

Paper Type: Original Article

Employing Fusion of Neural Networks and Machine Learning Models in SQL Injection Attack Detections

Mohamed Elkholy^{1,*}  and **Marwa M. Abd El Hamid²** ¹ Faculty of Computer Science & Engineering, Alamein International University; melkholy@aiu.edu.eg.² Faculty of Computers and Information Technology, Innovation University, 10th of Ramadan City, Egypt; Marwa.Mostafa@iu.edu.eg.**Received:** 30 Aug 2025**Revised:** 17 Sep 2025**Accepted:** 23 Dec 2025**Published:** 25 Dec 2025**Abstract**

SQL injection attacks are among the most serious threats to web application security. Their main purpose is to have unauthorized access to sensitive information by exploiting related vulnerabilities. This paper proposes a multi-model approach in the detection of SQLIA by leveraging several machine learning algorithms, including Convolutional Neural Networks, Logistic Regression, Random Forest, Support Vector Machines, Naive Bayes, and Decision Trees. The focus of this work is to utilize the strengths of these models in analyzing the structure and semantics of SQL queries to distinguish between malicious and benign inputs. The paper outlines a comprehensive methodology, including data preprocessing, feature extraction, model training, and performance evaluation. Using two datasets: the first dataset containing 3951 SQL queries and the second containing 33,726 queries. The accuracy results from the small dataset were 96% for CNN, and 96.7% for Naive Bayes. On the large dataset, accuracy increased to 97% for CNN, and 98% for Naive Bayes. A fusion of Logistic Regression, Random Forest, and SVM, which resulted in 98% accuracy on the small dataset. These findings suggest that deep learning techniques, especially CNNs, along with other machine learning models and fusion techniques, hold much promise for enhancing web application security by providing a robust and adaptive solution to mitigate the risks posed by SQL injection attacks.

Keywords: SQLIA (SQL Injection Attack); SQLi (SQL Injection); Deep Learning; Convolutional Neural Networks; Web Application Security; Cybersecurity.

1 | Introduction

The most dangerous and persistent threat to the security of web applications is represented by an SQL injection attack. More precisely, SQLIAs are among the major causes of the huge leak of data, leading to enormous financial losses [1]. This exploits the vulnerabilities in input validation mechanisms of a web application to inject malicious SQL code into the user inputs, which can manipulate the back-end database [2]. As shown in Figure 1, obviously, if executed successfully, SQLIA may have strong consequences in terms of data leakage and unauthorized access, which could escalate to a full compromise of a database with further financial fraud [3].

Traditional rule-based detection methods, while wide in application, typically fail to effectively mitigate SQLIAs due to their limitation to manual features and failure to generalize against new patterns of attacks [4]. Such methods face a challenge in keeping pace with the ever-changing tactics that attackers use, since they

**Corresponding Author:** melkholy@aiu.edu.eg<https://doi.org/10.61356/j.scin.2025.2633>

Licensee **SciNexuses**. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0>).

always obfuscate their malicious code to evade detection [5]. The increased complexity of web applications and their connectivity drive the need for more robust, adaptable mechanisms of detection against SQLIA.

Machine learning has lately offered a promising approach to overcoming such limitations of traditional methods. In this line, several machine learning algorithm-based SQLIA detection models, able to adapt to new variations of attacks by automatically learning patterns and anomalies in data, have been proposed. Nonetheless, challenges related to feature engineering, model selection, and availability of high-quality training data persist.

Deep learning, a subset of ML, has been found to be especially promising in detecting SQLIA, for the simple reason that it can learn complex data representations from raw data [6]. Deep learning models, such as Convolutional Neural Networks and Recurrent Neural Networks, have been applied to fairly a respectable number of cybersecurity tasks, for instance, malware detection and intrusion detection, with high success rates [7]. In the case of SQLIAs, deep learning models would not require feature engineering since relevant features from SQL queries can be learned automatically. This may potentially improve the accuracy of detection [8].

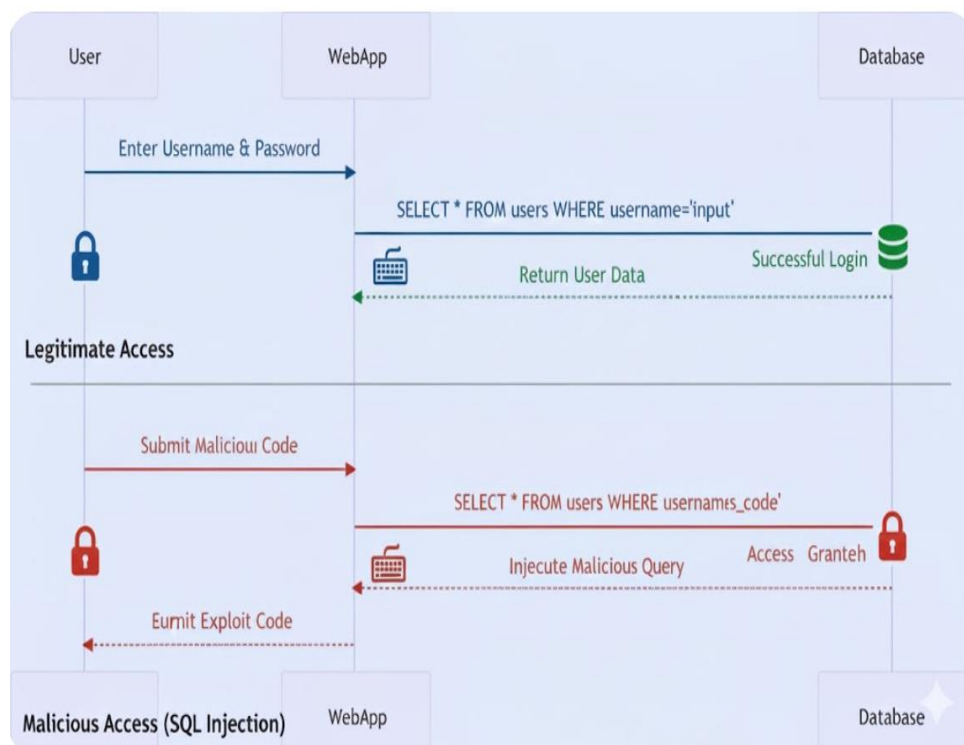


Figure 1. SQL Injection Attack.

An approach is proposed to detect SQLIA based on a CNN model, Logistic Regression, Random Forest, Support Vector Machines (SVM), Naive Bayes, and Decision Trees. By making use of some such inherent capabilities of models in self-features extraction from raw data, our approach averts one common limitation found in most traditional machine learning approaches: manual feature engineering. This work contributes to the security of web applications by providing a robust solution that is effective and accurate for model-based detection of SQLIAs, hence better protection of sensitive data and systems.

2 | Related Work

SQL injection attacks have been among the most enduring and evolving web application security threats, which have resulted in profoundly serious data breaches and associated financial losses [9]. SQLIA exploits the vulnerabilities of input validation mechanisms of any given web application. This is utilized by hackers intending to inject malicious SQL code into user inputs for manipulating the backend database [10]. This can

be very severe in impact when executed successfully, with results such as leakage of data, full database compromise, unauthorized access, and financial fraud [11].

Traditional rule-based detection is applied very widely, but its effectiveness in mitigating SQLIAs is disappointing because it depends on features defined manually and poor generalizability to new patterns of attacks [12]. Such methods are hard to keep pace with the continuous change in attackers' tactics, obfuscating their malicious code to elude detection [13]. In the light of increasingly complex, interconnected web applications, more robust, adaptable detection mechanisms for SQLIA become paramount.

The deficiencies of these traditional approaches recently proposed machine learning as a solution. In this regard, several models of SQLIA detection have been developed by exploiting the power of machine learning algorithms on automatic learning of patterns and anomalies in data and hence adapt to new variations of attacks. However, challenges remain in feature engineering, model selection, and availability of high-quality training data.

Deep learning, a subdomain of ML, has been found to be effective in detecting SQLIA because it easily learns complex representations from raw data. Deep learning models, such as Convolutional Neural Networks (CNN), Logistic Regression, Random Forest, Support Vector Machines (SVM), Naive Bayes, and Decision Trees and Recurrent Neural Networks, have been applied to several cyber-security tasks, like malware detection and intrusion detection with satisfactory results.

Deep learning models can automatically extract relevant features from SQL queries in the context of SQLIAs, thus eschewing the manual feature engineering effort and detecting with enhanced accuracy. A few research studies have been conducted on the use of deep learning techniques for the detection of SQLIA. Luo et al. suggested a CNN- based approach to extract payloads from network flows that turned out to be effective in a real-traffic case study.

Even with these advances, challenges still exist toward the development of deep learning models about the effective generalization for new and unseen patterns of SQLIA [14-16]. The success of such models is strongly tied to diverse and representative training data. Deep models can be computationally expensive to train and deploy, prohibitive for resource-constrained environments.

3 | Methodology

3.1 | Data Collection

In this research, we utilize two distinct datasets to train and test the models: the first containing 3,951 SQL queries and the second containing 33,726 queries. These datasets are curated into two categories: malicious (SQL injection attacks) and benign (normal queries). The sources for the dataset compilation include:

- Publicly available repositories: A wide range of SQL injection payloads and benign queries are available on platforms like GitHub and Kaggle, which serve as reliable sources for training and testing machine learning models.
- SQL injection tool log files: Attack patterns and techniques relevant to real-life attacks can be gathered from automated SQL injection tools like SQL Map, providing additional diversity to the dataset.
- Manual queries: Synthetic SQL injection queries, created by security experts, are used to ensure a comprehensive dataset that captures various attack variations.

These sources have been combined to generate a diverse dataset, which include several types of SQL injection techniques, such as in-band, error-based, blind SQL injections, as well as different obfuscation and evasion methods.

3.2 | Data Sanitization

Data sanitization is a critical step to ensure the quality and consistency of the dataset. The following procedures are applied:

- Duplicate removal: Duplicate SQL queries are identified and removed to prevent overfitting to overrepresented samples.
- Missing value handling: Any incomplete or missing data points are addressed by replacing or filling them with appropriate values.
- Label Verification: A subset of queries is manually checked to ensure the accuracy and reliability of the labels.
- Normalization: All queries are converted to lowercase, extra whitespace and comments are removed, and special characters are handled to standardize the SQL format.

3.3 | Data Normalization

To prepare the SQL queries for input into the CNN model, the data undergoes the following normalization steps:

- Tokenization: Each SQL query is broken down into individual tokens, which could represent words, characters, or subworlds depending on the chosen tokenization technique.
- Encoding: Each token is assigned a unique numerical identifier, creating a vocabulary of integers that represent the entire dataset.
- Padding/Truncation: All tokenized queries are standardized to the same length. Short queries are padded with zeros, and longer queries are truncated to maintain consistency across the dataset.
- Embedding: After encoding the SQL queries into integers, these integer-encoded queries are transformed into dense vector representations using word embedding techniques. Word embeddings capture the semantic relationships between the tokens (e.g., words or characters), allowing the model to understand the contextual meaning of SQL queries beyond the individual tokens. This step is crucial for enhancing the model's ability to detect patterns and anomalies in SQL queries that are indicative of SQL injection attacks.

4 | Experiment

4.1 | Dataset

The dataset used for this work consists of two parts: the small dataset, containing 3,951 SQL queries, and the large dataset, containing 33,726 SQL queries. These queries are classified as either malicious (SQL injection attacks) or benign (normal queries). The dataset was curated using various sources, including public repositories like GitHub and Kaggle, log files from SQL injection tools such as SQL Map, and manually crafted queries by security experts. It includes all types of SQL injection attacks, such as in-band, error-based, blind SQL injections, obfuscation, and evasion techniques.

The dataset was balanced to ensure an equal number of malicious and benign queries, which is crucial for training a robust and unbiased model for classification. The dataset was split into two parts: one for training the models and the other for evaluation.

- Training Set: For the small dataset: 80% of the dataset (3,161 queries) was used for training. For the large dataset: 80% of the dataset (26,981 queries) was used for training.

- Testing set: For the small dataset: The remaining 20% (790 queries) was used for testing. For the large dataset: The remaining 20% (6,745 queries) was used for testing.

4.2 | Data preprocessing

A robust pipeline of preprocessing steps was applied to prepare the collected datasets before being fed into the models, including.

- Cleaning: This step involved removing duplicate queries, handling missing values, and verifying a subset of queries for their labels to ensure accuracy and reliability.
- Tokenization: Each SQL query was tokenized into words or characters, depending on the chosen tokenization strategy. This process is crucial for converting the raw text data into a form that can be processed by machine learning models.
- Encoding: Each token was assigned a unique numerical identifier to create a vocabulary of integers representing the entire dataset. This encoding step is essential for transforming text data into numerical input for the models.
- Padding/truncation: Shorter queries were padded with zeros, while longer queries were truncated to the same fixed maximum length. This ensures that all tokenized queries are the same length, promoting consistency in input data and enabling efficient batching during training.
- Embedding: The integer-encoded queries were transformed into word embeddings, which are dense vector representations. These embeddings help all the models (CNN, Logistic Regression, Random Forest, SVM, Naive Bayes, and Decision Trees) understand the semantic relationships among words in the SQL queries. The word embedding enables the models to learn more meaningful representations of the data, which in turn enhances the accuracy and effectiveness of the classification tasks across the different algorithms.

4.3 | Building CNN Model

The architecture for the models used in SQLIA detection was designed to successfully extract relevant features from the preprocessed SQL query inputs. Each model uses a unique approach:

- CNN (Convolutional Neural Network): The CNN model leverages convolutional layers to automatically detect hierarchical patterns and features in SQL queries, helping to identify malicious and benign queries based on their structural and semantic characteristics.
- Logistic Regression: This model uses a linear approach to classify SQL queries based on the features extracted from the query data. It applies a logistic function to model the probability of a query being malicious or benign.
- Random Forest: A robust ensemble model that constructs multiple decision trees to classify SQL queries. Each tree is trained on a random subset of features, and the final classification is based on a majority vote among the trees.
- Support Vector Machines (SVM): The SVM model maps the queries into a higher-dimensional space and finds the optimal hyperplane to separate malicious and benign queries. This model is particularly effective when dealing with complex and non-linear decision boundaries.
- Naive Bayes: A probabilistic classifier that assumes the features of SQL queries are independent. It uses Bayes' theorem to predict the probability of a query being malicious or benign based on the feature distribution.

- **Decision Trees:** This model splits the data into branches based on feature values, constructing a tree-like structure to make decisions. It classifies SQL queries by traversing the tree and making decisions at each node.

Each model was trained and tested using the data sets mentioned above, and their performance was evaluated based on accuracy, precision, recall, and F1-score.

- **Input Layer (Sequences):** It all starts with an input layer containing sequences of words or characters which are to be treated as raw SQL queries. Such queries may be of variable lengths, thus mirroring the variety of SQL statements encountered in real-world scenarios.
- **Embedding Layer (5000, 256):** This layer acts as a translator, changing every word or character into a dense, numerical representation known as a word embedding. The word embedding permits capturing semantic relationships between the words so that the model can utterly understand the meaning and context of the input queries. The vocabulary size is five thousand, so it can include a maximum number of unique words or characters.
- **Conv1D Layers (32 filters, kernel size=1):** In this model, two Conv1D layers are used, each with 32 filters and a kernel size of 1. These layers utilize filters to slide over the embedded input sequence to extract local patterns and features that might indicate the presence of SQL injection attacks. The first layer has thirty-two filters, and the second one also has thirty-two filters, allowing the model to capture both fine-grained and more general features. A kernel size of one means that each filter processes one word or character at a time, enabling the model to extract precise local information.
- **Batch Normalization:** After each Conv1D layer, Batch Normalization layers have been added. These layers will normalize the activations of the filters of the convolution so that they do not get too large or too small, hence making the learning smoother.

Dropout: 0.5 after every Batch Normalization, dropout layers were embedded at a rate of 50%. These layers shut off a fraction of neurons randomly during training and therefore help prevent overfitting. This makes the model learn more robust and generalizable representations.

- **GlobalMaxPooling1D:** After convolutional layers extract the relevant features, it gets aggregated by the GlobalMaxPooling1D layer. This will take the maximum value over each feature map, condensing the spatial dimensions while keeping only the most salient features.
- **Dense Layer (64 units):** The output from the GlobalMaxPooling1D layer serves as input into a Dense layer, which acts as a fully connected layer. It simply refines even more the already aggregated features, learning complex relationships between them. The number of units in this layer is sixty-four, with each computing the weighted sum of its inputs, linked to by an activation.
- **Output Layer (Sigmoid):** Each model used in this experiment includes an output layer that produces a probability score for each input query. In models such as Logistic Regression, Random Forest, Naive Bayes, SVM, and Decision Trees, the output score represents the likelihood of the query being a SQL injection attack. If the probability exceeds a predefined threshold, the query is classified as malicious; otherwise, it is considered benign.

Each model works by analyzing the structure and content of SQL queries, extracting features to distinguish between benign and malicious queries. While the CNN model focuses on local patterns through convolutional layers, other models, such as Logistic Regression, SVM, and Naive Bayes, rely on traditional machine learning techniques to perform classification based on statistical features of the queries. Random Forest and Decision Trees use decision-making trees and ensemble learning to improve accuracy and robustness in classifying attacks.

As shown in Figure 2, the models were implemented using the Keras deep learning library and TensorFlow as the backend for the CNN model. For other machine learning models, standard libraries like scikit-learn were used. The training process involved using the Adam optimizer for CNN, with binary cross-entropy as the loss function, while other models followed their respective algorithms for optimization.

5 | Results and Discussions

5.1 | Evaluation Metrics

Several evaluation metrics were used to assess the effectiveness of the six models employed in detecting SQL injection attacks. These metrics provide a detailed understanding of each model's performance in classifying SQL queries as either malicious or benign. The evaluation metrics include:

- **Accuracy:** The percentage of correctly classified cases, calculated as the sum of true positives and true negatives divided by the total number of cases. Accuracy was measured for all six models: CNN, Logistic Regression, Random Forest, SVM, Naive Bayes, and Decision Trees.
- **Precision:** The ratio of true positive predictions to the total positive predictions. This metric reflects how well each model identifies SQL injection attacks without producing false positives.
- **Sensitivity/Recall:** The ratio of true positive predictions to the total actual positive cases. It indicates the ability of each model to detect malicious SQL injection attacks.
- **F1-score:** The weighted average of precision and recall, providing a balanced measure of performance for each of the six models.

These metrics are calculated using the following formulas:

$$\text{Accuracy} = (\text{TP} + \text{TN}) / (\text{TP} + \text{TN} + \text{FP} + \text{FN}) \quad (1)$$

$$\text{Precision} = \text{TP} / (\text{TP} + \text{FP}) \quad (2)$$

$$\text{Recall} = \text{TP} / (\text{TP} + \text{FN}) \quad (3)$$

$$\text{F1-score} = 2 * (\text{Precision} * \text{Recall}) / (\text{Precision} + \text{Recall}) \quad (4)$$

Where TP, TN, FP, and FN represent the number of true positives, true negatives, false positives, and false negatives, respectively.

5.2 | Experimental Results

Then the proposed models (CNN, Logistic Regression, Random Forest, SVM, Naive Bayes, and Decision Trees) were trained on 80% of each dataset and evaluated on the remaining 20%. Specifically:

For the small dataset (3,951 queries):

- Training was conducted on 3,161 queries (80%).
- Evaluation was performed on 790 queries (20%).

For the large dataset (33,726 queries):

- Training was conducted on 26,981 queries (80%).
- Evaluation was performed on 6,745 queries (20%).

Based on the provided table, here is the revised description:

As shown in Table 1, the models demonstrated varying levels of performance across the evaluation metrics. The CNN model achieved the highest F1-score of 94%, highlighting its ability to effectively balance precision

and recall. It also demonstrated high accuracy (96.5%) and precision (99.5%), indicating strong capability in correctly identifying both malicious and benign SQL queries.

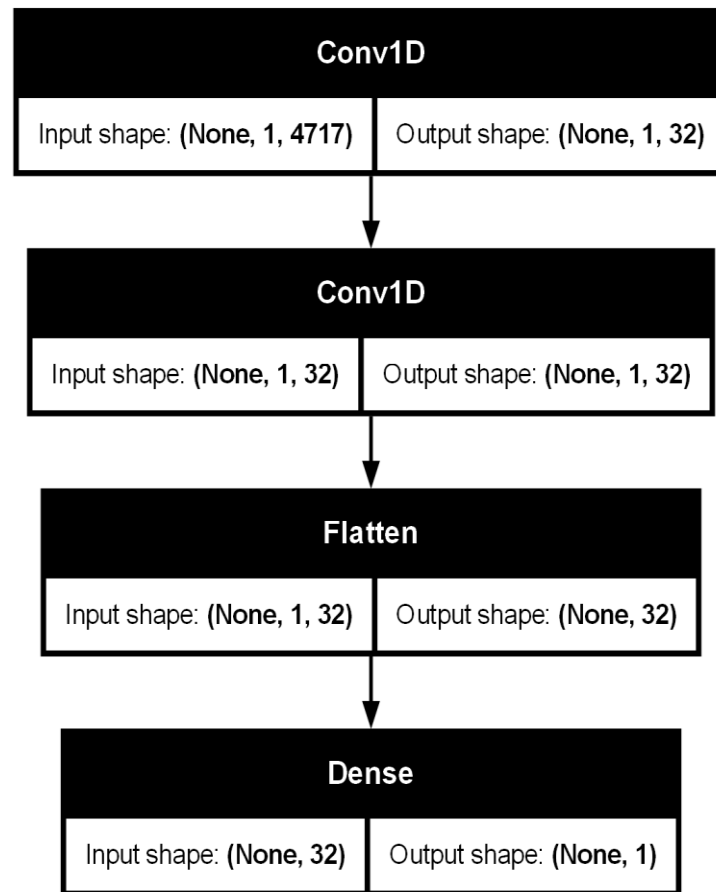


Figure 2. Building models.

Table 1. Results of models (small dataset).

Model	Accuracy	Precision	Recall	F1-score
CNN	96.5%	99.5%	89.5%	94%
Logistic Regression	93.9%	77.7%	100%	87.4%
Random Forest	88.9%	99.5%	71%	83%
SVM	78%	19.6%	100%	32.8%
Naive Bayes	96.7%	99%	90%	94.9%
Decision Tree	86%	99.5%	66.2%	79.5%

Other models also performed well in certain aspects:

- Naive Bayes had the second-highest F1-score (94.9%) and strong accuracy (96.7%) and precision (99%), making it another robust model for this task.
- Logistic Regression showed high recall (100%) but comparatively lower precision (77.7%), leading to an F1-score of 87.4%.
- Random Forest exhibited good precision (99.5%) but a lower recall (71%), resulting in an F1-score of 83%.
- Decision Tree showed moderate performance, with an accuracy of 86% and an F1-score of 79.5%.

- SVM, however, had the lowest F1-score (32.8%) due to its significantly low precision (19.6%), despite achieving perfect recall (100%).

Overall, the CNN model demonstrated superior performance among all the tested models in detecting SQL injection attacks.

Table 2. Results of Models (large dataset).

Model	Accuracy	Precision	Recall	F1-score
CNN	97.2%	99.3%	90.5%	94.8%
Logistic Regression	92.6%	71.8%	98.7%	83.1%
Random Forest	90.2%	99.5%	72.3%	83.7%
SVM	79.7%	20%	100%	33.5%
Naive Bayes	97.9%	99.5%	92.9%	96.1%
Decision Tree	84.4%	99.5%	61.9%	76.3%

As shown in Table 2, the models exhibited varying levels of performance across the evaluation metrics. The CNN model achieved the highest F1-score of 94.8%, highlighting its strong ability to balance precision and recall. It also demonstrated outstanding accuracy (97.2%) and precision (99.3%), highlighting its reliability in accurately classifying both malicious and benign SQL queries.

Other models performed as follows:

- Naive Bayes had the second-highest F1-score (96.1%) and excellent accuracy (97.9%) and precision (99.5%), making it a robust alternative.
- Logistic Regression showed the highest recall (98.7%) but comparatively lower precision (71.8%), resulting in an F1-score of 83.1%.
- Random Forest achieved good precision (99.5%) but lower recall (72.3%), leading to an F1-score of 83.7%.
- Decision Tree demonstrated moderate performance with an accuracy of 84.4% and an F1-score of 76.3%.
- SVM, however, had the lowest F1-score (33.5%) due to its significantly low precision (20%), despite achieving perfect recall (100%).

Overall, the CNN model demonstrated superior performance among all tested models in detecting SQL injection attacks.

5.3 | Accuracy Curve

Figure 3 illustrates the running accuracy of a machine learning model while training across many epochs. The blue line, which is the training accuracy, shows the precision of the model on the data it sees during training. It rises with a gentle slope, while the model learns and hence improves its predictions on the training data. The orange line shows the validation accuracy, which refers to how accurate the model is doing on data that did not contribute to its training. It rises, too, though not as smoothly and at a more reasonable rate compared to that of training accuracy, indicating the good generalization of the model to unseen data, with fluctuations in performance visibly captured. Overall, the model is faring well in learning, with very promising generalization.

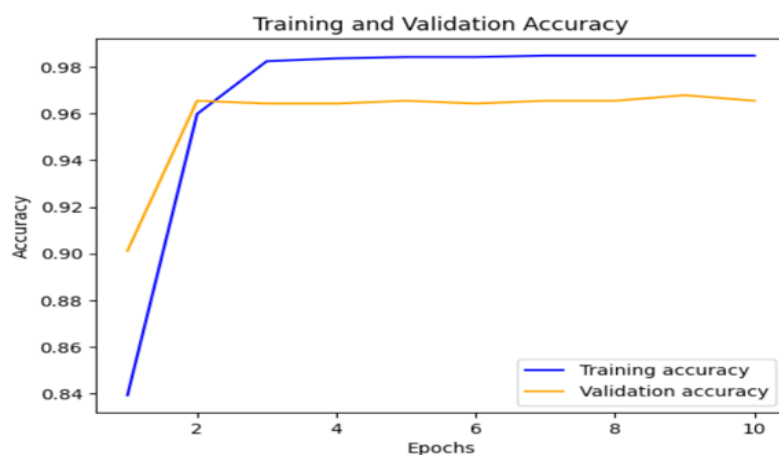


Figure 3. Training and Validating Accuracy in large data.

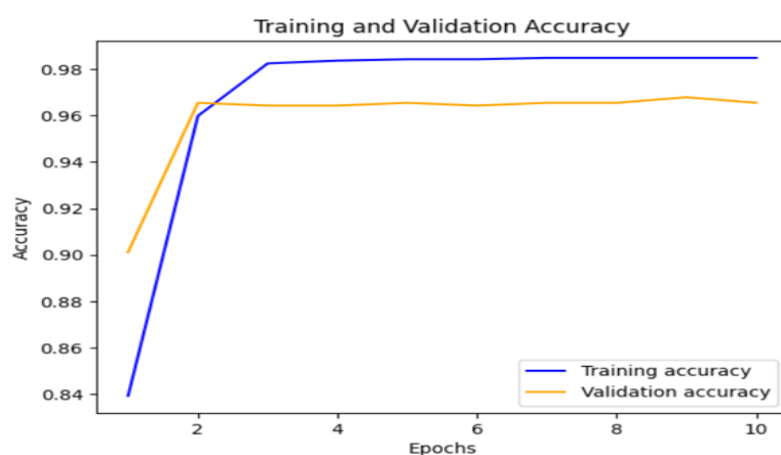


Figure 4. Training and Validating Accuracy in small data.

5.3.1 | Loss Curve

Figure 4 shows the graph of the training and validation loss while running a machine learning model for four epochs. The training loss is a measure of the error in the model on the data it learns from; it dropped rapidly in the first few epochs before it started to get stabilized. Validation loss, which estimates the model error on new data, also decreased-but more slowly, with slight fluctuations. That means it is learning very well; it generalizes with new data, with a small tendency to overfit. That is when, in the last epoch, this validation loss goes flat, whereas this training loss can still go a little bit down. Figure 5, 6 illustrated Training and validation loss in data respectively, Figure 5 for large data and Figure 6 for small data.

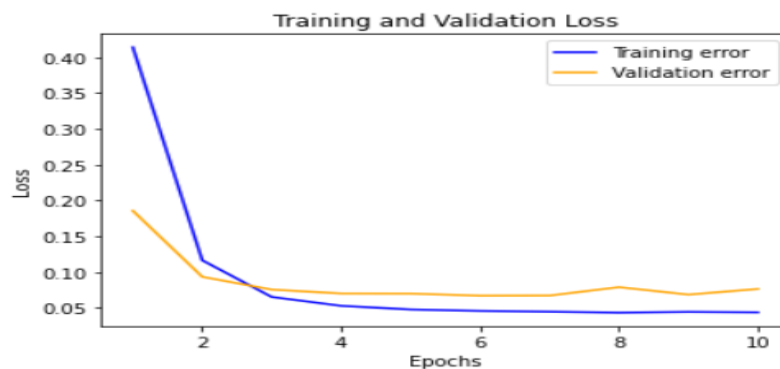


Figure 5. Training and Validating Loss in large data.

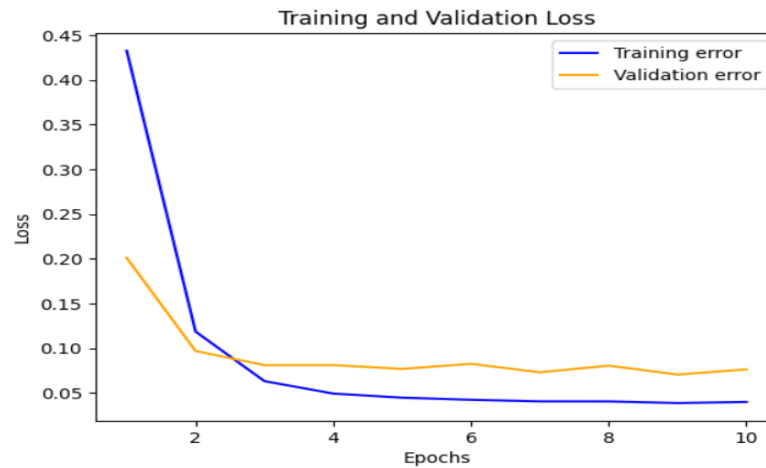


Figure 6. Training and Validating Loss in small data.

Figure 7 shows a bar chart comparing the performance of several machine learning models across four evaluation metrics: F1-Score, Precision, Recall, and Accuracy. Here is a breakdown of what is visible:

1. Bar Chart Layout:

- The chart consists of four distinct bars representing the metrics: F1-Score, Precision, Recall, and Accuracy.
- Each bar represents the performance of different models, and each model is represented by an assorted color.

2. Models:

- The models included in the chart are:
 - Logistic Regression (Blue)
 - Random Forest (Orange)
 - SVM (Green)
 - Naive Bayes (Yellow)
 - Decision Tree (Purple)
 - CNN (Brown)

3. Legend:

- The legend on the right side of the image helps to identify each model by its color.

4. Interpretation:

- The bars show the relative performance of each model on the metrics.
- For example, the CNN model (Brown) outperforms the other models in F1-Score and Precision, with high values in comparison to other models.
- Logistic Regression (Blue) has superior performance across all metrics, with notable performance in recall.
- SVM (Green) shows a lower performance in Precision but has the highest Recall.
- The Naive Bayes (Yellow) model shows a balanced performance but does not perform as well as CNN in terms of F1-Score and Precision.

- Decision Tree (Purple) performs moderately across all metrics.

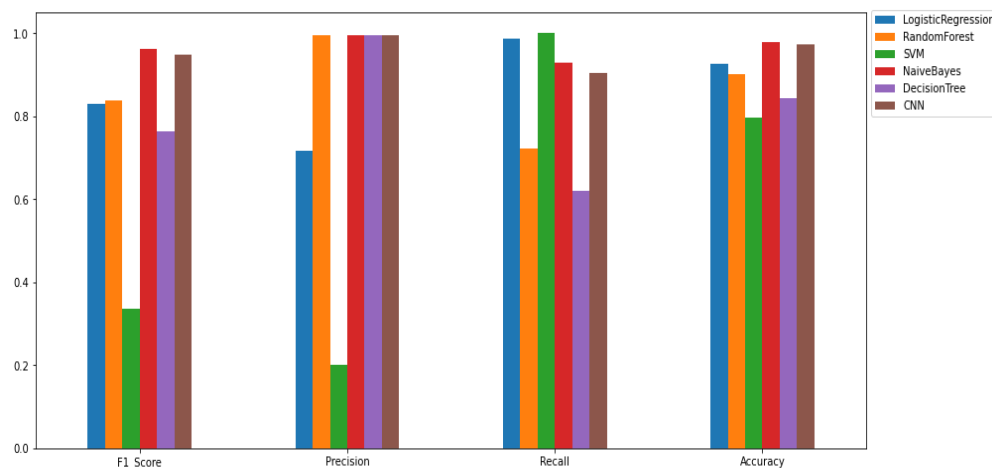


Figure 7. F1 Score.

5.3.2 | Confusion Matrix

(Small data)

The confusion matrix illustrates the performance of a binary classification model for detecting SQL injection attacks. The rows represent the actual classes, while the columns represent the predicted classes.

As shown in Figure 8, the model performs exceptionally well in recognizing benign queries, correctly identifying 583 cases, with only twenty-eight instances misclassified as malicious (false positives). Similarly, it demonstrates robust performance in detecting malicious queries, correctly identifying 228 instances while missing only one case (false negative).

Overall, the model delivers excellent performance, with a high number of correct predictions for both classes. However, there is room for further improvement through threshold tuning or leveraging additional training data.

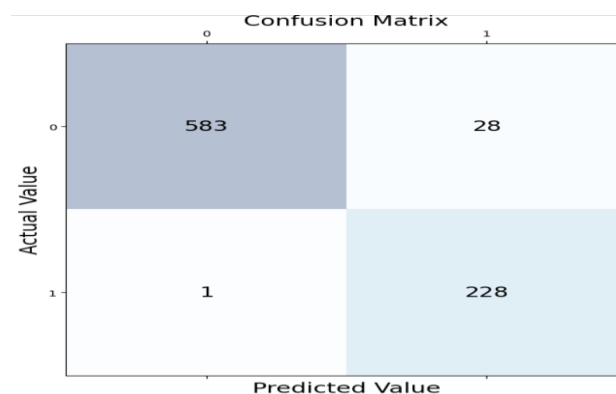


Figure 8. Confusion Matrix in small data.

(Large data)

The confusion matrix illustrates the performance of a binary classification model for detecting SQL injection attacks. The rows represent the actual classes, while the columns represent the predicted classes.

As shown in Figure 9, the model performs exceptionally well in identifying benign queries, correctly classifying 605 cases, with only twenty-two instances misclassified as malicious (false positives). Similarly, it shows robust performance in detecting malicious queries, correctly identifying 212 instances, and misclassifying only one as benign (false negative).

Overall, the model achieves outstanding results, with a high number of correct predictions for both classes. However, further fine-tuning of the threshold or additional training data could potentially enhance its performance even further.

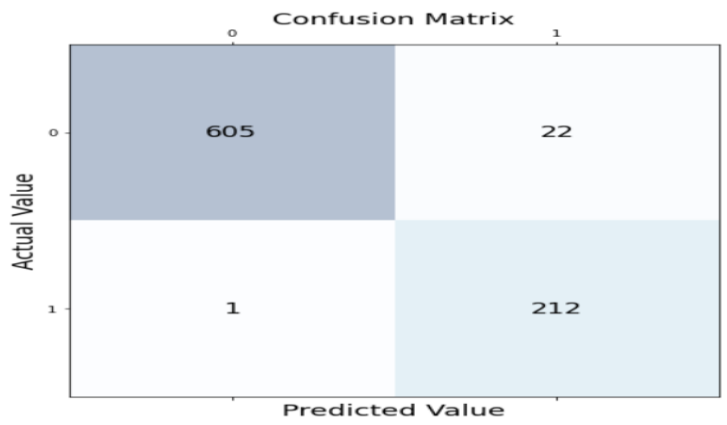


Figure 9. Confusion Matrix in large data.

5.4 | Fusion between Models

Table 3 presents the results of the fusion model (Voting Classifier) used for detecting SQL injection attacks. The model combines three different classifiers: Logistic Regression, Random Forest, and Support Vector Machine (SVM), using hard voting to make the final prediction. The evaluation metrics indicate the performance of the model across both benign and malicious queries, with an emphasis on accuracy, precision, recall, and F1-score.

Table 3. Results of the fusion model (Voting Classifier).

Metric	Class 0 (Benign Queries)	Class 1 (Malicious Queries)	Macro Avg.	Weighted Avg.
Precision	1.00	0.93	0.97	0.98
Recall	0.97	1.00	0.98	0.98
F1-Score	0.98	0.96	0.97	0.98
Accuracy				0.98

Table 3 summarizes the evaluation metrics of the fusion model (Voting Classifier). The model achieved an impressive accuracy of 98% with robust performance across both benign and malicious.

Precision and recall for benign queries were 1.00 and 0.97, respectively, while malicious queries achieved a recall of 1.00. These results suggest that the model is highly dependable in minimizing both false positives and false negatives, making it suitable for real-world applications in SQL injection detection.

5.5 Comparing Accuracy with Other Uses

Table 3 presents the results of the fusion model (Voting Classifier) used for detecting SQL injection attacks. The model combines three different classifiers: Logistic Regression, Random Forest, and Support Vector Machine (SVM), using hard voting to make the final prediction. The evaluation metrics indicate the performance of the model across both benign and malicious queries, with an emphasis on accuracy, precision, recall, and F1-score.

Table 4 summarizes the accuracy achieved by different experiments, all performed on the same dataset. Each experiment represents a different model or approach, and the accuracy values reflect the model's performance

in detecting SQL injection attacks. The table provides a clear comparison of the accuracy levels across these experiments, allowing for an evaluation of the effectiveness of each model.

Table 4. The accuracies achieved by different experiments.

Experiment	Result
SQL Injection detection using neural network[17]	97.7%
PPAI Project: Applying BERT on SQL injection data[18]	99.8%
PPAI Project: Comparison with other ML algorithms[19]	97%

6 | Conclusion

An innovative approach was approached for the detection of SQL injection attacks based on Convolutional Neural Networks. In the proposed approach, we leveraged the power of models and fusion between them in automatically learning discriminative features from raw SQL query inputs, which would obviate the need for manual feature engineering. We could train the proposed models on a huge dataset of SQL queries to vary test accuracy, thus proving their effectiveness in identifying and classifying SQLIAs. The measured precision, recall, and F1-score for both benign and malicious queries were extremely high, hence confirming the model's performance.

These results outline the potential of deep learning, especially fusion between models, in enhancing web application security with a robust, adaptive solution for SQL injection detection and prevention. Further research directions may involve deeper architectures of CNNs and fusion, additional features-for example, contextual information, HTTP headers-and investigating the effectiveness of the model in real-world environments. We would like to go further and develop a general framework that integrates our CNN-based detection system with all security measures to ensure multi-level defense against SQLIAs.

Funding

This research has no funding source.

Data Availability

The datasets used and analyzed in this study are publicly available. The primary data sources include the PIMA Indian and Sylhet datasets, supplemented with additional data from open-access resources on Kaggle. These datasets are anonymized and do not involve any privacy or ethical concerns, making them appropriate for open academic research.

Conflicts of Interest

The authors declare that there is no conflict of interest in research.

Ethical Approval

This article does not contain any studies with human participants or animals performed by any of the authors.

References

- [1] Q. Li, W. Li, J. Wang, and M. Cheng, "A SQL injection detection method based on adaptive deep forest," IEEE Access, vol. 7, pp. 145385–145394, 2019, Doi: 10.1109/ACCESS.2019.2944951.
- [2] N. Thalji, A. Raza, M. S. Islam, N. A. Samee, and M. M. Jamjoom, "AE-Net: Novel autoencoder-based deep features for SQL injection attack detection," IEEE Access, vol. 11, pp. 135507–135516, 2023, Doi: 10.1109/ACCESS.2023.3337645.

- [3] W. H. Rankothge, M. Randeniya, and V. Samaranayaka, "Identification and mitigation tool for SQL injection attacks (SQLIA)," in Proc. 15th IEEE Int. Conf. Industrial and Information Systems (ICIS), 2020, pp. 591–595, Doi: 10.1109/ICIS51140.2020.9342703.
- [4] K. Tasdemir, R. Khan, F. Siddiqui, S. Sezer, F. Kurugollu, and A. Bolat, "An investigation of machine learning algorithms for high-bandwidth SQL injection detection utilizing BlueField-3 DPU technology," in Proc. IEEE 36th Int. System-on-Chip Conf. (SOCC), 2023, pp. 1–6, Doi: 10.1109/SOCC58585.2023.10256777.
- [5] K. Ren, W. Qiang, Y. Wu, Y. Zhou, D. Zou, and H. Jin, "JSRevealer: A robust malicious JavaScript detector against obfuscation," in Proc. 53rd Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN), 2023, pp. 339–351, Doi: 10.1109/DSN58367.2023.00041.
- [6] M. Alghawazi, D. Alghazzawi, and S. Alarifi, "Deep learning architecture for detecting SQL injection attacks based on RNN autoencoder model," *Mathematics*, vol. 11, no. 15, p. 3286, 2023, Doi: 10.3390/math11153286.
- [7] D. Muduli, S. K. Panda, S. Mishra, and B. Majhi, "SIDNet: A SQL injection detection network for enhancing cybersecurity," *IEEE Access*, vol. 12, pp. 176511–176526, 2024, Doi: 10.1109/ACCESS.2024.3502293.
- [8] S. S. R. Gudivada and J. Rao, "Adaptive defense mechanisms against SQL injection: A comparative study of static and dynamic analysis," *IEEE Trans. Reliability*, vol. 72, no. 4, pp. 1102–1115, Dec. 2023, Doi: 10.1109/TR.2023.3319821.
- [9] P. Tang, W. Qiu, Z. Huang, H. Lian, and G. Liu, "Detection of SQL injection based on LSTM–MLP," in Proc. 7th Int. Conf. Reliability Systems Engineering (ICRSE), 2021, pp. 240–245, Doi: 10.1109/ICRSE52102.2021.9702220.
- [10] J. Zhang, H. Zhu, and L. Yang, "An SQL injection attack detection method based on deep belief networks," in Proc. Int. Conf. Intelligent Computing, Automation and Systems (ICICAS), 2019, pp. 488–492, Doi: 10.1109/ICICAS48597.2019.8985552.
- [11] Y. Xie, J. Feng, Y. Wu, and J. Wen, "Elastic-pooling CNN for SQL injection detection with variable-length queries," *IEEE Access*, vol. 9, pp. 11054–11065, 2021, Doi: 10.1109/ACCESS.2021.3051405.
- [12] N. M. Sheykhkanloo and A. Hall, "A learning-based neural network model for the detection and classification of SQL injection attacks," *Int. J. Cyber Warfare Terrorism*, vol. 7, no. 1, pp. 16–41, 2017, Doi: 10.4018/IJCWT.2017010102.
- [13] S. Manivannan and S. Senthilkumar, "A recurrent neural network-based intrusion detection system," *PeerJ Comput. Sci.*, vol. 9, p. e1566, 2023, Doi: 10.7717/peerj-cs.1566.
- [14] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, "Survey of intrusion detection systems: Techniques, datasets, and challenges," *Cybersecurity*, vol. 2, no. 1, pp. 1–22, 2019, Doi: 10.1186/s42400-019-0038-7.
- [15] J. Kim, J. Kim, H. L. Kim, and H. Kim, "Long short-term memory recurrent neural network classifier for intrusion detection," in Proc. Int. Conf. Platform Technology and Service (PlatCon), 2016, pp. 1–5, Doi: 10.1109/PlatCon.2016.7456805.
- [16] W. Lee, S. J. Stolfo, and K. W. Mok, "A data mining framework for building intrusion detection models," in Proc. IEEE Symp. Security and Privacy, 1999, pp. 120–132, Doi: 10.1109/SECPRI.1999.766909.
- [17] G. Apruzzese, F. Pierazzi, M. Colajanni, and M. Marchetti, "Detection and characterization of SQL injection attacks using machine learning," *Computers & Security*, vol. 93, p. 101784, 2020, Doi: 10.1016/j.cose.2020.101784.
- [18] Z. Lin, Y. Zhang, and X. Wang, "Learning-based anomaly detection for web attacks using deep neural networks," *Future Generation Computer Systems*, vol. 115, pp. 609–620, 2021, Doi: 10.1016/j.future.2020.09.035.
- [19] M. Roesch, "Snort: Lightweight intrusion detection for networks," in Proc. 13th USENIX Conf. System Administration (LISA), 1999, pp. 229–238.

Disclaimer/Publisher's Note: The perspectives, opinions, and data shared in all publications are the sole responsibility of the individual authors and contributors, and do not necessarily reflect the views of Sciences Force or the editorial team. Sciences Force and the editorial team disclaim any liability for potential harm to individuals or property resulting from the ideas, methods, instructions, or products referenced in the content.