Télécom ParisTech

Projet de filière SR2I

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

Antoine Urban, Yohan Chalier

encadré par
Jean-Philippe Monteuuis
Houda Labiod

15 juin 2018

Résumé

Chapitre 1

Démarche et stratégie

1.1 Première implémentation

1.1.1 Objectif

En premier lieu, nous souhaitions commencer par une vision globale des données et du travail à effectuer. Nous disposions d'une base de données contenant des mesures de voiture, provenant de CarQuery, et contenant 54808 lignes complètes. Dans cette partie, nous allons nous efforcer d'obtenir une première fonction de classification se basant sur des critères très simple : des régions de décision rectangulaires et arbitraires.

1.1.2 Mise en œuvre

Puisque l'objectif de cette étude est la détection d'anomalies dans la mesure de longueur et de largeur, nous avons extrait les deux colonnes correspondantes dans une DataFrame du module Pandas, en Python.

Après un premier affichage des données, 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 5026.

Manuellement, nous avons alors défini des zones simples (rectangulaires) en tant que régions de décision (Table 1.1). 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.

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 1.1 – 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 1.1 représente l'affichage de tous les points décrits plus tôt ainsi que des régions de décision.

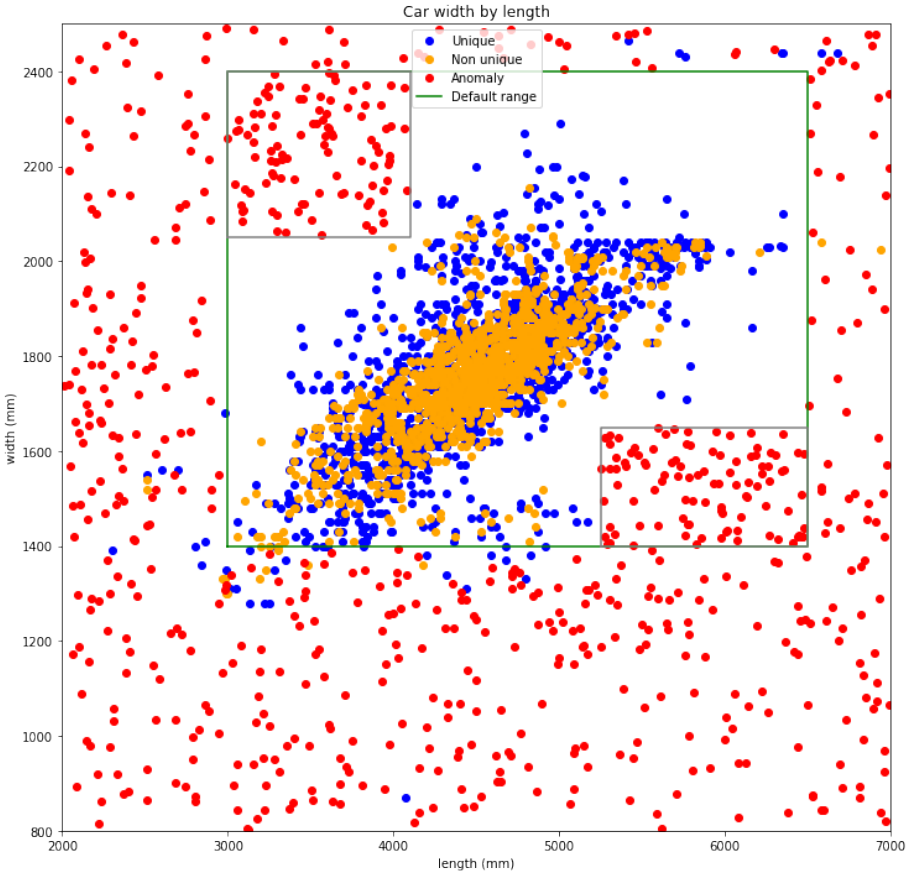

FIGURE 1.1 – Régions de décision manuelles pour des dimensions de voitures

Ainsi faite, notre classification possède, sur le jeu d'entraînement, une précision de 97,57%.

1.2 Recherche des bases de données

1.3 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 : github.com/ychalier/anomaly.

1.3.1 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`. 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 :

— `numpy`

— `pandas`
— `matplotlib`
— `jupyter`
— `scikit-learn`

1.3.2 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).

Génération des données malicieuses 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 Avec le *seed* généré précédemment, la grande matrice 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

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`.

1.3.3 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 *classifiers*, et l'affichage des données. Ses méthodes (Table 1.2) 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 données
	`append_odd_points`	Étape 7 du chargement des données
	`format`	Étapes 8 et 9 du chargement des données
Interface `sklearn`	`classify`	Entraîne un *classifier* et renvoie le score de test
	`tune_parameters`	Trouve le meilleur jeu de paramètres pour un *classifier*
	`predict`	Fonction finale de prédiction online
Affichage	`plot`	Affiche la matrice de données complètes
	`plot_decision_boudaries`	Affiche les régions de décisions d'un *classifier*

Table 1.2 – Méthodes de la classe `Detector`

1.4 Méthode d'évaluation

Chapitre 2

Paramétrage et résultats

2.1 Recherche des paramètres optimaux

Une majeure partie de notre temps a été investie dans la recherche des jeux de paramètres optimaux pour chacun des *classifiers* 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.

2.1.1 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 *classifier* donné à partir d'un dictionnaire dont les clés 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. 1 sous-échantillon servira à la validation du modèle, tandis que les $C - 1$ autres permettront l'entraînement du modèle. On répète cette opération C fois, afin que chaque sous-échantillon soit utilisé pour la validation une fois. La performance du modèle est ensuite évaluée en effectuant la moyenne des C erreurs quadratiques.

La fonction d'évaluation du score utilisée par `GridSearchCV` est par ailleurs modifiable. Dans le cadre de notre projet, nous nous intéressons au *f1-score* pour évaluer un modèle, et nous souhaitions aussi extraire, si possible, les tendances d'évolution du score en fonction des différents paramètres testés. Nous avons donc implémenté notre propre fonction d'évaluation, prenant en paramètre l'estimateur à évaluer et les matrices de *features* et de *labels* à utiliser pour cela. Cette fonction effectue les opérations suivantes :

1. Prédiction des *labels* à partir de la matrice de *features*
2. Calcul de la matrice de confusion entre la prédiction et la matrice donnée en consigne

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 de l'estimateur 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 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 l'exécution du script interminables. Notre plus long essai eut lieu sur le serveur InfRes lame10 doté pourtant de 80 cpus, sur une durée de plus de 18 heures. Malgré tout, il aura fallut interrompre l'exécution du programme manuellement pour rendre le serveur disponible. Le fichier généré pesait plus de 120Mo, et les deux CSV extraits pèsent respectivement 25Mo et 23Mo.

Enfin, du grand fichier généré contenant l'historique de tous les tests passés sont ensuite extraites les tables de score CSV qui seront présentées dans les paragraphes suivant. Ces dernières sont disponibles à l'adresse `https://perso.telecom-paristech.fr/ychalier/anomaly/params/`. 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 d'estimateur, 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.

2.1.2 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 2.1 et 2.2 sont le résultat du long test pour les paramètres du MLP. La Table 2.1 regroupe les différents paramètres testés.

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 2.1 – 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

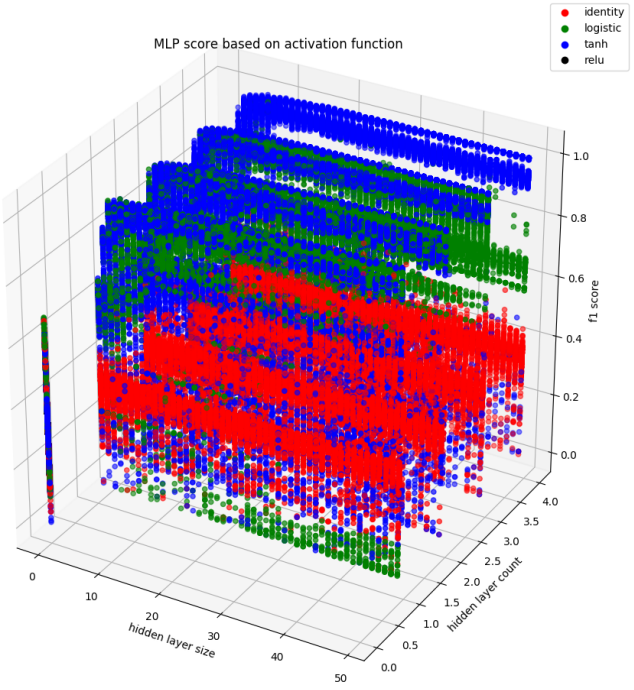

FIGURE 2.1 – Évolution du score pour MLP selon la fonction d'activation

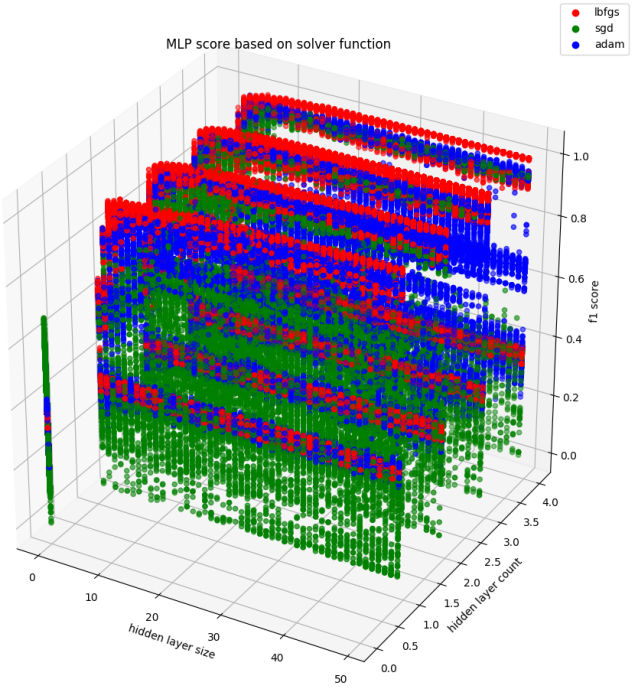

FIGURE 2.2 – Évolution du score pour MLP selon l'algorithme de descente du gradient

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

— *lbfgs* comme méthode du gradient (la documentation de sklearn prédit cette préférence lorsqu'il y a peu de données à exploiter)

Enfin, la Table 2.2 donne les résultats donnés par la recherche exhaustive et validation croisée.

paramètre	valeur
`learning_rate`	
`alpha`	
`activation`	`'tanh'`
`solver`	`'lbfgs'`
`hidden_layer_sizes`	

TABLE 2.2 – Paramètres optimaux pour MLP

2.1.3 AdaBoost

Comme pour MLP, la Table 2.3 liste les paramètres testés et la Figure 2.3 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 2.3 – Liste des paramètres testés pour AdaBoost

Ici, il y a peu d'interprétations possibles : les paramètres influent peu sur la performance de l'estimateur. Dès lors que les arbres sont suffisamment profonds, qu'il y a assez d'estimateurs agrégés et que le taux d'apprentissage et correct, les résultats sont vite optimaux. La fonction de recherche exhaustive nous a renvoyé la combinaison de paramètres présentée dans la Table 2.4 (pour un *f1-score* moyen maximal de 0,947).

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

TABLE 2.4 – Paramètres optimaux pour AdaBoost

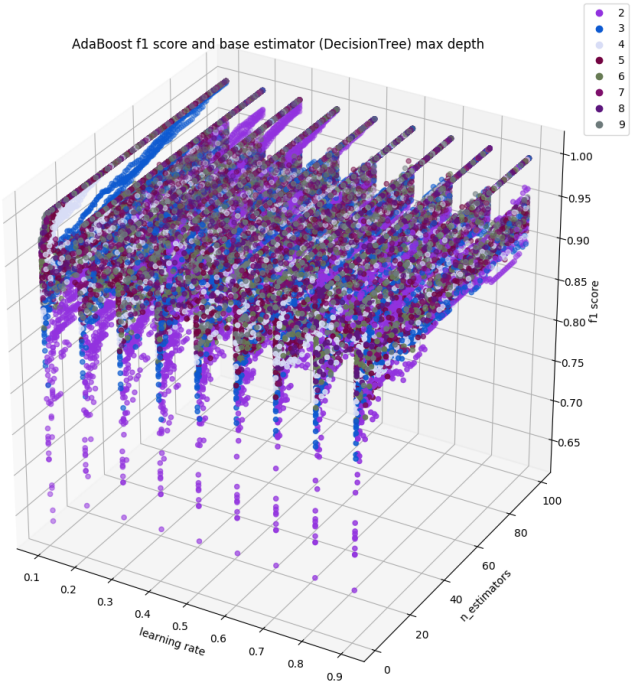

FIGURE 2.3 – Évolution du score pour AdaBoost selon la profondeur maximale de l'arbre de décision

2.1.4 SVM

2.2 Étude comparative des performances

2.3 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 *classifier*, dont les paramètres sont ceux résultant de l'optimisation effectuée précédemment, avec ces données
3. Attribuer ce *classifier* en tant que *classifier* 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`. La méthode, contenant une référence à l'estimateur sélectionné, possède tous les éléments pour effectuer les prédictions. Son utilisation peut alors être résumée à l'aide de la fonction suivante :

```
def classify(class_, length, width):
    import pickle
    file = open('anomaly_classifier.clf', 'rb')
    fun = pickle.load(file)
    file.close()
    if fun(class_, length, width):
        return "malicious"
    else:
        return "non-malicous"
```

Cette fonction est autonome et portable ; elle ne nécessite qu'un fichier contenant la méthode citée précédemment.