# 🧠 Rapport technique : Détection d'injections SQL par apprentissage automatique

**1. Objectif de la partie**

L'objectif de cette partie du projet est de concevoir, implémenter et intégrer un modèle de machine learning capable de détecter automatiquement les requêtes SQL malicieuses (SQL Injection) dans une interface web. Contrairement à une simple détection par règles, notre approche est basée sur des modèles statistiques entraînés sur des exemples réels.

**2. Technologies utilisées**

* **Langage principal** : Python 3.13
* **Bibliothèques ML** : scikit-learn, pandas, numpy, pickle, json
* **Interface web** :
  + Backend : Flask
  + Frontend : HTML, CSS, JavaScript, Chart.js
* **Outils de développement** : VS Code, Git, Terminal

**3. Pipeline de développement**

**a. 🔄 Préparation des données**

* Datasets fournis : SQL\_Injec\_NormTrain.csv, SQL\_Injec\_Valid.csv, SQL\_Injec\_Test.csv
* Chaque requête est transformée en vecteur de **25 features** (caractéristiques) : longueur, score d’injection, nombre de mots-clés, présence de symboles --, =, OR, parenthèses, etc.
* Fusion des trois jeux de données → Séparation :
  + **80%** pour entraînement + validation croisée
  + **20%** pour évaluation finale (test set)

**b. ✅ Sélection des colonnes**

* Toutes les colonnes fournies dans le dataset sont utilisées : pas de sélection manuelle.
* Le modèle choisit seul les colonnes les plus discriminantes lors de l’apprentissage.

**4. Entraînement des modèles**

Trois modèles de classification supervisée ont été entraînés et comparés :

| **Modèle** | **Validation (val\_acc)** | **Test Accuracy** |
| --- | --- | --- |
| Random Forest | **0.9977** | **0.9978** |
| Support Vector Machine (SVM) | 0.9957 | 0.9960+ (non retenu) |
| Logistic Regression | 0.9819 | 0.9820 |

* **Méthode d'entraînement** : GridSearchCV avec validation croisée (5-fold)
* **Sauvegarde automatique** : modèles .pkl et statistiques .json

**5. Évaluation des performances (RandomForest)**

* **Accuracy** : 99.78 %
* **F1-score** : 99.70 %
* **Précision** : 99.95 %
* **Rappel** : 99.46 %
* **Matrice de confusion** :
  + Faux positifs : 3
  + Faux négatifs : 32
  + Total testé : 15 901 requêtes
* **Top 10 des features importantes** :
  + SCORE\_INJECTION\_NORM
  + NB\_SPECIAL\_CHARS
  + CONTIENT\_EQUAL
  + NB\_COMMENTS\_NORM
  + etc.

**6. Intégration dans l’interface web**

* Backend (Flask) reçoit une requête SQL
* Extraction automatique des features
* Prédiction immédiate par le modèle
* Affichage dans l’interface :
  + Type (benign/malicious)
  + Confiance (%)
  + Détail des features analysées

**7. Problèmes rencontrés & solutions**

| **Problème** | **Solution mise en place** |
| --- | --- |
| ❌ Incohérence entre les features d’entraînement et de test | Ajout d’un export des colonnes utilisées, harmonisation automatique |
| ❌ Mismatch dans l’ordre des colonnes | Génération dynamique d’un DataFrame avec les bons noms |
| ❌ Résultats différents entre terminal et interface Flask | Uniformisation des modèles et fichiers chargés |
| ❌ Bugs dans l’affichage terminal | Nettoyage progressif et standardisation des logs |

**8. Bonnes pratiques respectées**

* Automatisation de l’entraînement + sauvegarde des modèles
* Modularité du code (fonctions claires, séparation logique)
* Tests systématiques avec des jeux de requêtes malicieuses / bénignes
* Utilisation rigoureuse de GridSearchCV pour améliorer la robustesse

**9. Améliorations futures et perspectives**

1. 🧠 Ajout d’une explication pour chaque prédiction : par exemple, ‘caractère suspect OR détecté’.  
2. 🔍 Page de recherche dans l’historique avec filtres par type ou mots-clés.  
3. 📂 Ajout d’un système d’upload de logs (fichier .log ou .txt).  
4. 📈 Statistiques d’analyse dans une page dédiée (nombre de requêtes, % malicieuses, graphiques évolutifs).  
5. 🔐 Authentification simple via sessions Flask pour restreindre certaines pages.

**10. Conclusion**

Cette partie du projet a permis de mettre en œuvre un système **performant**, **automatisé**, et **intégré** pour la détection d’injections SQL. Le modèle RandomForest a donné d’excellents résultats, et l’architecture mise en place pourra facilement être dupliquée pour les autres types d’attaques (ex: HTTP, XSS).

Le système est désormais fonctionnel à la fois en **terminal** et en **interface web**, avec une robustesse assurée par la validation croisée, une automatisation complète du pipeline, et une gestion intelligente des erreurs. L’ensemble est prêt à être déployé ou étendu.