ISSN 1846-6168 (Print), ISSN 1848-5588 (Online)
https://doi.org/10.31803/tg-20210205101347

Preliminary communication

# Ensemble Machine Learning Approaches for Detection of SQL Injection Attack

Umar Farooq

**Abstract:** In the current era, SQL Injection Attack is a serious threat to the security of the ongoing cyber world particularly for many web applications that reside over the internet. Many webpages accept the sensitive information (e.g. username, passwords, bank details, etc.) from the users and store this information in the database that also resides over the internet. Despite the fact that this online database has much importance for remotely accessing the information by various business purposes but attackers can gain unrestricted access to these online databases or bypass authentication procedures with the help of SQL Injection Attack. This attack results in great damage and variation to database and has been ranked as the topmost security risk by OWASP TOP 10. Considering the trouble of distinguishing unknown attacks by the current principle coordinating technique, a strategy for SQL injection detection dependent on Machine Learning is proposed. Our motive is to detect this attack by splitting the queries into their corresponding tokens with the help of tokenization and then applying our algorithms over the tokenized dataset. We used four Ensemble Machine Learning algorithms: Gradient Boosting Machine (GBM), Adaptive Boosting (AdaBoost), Extended Gradient Boosting Machine (XGBM), and Light Gradient Boosting Machine (LGBM). The results yielded by our models are near to perfection with error rate being almost negligible. The best results are yielded by LGBM with an accuracy of 0.993371, and precision, recall, f1 as 0.993373, 0.993371, and 0.993370, respectively. The LGBM also yielded less error rate with False Positive Rate (FPR) and Root Mean Squared Error (RMSE) to be 0.120761 and 0.007, respectively. The worst results are yielded by AdaBoost with an accuracy of 0.991098, and precision, recall, f1 as 0.990733, 0.989175, and 0.989942, respectively. The AdaBoost also yielded high False Positive Rate (FPR) to be 0.009.

**Keywords:** Boosting; ensemble learning; Light GBM; SQL injection; web security

## 1 INTRODUCTION

A Web Application is software that uses internet connected web browsers and has gained high importance for performing different tasks in social, commercial, academic, and other platforms. These web applications are connected to back-end relational databases operated by Structured Query Language (SQL) that hold a huge amount of information like usernames, passwords, bank details, etc., and are used for communication, online transactions, data storage, accessing social networks, etc. Despite all the importance of these web applications it provides a way for hackers and crackers to attack these databases. Securing the web data must be of the utter importance for developers of these web applications.

Almost 98% of web applications are prone to various attacks but the top most one is SQL Injection attack as is listed as number one in the top ten web application security risks by Open Web Application Security Project (OWASP) [1, 2]. This attack has been listed in top ten vulnerabilities by OWASP from last fifteen years [3]. Refined software and other tools are also used nowadays to perform injection attacks controlled by machines [4].

SQL injection is an exploitation technique that compromises the security at database layer of a web application. This vulnerability usually occurs due to insufficient validation of inputs and directly including them in a SQL query. By utilizing these vulnerabilities, an attacker can submit SQL queries legitimately to the database. Generally, any web application is prone to SQL injection attack when any of the following vulnerabilities are present in the web application:

- When filtration, validation, and sanitization of input data from the user is not applied by the web application.
- When the dynamic queries or non-defined calls are given directly to the interpreter.
- When hostile data is used to retrieve sensitive data from the database or dynamic query is concatenated with both hostile data and structure [5].

SQL injection attacks are classified into seven categories: tautologies, illegal/logically incorrect queries, piggy-backed queries, stored queries, inference and alternate encodings [6]. In SQL injection a malicious script is being embedded into a less secure web application through an entry node then bypassed to the back-end database. This script then forces the web application to produce results from the database through queries that shouldn't be executed normally or ever. Using this attack, an attacker can get all the data from the database by bypassing the authentication and authorization of the web application.

SQL injection is a code injection technique that can provide the attacker with an unauthorized access to the sensitive information in the database. It not only gets the unrestricted access but it can also be utilized to disturb data integrity by adding, deleting, or modifying the records in a database. SQL injection attack is primarily focused on exploiting vulnerability in the security of a web application that is when the user input is not correctly validated or filtered, and when user input is not typed strongly and executed unexpectedly. It also occurs when there is weakness in the code, programming language. It is an attack vector for web applications but also can be used to attack any kind of SQL database. Hackers can gain unauthorized access to underlying data, structure, and DBMS. The well understood example of SQL injection attack is tautological one, "SELECT * FROM Users WHERE User-id = 1 or 1=1", where the injection happens due to the true condition using OR. Attackers nowadays use other ways to perform mass SQL injection attacks such as refined tools or botnets for discovering of vulnerable sites [3].

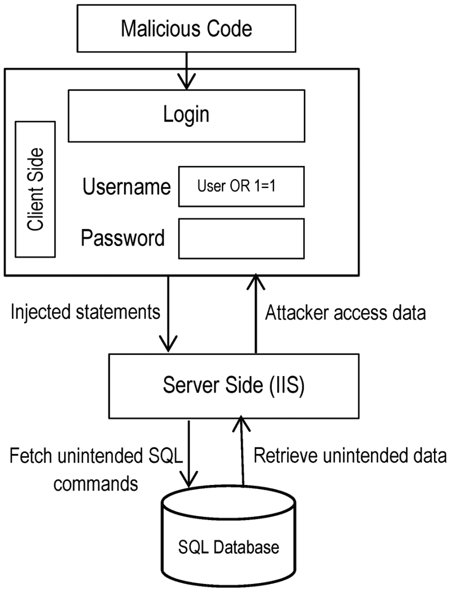

**Figure 1** Typical SQL Injection Attack

## 2 BACKGROUND

In this section, we will briefly mention out all the ten types of SQL injection attack.

### 2.1 Tautologies

The attacker uses a conditional query wherein the ‘WHERE’ clause is used to inject and make the condition a tautology that always happens to be true. In example “SELECT * FROM Users WHERE User-id = 1 or 1=1”, the query will result all the data in the database the condition of WHERE clause is true. This can be secured by restricting the users to input special characters like single quotes, double quotes, equality, and other symbols that are used to make the malicious queries [7].

Example: SELECT * FROM accountTable WHERE user login= or 1=1

### 2.2 Piggy-Backed Query

This type is used to retrieve data, modify database, execute commands and perform Denial of Services (DOS) attack. In this attack, attacker tries to inject other malicious queries along with the normal/original query. The original query is true and executed normally while as additional malicious queries are injected without checking. This can be secured by avoiding execution of multiple statements and checking for delimiter in all queries [7].

Example: SELECT * FROM accountTable WHERE user login=umar AND passwd=; drop accountTable user – AND pin=221

### 2.3 Union Query

This type is used to bypass authentication and extract all data from the database. In this attack, attacker inserts a UNION query into parameter that happens to be weak hence vulnerable. This can be secured by verifying the user inputs strictly and avoid execution of multiple queries on the side of database [7].

Example: SELECT * FROM accountTable WHERE user login= UNION SELECT * FROM accountTable WHERE No=10232 – AND passwd = AND pin=

### 2.4 Stored Procedures

This type is used to execute remote commands, perform DOS, and for privilege escalation. In this attack, the attacker uses delimiter “;” and stored procedure keywords such as “EXEC”, “SHUTDOWN”, etc. This can be secured by verifying the user input with a low privileged account for execution and executing stored procedures within a safe interface with appropriate roles [7].

Example: SELECT * FROM accountTable WHERE user login= ‘umar’ AND passwd = ‘farooq’; SHUTDOWN;–;

### 2.5 Illegal/Logically Incorrect Queries

This type is used to detect such parameters that are vulnerable to injection and then extract data from the identified database. In this attack, attacker tries to extract all information about database and structure. This can be secured by verifying inputs from user and avoiding the generation of error messages from database [7].

Example: SELECT * FROM accountTable WHERE user login= ’umar”’ AND passwd =

### 2.6 Inference

This type is used to detect such parameters that are vulnerable to injection and then extract data from the database with schema identified. This attack is launched on secured databases and is of two types: Inference blind SQL injection and Inference time SQL injection [7].

Example: 1; IF SYSTEM_USER='sa' SELECT 1/0 ELSE SELECT 5

### 2.7 Alternate Coding

This type is used to escape from being detected. In this attack, attacker injects encoded text to bypass detection techniques with the help of signatures like EXEC (), Char (), ASCII (), BIN (), HEX (), UNHEX (), BASE64 (), DEC (), ROT13 (), etc. This can be secured by verifying user inputs and prohibition of meta-characters [7].

Example: SELECT * FROM accountTable WHERE user login= ’umar’;exec(char(0x59842 352646f776e)) AND passwd =’farooq’ AND pin =; SHUTDOWN;–;

### 2.8 End of Line Comment

SELECT * FROM Accounts WHERE accountName = _admin‘--_AND password = _‘

This statement logs the hacker as admin user [8].

### 2.9 Blind Injection

This type is used for asking Boolean (true/false) questions and the information is extracted depending upon the behavior of the web page. The web page functions normally if the injection attack is true, otherwise the web page functions differently [8].

### 2.10 Timings Attacks

This type is used to derive information with the help of If-Then statements where the attacker notes the timing delays of responses from the database [8].

Generally, SQL injection attack is divided into three types depending upon the mode of transfer of incoming and outgoing data. The three types are in-band, out-of-band, and inferential [9]. In in-band SQL injection attack, the attacker extracts the information from the same channel that is used for sending the query or performing the attack. In out-of-band SQL injection attack, the attacker extracts the information with the help of another channel like email. In inferential SQL injection attack, the attacker does not extract the information using any channels rather launches other attacks to analyze the behavior of the web application.

## 3 RELATED WORK

Multiple studies and researches have been carried out so far on the field of SQL injection and it's detection by using various approaches like static & dynamic analysis, combined technique, machine learning, Hash technique, Black Box testing, etc. [10].

Static analysis checks whether each stream from a source to a sink is dependent upon an info approval and additionally input purifying routine [11]; though dynamic analysis depends on progressively mining the developer's planned query structure on any information and recognizes assaults by contrasting it against the structure of the real given query [12].

AMNESIA, as a consolidated methodology, is a model-based method that consolidates the static and dynamic analysis for detection and prevention of SQL injection attacks. It uses static analysis in order to make the SQL query models at the time of accessing the database. It then uses dynamic analysis before the queries are sent to database and compares them with the already built statically models [10]. But there are some queries and code snippets generation approaches that make this model less efficient with more error rate [13].

Hidden Markov Model (HMM) has been presented to detect malicious queries with the help of machine learning in two phases: training and running phase. The first phase focuses on collecting known malicious and benign queries and the second phase focuses on detecting injection attacks. Author, by himself, cleared that WHERE clause and piggybacked queries cannot be detected by this model [4].

Detection of SQL injection attack based on Naïve Bayes machine learning algorithm was proposed combined with the mechanism of role-based access [14]. The detection rate with this model is 93%, however future attack cannot be detected with this data and the classifier relies on the labeled data.

## 4 METHODOLOGY

The main motive of the proposed model is to detect SQL Injection attack. The whole procedure is performed in four stages:

1) The first stage focuses on collecting the dataset that contains proper SQL injection attack queries. For this issue, we created a dataset that contains SQL queries, SQL injection attack queries, and plain text. The labelling of the dataset is done in this stage.
2) The second stage deals with extracting all the features from all the queries and selecting the best of them (a.k.a. Feature extraction and feature selection). Tokenization is used in this stage to divide the queries into tokens.
3) The third stage deals with training the model. The model is trained in this phase with 70% of the dataset (a.k.a. Training part).
4) The fourth stage is focused on using the 30% of dataset that we separated from the collected dataset for testing and evaluating the proposed model with the selected best feature set (a.k.a. Testing part).

### 4.1 Dataset

The most important part in detecting a SQL injection attack is collecting a meaningful dataset that contains SQL injection attack queries. The main contribution in this paper is a labelled dataset that we manually collected for the said problem. The dataset not only contains SQL injection attack queries but also normal SQL injection queries and plain text queries so that the proposed model will properly comprehend and differentiate between normal and attacking SQL queries. The dataset is collected in three phases: 1) the normal SQL injection queries are collected in first phase, 2) the SQL injection attack queries are collected in the second phase, and 3) the plain text is collected in the third phase. We collected these queries in the text format and applied labelling and preprocessing methods on it and then converted it to a csv file. We applied tokenization on the dataset and formed a new tokenized dataset. The dataset contains a total of 35198 queries with 21 features. The dataset has the following three categories:

#### 4.1.1 Non-Malicious or Normal SQL Queries

These queries, non-malicious in nature, are used to create, maintain, and retrieve database in the form of tables (relational database). The tokens (keywords) used in this type are: (rename, drop, delete, insert, create, exec, update, union, set, Alter, database, and, or, information_schema, load_file, select, shutdown, cmdshell, hex, ascii). Also the dangerous characters used in this type are: --, #, /*, ', ", ||, \\, =, /**/,@@.

### 4.1.2 SQL Injection Attack Queries/Malicious SQL Queries

These queries are used to execute malicious SQL statements in a web application and bypass the security measures. These queries are also used to add, modify, and delete records in a database in an unrestricted way. The tokens (keywords) used in this type are: , *, ; , _, -, (, ), =, {, }, @, ., , &, [, ], +, -, ?, %, !, :, \, /. Also the SQL tokens used are: where, table, like, select, update, and, or, set, like, in, having, values, into, alter, as, create, revoke, deny, convert, exec, concat, char, tuncat, ASCII, any, asc, desc, check, group by, order by, delete from, insert into, drop table, union, join.

### 4.1.3 Plain Text

These are simply in the form of plain text. The tokens (keywords) used in this type are alphabets and digits. The plain text is used in this dataset in order to make sure that the proposed model properly comprehends and differentiated between the SQL query, SQL injection query and the plain text that the user inputs in the login node of any web app.

The detailed description of the collected dataset (features) is given below in Tabs. 1 and 2.

**Table 1** Description of features of dataset

| S. No. | Feature | Description |
|---|---|---|
| 1 | data | It contains all the full queries |
| 2 | no_sngle_quts | Total number of single quotations in a query |
| 3 | no_dble_quts | Total number of double quotations in a query |
| 4 | no_punctn | Total number of punctuations in a query |
| 5 | no_sgle_cmnt | Total number of single line comments in a query |
| 6 | no_mlt_cmnt | Total number of multi-line comments in a query |
| 7 | no_whte_spce | Total number of white spaces in a query |
| 8 | no_nrml_kywrds | Total number of normal keywords in a query |
| 9 | no_hmfl_kywrds | Total number of harmful keywords in a query |
| 10 | no_prctge | Total number of percentage (%) symbols in a query |
| 11 | no_log_oprtr | Total number of logical operators in a query |
| 12 | no_oprtr | Total number of operators in a query |
| 13 | no_null_valus | Total number of null values in a query |
| 14 | no_hexdcml_valus | Total number of hexadecimal values in a query |
| 15 | no_db_info_cmnds | Total number of database information commands in a query |
| 16 | no_roles | Total number of roles (e.g., Admin, user, etc.) in a query |
| 17 | no_ntwr_cmnds | Total number of network commands in a query |
| 18 | no_lanage-cmnds | Total number of language commands in a query |
| 19 | no_alphabet | Total number of alphabets in a query |
| 20 | no_digits | Total number of digits in a query |
| 21 | no_spl_chrtr | Total number of special characters in a query |

**Table 2** Description of labels

| S. No. | Label | Description | Count | Ratio |
|---|---|---|---|---|
| 1 | 0 | It represents the normal SQL queries | 6888 | 19.57% |
| 2 | 1 | It represents the SQL injection attack queries | 18369 | 52.19% |
| 3 | 2 | It represents the plain text | 9941 | 28.24% |

### 4.2 Tokenization

The keywords used in SQL injection attack are used to launch operations on the database tables. These keywords play an important role in launching SQL injection attack as the keywords perform the unexpected tasks. So, there is a need to differentiate these keywords form a normal and malicious query. The method of tokenization is used to perform such operation i.e., extract the tokens from the actual queries. In simple terms, tokenization is the process of dividing a query into a list of tokens (keywords). Depending upon these extracted tokens, the proposed model extracts features. Each query is represented by a sequence of numbers where each number represents one of the features represented in Tab. 1.

The suitable determination of these features plays an essential function in detection of SQL injection attack. The reasoning for picking these sorts of features is its capacity to recognize the greater part of SQIA types like redundancies/tautologies, union, piggybacked, illegal/logically incorrect, alternate encodings and stored procedures which are dealt with the same as SQL queries.

Let us take the example of **or 1=1** to understand the concept of tokenization.

By applying the tokenization to the above query, the output is given below and is in accordance with the features listed in Tab. 1:

| | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| or 1=1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 0 | 1 |

### 4.3 Training Ensemble Models

The main phase is to train the machine learning algorithms for the detection of SQL injection attack with the manually collected dataset. The selected ensemble learning algorithms that we used in our proposed model are Gradient Boosting Machine (GBM), Adaptive Boosting (AdaBoost), Extended Gradient Boosting Machine (XGBM), and Light Gradient Boosting Machine (LGBM). To have a better understanding of how the machine learning models would perform over the testing data we applied three and five-fold cross-validation where we split the dataset into 3 and 5 parts, respectively. The advantage of cross validation is that all the observations are utilized for both training and testing the models, and each observation is used for testing exactly once.

## 5 RESULTS AND DISCUSSION

As per the experiments that we conducted, we come to conclusion that our proposed system is enough to detect SQL injection attack queries from normal and plain text queries with 21 features. We focused on making the features as much as possible in order to make the proposed model robust and detect all types of SQL injection attack queries, efficiently. To evaluate the performance of our proposed model we applied the algorithms, ensemble boosting in nature, on the testing data (30% of the original dataset). The classification results that were evolved by the proposed model are near perfection and are depicted in the below tables and figures.

We separated the results in different tables, where in every table represents different classification metrics such as accuracy (Acc.), precision (Pr.), recall (Re.), f1 score (f1), false positive rate (FPR), root mean squared error (RMSE), mean absolute error (MAE), and mean squared error (MSE), to analyze the behavior of our system properly. The results are depicted in below Tabs. 3-14.

### 5.1 Classification Report

**Table 3** Accuracy report of our proposed model

| Accuracy | | | |
|---|---|---|---|
| Classifier | Partition Strategy | 3-CV | 5-CV |
| GBM | Training Set = 70%<br>Testing Set = 30% | 0.991856 | 0.990909 |
| AdaBoost | | 0. 991098 | 0. 991098 |
| XGBoost | | 0.992233 | 0.992233 |
| Light GBM | | 0.993371 | 0.993371 |

**Table 4** Precision report of our proposed model

| Precision | | | |
|---|---|---|---|
| Classifier | Partition Strategy | 3-CV | 5-CV |
| GBM | Training Set = 70%<br>Testing Set = 30% | 0.991791 | 0.990660 |
| AdaBoost | | 0.990733 | 0.990733 |
| XGBoost | | 0.991400 | 0.991400 |
| Light GBM | | 0.993373 | 0.993373 |

**Table 5** Recall report of our proposed model

| Recall | | | |
|---|---|---|---|
| Classifier | Partition Strategy | 3-CV | 5-CV |
| GBM | Training Set = 70%<br>Testing Set = 30% | 0.990388 | 0.989341 |
| AdaBoost | | 0.989175 | 0.989175 |
| XGBoost | | 0.990596 | 0.990596 |
| Light GBM | | 0.993371 | 0.993371 |

**Table 6** F1 score report of our proposed model

| F1 Score | | | |
|---|---|---|---|
| Classifier | Partition Strategy | 3-CV | 5-CV |
| GBM | Training Set = 70%<br>Testing Set = 30% | 0.991084 | 0.989997 |
| AdaBoost | | 0.989942 | 0.989942 |
| XGBoost | | 0.992234 | 0.992234 |
| Light GBM | | 0.993370 | 0.993370 |

**Table 7** MAE report of our proposed model

| MAE | | | |
|---|---|---|---|
| Classifier | Partition Strategy | 3-CV | 5-CV |
| GBM | Training Set = 70%<br>Testing Set = 30% | 0.010321 | 0.011590 |
| AdaBoost | | 0.011553 | 0.011553 |
| XGBoost | | 0.011742 | 0.011742 |
| Light GBM | | 0.009280 | 0.009280 |

**Table 8** MSE report of our proposed model

| MSE | | | |
|---|---|---|---|
| Classifier | Partition Strategy | 3-CV | 5-CV |
| GBM | Training Set = 70%<br>Testing Set = 30% | 0.014678 | 0.016590 |
| AdaBoost | | 0.016856 | 0.016856 |
| XGBoost | | 0.017992 | 0.017992 |
| Light GBM | | 0.014583 | 0.014583 |

**Table 9** RMSE report of our proposed model

| RMSE | | | |
|---|---|---|---|
| Classifier | Partition Strategy | 3-CV | 5-CV |
| GBM | Training Set = 70%<br>Testing Set = 30% | 0.121152 | 0.128805 |
| AdaBoost | | 0.129830 | 0.129830 |
| XGBoost | | 0.134135 | 0.134135 |
| Light GBM | | 0.120761 | 0.120761 |

**Table 10** FPR report of our proposed model

| False Positives | | | |
|---|---|---|---|
| Classifier | Partition Strategy | 3-CV | 5-CV |
| GBM | Training Set = 70%<br>Testing Set = 30% | 0.008 | 0.009 |
| AdaBoost | | 0.009 | 0.010 |
| XGBoost | | 0.008 | 0.008 |
| Light GBM | | 0.007 | 0.007 |

### 5.2 Confusion Matrix

Confusion matrix is a performance measurement for machine learning classifiers with different combinations of actual and predicted values. The above results are calculated with the help of confusion matrix that is used to evaluate the overall performance of our proposed classification system. As the problem we chose is multi-class classification with three classes (normal SQL query, SQL injection attack query, and plain text), hence the confusion matrix is 3×3. The following classification metrics are evaluated:

$$Accuracy = \frac{TP + TN}{TP + TN + FN + FP} \tag{1}$$

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

$$Recall = \frac{(TP)}{(TP + FN)} \tag{3}$$

$$F1\,Score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{4}$$

$$MAE = \frac{\sum_{i=1}^{n} abs(y_i - \hat{y})}{n} \tag{5}$$

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{6}$$

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \quad (7)$$

$$FPR = \frac{FP}{FP + TN} \text{ or } 1 - Recall \quad (8)$$

The confusion matrix of our algorithms is given below where 0, 1, and 2 represent normal SQL queries, SQL injection attack queries, and plain text, respectively.

**Table 11** Confusion matrix of AdaBoost

| AdaBoost | | | | |
|---|---|---|---|---|
| | | Actual | | |
| | | 0 | 1 | 2 |
| Predicted | 0 | 1966 | 12 | 21 |
| | 1 | 7 | 5473 | 37 |
| | 2 | 6 | 20 | 3018 |

**Table 12** Confusion matrix of GBM

| GBM | | | | |
|---|---|---|---|---|
| | | Actual | | |
| | | 0 | 1 | 2 |
| Predicted | 0 | 2078 | 12 | 15 |
| | 1 | 3 | 5461 | 22 |
| | 2 | 8 | 26 | 2935 |

**Table 13** Confusion matrix of XGBoost

| XGBoost | | | | |
|---|---|---|---|---|
| | | Actual | | |
| | | 0 | 1 | 2 |
| Predicted | 0 | 2060 | 21 | 37 |
| | 1 | 7 | 5388 | 41 |
| | 2 | 4 | 28 | 2974 |

**Table 14** Confusion matrix of LGBM

| Light GBM | | | | |
|---|---|---|---|---|
| | | Actual | | |
| | | 0 | 1 | 2 |
| Predicted | 0 | 2095 | 6 | 17 |
| | 1 | 1 | 5418 | 17 |
| | 2 | 11 | 18 | 2977 |

The classification report of our proposed system is given in Fig. 2 wherein we represented it in graphical form.

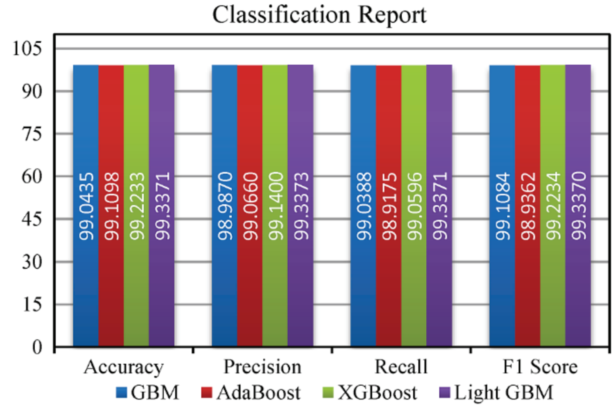

**Figure 2** Classification report

The error report, in graphical form, of our proposed system is given in Fig. 3.

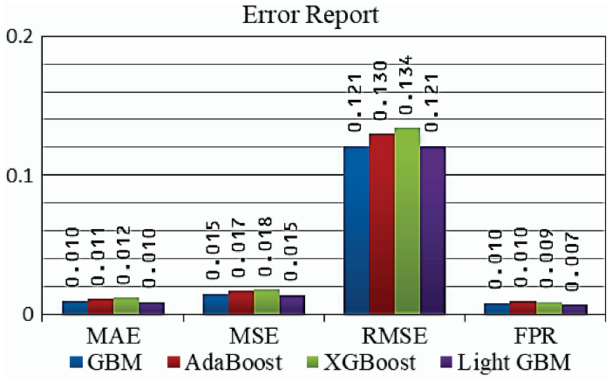

**Figure 3** Error report

The classification reports evaluated by our four models are given in Fig. 4.

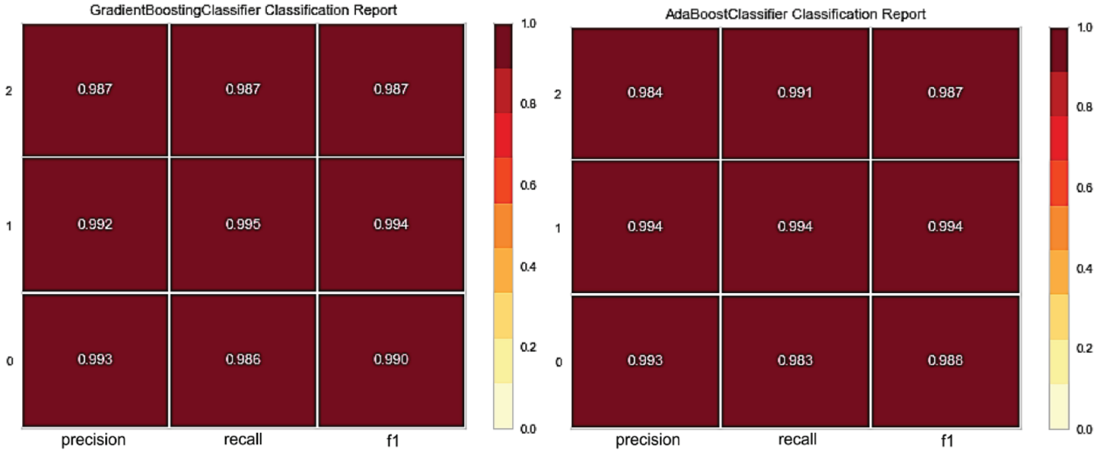

**Figure 4** Classification report from GBM, AdaBoost, XGBM, and LGBM, respectively

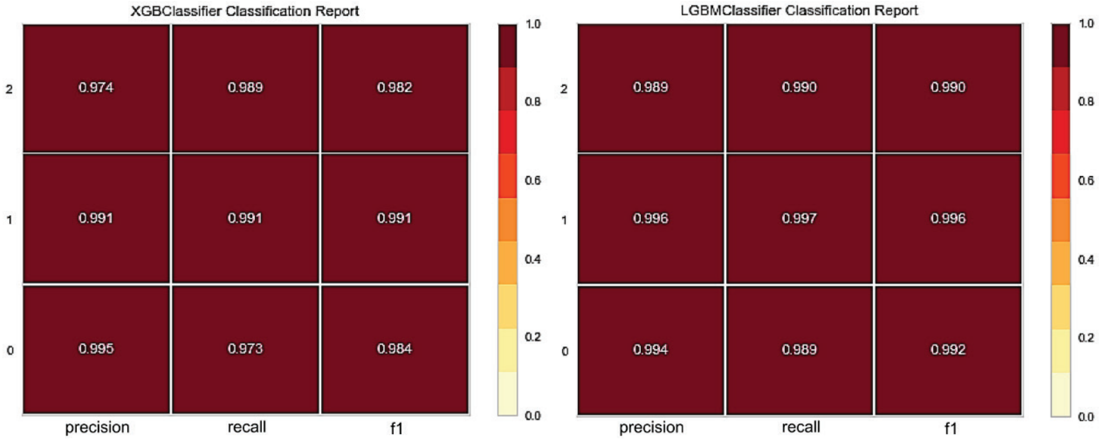

**Figure 5** Classification report from GBM, AdaBoost, XGBM, and LGBM, respectively (continuation)

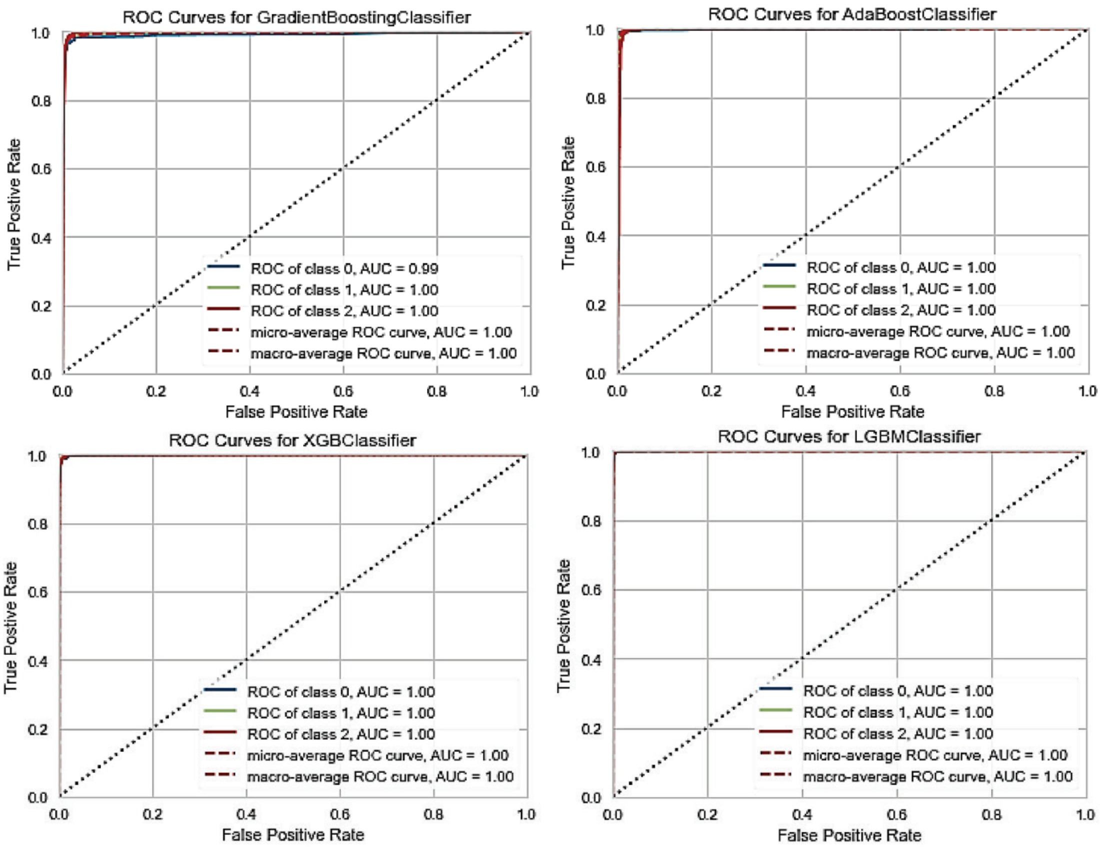

**Figure 6** ROC results from GBM, AdaBoost, XGBM, and LGBM, respectively

## 5.3 Roc Curves

The ROC values evaluated by our algorithms are given in Tab. 15.

**Table 15** ROC values of our proposed models

| Algorithms | GBM | AdaBoost | XGBoost | Light GBM |
|---|---|---|---|---|
| ROC Value | 0.995449 | 0.997657 | 0.999548 | 0.999845 |

## 5.4 Comparative Analysis

The comparative analysis for the research that has been made on SQL injection attack is depicted in the table below (Tab. 16) and we compared them with the proposed model in terms of accuracy. Our proposed model dominates other existing models in terms of accuracy with less error rate.

**Table 16** Comparative analysis

| Classifiers/Models | Accuracy |
|---|---|
| SVM, Naïve Bayes, GBM, REGEX [15] | 97% |
| Neural Network system [16] | 96.8% |
| Genetic- fuzzy rule-based system [17] | 98.4% |
| SVM [18] | 98% |
| K-means [19] | 98.36% |
| Our Proposed model (GBM, AdaBoost, XGBM, LGBM) | 99.34% |

## 6 CONCLUSION

In this research work, we proposed SQL injection attack detection model based on 21 features in order to increase the efficiency of our classifiers. The main target of our system was particularly SQL injection attack that is increasing day by day while being used with some malicious content to gain unrestricted access to databases and extract sensitive information. These malicious queries can bypass authentication and authorization and can finally alter, modify, and delete the database. Keeping this as our objective, we proposed a robust model for detection of SQL injection attack queries from normal queries and plain text. In this work, the foremost step we carried out was to create a balanced dataset that contains normal and malicious SQL queries. We also introduced plain text to this dataset in order to make the proposed model perform well and differentiate malicious queries from normal and plain text.

The proposed model when applied to the dataset achieves an average accuracy of more than 99% with almost negligible error rate that indicates the selected feature set is quite efficient to discriminate SQL injection attack queries from normal SQL queries and plain text. For real world detection systems, the analysis indicate that our proposed system that is based on ensemble machine learning with the selected features can be applied in such SQL injection attack detection systems. The best test accuracy happens to be 99.34% with 0.007 percent FPR while as the lowest one is 99.11% with 0.009 percent FPR, yielded by LGBM and AdaBoost, respectively. The other two algorithms GBM and XGBM that we used yielded accuracy of 99.19% and 99.22%, respectively.

### Notice

This paper was presented at IC2ST-2021 – International Conference on Convergence of Smart Technologies. This conference was organized in Pune, India by Aspire Research Foundation, January 9-10, 2021. The paper will not be published anywhere else.

## 7 REFERENCES

[1] OWASP. https://owasp.org/www-project-top-ten/. (Accessed on 18.11.2020).

[2] Farooq, U. (2020). Real Time Password Strength Analysis on a Web Application Using Multiple Machine Learning Approaches. *International Journal of Engineering Research & Technology (IJERT), 9*(12), 359-364.

[3] Moh, M., Pininti, S., Doddapaneni, S., & Moh, T. (2016). Detecting Web Attacks Using Multi-stage Log Analysis. *2016 IEEE 6th International Conference on Advanced Computing (IACC)*, Bhimavaram, 733-738. https://doi.org/10.1109/IACC.2016.141

[4] Kar, D., Agarwal, K., Sahoo, A., & Panigrahi, S. (2016). Detection of SQL injection attacks using Hidden Markov Model. *2016 IEEE International Conference on Engineering and Technology (ICETECH)*, Coimbatore, India. https://doi.org/10.1109/ICETECH.2016.7569180

[5] OWASP. https://owasp.org/www-project-top-ten/2017/A1_2017-Injection. (Accessed on 19.11.2020).

[6] Moosa, A. (2010). Artificial Neural Network based Web Application Firewall for SQL Injection. *World Academy of Science, Engineering and Technology, International Journal of Computer and Information Engineering, 4*(4), 610-619. https://panel.waset.org/publications/1001/pdf

[7] Sheykhkanloo, N. M. (2015). SQL-IDS: Evaluation of SQLi Attack Detection and Classification Based on Machine Learning Techniques. *The 8th International Conference on Security of Information and Networks (SIN15)*, Sochi, Russia. https://doi.org/10.1145/2799979.2800011

[8] Kaur, M. & Agrawal, A. P. (2012). Token Sequencing Approach to Prevent SQL Injection Attacks. *IOSR Journal of Computer Engineering (IOSRJCE), 1*(1), 31-37. https://doi.org/10.9790/0661-0113137

[9] Sadeghian, A., Zamani, M., & Ibrahim, S. (2013). SQL injection is still alive: a study on SQL injection signature evasion techniques. In *International Conference on Informatics and Creative Multimedia*, Kuala Lumpur, Malaysia, 265-268. https://doi.org/10.1109/ICICM.2013.52

[10] Halfond, W. G. & Orso, A. (2005). AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 174-183. https://doi.org/10.1145/1101908.1101935

[11] Shar, L. K. & Tan, H. B. K. (2013). Defeating SQL injection. *Computer, 46*, 69-77. https://doi.org/10.1109/MC.2012.283

[12] Tajpour, A. & Shooshtar, M. J. Z. (2010). Evaluation of SQL injection detection and prevention techniques. In *Second IEEE International Conference on Computational Intelligence, Communication Systems and Networks*, Liverpool, UK, 216-221. https://doi.org/10.1109/CICSyN.2010.55

[13] Dharam, R. & Shiva, S. G. (2013). Runtime monitors to detect and prevent union query based SQL injection attacks. In *Tenth International Conference on Information Technology: New Generations*, Las Vegas, USA, 357-362. https://doi.org/10.1109/ITNG.2013.57

[14] Joshi, A. & Geetha, V. (2014). SQL Injection detection using machine learning. In *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, Kanyakumari, IEEE, 1111-1115. https://doi.org/10.1109/ICCICCT.2014.6993127

[15] Kranthikumar, B. & Velusamy, R. L. (2020). SQL injection detection using REGEX classifier. *Journal of Xi'an University of Architecture & Technology, 12*(6), 800-809.

[16] Sheykhkanloo, N. M. (2015). SQL-IDS: evaluation of SQLi attack detection and classification based on machine learning techniques. In *Proceedings of the 8th International Conference on Security of Information and Networks*, USA, 258-266. https://doi.org/10.1145/2799979.2800011

[17] Basta, C., Elfatatry, A., & Darwish, S. (2016). Detection of SQL Injection Using a Genetic Fuzzy Classifier System. *International Journal of Advanced Computer Science and Applications (IJACSA), 7*(6), 129-137. https://doi.org/10.14569/IJACSA.2016.070616

[18] Jagadessan, J., Shrivastava, A., Ansari, A., Kar, L. K., & Kumar, M. (2019). Detection and Prevention Approach to SQLi and Phishing Attack using Machine Learning. I*nternational Journal of Engineering and Advanced Technology (IJEAT), 8*(4), 791-799.

[19] Patel, M. P. & Sivaraman, D. B. (2017). SQL injection Detection for Secure Atomic and Molecular Database node for India. *International Journal of Advance Research and Innovative Ideas in Education (IJARIIE), 3*(2), 3867-3879.

**Author's contact:**

**Umar Farooq,**
Department of Computer Science & Technology (Cyber Security),
Central University of Punjab,
City Campus, Mansa Road, Bathinda 151001, Punjab, India
soulatf3@gmail.com