémoire abordera les modèles de base comme la classification binaire et d’autres plus avancés avec les forêts aléatoires et les systèmes de vote. Enfin, nous présenterons un procédé d’évaluation des performances d’un modèle, suivi par l’entraînement et l’évaluation de quelques modèles sélectionnés afin d’en tirer une conclusion.

Mots-clés : EMBER, LIEF, PE, Classification, Programme malveillant, Apprentissage automatique

Abstract

Identifying malware by machine learning model : how do machine learning models identify malware from legitime software ?

After presenting the EMBER project, a dataset with more than one million of Portable Executable LIEF analysis, this papier will present basic models like binary classification and some more advanced with random forest and voting system. Finally, we will present a process for evaluating the performance of a model, followed by training and evaluating a few selected models in order to draw a conclusion.

Keywords : EMBER, LIEF, PE, Classification, Malware, Machine Learning