Télécom ParisTech

Projet de filière SR2I

Détection d'anomalies de classification dans l'IoT via Machine Learning

Antoine Urban, Yohan Chalier

encadrés par
Jean-Philippe Monteuuis
Houda Labiod

22 juin 2018

Résumé

Dans le contexte de l'internet des objets, les messages peuvent contenir de fausses informations générées par un utilisateur authentifié. Par conséquence, les mécanismes de sécurité reposant sur le chiffrement et la signature sont inutiles face à ces attaques. Les algorithmes reposant sur l'apprentissage machine (Machine Learning) permet de classer si un objet émettant des données est malicieux ou non.

Chapitre 1

Introduction

L'exercice de la conduite est synonyme de changements constants. Ainsi, lorsque nous conduisons, notre attention est entièrement consacrée à notre environnement, car notre sécurité et celle des personnes qui nous entourent sont en jeu. Nous prêtons une attention particulière aux obstacles qui peuvent surgir, qu'il s'agisse tout simplement d'autres voitures partageant la route, des piétons ou encore des motocyclistes.

Longtemps fantasmés, les véhicules autonomes sont aujourd'hui réalité grâce aux incroyables progrès réalisés dans les domaines de l'intelligence artificielle et du développement de capteurs. Si ces processus visent à remplacer le conducteur humain, il semble naturel qu'ils prêtent la même attention aux obstacles. La Figure 1.1 présente la complexité introduite dans ce nouvel environnement. Un nombre important de capteurs est en effet nécessaire pour avoir une vue complète du contexte extérieur. Cela comprend à la fois des communications Véhicule-à-Véhicule et l'analyse des données des capteurs.

Dans ce travail, nous nous concentrerons sur les trois types d'obstacles les plus communs sur une route :

— une voiture,
— une moto et
— un piéton.

Tout travail sur l'analyse d'obstacles nécessite la détection préalable de régions d'intérêt en utilisant par exemple la méthode du "point-in-polygon" [1], ou encore à l'aide de capteurs à ultrasons et d'analyse de signal [2]. Ce travail n'a pas comme objectif de revenir sur l'utilisation de ces techniques, mais se concentrera sur l'analyse à posteriori des dimensions ainsi collectées avec pour but de proposer une classification efficace. La Figure 1.2 récapitule le flot de données dans un véhicule autonome. Comme nous pouvons le voir, en collectant les données des capteurs et provenant des véhicules aux alentours, le travail de classification (ou "perception") permet la modélisation de l'environnement et de déterminer le comportement du véhicule.

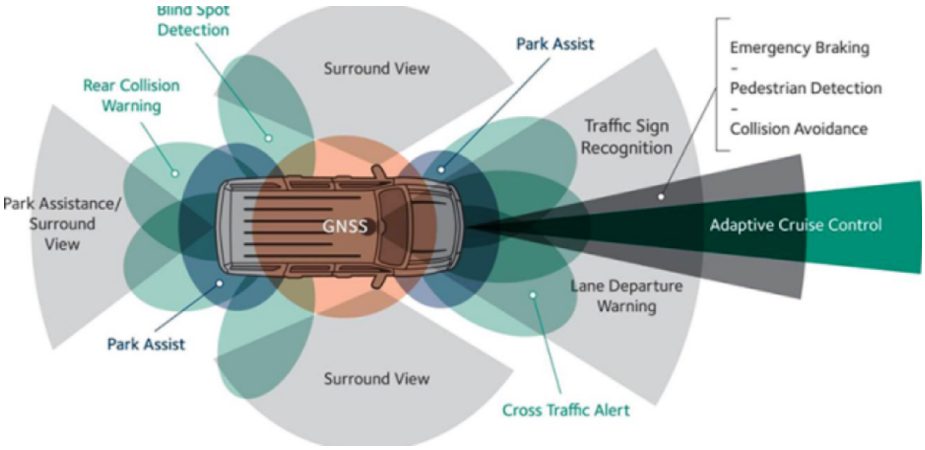

FIGURE 1.1 – Ensemble des capteurs présents dans le véhicule

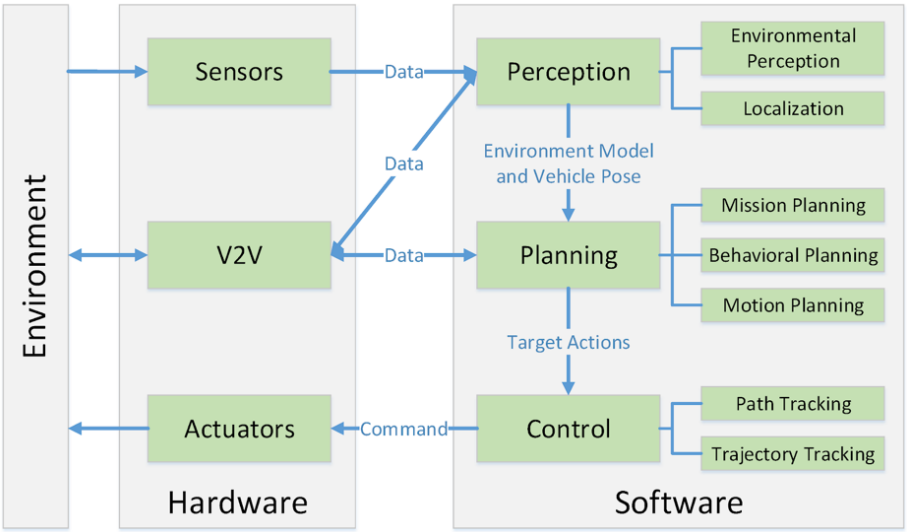

FIGURE 1.2 – Flux de données

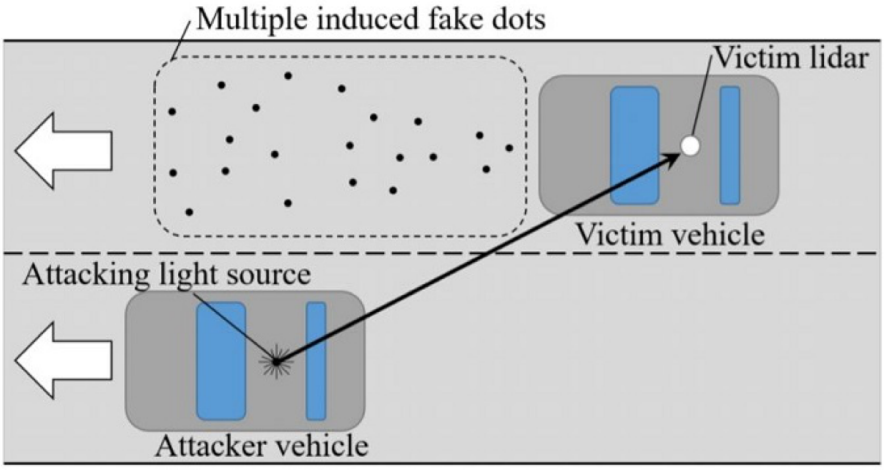

FIGURE 1.3 – Attaque par aveuglement

1.1 Attaques

Comme nous l'avons vu grâce aux figures 1.1 et 1.2, le nombre important de capteurs et la collecte de messages provenant de véhicules rendent la surface d'attaque importante. Cette section a pour objectif de décrire les plus importantes.

Attaque par aveuglement des capteurs La présence de capteurs optiques rend possible une attaque par aveuglement. Dans ce schéma [3], un attaquant utilise un laser pour aveugler ces capteurs ce qui a pour conséquence une mauvaise perception de l'environnement. De ce fait, la victime pourra freiner ou dévier de sa trajectoire et ainsi provoquer un accident.

Attaque par modification Un attaquant peut envoyer des faux messages à la victime. Cela aura pour conséquence de causer une fausse perception de l'environnement et incidemment créer un accident.

1.2 Travaux connexes

Le sujet de la détection et de la classification d'obstacle fait l'objet de nombreux travaux de recherche, tant leurs résultats sont nécessaires pour le bon fonctionnement des objets autonomes. Ainsi, plusieurs méthodes utilisant diverses techniques ont été développées. Si certains [4] utilisent la vitesse relative des objets pour détecter, suivre et reconnaître des séquences d'images, d'autres [5] ont basé leur méthode d'extraction sur la reconnaissance de contours utilisant les techniques HOG (Histogram of Oriented Gradients) et SURF (Speeded-Up Robust Features). Ces techniques ont en commun l'utilisation d'algorithmes d'apprentissage supervisé (à savoir SVM et des réseaux de neurones) pour leur classification.

Des chercheurs ont également travaillé à s'adapter au maximum à l'environnement urbain, où le trafic peut provenir de directions arbitraires. En d'autres

termes, la majorité des modèle sont performants sur du trafic autoroutiers mais se dégradent dans des situations urbaines. Ainsi, Darms, Rybski et Baker proposent [6] une méthode de suivi qui vise à prendre en compte la dimension dynamique de l'environnement extérieur.

Enfin, contrairement aux techniques citées jusqu'alors qui utilisent des techniques de classification, des chercheurs [7] ont également proposé un modèle de régression en cascade pour l'analyse de trafic issue de vidéos de surveillance.

1.3 Objectifs

A travers un cas simple, nous cherchons à vérifier l'appartenance d'un objet a‘ une classe (e.g. un véhicule) en utilisant un algorithme de Machine Learning et les dimensions de l'objet. Il s'agit donc de proposer un modèle de classification multi-classes en réalisant un classeur à partir d'un algorithme d'apprentissage supervisé.

Au préalable un travail de préparation est nécessaire. Ainsi, nous nous assurons de trouver des bases de données (cf. Figure 1.4 module A) avec les dimensions pour chacune des classes pour entraîner et valider notre modèle.

Ensuite, nous proposerons une méthodologie de sélection d'un algorithme d'apprentissage supervisé en évaluant leur précision et leur efficacité respectives sur des critères détaillés dans la partie 2.4 (cf. Figure 1.4 module D). Dans ce travail, nous avons considéré les algorithmes suivant :

— Réseau de Neurones ;
— Adaboost ;
— SVM ;
— Random Forest.

L'objectif final de ce projet est la mise à disposition d'une fonction de prédiction en-ligne (cf. Figure 1.4 module B) permettant de classer des données relevées par un ensemble de capteurs en malicieux / non-malicieux. Le pseudo code 1 permet de résumer le déroulé du processus de prédiction.

Algorithme 1 : Fonction de prédiction

Données : Un message i envoyé par un objet communicant contenant la classe de l'objet (classe$_i \in$ {voiture, moto, piéton}) et les dimensions de l'objet (longueur$_i$; largeur$_i$)

Résultat : Détection des objets malicieux

si *fonction(classe$_i$, longueur$_i$, largeur$_i$)* **alors**
  retourner *"malicieux"*
sinon
  retourner *"non-malicieux"*
fin

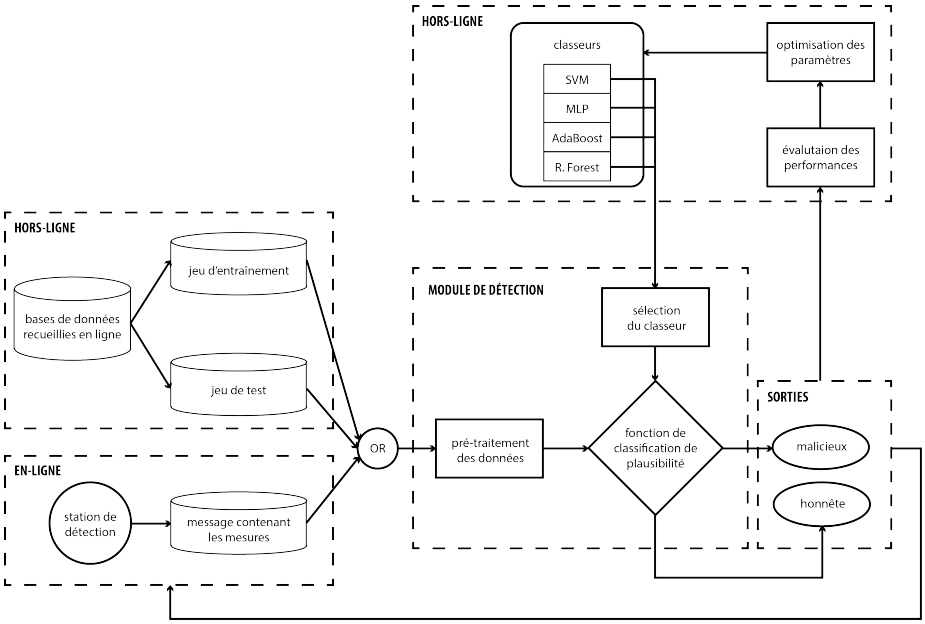

FIGURE 1.4 – Schématisation du fonctionnement global du détecteur

Chapitre 2

Démarche et stratégie

2.1 Première implémentation

2.1.1 Environnement Python

Afin d'éviter d'éventuels problème de version, nous avons opté pour l'utilisation d'environnements virtuels à l'aide du module Virtualenv [8]. Nous avons choisi le noyau Python 3.6. Les modules utilisés sont regroupés dans le fichier requirements.txt présent dans le dépôt git. Les principaux sont :

— Pandas (analyse de données) [9]
— scikit-learn (Machine Learning) [10]
— Numpy (calcul scientifique) [11]
— Matplotlib (tracé et graphe) [12]
— Jupyter (interface interactive de programmation) [13]

2.1.2 Objectif

En premier lieu, nous souhaitions commencer par une vision globale des données et du travail à effectuer. Dans cette partie, nous allons nous efforcer d'obtenir une première fonction de classification se basant sur des critères très simples (les extremums de largeur et de longueur) : des régions de décision rectangulaires et arbitraires.

2.1.3 Mise en œuvre

L'objectif de cette étude est la détection d'anomalies dans la mesure de longueur et de largeur. Nous disposions d'une base de données contenant des mesures de voiture, provenant de CarQuery [14]. Nous avons extrait les deux colonnes de largeur et longueur dans une DataFrame du module d'analyse de données, Pandas, en Python. Cette classe permet entre autres de représenter efficacement des tableaux de taille importante et de nommer les colonnes.

Après un premier affichage des données (Figure 2.1), il est apparu que beaucoup de points apparaissaient en plusieurs fois, aussi la séparation de la base de données en points uniques et points non-uniques se révéla pertinente. Cela permit de réduire le nombre de lignes à de 54808 à 5026, soit un facteur dix (Figure 2.2).

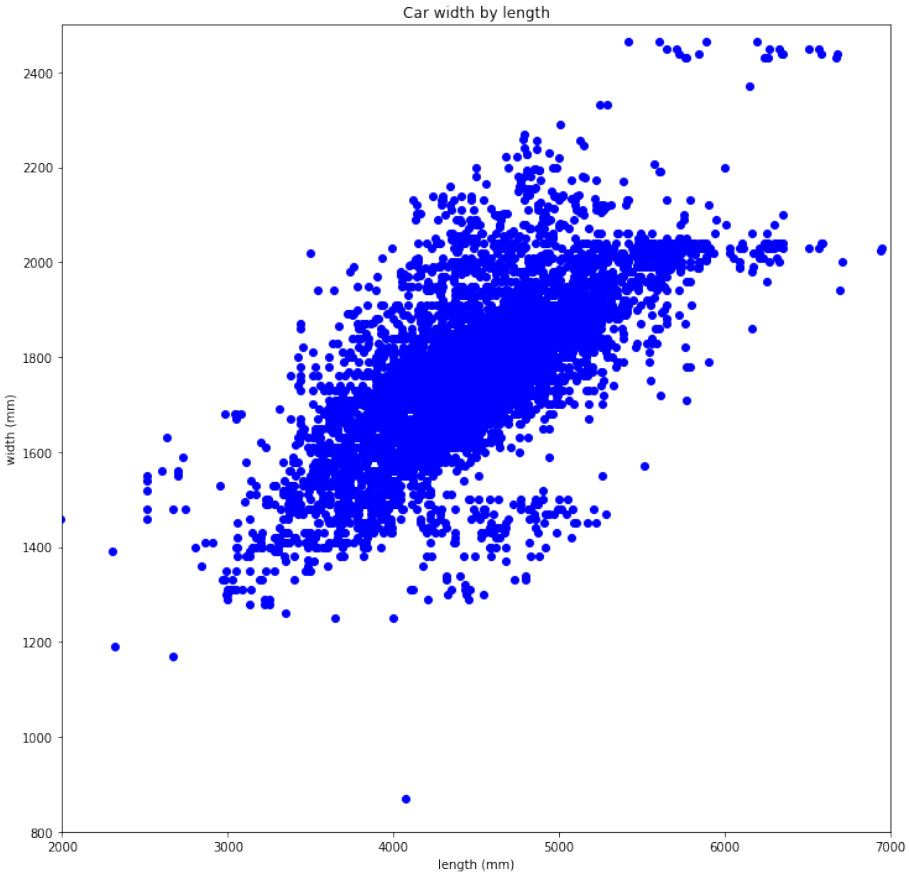

FIGURE 2.1 – Premier affichage des données

```
25 print("Original shape:", sizes.shape[0])
26 print("Shape after uniqueness pruning:", len(unique) + len(non_unique))
```

```
Original shape: 54808
Shape after uniqueness pruning: 5026
```

FIGURE 2.2 – Capture d'écran du nombre de lignes avant et après élagage

Manuellement, nous avons alors défini des zones simples (rectangulaires) en tant que régions de décision de la plausibilité des dimensions des véhicules (Table 2.1). Ces zones ont été définies au jugé, afin d'encadrer le plus de points valides sans toutefois englober une zone de l'espace trop large. Une approche pour déterminer ces seuils peut être l'utilisation des records de taille enregistrés au cours de l'histoire.

cadre	validité	intervalle de longueur	intervalle de largeur
vert	non-malicieux	3 à 6,5 mètres	1,4 à 2,4 mètres
gris	malicieux	3 à 4,1 mètres	2,05 à 2,4 mètres
gris	malicieux	5,25 à 6,5 mètres	1,4 à 1,65 mètres

TABLE 2.1 – Dimensions des régions de décision arbitraires

Hors de la zone verte, et dans les deux cadres gris, nous avons alors généré aléatoirement 700 points définis comme malicieux. La Figure 2.3 représente l'affichage de tous les points décrits plus tôt ainsi que des régions de décision. Ainsi faite, notre classification possède, sur le jeu d'entraînement, une précision de 97,83%, et une précision de 97,31%. Ces scores peuvent sembler suffisants, mais les régions de décisions montrées dans la Figure 2.3 montrent qu'il y a des zones mortes (par exemple, pour une largeur supérieure à 2,1 mètres et une longueur supérieure à 5 mètres, dans lesquelles un attaquant pourrait positionner ses points malicieux. Ce modèle présente donc de l'*underfitting*.

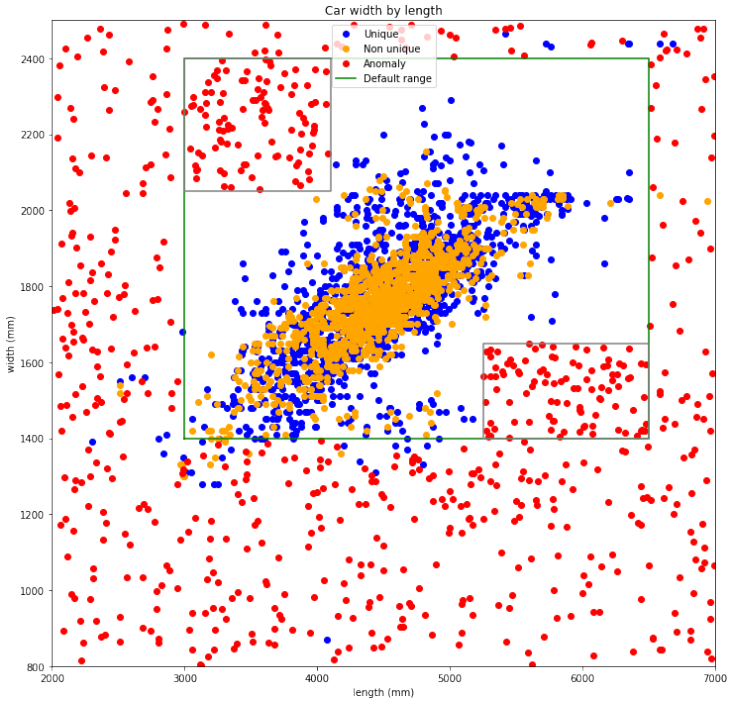

FIGURE 2.3 – Régions de décision manuelles pour des dimensions de voitures

2.2 Recherche des bases de données

Dans cette partie nous décrivons les bases de données que nous avons utilisé pour construire notre modèle. Pour ce faire, nous avons exploité quatre jeux de données pour les différentes classes, à savoir :

- Pour les voiture, nous avons utilisé la base de données CarQuery [14], qui présente 70847 références de véhicules, comprenant notamment l'année, le modèle et des spécifications diverses comme la longueur ou la largeur.
- Pour les motos, nous avons utilisé deux bases de données [15] [16] que nous avons concaténées, qui présentent 202 références de motos ainsi que leurs dimensions.
- Pour les piétons, nous avons utilisé les résultats d'un travail de recherche [17], qui présente les dimensions de 507 humains, comprenant notamment leurs tailles, et leurs largeur d'épaule.

Nous disposons donc des données suivantes :

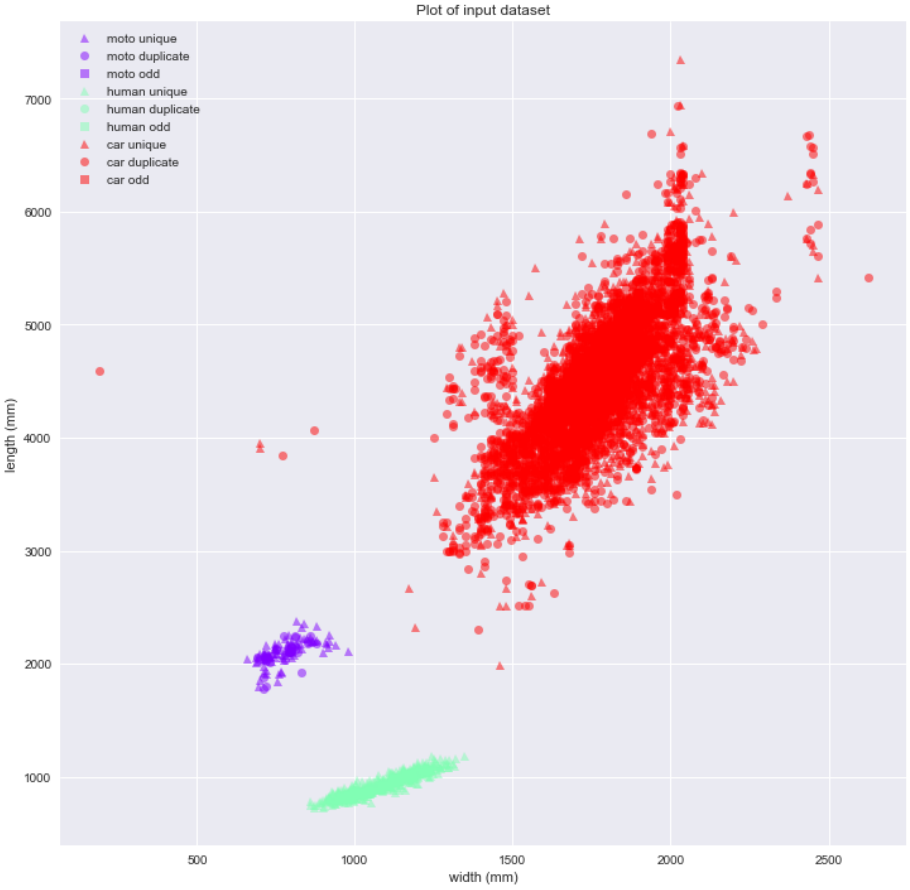

FIGURE 2.4 – Affichage des données brutes

2.3 Environnement de travail

Dans cette partie, nous décrivons les outils utilisés et développés pour poursuivre notre étude. Ces éléments se retrouvent sur le dépôt git que nous avons utilisé pour sauvegarder notre code [18].

2.3.1 Chargement des bases de données

Afin de centraliser le chargement des bases de données explicitées plus tôt entre tous les scripts en ayant besoin, nous avons implémenté une fonction de chargement nommée `load_detector` dans loader.py. Cette fonction instancie un object de la classe Detector, que nous décrirons dans la partie suivante. Elle procède de la façon suivante :

1. Pour chaque jeu de données au format CSV
   1.1. Lire les colonnes contenant la longueur et la largeur
   1.2. Renommer ces colonnes en "`length`" et "`width`"
   1.3. Supprimer les lignes incomplètes
   1.4. Si nécessaire, convertir les données en flottant et en millimètres
   1.5. Ajouter une colonne contenant la classe correspondant au jeu de données considéré
   1.6. Appliquer un premier filtre sur la longueur ou la largeur pour supprimer les points extrêmes isolés
2. Fusionner toutes les matrices précédentes en une seule
3. Créer un nouvel objet `Detector` avec cette matrice en attribut
4. Supprimer les éventuels redondances
5. Ajouter une colonne "`odd`" à la matrice, initialisée à `False`
6. **Générer les données malicieuses**
7. Ajouter les données malicieuses à la base de données, en rajoutant la colonne "`odd`" initialisée à `True`
8. Remplacer les valeurs des classes (originnellement des chaînes de caractères comme `"car"` ou `"human"`) par des entiers
9. **Séparer la matrice en un jeu d'entraînement et un jeu de test**
10. Renvoyer l'objet 'Detector' ainsi initialisé

Dans le cas des bases de données décrites au paragraphe précédent, l'étape 1.6. permet de supprimer quelques points, par exemple une moto de longueur supérieure à 20 mètres, ou une voiture de 6 mètres de large. Bien que réelles, ces données sont trop isolées pour être considérés dans le reste de notre travail.

La DataFrame finale possède 4 colonnes, plus une pour l'index. Ces colonnes sont la classe du véhicule (entier), la longueur (flottant), la largeur (flottant), et le caractère malicieux (booléen).

```
In [8]: 1 detector.xtrain
```

Out[8]:

	class	length	width
0	2	4920.0	1880.0
1	2	4010.0	1690.0
2	2	4290.0	1690.0
3	2	4410.0	1770.0
4	0	776.0	977.0
5	2	4854.0	1933.0
6	2	4040.0	1510.0
7	0	958.0	1092.0
8	2	4020.0	1500.0
9	2	3910.0	1480.0
10	0	3142.0	660.0

FIGURE 2.5 – Premières lignes de la matrice de *features* du jeu d'entraînement

Génération des données malicieuses *Toutes les données que nous avons recueillies proviennent de base de données légitimes. Afin d'obtenir des données malicieuses afin d'entraîner nos classeurs, nous avons procédé à une génération manuelle de ces dernières. Nous nous sommes concentré sur une génération uniforme, ce qui peut correspondre à une attaque d'aveuglement des capteurs.*

Pour un nombre de points à générer donné, le programme génère des points uniformément dans la zone rectangulaire définie par les minimums et maximums de longueur et de largeur de la base de données initiales. À chacun de ces points est associé, uniformément, une classe aléatoire parmi les classes présentes dans la base de données. La génération utilise un *seed* entier entre 0 et $2^{32} - 1$, ré-utilisable ultérieurement pour générer le même jeu de données.

Séparation de la matrice La matrice est pour l'instant stockée en tant qu'attribut de l'objet `Detector`. Avec le *seed* généré précédemment, elle est tout d'abord mélangée pour éviter d'avoir toutes les données triées. Puis, elle est coupée en deux moitiés :

— le jeu d'entraînement
— le jeu de test

La première moitié servira à entraîner nos classeurs, tandis que la seconde nous permettra de confronter ces classeurs à des données réelles, sélectionner les paramètres et définir le meilleur classeur.

Enfin, on procède à la division de chacune de ces matrices en deux matrices, une pour les *features* et une pour le label de sortie (le caractère malicieux). Au final, chacune de ces DataFrames (`x_train`, `y_train`, `x_test` et `y_test`) est stockée dans l'objet `Detector`. La figure 2.5 montre les 10 premières lignes de `x_train`.

2.3.2 Classe `Detector`

Comme expliqué précédemment, cette classe stocke les jeux de données utilisés pour l'entraînement et la prédiction. Elle va aussi permettre de centraliser les tests de classeurs, et l'affichage des données. Ses méthodes (Table 2.2) sont donc une sorte d'API pour la réalisation de la fonction de prédiction finale, objectif du projet.

Pre-processing	`clean`	Étapes 4 et 5 du chargement des bases de données
	`append_odd_points`	Étape 7 du chargement des bases de données
	`format`	Étapes 8 et 9 du chargement des des bases de données
Interface `sklearn`	`classify`	Entraîne un classeur et renvoie la matrice de confusion sur le jeu de test
	`tune_parameters`	Trouve le meilleur jeu de paramètres pour un classeur
	`predict`	Fonction finale de prédiction online
Affichage	`plot`	Affiche la matrice de données complètes
	`plot_decision_boudaries`	Affiche les régions de décisions d'un classeur

TABLE 2.2 – Méthodes de la classe `Detector`

2.4 Méthode d'évaluation

2.4.1 Matrice de confusion

Une technique pour évaluer la performance d'un algorithme de classification est d'utiliser une matrice de confusion. Il s'agit d'un résumé des résultats de prédiction sur un problème de classification, qui donne un aperçu du degré de confusion de notre modèle.

Terminologie

Pour définir de manière formelle une matrice de confusion, il convient d'introduire les termes suivants :

— TP (Vrai Positif) : item correctement détecté positif
— FP (Faux Positif) : item déclaré positif, là où il est en réalité négatif
— FN (Faux négatif) : item déclaré négatif alors qu'il était en réalité positif
— TN (Vrai négatif) : item correctement détecté comme négatif

Ces derniers permettent de définir la classification, correcte ou non, des objets données en entrée de notre fonction. Nous pouvons dès lors construire la matrice suivante :

		Classe réelle : Positif	Classe réelle : Négatif		
Classe prédite	Positif	TP	FP	PPV	FDR
	Négatif	FN	TN	FOR	NPV
		TPR	FPR		
		FNR	TNR		

Les colonnes représentent le nombre d'occurrences d'une classe de référence (ie des observations issues de notre base de données). Les lignes quant à elles le nombre d'occurrences d'une classe estimés par notre modèle de classification.

Pour illustrer cette terminologie, nous introduisons la matrice de confusion suivante, obtenues en appliquant l'algorithme RandomForest sur notre base de données :

$$\begin{bmatrix} 2801 & 19 \\ 36 & 467 \end{bmatrix}$$

Ainsi, parmi les 2837 observations positives, 2801 seront estimés comme tels et 36 comme étant négatives. De même, sur les 2820 observations que notre modèle a estimé comme positives, 19 sont en réalité négatives. Le même raisonnement s'applique pour les observations négatives.

Pour la suite, et pour comprendre cette matrice, il convient d'introduire les résultats suivants :

— ratio de vrai positif : $TPR = \dfrac{TP}{TP + FN} = 0.987$ (true positive rate)

Ce ratio montre la capacité de notre modèle à détecter tous les valeurs positives. Ici, cette détection est très efficace. Néanmoins, il est important de noter que nous pouvons facilement avoir un très bon TPR en prédisant systématiquement "positif". Cette valeur doit donc s'interpréter avec celle du TNR.

— ratio de vrai négatif : $TNR = \dfrac{TN}{TN + FP} = 0.959$ (True negative rate)

Aussi appelé sensibilité. Mesure la capacité de notre modèle à détecter les valeurs négatives. De la même manière que pour le TPR, on peut facilement avoir un très bon TNR en prédisant systématiquement négatif. Ici, la valeur importante combinée à celle du TPR renforce la performance du modèle.

— ratio de faux positif : $FPR = \dfrac{FP}{TP + TN} = 0.04$ (False positive rate)

Ce ratio décrit en quelque sorte la probabilité de fausse alarme.

— ratio de faux négatif : $FNR = \dfrac{FN}{TP + FN} = 0.01$ (False negative rate)

Décrit en quelque sorte le ratio de mauvaise classification des items positifs. Un taux égal à 1 signifierait que 100% des items positifs ont été

déclaré négatifs. On remarque que notre modèle classifie avec efficacité les items positifs.

— valeur prédictive positive : $PPV = \frac{TP}{TP+FP} = 0.993$ (positive predicted value)

Une PPV importante signifie qu'un item marqué positif est réellement positif (il y a un faible nombre de faux positifs).

— valeur prédictive négative : $NPV = \frac{TN}{TN+FN} = 0.928$ (negative predicted value)

A l'inverse du PPV, un NPV important signifie qu'un item marqué négatif est réellement négatif (il y a un faible nombre de faux négatif).

— taux de fausses découvertes : $FDR = \frac{FP}{TP+FP} = 0.001$ (False discovery rate)

Il s'agit du complémentaire de la précision. Un FDR important signifie l'incapacité de notre modèle à détecter les items positifs.

— taux de fausses omissions : $FOR = \frac{FN}{TN+FN} = 0.072$ (False omission rate)

Il s'agit du complémentaire du NPV. Un FOR important indique l'incapacité de notre modèle à détecter les items négatifs.

Ce mode de représentation permet non seulement de mettre en évidence les erreurs qui sont faites par notre classeur, mais surtout des types d'erreurs (mauvaise détection d'une classe ou surestimation du des items positifs par exemple) qui sont commises. En effet, l'utilisation de la seule "accuracy" peut être trompeuse dans la mesure où le nombre d'observations dans chaque classe est inégale.
Pour illustrer ce phénomène, prenons un exemple précis. Considérons l'algorithme SVM (support vector Machine) avec les paramètres suivants :

C=0.1, cache–size=200, class–weight=None, coef0=0.0, decision–function–shape='ovr', degree=3, gamma='auto', kernel='linear', max–iter=-1, probability=False, randomState=None, shrinking=True, tol=1e-05, verbose=False

Cet algorithme renvoie avec notre base de donnée, la matrice de confusion suivante :

$$\begin{bmatrix} 2771 & 74 \\ 239 & 239 \end{bmatrix}$$

L'accuracy est égale à $\frac{TP+TN}{Total\_items} = 0.91$

Nous remarquons que 313 items ont mal été classifiés. Dans le cadre d'une classification d'obstacles, cela est loin d'être satisfaisante alors que le score d'accuracy est, lui, relativement important. Il y a en effet dans ce dernier aucune pondération. Si une classe contenant un nombre de paramètres important est bien classifiée, elle peut "masquer" les mauvais résultats obtenus pour d'autres classes plus petites.

Score F1

Comme nous l'avons vu, le taux d'accuracy reste trop général pour correctement caractériser la performance d'une classification. En effet, il n'est pas pondéré selon le nombre d'élément de chaque classe ce qui peut induire des erreurs d'appréciation. Nous allons donc utiliser une autre métrique : le score F1. Pour ce faire, introduisons deux termes :

La précision : la précision est le ratio d'observations positives correctement prédites sur le total des observations positives prédites. On a donc :

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.1}$$

Le taux de rappel : le taux de rappel est le ratio d'observations positives correctement prédites sur le total des observations de cette classe. On a donc :

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.2}$$

Il est souvent intéressant de comparer deux versions d'un classeur pour déterminer lequel est le plus performant. Une idée intuitive serait de faire la moyenne de la précision et du taux de rappel. Pourtant cette idée est loin d'être la meilleure. Considérons par exemple deux cas :

- Cas 1 : Taux de rappel élevé, faible précision : Cela signifie que la plupart des exemples positifs sont correctement reconnus (FN faible) mais il y a beaucoup de faux positifs.
- Cas 2 : Faible taux de rappel, haute précision : Cela montre que nous manquons beaucoup d'items positifs (FN élevé) mais ceux que nous pronostiquons positifs le sont en effet (FP faible)

Ainsi, dans le cas d'une précision très mauvaise et d'un taux de rappel très élevé (ou inversement), nous pouvons rapidement arriver à une mauvaise estimation de la performance de notre algorithme. C'est pour cette raison que nous introduisons une nouvelle métrique : **le score F1**. Elle est définie comme étant la moyenne harmonique de la précision et du taux de rappel. On a donc :

$$\text{f1-score} = \frac{2 \times (\text{Recall} \times \text{Precision})}{(\text{Recall} + \text{Precision})} = 2 \times \frac{PPV \times TPR}{PPV + TPR} \tag{2.3}$$

Dans notre cas, nous cherchons à minimiser le nombre de faux positifs et de faux négatifs. L'utilisation du score F1 nous permet de les prendre tous deux en compte.

Chapitre 3

Paramétrage et résultats

3.1 Recherche des paramètres optimaux

À cette étape du projet, nous avions utilisés différents classeurs mais laissé les paramètres de ces derniers aux valeurs par défaut. Les résultats étaient assez mauvais pour SVM, le perceptron multi-couches et AdaBoost. Seule la Random Forest semblait se comporter correctement. Nous nous sommes donc penché sur l'optimisation des paramètres des trois premiers classeurs cités.

Une majeure partie de notre temps a été investie dans la recherche des jeux de paramètres optimaux pour chacun des classeurs sélectionnés. Nous avons procédé en deux temps. Premièrement, nous avons implémenté un script testant des intervalles de paramètres et renvoyant les meilleurs combinaisons. Puis, nous avons utilisé ce script pour des intervalles de valeurs de paramètres que nous avons jugé pertinents.

3.1.1 Recherche exhaustive et validation croisée

Notre algorithme de recherche des paramètres est basé sur une fonction de scikit-learn, dans le module de sélection de modèle : `GridSearchCV`. Cette fonction est utilisée dans la méthode `tune_parameters` du `Detector`. Cette fonction effectue une recherche exhaustive d'un jeu de paramètres optimal pour un classeur donné à partir d'un dictionnaire Python [1], dont les clés (index) sont les paramètres à faire varier et les valeurs les listes des valeurs prises par ces paramètres.

Cette fonction utilise de plus la technique de validation croisée. En quelques mots, cette technique consiste en la division des données d'entraînement en C sous-échantillons. De façon cyclique, 1 sous-échantillon servira à la validation du modèle, tandis que les $C-1$ autres permettront l'entraînement du modèle. On répète la technique de validation croisée C fois, afin que chaque sous-échantillon soit utilisé pour la validation une fois. La performance du modèle est ensuite évaluée en moyennant les scores de chaque itération. Cette méthode permet de prédire l'efficacité d'un classeur en évitant tout phénomène d'*overfitting* sans avoir à disposer d'un jeu de test indépendant.

1. Conteneur pouvant être indexé par n'importe quel objet dont Python peut produire un *hash*. Le fonctionnement est similaire à une table de hachage en C ou une Map en Java.

La fonction d'évaluation du score utilisée par `GridSearchCV` est par ailleurs paramétrable. Dans le cadre de notre projet, nous nous intéressons au *f1-score* pour évaluer un classeur, et nous souhaitions aussi extraire les tendances de l'évolution du score en fonction des différents paramètres testés. Nous pourrons alors interpréter l'impact des différents paramètres, et expliquer leur rôle. Nous avons donc implémenté notre propre fonction d'évaluation, prenant en paramètre le classeur à évaluer (Section 1.3) et les matrices de *features* et de *labels* (Section 2.3.1) à utiliser pour cela. Cette fonction effectue les opérations suivantes :

1. Classification des éléments du jeu de test (ici, le sous-échantillon mentionné dans la validation croisée). Production d'une matrice de *labels*, contenant les classes résultant de la classification.
2. Calcul de la matrice de confusion entre la prédiction et la matrice donnée en consigne. Parmi les données qui viennent d'être classé, on compte le nombre de vrais positifs, de vrais négatifs, de faux positifs et de faux négatifs.
3. Sauvegarde des paramètres et du score
4. Retour du *f1-score*

L'étape 3 consiste en l'écriture dans un fichier d'une ligne contenant la description détaillée des paramètres du classeur ainsi que les résultats des scores. La fonction de recherche exhaustive étant fortement parallèle, chaque processus écrit dans son propre fichier. Une fois tous les tests effectués, ces fichiers sont concaténés en un grand fichier. Cette méthode d'enregistrement continu permet en outre de sauvegarder les résultats des tests successifs des itérations de la validation croisée au fur et à mesure de l'exécution du programme, ce qui se révèlera important lors d'interruptions prématurées de cette exécution.

En effet, ce fut l'un de nos plus importants problèmes lors de ce projet. Pour le Multi-layered Perceptron, il y avait beaucoup de paramètres en jeu, et la complexité combinatoire mélangée avec le temps de convergence des réseaux neuronaux ont rendu le temps d'exécution du script interminable. Notre plus long essai eut lieu sur le serveur InfRes lame10 doté pourtant de 80 CPUs, sur une durée de plus de 18 heures ; l'intégralité des tests n'aura pu être effectuée, nous avons dû manuellement interrompre l'exécution.

Enfin, du grand fichier généré contenant l'historique de tous les tests passés sont ensuite extraites les tables de score CSV [2] qui seront présentées dans les paragraphes suivant. Pour chaque phase de test, le script génère :

— Un fichier de scores (TPR, FPR, TNR, FNR, PPV, *f1-score*, temps de prédiction)
— Pour chaque classe différente de classeur, un fichier contenant tous les jeux de paramètres

Ces tables sont, à la manière des tables en SQL, liées par une clé primaire externe qui est l'identifiant (unique) de la combinaison de paramètres.

2. Ces dernières sont disponibles à l'adresse `perso.telecom-paristech.fr/ychalier/anomaly/`.

3.1.2 Multi-layered perceptron

À l'aide d'un autre petit script Python, nous avons pu afficher les données récoltées dans les CSV mentionnés plus tôt. Les Figures 3.1 et 3.2 sont le résultat du test pour les paramètres du MLP. La Table 3.1 regroupe les différents paramètres testés, à savoir :

learning rate Taux de modification des poids à chaque mise à jour (i.e. à chaque rétropropagation)

alpha Pénalité de régularisation (en utilisant la norme l_2)

activation Fonction d'activation du réseau neuronal

solver Algorithme de descente du gradient

hidden layers size Nombre et taille des couches cachées du réseau

paramètre	ensemble des valeurs testées
`learning_rate`	`'constant'`, `'invscaling'` et `'adaptive'`
`alpha`	$\{10^{-k} \mid k \in [\![4, 7]\!]\}$
`activation`	`'identity'`, `'logistic'`, `'tanh'` et `'relu'`
`solver`	`'lbfgs'`, `'sgd'` et `'adam'`
`hidden_layer_sizes`	0 à 5 *layers*, de taille variant de 1 à 49

TABLE 3.1 – Liste des paramètres testés pour le multi-layered perceptron

L'exploitation des résultats est rendue très difficile par le manque de consistance des scores. Cependant, nous avons pu remarqués que des paramètres comme la régularisation (`alpha`) ou la taille et nombre de *layers* n'avaient que peu d'influence sur les résultats. Par contre, deux éléments semblent ce dégager :

- la tangente hyperbolique comme fonction d'activation (Figure 3.1)
- *lbfgs* comme méthode du gradient (la documentation de sklearn prédit cette préférence lorsqu'il y a peu de données à exploiter) (Figure 3.2)

Enfin, la Table 3.2 donne les résultats donnés par la recherche exhaustive et validation croisée (pour un *f1-score* moyen maximal de 0.937, ce qui est très prometteur).

paramètre	valeur
`learning_rate`	`'constant'`
`alpha`	10^{-6}
`activation`	`'tanh'`
`solver`	`'lbfgs'`
`hidden_layer_sizes`	`[28, 28, 28]`

TABLE 3.2 – Paramètres optimaux pour MLP

Remarque Sur les Figures 3.1 et 3.2, nous pouvons observer une "barre" en (0,0). Ces points correspondent aux tests avec 0 couches cachées (qui sont, par convention, de taille nulle). Les résultats ne sont donc pas étalés comme pour les autres tailles.

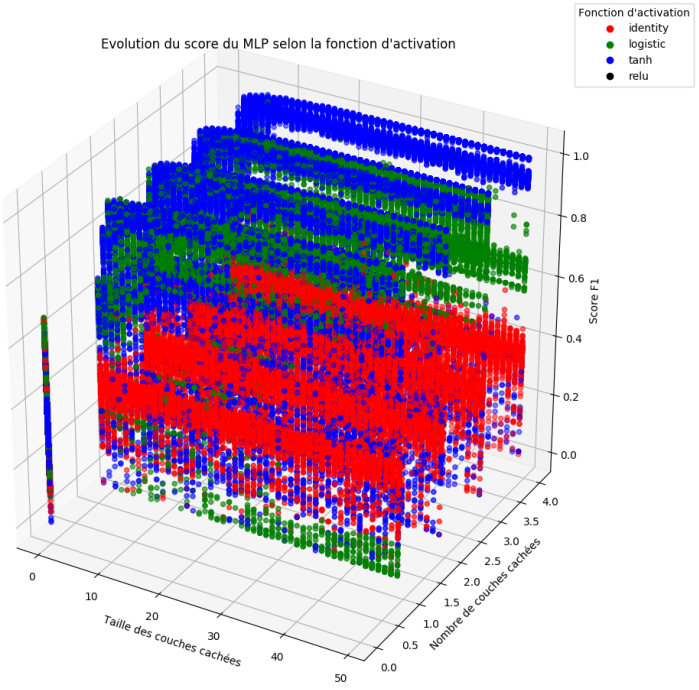

FIGURE 3.1 – Évolution du *f1-score* pour MLP selon la fonction d'activation

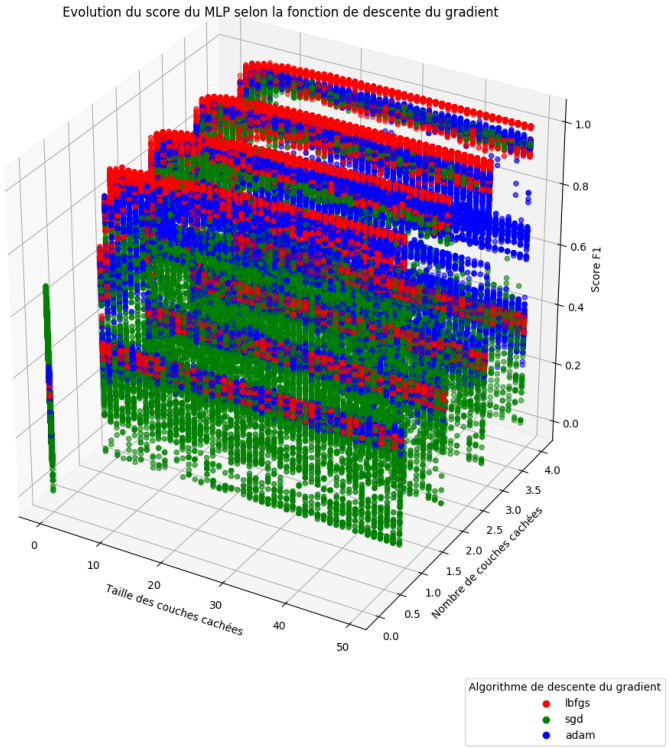

FIGURE 3.2 – Évolution du *f1-score* pour MLP selon l'algorithme de descente du gradient

3.1.3 AdaBoost

Comme pour MLP, la Table 3.3 liste les paramètres testés et la Figure 3.3 montre l'évolution du score en fonction de ces paramètres.

paramètre	ensemble des valeurs testées
`n_estimators`	$[\![1, 99]\!]$
`learning_rate`	$\{k/10 \mid k \in [\![1, 9]\!]\}$
`base_estimator`	Arbres de décision de profondeur maximale dans $[\![2, 9]\!]$

TABLE 3.3 – Liste des paramètres testés pour AdaBoost

Ici, il y a peu d'interprétations possibles : les paramètres influent peu sur la performance du classeur. Dès lors que les arbres sont suffisamment profonds, qu'il y a assez d'estimateurs agrégés et que le taux d'apprentissage est correct (supérieur à 0.1), les résultats sont vite optimaux. La fonction de recherche exhaustive nous a renvoyé la combinaison de paramètres présentée dans la Table 3.4 (pour un *f1-score* moyen maximal de 0,947, proche de celui du MLP, tout aussi prometteur).

paramètre	valeur
`n_estimators`	46
`learning_rate`	0,3
`base_estimator`	Arbres de décision de profondeur maximale 3

TABLE 3.4 – Paramètres optimaux pour AdaBoost

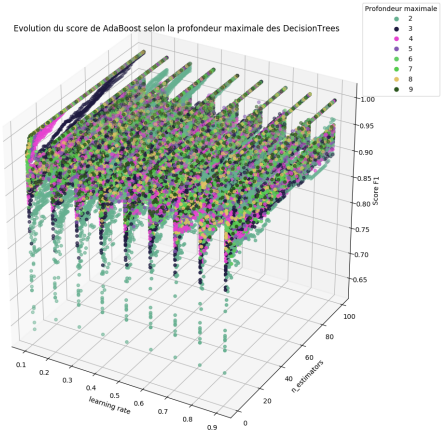

FIGURE 3.3 – Évolution du *f1-score* pour AdaBoost selon la profondeur maximale de l'arbre de décision

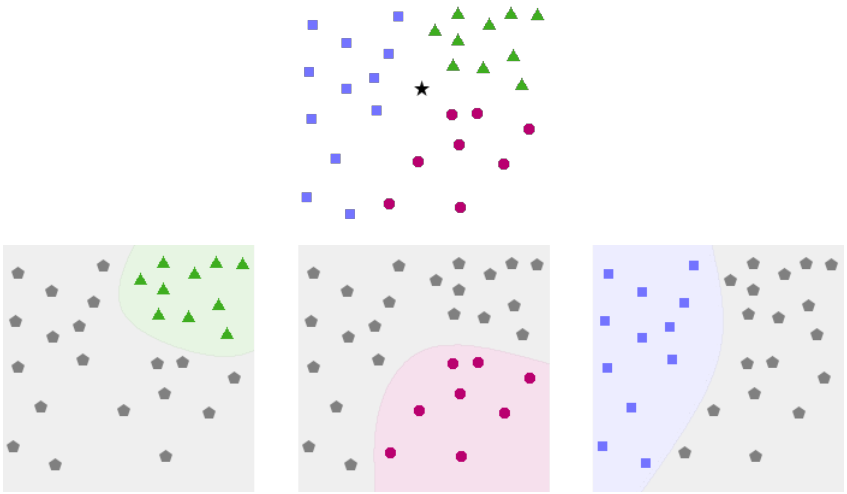

FIGURE 3.4 – Problème multi-classe et séparation "One-Versus-the-Rest"

3.1.4 SVM

L'algorithme du SVM est initialement défini pour la discrimination d'une variable qualitative binaire. Il est à ce titre particulièrement indiqué dans le cas de problèmes linéairement séparables. Or, nous sommes en présence d'un problème à trois classes. Nous allons donc devoir adapter la méthode générale. Plusieurs adaptations existent. Parmi ces dernières, nous avons considéré les deux suivantes :

"One-Versus-the-Rest" La plus simple et la plus ancienne des méthodes d'otpimisation indirectes. Il s'agit d'une extension au cas multiclasses proposée par Vapnik [19]. Elle consiste à construire p estimateurs binaires (à vecteurs de supports) pour classifier p classes (voir Figure 3.4). S'il s'agit d'une extension facile à mettre en place, cette méthode risque d'introduire artificiellement un déséquilibre des classes dans la construction des modèles individuels. Dans notre cas, nous avons 70847 points représentants des voitures contre 202 représentants des motos. Aussi, la méthode SVM obtiendra des moins bons résutlats pour séparer ces deux classes.

Méthode directe de Crammer et Singer [20] À l'encontre des schémas de décomposition indirect (comme la méthode "OVR"), cette approche consiste à séparer les classes en résolvant un unique problème d'optimisation. Ainsi, cela revient à résoudre un problème d'optimisation à $n(p-1)$ contraintes (où p est le nombre de classe et n le nombre d'observations).

paramètre	ensemble des valeurs testées
`multi_class`	`'ovr'`, `'crammer_singer'`
`C`	$\{10^k \mid k \in [\![-2, 3]\!]\}$
`tol`	$\{10^{-k} \mid k \in [\![3, 6]\!]\}$

TABLE 3.5 – Liste des paramètres testés pour SVM

Comme pour les méthodes précédentes, la Table 3.5 liste les paramètres testés. Nous noterons que pour tous les essais, `LinearSVC` a eu pour paramètres

`penalty='l1'` et `dual=False`

En effet, comme nous avons trois classes pour deux caractéristiques de sélection, il est plus performant de résoudre le problème primal (par défaut, le dual est sélectionné). De plus, `GridsearchCV` ne permet pas la combinaison du

paramètre	valeur
`multi_class`	crammer_singer
`C`	100
`tol`	0.00001

Table 3.6 – Paramètres optimaux pour SVM

Pour ces paramètres, nous obtenons la matrice de confusion suivante :

$$\begin{bmatrix} 2836 & 1 \\ 400 & 86 \end{bmatrix}$$

Il faut tout de même noter que ces résultats sont sujets à des fortes variations.

3.2 Performances des classeurs

En utilisant les paramètres optimaux présentés dans les Tables 3.2, 3.4 et 3.6, nous avons pu entraîner les différents classeurs qui nous intéressaient pour pouvoir comparer leurs performances. Les scores obtenus sont reportés dans la Table 3.7.

Le classeur AdaBoost possède le meilleur score F1 des quatre. Juste derrière, la Random Forest semble un très bon choix également, avec tout de même un taux de vrais positifs légèrement inférieur. La Figure 3.5 représente les régions de décision pour le classeur AdaBoost.

Le réseau de neurones possède un score F1 de 0.71, ce qui est assez décevant compte tenu des scores atteints lors de la recherche des paramètres optimaux (Section 3.1.2). Le taux de vrais positifs est assez bas (environ 57%) alors que le taux de faux négatifs est important (43%).

Enfin, le classeur SVM arrive en dernière position. Avec un taux de faux positif de 100% et de vrais négatifs de 0%, la méthode donne des résultats plus que décevants.

	TPR	**FPR**	**TNR**	**FNR**	**PPV**	***f1-score***
MLP	0.5677	0.0053	0.9947	0.4323	0.9493	0.7105
AdaBoost	0.9354	0.0103	0.9897	0.0646	0.9411	0.9382
SVM	0.9657	1.0	0.0	0.0343	0.1446	0.2515
Random Forest	0.9172	0.0067	0.9933	0.0828	0.9598	0.938

Table 3.7 – Tableau récapitulatif des performances des classeurs

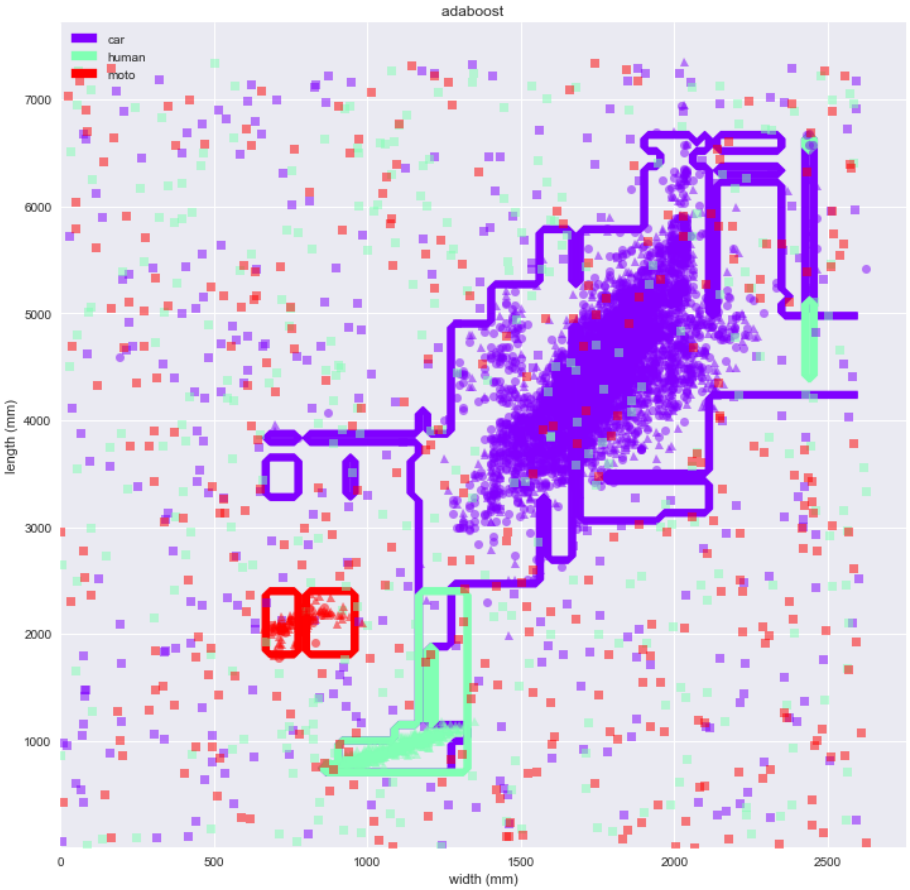

FIGURE 3.5 – Régions de décisions pour AdaBoost

3.3 Utilisation de la prédiction en-ligne

L'objectif final de ce projet est la mise à disposition d'une fonction de prédiction autonome permettant de classer des données relevées par un ensemble de capteurs en malicieux / non-malicieux.

Cette fonction est déjà implémentée dans le code, il s'agit de la méthode `predict` de la classe `Detector`. Le processus pour initialiser cette fonction est le suivant :

1. Créer un objet `Detector` en chargeant les bases de données récoltées
2. Entraîner un classeur, dont les paramètres sont ceux résultant de l'optimisation effectuée précédemment, avec ces données
3. Attribuer ce classeur en tant que classeur de prédiction pour le `Detector`
4. Sauvegarder la méthode `predict`

La dernière étape de ce processus consiste en la sérialisation de la méthode, à l'aide du module Pickle. Ce module permet d'écrire octet par octet la fonction telle qu'elle est représentée dans la RAM de Python, pour pouvoir la recharger plus tard et l'appeler telle une fonction normale. En effet, la méthode contient une référence au classeur sélectionné (en attribut de l'objet référent). Par un effet de fermeture [3], cette valeur sera aussi enregistrée lors de la sérialisation de la méthode. La fonction possède donc tous les éléments pour effectuer les prédictions. Elle est stockée dans un fichier *anomaly_classifier.clf*. Son utilisation peut alors être résumée à l'aide de la fonction suivante :

```
def classify(class_, length, width):
    import pickle

    # Ouverture du fichier contenant la fonction
    # en mode de lecture binaire
    file = open('anomaly_classifier.clf', 'rb')

    # Chargement du contenu du fichier dans la RAM,
    # et stockage de la référence de la fonction
    # dans la variable fun
    fun = pickle.load(file)
    file.close()

    # Classification
    if fun(class_, length, width):
        return "malicious"
    else:
        return "non-malicous"
```

Cette fonction est autonome, elle n'a plus besoin de l'objet `Detector` pour fonctionner. De plus, elle est portable, puisqu'elle ne nécessite qu'un fichier contenant la méthode citée précédemment, fichier pesant environ 400Ko, ce qui est bien inférieur à la taille des bases de données nécessaires pour entraîner le classeur (14 Mo).

3. La fonction utilise des variables indépendantes (utilisées localement mais définies dans la portée englobante). Autrement dit, ces fonctions se souviennent de l'environnement dans lequel elles ont été créées.

Chapitre 4

Conclusions

Dans ce travail, nous avons implémenté un algorithme de classification d'obstacles utilisant quatre algorithmes d'apprentissage supervisé. Une étude comparative des performance de ces derniers a été mené en prenant comme critère d'efficacité leur score F1 respectif. A la suite de ce travail, il apparaît que les méthodes de Random Forest et de SVM optimisé permettent d'atteindre des score F1 supérieur à 0.93. Cette classification fiable laisse entrevoir une utilisation de ce modèle dans des systèmes autonomes.

Dans des travaux futurs, une attention toute particulière devra être portée sur la sécurité du dispositif. A mesure que les systèmes s'automatisent, la résistance générale des dispositifs de prédiction à des actions malveillantes doivent devenir une priorité des futurs développement.

Bibliographie

[1] N. Deepika and V. V. Sajith Variyar. Obstacle classification and detection for vision based navigation for autonomous driving. In *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 2092–2097, Sept 2017.

[2] Gerard Gibbs, Huamin Jia, and Irfan Madani. Obstacle detection with ultrasonic sensors and signal analysis metrics. *Transportation Research Procedia*, 28 :173 – 182, 2017. INAIR 2017.

[3] Hocheol Shin, Dohyun Kim, Yujin Kwon, and Yongdae Kim. Illusion and dazzle : Adversarial optical channel exploits against lidars for automotive applications. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 445–467, Cham, 2017. Springer International Publishing.

[4] Mourad Bendjaballah, Stevica Graovac, and Mohammed Amine Boulahlib. A classification of on-road obstacles according to their relative velocities. *EURASIP Journal on Image and Video Processing*, 2016(1) :41, Dec 2016.

[5] D. V. Prokhorov. Road obstacle classification with attention windows. In *2010 IEEE Intelligent Vehicles Symposium*, pages 889–895, June 2010.

[6] M. S. Darms, P. E. Rybski, C. Baker, and C. Urmson. Obstacle detection and tracking for the urban challenge. *IEEE Transactions on Intelligent Transportation Systems*, 10(3) :475–485, Sept 2009.

[7] M. Liang, X. Huang, C. H. Chen, X. Chen, and A. Tokuta. Counting and classification of highway vehicles by regression analysis. *IEEE Transactions on Intelligent Transportation Systems*, 16(5) :2878–2888, Oct 2015.

[8] Virtualenv. Disponible sur `https://virtualenv.pypa.io/en/stable/`. Consulté le 2018-06-20.

[9] Pandas, python data analysis library. Disponible sur `http://pandas.pydata.org`. Consulté le 2018-06-20.

[10] scikit-learn : machine learning in python. Disponible sur `http://scikit-learn.org/stable/index.html`. Consulté le 2018-06-20.

[11] Numpy. Disponible sur `http://www.numpy.org`. Consulté le 2018-06-20.

[12] Matplotlib : Python plotting. Disponible sur `https://matplotlib.org`. Consulté le 2018-06-20.

[13] Project jupyter. Disponible sur `http://jupyter.org`. Consulté le 2018-06-20.

[14] Carquery, the vehicle data api & database. Disponible sur `http://www.carqueryapi.com`. Consulté le 2018-06-20.

[15] base de donn moto 1. Disponible sur `http://www.teoalida.com/cardatabase/motorcycles/`. Consulté le 2018-06-20.

[16] base de donn moto 2. Disponible sur `http://www.motocicliste.net/moto/db.asp`. Consulté le 2018-06-20.

[17] Grete Heinz, Louis J. Peterson, Roger W. Johnson, and Carter J. Kerk. Exploring relationships in body dimensions. *Journal of Statistics Education*, 11(2) :null, 2003.

[18] anomaly. Disponible sur `https://github.com/ychalier/anomaly/`. Consulté le 2018-06-20.

[19] V. N. Vapnik. An overview of statistical learning theory. *IEEE Transactions on Neural Networks*, 10(5) :988–999, Sep 1999.

[20] Koby Crammer and Yoram Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *Journal of machine learning research*, 2(Dec) :265–292, 2001.

Table des figures

Liste des tableaux