Comprehensive Machine Learning Methods in Detecting SQL Injection Vulnerability

ZIHAN ZHOU, University of Alberta, Canada
JINGTONG YANG, University of Alberta, Canada
YUZHOU LI, University of Alberta, Canada

This research presents a novel approach to detecting SQL injection vulnerabilities using comprehensive machine learning models. With increasing data security challenges, SQL injection remains a significant threat. The research applies the sequential text vectorization technique *Doc2Vec* for SQL semantics preprocessing. For injection analysis, multiple recognition results are weighted and aggregated twice by a two-layer intelligent detection system constructed. This innovative approach excels in terms of recognition quality, cost control and generalization and provides an affordable and effective solution for SQL injection detection, contributing to the practical application and popularization of ML-based injection analysis.

Additional Key Words and Phrases: SQL Injection, Machine Learning, Vulnerability Detection, Static Analysis

ACM Reference Format:
Zihan Zhou, Jingtong Yang, and Yuzhou Li. 2023. Comprehensive Machine Learning Methods in Detecting SQL Injection Vulnerability. 1, 1 (December 2023), 20 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The digital and intelligent age, characterized by the massive use of web-based products, has conveyed new challenges in the domain of data security and program analysis. With more cutting-edge networking techniques continuing to be developed, the time also sees more threats that target them. Among the most notorious and classic threats to web applications is the Structured Query Language (SQL) injection attack. According to the survey *OWASP Top Ten* [OWASP 2021] published by the famous software security community *OWASP*, the year 2017 saw the “injection attack" being considered “the most serious" vulnerability in all. Even though it has been known for over 20 years, it still has “the third place" on the rank (see Figure 1).

Vulnerability analysis and dynamic attack warning processes based on intelligent systems are gradually replacing traditional strategies based on program analysis. In recent years, with the maturity of Natural Language Processing (NLP) techniques, vectorized text can be directly learned by Machine Learning (ML) models for pattern recognition. The trend allows more complicated text detection models to be involved in SQL injection detection tasks, but paying a high price does not lead to significant performance improvement, but rather hinders the practical application and popularization of the new technology.

Increasing the model complexity with more expensive training and application costs does not lead to better solutions for all problems. This research aims to provide a novel realization thought

Authors’ addresses: Zihan Zhou, University of Alberta, Edmonton, Canada, zhou9@ualberta.ca; Jingtong Yang, University of Alberta, Edmonton, Canada, jyang9@ualberta.ca; Yuzhou Li, University of Alberta, Edmonton, Canada, yuzhou5@ualberta.ca.

XXXX-XXXX/2023/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

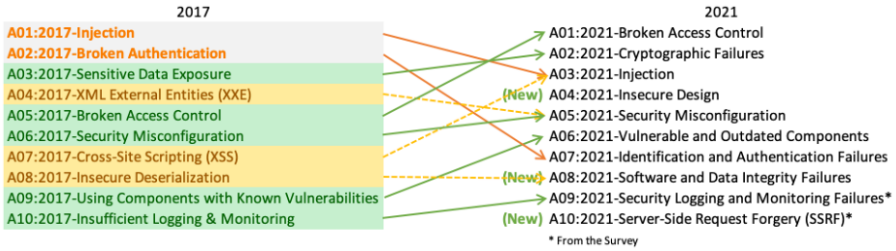

Fig. 1. Attack Method Rank List taken from the *OWASP Top Ten* survey [OWASP 2021].

for SQL injection detection, which allows the performance merging of affordable and efficient models to achieve predictive performance that is as good as or even better than any expensive advanced models. In this paper, based on the more advanced sequential text vectorization technique Doc2Vec, a novel detection strategy based on the comprehensive analysis of multiple machine learning models is proposed. Secondary screening and aggregation of multiple recognition results are performed by building a double-layer weighting model. In the latest "ultimate blind test", the comprehensive weighted model demonstrated more impressive capacity regarding recognition precision and overfitting resistance than the experimented best classic model SVM.

2 BACKGROUND AND MOTIVATION

2.1 How SQL Injection Attack Works?

SQL injection attacks are easy to execute which exploit web vulnerabilities in database query functions by adding malicious codes into some potential vulnerability that existed behind the website interface to manipulate SQL queries, attackers can easily access the application's database to steal unauthorized or sensitive data. This attack can also disrupt the normal website running which may lead to system shutdown with huge business loss. "On average, web attacks caused by SQL injection cause nearly $10 billion loss to the US economy every year." [Wei Zhang 2022] These vulnerabilities are typically caused when the website fails to properly check or sanitize illegal user input before incorporating it into an SQL query.

SQL injection detection and precluding is an active research topic in both industry and academia. Classic SQL injection analysis methods are primarily dropped into two mainstream categories: dynamic analysis and static analysis. The dynamic analysis approaches focus on examining application runtime performance. However, it may not identify the vulnerabilities and risks before the attack really happens. By contrast, the methods in static analysis involve checking the application codes without actual execution. It analyzes the given codes based on precise mathematical and logical control of data flows but is often time-consuming and prone to *False Positive* cases.

2.2 Research Motivation

As researchers argued, web attackers "increasingly sophisticated tools for automatic injection attacks, and the traditional-based approach often cannot cope well." [Wei Zhang 2022] Although static analysis provides a reasonable solution for "injections", the continuing evolution of attack patterns causes the static analysis approaches to rely heavily on the experience of security engineers. To figure out a more effective strategy for handling this issue, novel approaches with more advanced algorithms are required. Employing machine learning can be advantageous since it can uncover some insignificant features of vulnerable cases that cannot be easily defined by knowledge-based dataflow analysis. As long as we continuously update and feed new attack cases to the training

dataset, the fine-tuned machine learning detector can quickly adapt to new injection patterns to cost-effectively defend against emerging attacks.

Deploying intelligent detection solutions to replace traditional analysis-based methods is becoming a trend in ongoing research, but several significant inquiries remain unresolved. Even though we have seen many AI-based detectors implemented in years, "none of them are complete or accurate enough to guarantee an absolute level of security on web applications" [Shehu 2014] and we haven't found a mature approach which provides systematic solutions in dealing and preventing potential injection attacks.

In this research, we would like to continue the study by using machine learning methods in SQL injection detection, but a novel approach about applying multiple dissimilar classifiers on the same tasks provided, with some unique designs raised for integrating model performance to create a comprehensive detection report. For the first stage, we focus on building and operating various machine-learning models exposed to malicious information, aiming to investigate their independent performance in injection defining. And then in the second part, we aggregate and weigh the outcomes from models to form a more powerful intelligent detector to provide an all-around analysis.

Although this study does not directly propose a vulnerability learning model as powerful as a deep neural network-based model, it provides a better detection solution based on lower cost and faster detection. Comprehensive designs based on more stable models also provide generalization and resilience beyond a single complex neural network model. More importantly, this research supplies in-depth insights into multi-model comprehensive analysis and advanced detection systems which may come up with a solution to the utmost extent for handling SQL injection attacks.

2.3 Problem Formulation

Here the architecture flowchart (see Figure 2) about our project method is provided, there are three core phases illustrated: the *Data Phase*, the *Model Phase*, and the *Detector Phase*, which stands for data processing, model classifier works, and comprehensive detector strategy, respectively.

Pt. 1: Data Phase. The beginning part of the *Data Phase* accepts a facts input as the "original data" $\mathcal{D} \in \mathbb{R}^n$, which contains labels $y_i \in y$ and lines of features $x_i \in x$ that corresponds to each label y_i, holding mixed data of normal and malicious SQL queries. After confirmation of the input data, the data preprocessing is taken to convert document-based tokens to numeric vectors by steps of normalization and tokenization. Then the splitting step will follow a settled *splitting ratio* to divide the processed dataset into two pieces dataset for "training/validation" $\mathcal{D}_{t\&v}$ and dataset for "test" $\mathcal{D}_{test}$ automatically, which means we have 90% for training and validation and 90% for examinations (see subsection 4.1 for more details). All these processes contribute to transforming data into formats that can be recognized and learned by the machine learning models designed.

Pt. 2: Model Phase. The data divisions will be processed by the *Model Phase*. We use the same training dataset to train each detection model to ensure fairness and standardization. There are some models constructed for the detection task, including *Logistic Regression*, *Naive Bayes*, *k-nearest Neighbours*, *Support Vector Machines*, *Decision Tree*, and *Gradient Boost Decision Tree*. More details can be found in section 4.2.

We evaluate the performance of different Hyperparameter choices using K-fold cross-validation by evaluating *Mean Square Error (MSE)*, both *macro and micro F1 scores*, and *balanced accuracy*. We will also report the confusion matrix distribution for each model performance and the *macro F1 score* works as the core metric that helps determine the final selection of the K value and specific hyperparameters for each model (refer to subsection 4.3 for more details). After optimization, formal models with the optimal hyperparameter will get access to the vector information and each of

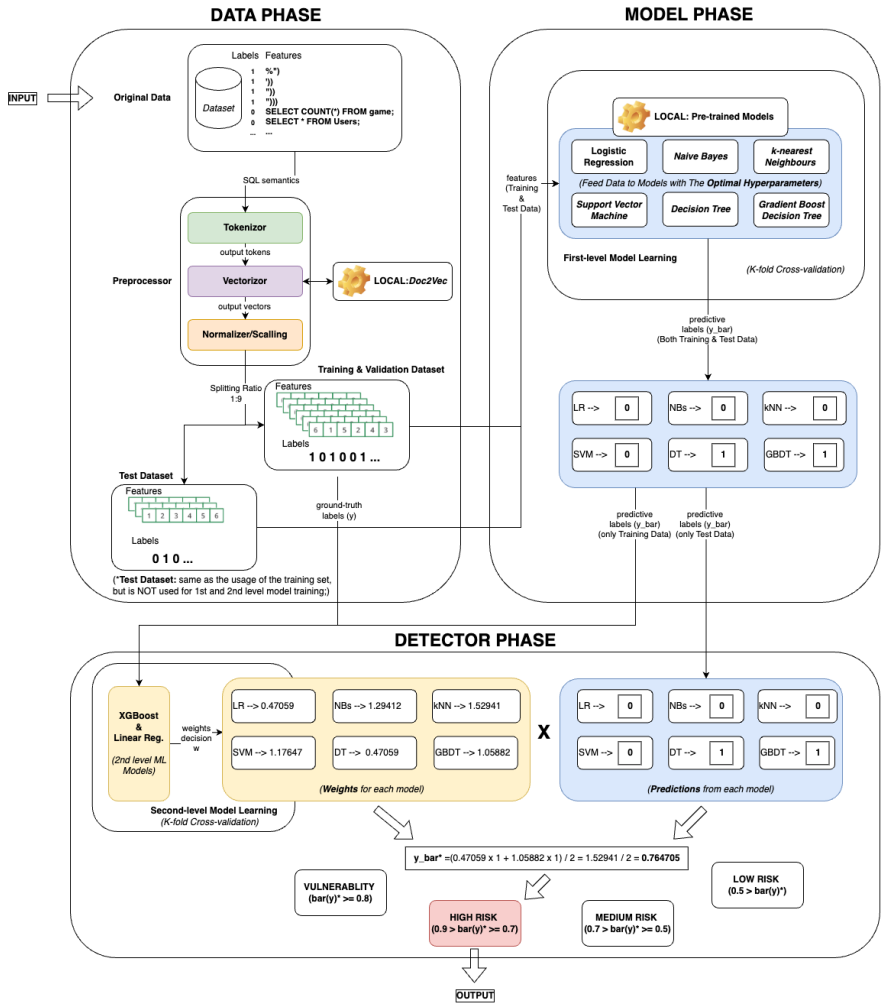

Fig. 2. The architecture flowchart of our comprehensive intelligent system for SQL injection vulnerability detection. Three primary parts are formed and concluded: the *Data Phase*, the *Model Phase*, and the *Detector Phase*.

them will make predictions on it. The outcomes will be collected and used for later procedures like "training weighting models" or "making final comprehensive detection". The test set will be applied similarly, but it will only be utilized to generate predictions instead of training and optimization.

Pt. 3: Detector Phase. In the *Detector Phase*, a novel method is provided to offer insights about how to merge multiple model outcomes by weighting and combining, to form an overall comprehensive detection report. By feeding the detection model outcomes to the second-level weights learning models, the models will provide weight indications for the predictions that come from various models. There are two models prepared for this part: *XGBoost* and *Linear Regression*, and an empiricism-based weight selection method (see subsection 4.4 for more details) as the baseline. All

these weights learning models will also be trained and validated under the same setting of K-fold cross-validation to have sufficient generalization assurance.

Subsequently, those weights learned will be combined with the predictions made by detection models for any arbitrary SQL semantics inputs to form an overall comprehensive detection report. By checking the outcome distribution, the final comprehensive risk probability can be dropped into any one of the four levels: if the risk probability is greater than 0.9, we consider the corresponding query a "*Vulnerability*"; if the number output is less than 0.9 and greater than 0.9, we label it as "*High Risk*"; if the number output is larger than 0.5 and less than 0.7, we label it as "*Medium Risk*"; otherwise, the queries left will be collected in the category of "*Low Risk*." The comprehensive detection report will be output finally. The label designs offer a more practical risk level indication for each SQL semantics input.

3 EXISTING WORKS

According to the literature review, there are many detection methods and tools developed for detecting or preserving SQL injection attacks. Some brief overviews of these researches are provided below that focus on two principal types: the first is some methods of *static analysis-based detection* and then some *machine learning-based detection* approaches.

3.1 About Statics Analysis Detection Methods

Notable research *A Static Analysis Framework For Detecting SQL Injection Vulnerabilities* by Fu et al [X. Fu and Tao 2007] developed a static analysis framework, SAFELI, designed to identify security holes in ASP.NET Web applications that permit SQL Injection Attacks (SIAs). SAFELI operates by studying SQL query submission points through symbolic execution and by looking at the MSIL bytecode of these programs. It makes use of a hybrid constraint solver to find user inputs that can lead to information security breaches. The tool's satisfiability decision/approximation technique for string restrictions is what makes it innovative. SAFELI makes use of static analysis and targets ASP.NET Web applications in particular. It uses a hybrid constraint solver at hotspots that submit SQL queries and uses symbolic execution to investigate MSIL bytecode. Through this method, SAFELI can identify more vulnerabilities than traditional methods. Some insights from SAFELI, like the importance of compile-time checks and symbolic execution, could be integrated into our ML approach, potentially as a pre-processing or filtering step to enhance the accuracy of our models.

Another article that attracted our interest is *An Approach for SQL Injection Vulnerability Detection* published by M. Junjin [Junjin 2009]. The researchers developed an SQLInjectionGen tool that combines static analysis, runtime detection and automatic testing to detect SQLi. The article also compares the effectiveness of SQLInjectionGen with another static analysis tool FindBugs. FindBugs, as a well-known detection tool, often only issues warnings for SQl statements with variables rather than constant values. By comparison, it was found that SQLInjectionGen only had a few false negatives. Therefore, SQLUnitGen is particularly helpful for finding and resolving SQLIA vulnerabilities in legacy code as well as new code that does not take advantage of newer, more secure technologies. This method is more focused on automatically generating test cases and analyzing them in conjunction with static analysis tools, where this method can be considered in combination with machine learning models in future work.

3.2 About Machine Learning Detection Methods

The paper *SQL Injection Detection Using Machine Learning Techniques and Multiple Data Sources* by Kevin Ross [Ross 2018], discusses a novel approach to detect SQL injection attacks using machine learning. Traditional defences came from the mainstream types of static, signature-based intrusion detection system (IDS) rules, which fail against new, or "zero-day", attacks. The paper focuses on

utilizing machine learning techniques to identify unknown attacks. They collect traffic from two points: the web application host and a Datiphy appliance node located between the web app host and the associated MySQL database server. Analysis of these datasets, along with a correlated dataset, showed that the accuracy obtained using rule-based and decision-tree algorithms is nearly the same as with a neural network algorithm, but with greatly improved performance. The authors managed to combine the speed and efficiency characteristic of signature detection techniques with the potentially increased accuracy and ability to detect new attacks of machine learning techniques. The paper suggests that this multi-source, machine-learning approach could be a significant step forward in countering SQL injection attacks, combining the speed of signature-based detection with the adaptability of machine learning.

The other paper *SQL Injection Detection using Machine Learning* published by Anamika Joshi and Geetha V [Joshi and Geetha 2014] also provides insights about combining machine learning predictions with other incoming data. They delve into the problem of SQL Injection Attacks (SQLIA) and the proposed solutions. SQLIA is a popular type of SQL injection attack by simply manipulating SQL queries through unrestricted parameters for user input, in order to alter the intended query's logic. Researchers first explain SQLIA types, including tautology attacks, union-based attacks, and blind SQL injections. Existing solutions like *AMNESIA* and *SQLrand* are reviewed, pointing out their limitations. A classifier that uses a combination of the Naive Bayes machine learning algorithm and Role-Based Access Control mechanism is raised for taking SQLIA detection. Role-Based Access Control is a method of regulating access to computer or network resources based on the roles of individual users within an organization. This model was tested on three types of SQLIA: comments, union, and tautology.

4 METHODOLOGY

In this section, we delve into the specifics of our approach to detecting SQL injection vulnerabilities using comprehensive ML methods. The subsections listed in the methodology correspond to the three principal phases displayed in our architecture diagram: the *Data Phase*, the *Model Phase*, and the *Detector Phase*. All these arrangements ensure the robustness and reliability of our findings of the intelligent system and provide a formal strategy for handling the SQL injection issues to the utmost extent.

4.1 Data Availability

Data Usage. To enhance the data variety and model generalization, the datasets formulated for SQL Injection Vulnerability learning are collected from *GitHub* op en resources (see references):

- Dataset of *burp*;
- Dataset of *fuzzdb*;
- Dataset of *misc*;
- Dataset of *OWASP*;
- Dataset of *safe*; and
- Dataset of *blindTest* (*only for *Ultimate Blind Test*)

with 8790 samples in total. Includes 4424 positive cases (cases with *High Injection Risk*) and 5351 negative cases (cases with *Low Injection Risk*). The overall dataset is considered "relatively balanced" with a positive-negative ratio of 1:1.21.

Each line in the data files ending with "*–sqli*" indicates the feature of one instance related to SQL injection Vulnerability. Correspondingly, each line in the data files ending with "*–labels*" is identified by a unique label of 0 or 1, which stands for *High Injection Risk* and *Low Injection Risk*, respectively. A formal instruction about data usage can be found in the *README.md* file

under the program directory. The program supports more datasets to be added. Training with more high-quality data enables the models to perform more effectively but with higher computational costs and over-fitting risk.

Normalization Phase. The tokenization step contributes to converting each SQL semantics into individual tokens and removing useless space around the tokens. The tokenizer *word_tokenize* from the popular NLP dependency *nltk* is applied. To keep more indicative features from the coding language, no “stemming" or “lemmatization" is set to formalize the token format. For example, the word “Methods" is kept instead of the normalized edition of “method".

Additionally, the scaling option may help standardize the raw data by pruning or reorganizing extreme values relying on the whole data distribution. Using the *StandardScaler* method from *sklearn* for faster and more reasonable processing. This is an optional step, since it may be expensive to scale unseen data based on the features acquired from a whole training set. To process the scaling in predictions, users should guarantee that the pre-models can visit the training data during the prediction to have a scaling strategy with the same data operation distribution.

Vectorization Phase. Vectorization is essential for converting string expressions into numerical vector representations in Natural and Machine Language Processing. Although some models (e.g., Naive Bayes) work well in processing token-based data, most popular ML classification models are designed for vector-based numerical inputs to perform best. In this way, we implement the vectorization technique of *Doc2Vec* to process the token embedding, which is an improved edition of the popular model *Word2Vec* by rebalancing the distinctions caused by the amounts of tokens in a given document (SQL semantics in this task).

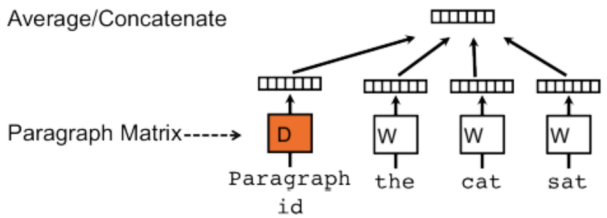

Fig. 3. The figure from [Shperber 2023] illustrates how the sequential text-based *Doc2Vec* model works in taking vectorization.

Some other approaches can also be used to reach the same purpose of vectorization. To reach the best model performance, a few methods are discussed and examined. The most classic method first being considered is the *Bag of Words* (BoW) method. With the independent assumption of in-document tokens, BoW treated each document as a “bag of words" disregarding its grammar and acquiring corresponding numerical expressions by counting the occurrence frequency. Some existing research about vulnerability detection chose the BoW model as their embedding strategy, they “[c]onvert each PHP test sample file into a 98-dimensional vector through the bag-of-words model as the final training data" [Hu et al. 2020] and got good training performance. However, the BoW model significantly suffers more from *Unknown Words* (UNK) issues, and it is more costly to save and inquire (especially when we want to deploy in a dynamic web-based environment).

The method of *Word2Vec* may also be used, but it has some limitations due to the unbalanced length of input semantics. Applying a second-level filter or taking information retrieval techniques such as “*tf-idf*" may contribute to re-balance the influence but is time-consuming. Besides, because the *Word2Vec* model does not care about the relationship between in-document elements, it may

not provide any valuable sequential information during the data embedding. These reasons push us to abandon it and move to *Doc2Vec*.

By contrast, the approach of *Doc2Vec* inherits the advantages of *Word2Vec* and provides document-based vectorization solutions for semantics with various sizes. It is more robust in dealing with UNK issues and maintaining the between-token sequential dependencies (see *Fig. 1*) facilitating the model learning process. By building the *Word2Vec* model once and saving it locally, the same vectorization strategy will be used for both the training and prediction datasets, for similar data distribution and presentation. This mechanism also accelerates the detection process by avoiding storing or reprocessing the training dataset.

Reorganization Phase. Reorganization of the input dataset will be taken after the pre-processing steps. The data splitting ratio is "9:1", which means 90% of the existing data are used for model training and validation (more details see *subsection 4.3 - Cross-validation Method*) and the 10% left is considered as the "unseen" test set. The process will also reorganize the data structure to facilitate the model learning processes. The splitting phase is processed after a *random data shuffle*, which allows *random sampling* and guarantees the data variety. This method improves model robustness and generalization but may lead to unequal outcomes for each execution time.

4.2 Model Construction

Six machine-learning models are constructed for the first-level detection task. This section provides some thoughts about their mathematical logic, hyperparameter preparations, model preferences and construction procedures.

Logistic Regression. Logistic regression is a straightforward and interpretable classification method, commonly used to model the relationship between input features and the probability of binary outcomes.

The algorithm design allows a logistic regression classifier to learn the weights $w_i \in W$ for some input training features $x_i \in X$, as shown in the equation

$$z = w_1x_1 + w_2x_2 + \ldots + w_mx_m + b = w^Tx + b \tag{1}$$

When unseen test data $\bar{x}_i \in \bar{X}$ applied, these weights learned are linearly combined with the test data to obtain a probability z, which can be used in the *Sigmoid* function

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2}$$

to map the z value to a prediction probability $\sigma(z)$ between 0 and 1. This probability represents the likelihood of the input belonging to a certain category.

The logistic regression classifier is established over the packed model in *sklearn*, with a hyperparameter C for the inverse of regularization strength. More miniature C value leads to more muscular regularization by putting more penalties on parameter norms, and vice versa. Regularization is commonly used to prevent overfitting issues and improve model generalization.

Naive Bayes (NBs). Recall and compare to the classic Bayesian equation of

$$p(c|d) = \frac{P(d|c)P(c)}{P(d)} \tag{3}$$

model NBs is treated as a simplified edition of the Bayesian equation that serves as a generative model for

$$p(c|d) = p(d|c)p(c) \tag{4}$$

where the posterior $p(c|d)$ presents the probability that document d (i.e., a string of SQL semantics) appears in class c, the likelihood $p(d|c)$ stands for the chance that document d appears in class c, and the prior probability $p(c)$ indicates the probability that the class c appears in all classes C.

$$p(c|d) = P(x_1, x_2, ..., x_n|c)P(c) \tag{5}$$

The expression of a document d can also be expanded to written in feature tokens of $x_i \in d$ to fit the consideration of "tokenization" and "Word2Vec transferring". Thanks to the "naive" assumption of independence made between each in-class element, the prior probability of predictor $p(x)$ can be dropped to simplify the prediction process. The module built-in *sklearn* is utilized with a hyperparameter α selected for model fine-tuning, which determines the "Laplace smoothing extent" (aka., "add-k" smoothing) of the model.

k-nearest Neighbors (k-NN). k-NN is another widely used spatial classification method. It operates by comparing the distances between an unknown sample $\bar{x}$ and all other learned samples $x_i \in X$ in the training dataset to determine the class of an unidentified instance by uncovering the types of the k closest data points (see *Fig. 2*). This model is edited and launched on the $KNeighborsClassifier$ approach provided by *sklearn*.

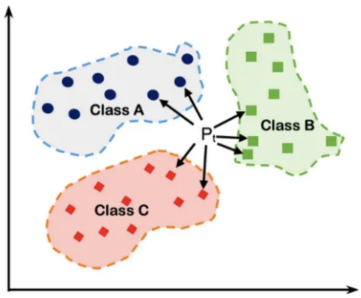

Fig. 4. [Soo 2023]'s work delivers some insights about how a new sample being classified by the kNN model.

The hyperparameter k specifies the number of neighbours $x_1, ..., x_k \in X$ to be assessed in *Euclidean* distance measuring by

$$d(x, y) = \sqrt{\sum_{i=1}^{t} (x_i - y_i)^2} \tag{6}$$

where $x_i \in x$ and $y_i \in y$ implicit the distance between two data points in a t-dimensional environment. The k nearest samples with the closest distance to the unknown type sample are selected based on a voting rule that follows the majority. Based on various choices of k values, the k-NN classifier performs unequally in detecting SQL injection vulnerability. Normally, a diminutive k leads to a higher over-fitting probability and an oversized k may consider more noisy samples in learning. Taking careful fine-tuning over the hyperparameter of k-NN helps avoid extreme cases.

Support Vector Machine (SVM). SVM is considered one of "the most powerful out-of-the-box supervised machine learning algorithms" [Fan 2018] that map samples' features to points in a vector space. It learned to construct a decision boundary of hyperplane (see *Fig. 2*) that contributes to differentiating points into distinct categories. The main goal of SVM is to find the optimal hyperplane $w^T x + b = 0$ that separates different data categories to "maximize the margin" of $\frac{2}{||W||}$. This hyperplane is chosen in a way that maximizes its gap to the nearest vector data point (i.e.,

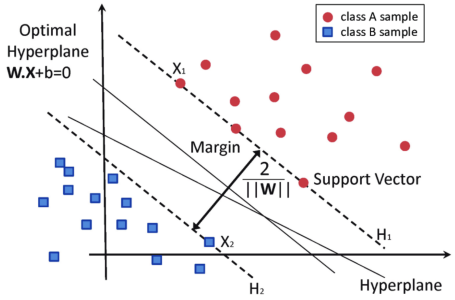

Fig. 5. The figure taken from [Esperanza et al. 2016] illustrates the data distribution and decision boundary build by SVM.

support vectors) for better generalization consideration. This decision boundary can be easily applied to new feature points, which caused the SVM classifier to operate efficiently in high dimensional conditions and be effective in non-linear classification tasks. The decision function $\bar{y}$ can be written as

$$\bar{y} = \sum_{i=1}^{N} \alpha_i y_i \kappa(x_i, \bar{x}) + b = 0 \tag{7}$$

which is used to make predictions for an unseen instance $\bar{x}$. $\kappa(x_i, \bar{x})$ is a *kernel function* which is used to compute the similarity between the unseen instance $\bar{x}$ and each of the support vectors $x_i \in x$ for $x \in \mathbb{R}^N$. Label information for each element of x_i is provided by $y_i \in y$ for $y \in \mathbb{R}^N$. b stands for the bias term and $\alpha_i \in \alpha$ indicates the *Lagrange multipliers* which can be viewed as a weighting parameter learned from the training that shares the same dimension with the training dataset. In this model, the *Gaussian Kernel*, also known as the *Radial Basis Function* (RBF), is selected as the *Kernel Function* for the SVM classifier:

$$\kappa(x_i, x_j) = exp\left(-\frac{(x_i - x_j)^2}{2\sigma^2}\right) \tag{8}$$

where the parameter σ controls the shape of the decision boundary in SVM, which can be designed for specific tasks. To implement an SVM classifier for this vulnerability classification task, *svm.SVC* function from *sklearn* is utilized. The process has a regularization term of C, which is the hyperparameter to control the intensity. Similar to what we explained for Logistic Regression, an advancing C value leads to more powerless regularization and vice versa.

(Optional) This paragraph provides more details about SVM training and optimization. Because the gradient-based optimization over high-dimensional space may be problematic, the *Duality* concept is applied in the training stage of SVM to transform the *Primal Problem* (i.e., a problem that may be challenging to solve directly) into an equivalent or approximate *Dual Problem* (i.e., a problem with a much simpler solution), in the form of

$$\min_{\alpha} \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i y_i \alpha_j y_j \kappa(x_i, x_j) - \sum_{i=1}^{N} \alpha_i \tag{9}$$

where it associated with each data point, and y_i and y_j indicate their corresponding category labels. Recall the goal of SVM training is to find the vector x of Lagrange multipliers that minimizes the function above. Kernel Function $\kappa(x_i, x_j)$ helps map the input vectors x_i and x_j to a higher dimension

to calculate the similarity between each pair of training instances. All these strategies facilitate the classification process by providing a clearer decision boundary in the higher dimensional space.

Decision Tree (DT). DT models are highly conditional and represent decision rules in a tree structure that can assign instances to different categories or predict outcomes (an example see *Fig. 3*). Although it may not perform as strongly as the SVM model, one of the most significant

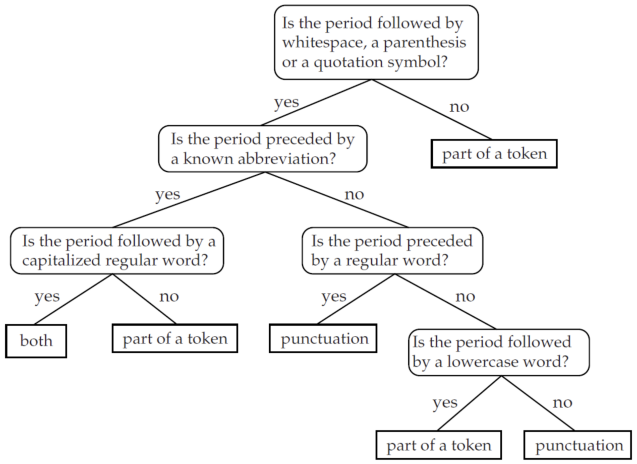

Fig. 6. [Soo 2023] A sample decision tree model developed for natural language processing. We construct a similar structure but for the numerical expressions of the machine languages.

advantages of the DT model is that it provides much more readable and interpretable solutions. However, sometimes it may also suffer from overfitting, where the model is trained on data with too complicated features. To maintain the model generalization, we control the hyperparameter of "tree depth" in this project by fine-tuning it until a reasonable convergence rate is found.

Gradient Boost Decision Tree (GBDT). GBDT is the last model we installed for injection detection which is a sort of *Boosting algorithm* that combines multiple "weak learners" to establish a "strong learner". GBDT is a complicated composite model developed based on decision trees, it learned to reduce prediction errors and improve prediction accuracy gradually by taking

$$F_{m+1}(x_i) = F_m(x_i) + h_m(x_i) = \bar{y}_i \tag{10}$$

where $m \in M$ indicates the learning round (i.e., the index of the new model being added), F_m and F_{m+1} presents the overall GDBT tree model before and after the update, and $h_m(x_i)$ presents the new "weak learner" model being "boosted" to GBDT, which is also trained on data samples $x_i \in x$. The forecast of this equation is known as $\bar{y}_i \in \bar{y}$ which provides a corresponding prediction for each feature element of x_i. The predictions will also be applied to compute the gradient of the loss function to provide insights for update in the next round $m + 1$. Commonly in each training round, a new tree model is built over the existing models. This "boosting" aims to reduce the loss seen from the earlier validation round by training new models on the loss gradient direction with the residuals of the predicted values and true values of the original model. The ensemble model *GradientBoostingClassifier* is taken from *sklearn*. Two hyperparameters related are designed for tuning: *tree depth* for fitting extent control and n for the number of "weak learners".

4.3 Learning Design

This section provides guidance and some methodological overviews for a few special designs in the model learning process.

Evaluation Benchmarks. The evaluation strategy of this research project will be around the *confusion matrix* concept. The confusion matrix is a table that is used to describe the performance of a classification model on a set of data for which the ground truth values are known. There are a few confusion matrix-related metrics reported, they are

- **Confusion Matrix Information**: includes True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN);
- **Balanced Accuracy**: an accuracy metric for binary classification problems, useful for imbalanced datasets;
- **Micro F1-Score**: calculates globally by counting the total TP, FN, and FP; and
- **Macro F1-Score**: calculates for each class independently and then takes the average;

Besides, we also report the minimizing “test loss" and the optimal “hyperparameters" selected according to the indications from “Macro F1-Score", which is treated as the core evaluation metric. For the consideration that an “unbalanced dataset" may be used, we selected these “balanced benchmarks" of “F1-Scores" and “balanced accuracy" to provide some more reasonable evaluations. The information listed in the confusion matrix is also reported so we can process the analysis in detail by browsing some notable sample cases. These measures provide insights for us to fine-tune the comprehensive detector for optimization.

Hyperparameter Fine-tuning. A systematic hyperparameter fine-tuning strategy is implemented, which permits the optimal hyperparameter to be found, reported and applied directly.

As discussed in the *section 4.2*, the performance of each machine learning model deployed may be controlled by one or more “hyperparameters" by attempting various choices of hyperparameters and examining the relevant model performance, the optimal hyperparameter selections for each model can be determined. *Micro F1-score* is the key metric for deciding the best hyperparameter. The “hyperparameter scope" is designed by empiricism, and the selection is processed by taking loop examinations, similar to the idea of *Grid Search*. All other environment settings will be the same for variable control until the optimal hyperparameter is found and reported. The model hyperparameter tuning stands for

- **Logistic Regression** - C, for Regularization Intense;
- **Naive Bayes** - α, for Laplace Smoothing Intense;
- **k-nearest Neighbors** - k, for Considered Neighbor Number;
- **Support Vector Machine** - C, for Regularization Intense;
- **Decision Tree** - *depth*, for Rule Fitting Extent;
- **Gradient Boost Decision Tree** - *depth*, for Rule Fitting Extent, and
- **Gradient Boost Decision Tree** - n, for Weak Learner Number;

To guarantee the generalization, the fine-tuning is processed under the *cross-validation* setting (see *subsection 4.3 - Cross-validation Method* for more details). That means the program will compute the *Micro F1-score* for each sample model in every round of the *cross-validation* and take the *Mean Micro F1-score* over round performances for each hyperparameter choice. After fine-tuning, a formal model with the optimal hyperparameter is trained and saved locally as the pre-trained models.

Cross-validation Method. The strategy of *K-fold Cross-validation* is applied to the training stage of the models we mentioned before. “*Cross-validation*" is a practical statistical method of cutting data samples into smaller subsets to evaluate the predictive performance of a model. The essential

thought of "*K-fold*" is to divide the input training dataset into K subsets, and taking different part for validation in each round $k_i \in K$: For each round k_i, treat the union of $K-1$ subsets as the "training set", and the one left as the "test set" (see *Fig. 5* below). Taking a "random shuffle" before folding helps improve model robustness to reduce the probability of over-fitting but may result in unequal outcomes for each time of processing.

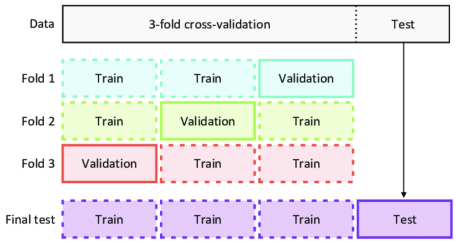

Fig. 7. This diagram produced by [Duran-Lopez et al. 2020] indicates how 3-Fold Cross-validation works. Our project follows the same setting of data allocation.

Due to the limited data set we have, we are not sure whether a single evaluation of the model is stable, so we need to divide the sample data into K groups for multiple cross-validations. The value of $K = 10$ is very common in the field of machine learning. Based on existing experience, this value is neither affected by high bias nor high variance. During our training, we tried the performance of K from 3 to 10. Our research found that except for a few models that were less affected by the K value, most models are optimized when $K = 3$ (comprehensive accuracy and time cost).

Model Saving and Reloading. The "Saving" and "Reloading" mechanisms are prepared for the *Doc2Vec* vectorization model and all first-level detection models in this task. This measure helps maintain the stability of model performance and avoid unnecessary additional training and consumption. For the vectorization dictionary model, we only retrain it once no existing dictionary is found. This design guarantees all detection models can access the same preprocessed data inputs and allows fast detection to be completed under dynamic environmental demands.

The first-level machine learning models will be updated when executing the development program *classifier.py* (p.s., to redo the whole training and saving process) or the "*rebuild*" option inside the interactive program *detector.py* (p.s., to rebuild a specific model). The "rebuilding" process will also obey the existing vectorization model — if any exists — to make sure that diverse classifiers are compared with the same input data distributions. This approach does not work for the second-level weighting models since only the "weight outcomes" are needed, which have been deployed to the comprehensive detector. Model reusing significantly reduces the time taken for executing SQL injection detection and lowers the time and computation power consumption.

4.4 Detection Strategy

In the *Detection Phase*, the "*scan*" (i.e., scan a line input) and "*file*" (i.e., scan a file input) options of *detector. py* is used to process a multi-classifier detection, and the "*detect*" option may process an analysis based on a specific classifier for comparisons. For each vector input of a SQL semantics x, the detector is expected to contact and collect all model predictions to form an all-around detection report. A formal prediction collection gathered for input from all six models can be $\bar{y} = \{\bar{y}_1 = 0, \bar{y}_2 = 0, \bar{y}_3 = 0, \bar{y}_4 = 1, \bar{y}_5 = 1, \bar{y}_6 = 0\}$ for $\bar{y}_i \in \bar{y}$, where having whether $\bar{y}_i = 0$ or $\bar{y}_i = 1$ from each prediction that stands for "prejudged vulnerability" or "prejudged safe".

Multi-model Comprehensive Detection. To form an overall report and provide a precise vulnerability analysis, all prediction outcomes should be put together to form an all-around evaluation. Because distinct models may focus on diverse features of the given data, having multiple models and considering a comprehensive prediction can bring a much more robust model with better generalization ability in dealing with the unseen data. But, what algorithm should be employed to complete the comprehensive detection? How can we compare the outcomes and get the risk? What can we do to formalize it as a real "intelligent detection system"?

There are two main strategies made with various attempts: the *empiricism-based weights selection*, and the *learning-based weights decision* over the second-level Machine Learning *Regression Models*. All the strategies assume the base weight for each model is $w_i = 1$, and the total weights should be $\Sigma_i^n w_i = n$.

Weights Selection: Empiricism-based Mean Determination. For the first choice of the *empiricism-based weights selection*, the weights for models are determined by the knowledge gained from previous research or examinations. Among that, "mean addition" is a choice of the empiricism-based selection, which brings the same weight $w_i \in w$ for every model where

$$\bar{y}^*_{Emp} = \frac{\Sigma_i^n \bar{y}_i}{n} \tag{11}$$

by adding the prediction $\bar{y}_i \in \bar{y}$ of each model together and then taking the mean value as the comprehensive prediction $\bar{y}^*_{Emp}$. This naive strategy provides each model with an "equal voice" in making the final detection. Still, it may suffer more from the outlier predictions made by extremely weak predictors, and more nominal highlighting the attitude of the supermodels.

Weights Learning: The 2^{nd} Level Machine Learning Model. To overcome the existing issues and make the comprehensive detection design more powerful, we have more advanced learning-based weighting methods deployed. Because we expect the problem to have continuous outcome expressions which drop into specific "Risk Levels", two regression models were selected for the weighting task: the *Linear Regression* model that comes with a quick answer, and the *XGBoost* model that is similar to the GBDT described before, as a more advanced "implementation of gradient-boosting decision trees." [Simplilearn 2023].

For the first weights learning model of Linear Regression, we simply establish a polynomial to fit the rules and reduce the gaps between the predictions from the detection learning models and the ground-truth labels. To learn a weight $w_i \in w$ for each model outcome $\bar{y}_i \in \bar{y}$, the learning goal can be formalized as

$$\bar{y}^*_{LR} = w_0 + w_1\bar{y}_1 + w_2\bar{y}_2 + \ldots + w_n\bar{y}_n \tag{12}$$

where w_0 indicates the potential bias in learning. The multiplications at the RHS should lead to a comprehensive prediction that minimizes the gap between the sum of the weighted parts and the golden category given. The model is also established based on a pre-designed model taken from *sklearn* and fine-tuned under a 3-fold cross-validation environment.

The XGBoost model unpacked from module *XGBoost* serves as the second weights learning model. XGBoost also follows the strategy of *Gradient Boost* by adding more "weaker learners" h_m to improve the performance of the major body to form an ultimate "strong learner" F_n (see section 4.2). Under such a design, six "weak learners" will be established to adapt to six detection predictions in one round respectively. In simple terms, the weights $w_i \in w$ for each "weak learner" h_m are needed, such as

$$F_{i+1}(\bar{y}_i) = F_i(\bar{y}_i) + w_i h_i(\bar{y}_i) = \bar{y}_{XGB} \tag{13}$$

where the ultimate outcome after the whole process of “gradient boost" indicates

$$\bar{y}^*_{XGB} = F_n(\bar{y}_i) \tag{14}$$

Comparing the simple model of Linear Regression, XGBoost is much more robust in learning the rules of regression problems. Similar to what we did for the first model, the XGBoost model is trained and turned with the same cross-validation settings.

The Intelligent System. All the designs and preparations illustrated before contribute to framing the problem and converting the classification detector to an intelligent system with a novel method. After gaining outcomes from each “first-level ML detection model" and acquiring the relevant weights from any “second-level weights decision approach", we finally get the capability to solve the SQL injection vulnerability detection task to the utmost extent. By collecting and weighing the “prejudged predictions", the final comprehensive detection report is formed.

Distinct from any single detection result, although the performance of the intelligence system may not always be as influential as an individual SVM or a deep neural network model, the all-around detection provides much better robustness and generalization. The ensemble intelligence system learns to consider the insights from each detection model built-in to focus much more on learning features that cannot be provided by any single classifier. The comprehensive model provides more more muscular analysis ability in dealing with unseen data, where it learns to enhance the advantages and diminish the drawbacks of multiple models by fine-tuning and carefully validating each level of the detection system. Besides, the intelligence system offers much more anti-overfitting ability that will not easily fall into a dilemma of “over-learning" on a few specific features.

5 OUTCOME ANALYSIS

In this section, some evaluations and comparisons are provided for the detection results from both the *Single Detectors* and the *Comprehensive Detectors*. Some indepth thoughts about optimization and cause analysis will also be discussed.

5.1 Performance and Optimization of Single Detectors (SDs)

Hyperparameter Optimization. When selecting the best hyperparameters for each model, our primary consideration is the F1-macro score. F1-macro is a commonly used metric for evaluating classification models that balance precision and recall across all classes. By optimizing the metric of *F1-macro*, we aim to achieve good overall performance across the categories in the datasets. However, we also take into account other indicators such as *loss*, *F1-micro*, and *balanced accuracy*. These additional metrics provide us with a more comprehensive understanding of the model’s performance. To assist in this process, we utilize *hyperparameter vs. performance change* diagrams (see Figure 8). These diagrams allow us to visualize how the performance of the model changes concerning different hyperparameter settings. By analyzing these diagrams, we can identify the optimal hyperparameter values of

- **Logistic Regression** - C=100;
- **Naive Bayes** - α=50;
- **k-nearest Neighbors** - k=3;
- **Support Vector Machine** - C=50;
- **Decision Tree** - *depth*=10;
- **Gradient Boost Decision Tree** - *depth*=5, and
- **Gradient Boost Decision Tree** - n=30;

that maximize our desired metrics and optimize the performance of every individual ML-based injection detector.

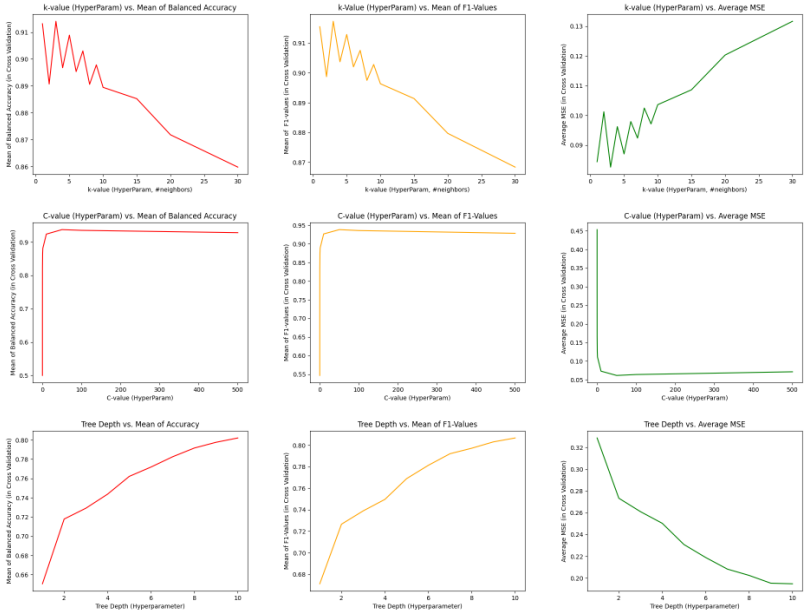

Fig. 8. This diagram indicates the performance changes of models (k-NN, SVM, and DT) depending on various selections of the hyperparameters (*k*, *C*, and *depth*), respectively.

Metric	LR	NB	kNN	SVM	DT	GBDT
TP	340	241	376	406	333	386
FP	47	122	28	19	80	30
TN	502	427	521	530	469	519
FN	91	190	55	25	98	45
Accuracy	0.8592	0.6816	0.9153	0.9551	0.8184	0.9235
F1_micro	0.8592	0.6816	0.9153	0.9551	0.8184	0.9235
F1_macro	0.8552	0.6697	0.9134	0.9544	0.8148	0.9220
Test Loss	0.3923	2.4759	1.0288	0.1474	1.9484	0.2143

Table 1. Performance comparison of individual machine learning detection models based on multiple metrics, the optimal hyperparameter choices are also reported.

Performance Comparison: Test Set Examination. In this subsection, we first examine the outcomes from every single detection model constructed where the results can be found in Table 1. This table provides a comparative evaluation of six machine learning models based on the "unseen test dataset": Logistic Regression (LR), Naive Bayes (NB), k-nearest Neighbors (kNN), Support Vector Machines (SVM), Decision Trees (DT), and Gradient Boosted Decision Trees (GBDT). These models are assessed on various performance indicators, including True Positives (TP), False Positives (FP), True Negatives (TN), False Negatives (FN), Balanced Accuracy, micro and macro F1-scores, and Test Loss.

Upon analysis, NB performed the worst on all counts. It had the lowest TP and TN of 241 and 427 respectively. The false part also saw the highest FP and FN scores of 122 and 190. The poorest accuracy (0.6816) and F1 scores (0.6816) suggest the highest error and the worst performance in

comparison. On the contrary, the single detector of SVM exhibits superior performance on all metrics. It correctly identified the most positive and negative instances, as reflected in the highest TP and TN scores of 409 and 530, respectively. It made the fewest incorrect predictions, indicated by the lowest FP (19) and FN (25) scores. Furthermore, it achieved the highest balanced accuracy of 0.9551 and a micro F1-score of 0.9551, implying excellent overall performance and a perfect balance between precision and recall. Besides, its lowest test loss (0.1474) also indicates that the SVM model works the best in all six classic detection models we examined.

Due to its greater performance in injection prediction, the SVM is also selected as the first single detector (SD) which will be used as a baseline to have a comparison with our comprehensive detectors (CDs) in the following examinations.

5.2 Performance and Optimization of Comprehensive Detectors (CDs)

Weighting Optimization. As we discussed in the detector strategy, there are three strategies used to determine the weights for model performance merging: Empiricism Selection (EMP), Linear Regression Learning (LR), and XGBoost Learning (XGB). Some goals of weight optimization include:

- **1. Reduce overly sensitive predictions.** Especially for cases in the category 'VULNERABILITY OR HIGH RISK.' We found most powerful models are over-sensitive and often overestimate the risk level of some safe samples;
- **2. Lower the computational cost.** We don't want the comprehensive weighted model to be too expensive to use in both time and computational power;
- **3. Keeping an output category of "MEDIUM".** Due to the comprehensive design, it is too aggressive to map all merging outcomes to the discrete cases of 0 and 1. Furthermore, the "MEDIUM" zone allows users to have a perspective recognition of the potential risks and take more tests;
- **4. Lower the sample number that dropped in the category "MEDIUM".** Even though the design may contribute to providing more effective indications, it may lower the system's accuracy in injection detection. Samples as low as possible are expected to fall into the "MEDIUM" category.

For the CD with empiricism weight selection, we manually define the weights for each detection model outcome as $[1, 1, 1, 1, 1, 1]$, which follows the approach of *Mean Determination* we discussed in subsection 4.4 that every detection model is equally considered and averaged. By contrast, the weights acquired from the second-level learning models are significantly distinct. The *Linear Regression* model learns the weights to be $[-0.038, -0.124, 2.870, 1.719, 0.252, 1.321]$ which even put some *negative weights* on models with more inadequate performance to calibrate the overall comprehensive prediction. The weighting model of *XGBoost* also makes an effort to enhance the overall accurateness by enlarging the weights for the "best agent" contained. The strategy results in a weight list of $[0.500, 0.300, 1.000, 1.900, 1.000, 1.300]$ where the most powerful single model of SVM got the largest weight of 1.900 in all sixes.

Performance Comparison: Ultimate Blind Test. For the "Ultimate Blind Test", we use the unseen dataset of *blindTest.txt* from online resources. It is called "blind" since we do not even know if the dataset shares the same data distribution with our learning dataset. By contrast, the test and validation datasets used before are "unseen", but still share similar data distribution with the training data. Refer to the outcomes listed in Table 2 for three rounds, there are three *comprehensive detectors* (CDs) combined with various weights determination strategies of EMP, LR and XGB respectively, and two baseline *single detectors* (SDs) of SVM (i.e., the most powerful individual model detector we constructed earlier) and DNN (i.e., a *Deep Neural Network* model trained under the same settings

Risk Level (Label)	Ground Truth	SD of SVM	SD of DNN	CD with EMP	CD with LR	CD with XGB
VULNERABILITY OR HIGH RISK (~1)	121 (56%)	146 (68%) 148 (69%) 151 (70%)	147 (68%) 141 (65%) 143 (66%)	121 (56%) 125 (58%) 128 (59%)	138 (64%) 136 (63%) 131 (61%)	131 (61%) 132 (61%) 133 (62%)
MEDIUM RISK (0.5)	-	-	-	45 (21%) 43 (20%) 38 (18%)	2 (.9%) 7 (3%) 7 (3%)	13 (6%) 22 (10%) 12 (6%)
LOW RISK (~0)	95 (44%)	70 (32%) 68 (32%) 65 (30%)	69 (32%) 75 (35%) 73 (34%)	50 (23%) 48 (22%) 50 (23%)	76 (35%) 73 (34%) 78 (36%)	72 (33%) 62 (29%) 71 (33%)
TOTAL	216	216	216	216	216	216
Weight(s)	-	1	1	[1, 1, 1, 1, 1, 1]	[-0.038, -0.124, 2.870, 1.719, 0.252, 1.321]	[0.500, 0.300, 1.000, 1.900, 1.000, 1.300]

Table 2. Performance comparison in three rounds of two single detectors (SD-SVM and SD-DNN) and three comprehensive detectors with weights decided by Empiricism (CD-EMP), Linear Regression (CD-LR), and XGBoost (CD-XGB) in the "Ultimate Blind Test."

with 200 hidden nodes in layers) are also prepared for the performance comparison. The trends of prediction outcomes are visualized in Figure 9.

The performance evaluation is conducted based on the distribution of risk levels —— HIGH (VULNERABILITY), MEDIUM, and LOW —— that each method predicts, and these are compared to the actual ground truth.

In the high-risk category, the ground truth indicates that 56% of the total cases fall under this category. Both SD models (SD-SVM and SD-DNN) significantly overestimate the number of high-risk instances, with percentages ranging from 65% to 70%. This condition implies that SD models are too sensitive with numerous “false alarms". On the other hand, the CD models provide estimates closer to the ground truth, with CD-EMP even matching the ground truth in the first round. Their percentages range from 56% to 64%, suggesting the strategy of CD models is better calibrated for high-risk prediction.

In the medium-risk category, the SD models do not provide any estimates with a lack of sensitivity to cases that fall between high and low-risk levels. This implies that CDs offer a more nuanced risk

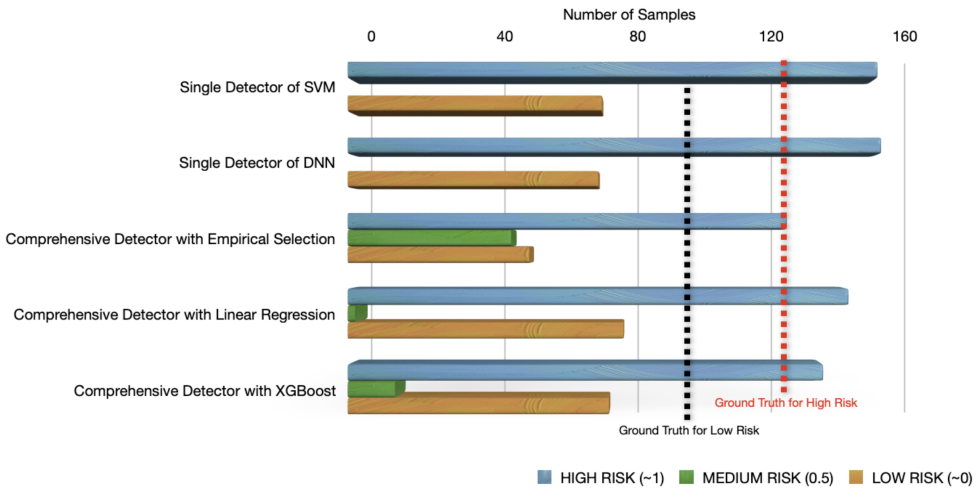

Fig. 9. Performance comparison of two single detector and three comprehensive detectors in the "Ultimate Blind Test".

assessment by acknowledging a “MEDIUM" risk level, which the SD omits. Among the CD models, CD-EMP predicts the highest percentages (18% to 21%), significantly higher than the CD-LR and CD-XGB models, which predict less than 3% and 6% to 10% respectively. Recall our optimization expectations, we expect to “lower the sample number that dropped in the category MEDIUM". Although CD-EMP has perfect high-risk detection, it has too many cases that fall into the “gray area", which means the CD-EMP has poorer performance in detailed detection compared to CDs weighted by model learning.

For low-risk predictions, the ground truth indicates 44% of the cases. Both SD models underestimate this category, predicting only 30% to 35% of low-risk cases. This could potentially lead to a high number of miåssed detections. CD models also underestimate, but the CD-LR and CD-XGB provide closer estimates of the *ground truth* (29% to 36%). Overall, the CD models, particularly CD-EMP, CD-LR, and CD-XGB, align better with the ground truth across all risk categories. This suggests that these models might be more reliable and accurate in risk prediction tasks.

The overall test outcome from the “blind test" also indicates that the comprehensive strategy brings stronger overfitting resistance concerning complicated single models. Beyond the on-table metrics, the advanced model SD-DNN consumed more than 5 minutes to train, but even the slowest composition model GBDT in our CDs only spent about 40 seconds in preparation. The acceleration performance will be much more significant as larger datasets are used in training.

6 CONCLUSION

Intending to improve the detection quality and control the costs of SQL injection detection based on ML methods, increasing the complexity and training depth of a single model is not the only way. This research proposes a novel strategy based on a comprehensive analysis and performance merging of multiple simple models to reach an affordable and efficient prediction. The novel method forms an intelligent detection system that takes secondary screening and aggregation of multiple recognition results from the double-layer weighting model to provide an all-around accurate report, which includes three primary phases of data processing, model construction and

detector optimization. Besides, the experiment revealed that the simple comprehensive models delivered the same or even better recognition precision and overfitting resistance compared to some powerful single-detector models of SVM and DNN.

Furthermore, it is also noteworthy to acknowledge potential future challenges and improvements. Although this project proves that the comprehensive model can exceed some single advanced models, it can do better by updating some composition models to more powerful choices. For instance, by replacing the poorest detection model of “Naive Bayes", the overall comprehensive performance will be boosted considerably. Another phase that can be improved is to test more variants of the weighting models. We only have two weighting models examined in this task but more research could be done to provide valuable indications about the weighting of the comprehensive model. Additionally, applying more advanced NLP techniques to pre-process the input of SQL semantics can also be considered an enhancement direction. Some developed relation extraction methods could be used to replace the simple sequential method of *Doc2Vec*. Further studies may also attempt to remove *Stopwords* (i.e., Words with extremely high frequency) by filtering unnecessary information to reinforce the weights of the sample essence. Comprehensive analysis of the intelligent system or bringing a new development direction to ML-based SQL injection detection.

REFERENCES

Lourdes Duran-Lopez, Juan Pedro Dominguez-Morales, Antonio Conde-Martin, Saturnino Vicente Díaz, and Alejandro Linares-Barranco. 2020. PROMETEO: A CNN-based computer-aided diagnosis system for WSI prostate cancer detection. *IEEE Access* PP (07 2020), 1–1. https://doi.org/10.1109/ACCESS.2020.3008868

García-Gonzalo Esperanza, Fernández-Muñiz Zulima, Garcia Nieto Paulino Jose, Sánchez Antonio, and Menéndez Marta. 2016. Hard-Rock Stability Analysis for Span Design in Entry-Type Excavations with Learning Classifiers. *Materials* 9 (06 2016), 531. https://doi.org/10.3390/ma9070531

Shuzhan Fan. 2018. Understanding the mathematics behind Support Vector Machines — shuzhanfan.github.io. https://shuzhanfan.github.io/2018/05/understanding-mathematics-behind-support-vector-machines/. [Accessed 30-11-2023].

Jianwei Hu, Wei Zhao, Zheng Yan, and Rui Zhang. 2020. Analysis and Implementation of SQL Injection Vulnerability Mining Technology Based on Machine Learning — netinfo-security.org. http://netinfo-security.org/EN/10.3969/j.issn.1671-1122.2019.11.005.

Anamika Joshi and V Geetha. 2014. SQL Injection Detection using Machine Learning. In *International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*. IEEE.

M. Junjin. 2009. An Approach for SQL Injection Vulnerability Detection. *2009 Sixth International Conference on Information Technology: New Generations, Las Vegas, NV, USA,* (2009). https://doi.org/doi:10.1109/ITNG.2009.34.

OWASP. 2021. OWASP Top Ten | OWASP Foundation — owasp.org. https://owasp.org/www-project-top-ten/.

Kevin Ross. 2018. *SQL Injection Detection Using Machine Learning Techniques and Multiple Data Sources.* Master’s thesis. San Jose State University. https://doi.org/10.31979/etd.zknb-4z36

Xhuvani Aleksander Shehu, Bojken. 2014. A Literature Review and Comparative Analyses on SQL Injection: Vulnerabilities, Attacks and their Prevention and Detection Techniques - ProQuest — proquest.com. https://www.proquest.com/openview/d4d7a48db7ba4002b3e773ff03125828/1?pq-origsite=gscholar&cbl=55228.

Gidi Shperber. 2023. A gentle introduction to Doc2Vec — medium.com. https://medium.com/wisio/a-gentle-introduction-to-doc2vec-db3e8c0cce5e.

Simplilearn. 2023. What is XGBoost? An Introduction to XGBoost Algorithm in Machine Learning | Simplilearn — simplilearn.com. https://www.simplilearn.com/what-is-xgboost-algorithm-in-machine-learning-article.

Go Min Soo. 2023. Deep Learning Bible - H. Traditional NLP - Eng. DeepLearningBible-H.TraditionalNLP-íĮœêÿĂ. [Accessed 30-11-2023].

Xiaofeng Li Minggang Shao Yajie Mi Hongli Zhang Guoqing Zhi Wei Zhang, Yueqin Li. 2022. Deep Neural Network-Based SQL Injection Detection Method — hindawi.com. https://www.hindawi.com/journals/scn/2022/4836289/.

B. Peltsverger S. Chen K. Qian X. Fu, X. Lu and L. Tao. 2007. A Static Analysis Framework For Detecting SQL Injection Vulnerabilities. *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)* (2007). https://doi.org/10.1109/COMPSAC.2007.43