# Détection d'Injections SQL par Apprentissage Automatique

USCB1F - Alternance Master Sécurité informatique 2025-2027 : Intelligence Artificielle (2025 - 2026 Annuel)

Auteur : GALLE Fabien
Entreprise: Devensys Cybersecurity
Poste: Cybersecurity Analyst


## Introduction

Les injections SQL représentent l'une des vulnérabilités web les plus critiques. En consultant le classement OWASP Top 10, j'ai constaté qu'elles figurent régulièrement parmi les principales menaces depuis plus de vingt ans. Les approches classiques comme les expressions régulières ou les WAF (Web Application Firewall) montrent leurs limites face aux attaquants qui trouvent constamment de nouvelles façons de contourner les protections via des variations syntaxiques (encodages diverses et variés, obfuscation, etc.).

C'est dans ce contexte que j'ai voulu explorer une approche basée sur l'apprentissage automatique. Mon hypothèse de départ était assez simple : si un modèle est entraîné sur suffisamment d'exemples de requêtes légitimes et malveillantes, il devrait être capable d'identifier les patterns caractéristiques des injections SQL, même celles légèrement modifiées.

Pour ce faire, j'ai utilisé le "Modified SQL Dataset" disponible sur Kaggle, qui contient 30 919 requêtes SQL réparties en deux catégories : 19 464 requêtes normales (soit 63% du dataset) et 11 455 injections SQL (37%). Cette proportion m'a semblé assez représentative d'un cas réel où les requêtes légitimes sont majoritaires.

https://www.kaggle.com/datasets/sajid576/sql-injection-dataset

# Solution Proposée

## Prétraitement et vectorisation

Le premier défi auquel j'ai été confronté était de transformer les requêtes SQL (qui sont du texte) en données numériques que les algorithmes de machine learning peuvent traiter. Au départ, j'ai pensé utiliser une simple vectorisation par mots (bag-of-words), mais après réflexion, cette approche ne semblait pas adaptée aux requêtes SQL qui contiennent beaucoup de caractères spéciaux et de symboles significatifs.

J'ai donc opté pour une vectorisation TF-IDF basée sur des n-grammes de caractères (de 1 à 3 caractères). L'avantage de cette méthode est qu'elle capture les patterns syntaxiques caractéristiques des injections comme `' OR`, `1=1`, `--`, etc., qui ne seraient pas détectés avec une tokenisation classique par mots. Cette approche m'a paru plus pertinente pour ce type de problème.

## Algorithmes testés

Pour choisir le meilleur modèle, j'ai décidé de tester six algorithmes de classification supervisée différents. J'ai principalement utilisé ceux qu'on a vus en cours dans les exercices précédents, ce qui m'a permis de mieux comprendre leur fonctionnement :

- **Régression Logistique**
- **Arbre de Décision**
- **Forêt Aléatoire (Random Forest)**
- **SVM (Support Vector Machine)**
- **Naive Bayes**
- **Gradient Boosting**

L'idée était de comparer leurs performances pour identifier celui qui donnerait les meilleurs résultats sur ce type de données.

## Méthodologie d'évaluation

J'ai divisé le dataset en 80% pour l'entraînement (soit 24 735 requêtes) et 20% pour les tests (6 184 requêtes), en utilisant la stratification pour maintenir la même proportion de classes dans chaque ensemble.

Pour évaluer les modèles, j'ai utilisé plusieurs métriques : Accuracy, Precision, Recall, F1-Score et ROC-AUC. Dans le contexte de la sécurité, j'ai vite compris que le recall était particulièrement important car un faux négatif (c'est-à-dire une injection non détectée)

représente un vrai risque de sécurité. Cependant, il faut aussi faire attention aux faux positifs : trop de fausses alertes bloqueraient des requêtes légitimes et dégraderaient l'expérience utilisateur.

Enfin, j'ai réalisé une validation croisée à 5 plis sur le meilleur modèle pour vérifier qu'il était stable et qu'il ne dépendait pas trop de la façon dont les données avaient été divisées.

# Résultats

## Performances comparatives

Après avoir entraîné les six modèles sur l'ensemble d'entraînement, j'ai obtenu les résultats suivants sur l'ensemble de test :

| Modèle | Accuracy | Precision | Recall | F1-Score | ROC-AUC |
|---|---|---|---|---|---|
| Gradient Boosting | 0.9871 | 0.9848 | 0.9817 | 0.9833 | 0.9979 |
| Random Forest | 0.9869 | 0.9796 | 0.9880 | 0.9838 | 0.9975 |
| Logistic Regression | 0.9784 | 0.9708 | 0.9751 | 0.9729 | 0.9956 |
| SVM | 0.9751 | 0.9684 | 0.9697 | 0.9691 | 0.9942 |
| Naive Bayes | 0.9572 | 0.9338 | 0.9627 | 0.9480 | 0.9925 |
| Decision Tree | 0.9513 | 0.9273 | 0.9595 | 0.9431 | 0.9522 |

Les résultats sont globalement très bons, ce qui m'a plutôt agréablement surpris. Le Gradient Boosting et le Random Forest sont au coude à coude avec des performances très proches. J'ai finalement choisi le Random Forest comme modèle final pour deux raisons : son F1-Score est légèrement supérieur (0.9838 contre 0.9833), mais surtout il offre la possibilité d'analyser l'importance des features, ce qui permet de mieux comprendre quels patterns le modèle utilise pour détecter les injections.

## Analyse détaillée du Random Forest

En examinant la matrice de confusion sur l'ensemble de test, j'obtiens :

- Vrais Négatifs : 3 877 | Faux Positifs : 63
- Faux Négatifs : 28 | Vrais Positifs : 2 316

Cela correspond à un taux de faux négatifs de seulement 1.2%, ce qui signifie que le modèle détecte correctement 98.8% des injections SQL. Le taux de faux positifs est aussi raisonnable (1.6%), ce qui est acceptable pour ne pas bloquer trop de requêtes légitimes.

Ce qui est intéressant, c'est que l'analyse des features importantes montre que le modèle a bien appris à repérer les patterns caractéristiques des injections : les commentaires SQL (`--`, `/*`), les opérateurs suspects comme `' OR` ou `1=1`, les caractères d'échappement (`\'`, `%27`), et les mots-clés SQL qui apparaissent dans des positions inhabituelles. Le modèle semble donc avoir capturé la "logique" des attaques plutôt que simplement mémoriser des exemples.

### Validation croisée

La validation croisée à 5 plis donne un F1-Score moyen de 0.9836 (± 0.0016), avec des scores individuels allant de 0.9817 à 0.9852. Cette faible variance confirme que le modèle est stable et ne dépend pas trop du découpage particulier des données d'entraînement et de test, ce qui est rassurant.
Logistique

### Tests sur nouvelles requêtes

Pour vérifier la capacité de généralisation, j'ai testé le modèle sur 8 requêtes externes au dataset 4 normales et 4 injections SQL classiques). Toutes ont été correctement classifiées avec des niveaux de confiance supérieurs à 95%. Bien sûr, c'est un échantillon limité, mais ça montre que le modèle ne se contente pas de mémoriser le dataset et peut s'adapter à de nouvelles requêtes.

## Conclusion

Ce projet m'a permis de constater que l'apprentissage automatique peut effectivement être une solution efficace pour détecter les injections SQL. Le Random Forest que j'ai retenu atteint un F1-Score de 98.38%, ce qui est franchement au-delà de ce que j'espérais au départ.

Ce qui m'a aussi surpris, c'est qu'on n'a pas besoin de modèles ultra-complexes ou de deep learning pour obtenir de bons résultats sur ce problème. Les algorithmes "classiques" de machine learning suffisent largement.

Cependant, il faut rester conscient des limites de cette approche. D'abord, le dataset vient d'une seule source (Kaggle) et n'est peut-être pas totalement représentatif de tous les types d'injections SQL qui existent dans la nature. Ensuite, même si 1.2% de faux négatifs, ça paraît faible, dans un contexte de production avec des millions de requêtes, ça peut quand même représenter des vulnérabilités réelles. Enfin, des attaquants sophistiqués pourraient développer de nouvelles techniques d'injection spécifiquement conçues pour contourner ce type de détection.

Pour un déploiement réel, je pense qu'il ne faudrait pas utiliser ce système seul, mais plutôt comme une couche de défense supplémentaire en complément des bonnes pratiques

(requêtes préparées, validation des entrées, WAF). Il faudrait aussi mettre en place un monitoring pour détecter si le taux de faux positifs/négatifs évolue avec le temps et réentraîner le modèle régulièrement avec de nouvelles données.

En résumé, ce miniprojet m'a montré que l'apprentissage automatique a vraiment sa place dans la cybersécurité, mais qu'il faut l'utiliser avec discernement en gardant en tête qu'aucune solution n'est parfaite et que la sécurité repose toujours sur plusieurs couches de défense.

---

### Références


- Dataset : Modified SQL Dataset, Kaggle (2021)
- OWASP Top 10 : https://owasp.org/www-project-top-ten/
- Scikit-learn Documentation : https://scikit-learn.org/