**Automated SQL Injection Detection Using Autoencoder Networks and XGBoost Classifier**

*Reported submitted to the SASTRA Deemed to be*

*University in partial fulfillment of the*

*requirements for the award of the degree of*

B. Tech.

*Submitted by*

Mervinth M (Reg. No.: 126157031, B. Tech(CS & BT))

Mithilesh P S (Reg. No.: 126157032, B. Tech(CS & BT))

Samudrala Koushik (Reg. No.: 126157048, B. Tech(CS & BT))

April 2025



**SCHOOL OF COMPUTING THANJAVUR,**

**TAMIL NADU, INDIA – 613401**

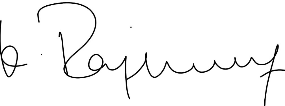


**SCHOOL OF COMPUTING**

**THANJAVUR – 613 401**

**Bonafide Certificate**

This is to certify that the report titled “**Automated SQL Injection Detection Using Autoencoder Networks and XGBoost Classifier**” submitted as a requirement for the course, CSE300 / INT300 / ICT300: MINI PROJECT for B.Tech. is a bonafide record of the work done by **Mr. Samudrala Koushik** (Reg no: 126157048) **Mr.Mervinth M** (Reg no: 126157031) **Mr. Mithilesh P S** (Reg no: 126157032), during the academic year 2024-25, in the School of Computing, under my supervision.

**Signature of project Supervisor** : 

**Name with Affiliation** : Dr. Rajkumar K,Sr. Asst. Professor, School of Computing

**Date**  :

Mini Project *Viva voce* held on **21/05/2025**

Examiner 1 Examiner 2

**ACKNOWLEDGEMENTS**

We would like to thank our Honorable Chancellor **Prof. R. Sethuraman** for providing us with an opportunity and the necessary infrastructure for carrying out this project as a part of our curriculum.

We would like to thank our Honorable Vice-Chancellor **Dr. S. Vaidhyasubramaniam** and **Dr. S. Swaminathan**, Dean, Planning & Development, for the encouragement and strategic support at every step of our college life.

We extend our sincere thanks to **Dr. R. Chandramouli,** Registrar, SASTRA Deemed to be University for providing the opportunity to pursue this project.

We extend my/our heartfelt thanks to **Dr. V. S. Shankar Sriram**, Dean, School of Computing, **Dr. R. Muthaiah,** Associate Dean, Research, **Dr. K.Ramkumar**, Associate Dean,Academics, **Dr. D. Manivannan**, Associate Dean, Infrastructure, **Dr. R. Alageswaran**, Associate Dean, Students Welfare.

Our guide **Dr. Rajkumar K**, Sr. Asst. Professor, School of Computing was the driving force behind this whole idea from the start. His deep insight in the field and invaluable suggestions helped us in making progress throughout our project work. We also thank the project review panel members for their valuable comments and insights which made this project better.

We would like to extend our gratitude to all the teaching and non-teaching faculties of the School of Computing who have either directly or indirectly helped us in the completion of the project.

We gratefully acknowledge all the contributions and encouragement from my family and friends resulting in the successful completion of this project. We thank you all for providing us an opportunity to showcase my/our skills through project.

**LIST OF FIGURES**

|  |  |  |
| --- | --- | --- |
| **Figure No.** | **Title** | **Page No.** |
| 1 | Sample Architecture | 3 |
| 3.1.1 | The architecture workflow analysis of novel proposed deep feature extraction approach | 8 |
| 3.1.2 | Extreme Gradient Boosting (XGB) Architecture | 9 |
| 3.2.2 | Dataset Review | 11 |
| 3.2.3 | Class Distribution and Visualization before preprocessing | 13 |
| 3.2.4 | Class Distribution and Visualization after preprocessing | 14 |
| 3.3.5 | Autoencoder Training vs Validation Loss Plot | 21 |
| 3.4.1 | Confusion Matrix | 22 |
| 3.4.2 | Classification Report | 22 |
| 3.6 | Prediction SQLi | 27 |

**LIST OF TABLES**

|  |  |  |
| --- | --- | --- |
| Figure No | Topics | Page No |
| 1 | Abbreviations | v |
| 2 | Classification Report | 22 |

**ABBREVIATIONS**

SQL Structured Query Language

AI Artificial Intelligence

AE-Net AutoEncoder network

KNN K-Nearest Neighbors

XGB Extreme Gradient Boosting

TFIDF Term Frequency- Inverse Document Frequency

**Automated SQL Injection Detection Using Autoencoder Networks and XGBoost Classifier**

**ABSTRACT**

SQL injection (SQLi) attacks exploit vulnerabilities in database-driven systems that allow them to modify, access, or delete sensitive information. Currently, we see significant downsides in scalability, accuracy, and responsiveness with SQLi detection schemes. This makes it clear that there is a need for better approaches to SQLi that can automate and streamline detection. In this research, we develop a new method with an Autoencoder Network (AE-Net) approach to feature engineering, where high-level representations are extracted from a dataset of 46,392 SQL queries.

The extracted features were subsequently assessed with machine learning models to effectively classify SQLi queries. The model trained with Extreme Gradient Boosting (XGBoost) performed the best with a k-fold accuracy of 0.99. All the models were hyper-parameter tuned and k-fold cross-validation was utilized to ensure the reliability and robustness of the results. Our results from the statistical t-test suggested we could replicate our performance, thereby highlighting the success of the suggested method. This research demonstrated a scalable, automated detection approach to help mitigate SQLi attacks and secure modern systems.

**KEY WORDS**: Machine Learning, SQL injection detection , XGBoost classifier , Gradio GUI deployment

**TABLE OF CONTENT**

|  |  |
| --- | --- |
| **Title** | **Page No** |
| Bonafide Certificate | ii |
| Acknowledgements | iii |
| List Of Figures | iv |
| List Of Tables | iv |
| Abbreviations | v |
| Abstract | vi |
| 1. Summary Of Base Paper` 2. Introduction 3. Related Work 4. Proposed Solution and System Architecture 5. Study Area 6. Methodology & Implementation | 1  1  2  2  2  2 |
| 1. Merits And Demerits of Base Paper 2. Literature Survey 3. Merits 4. Demerits | 5  5  6  7 |
| 1. Source Code And Implementation 2. Methodology 3. Autoencoder Feature Extraction 4. Extreme Gradient Boosting (XGB) 5. Module 1 : Data Processing and EDA 6. Data Cleaning 7. Imports and Dependencies 8. Exploratory Data Analysis 9. Data Preprocessing 10. Module 2 : Feature Extraction and Model Training 11. TF -IDF Vectorization and Data Split 12. Autoencoder Model for extraction viewing 13. Training Autoencoder for Feature Encoding 14. XGBoost Model Training and Accuracy Evalution 15. Autoencoder Training and Validation Loss Plot 16. Module 3 : Model performances 17. XGBoost Confusion Matrix 18. Classification Report 19. K-Fold Cross Validation Statistical Significance Testing 20. Model Performance Evaluation Metrics 21. Module 4 : Predictions 22. SQL Injection Prediction 23. Gradio Interface for SQL Injection Detection 24. Output Snapshots | 8  8  8  9  10  10  10  12  13  15  15  16  19  20  20  21  21  22  23  23  24  24  24  27 |
| 1. Conclusion | 28 |
| 1. References | 29 |
| 1. Appendix | 30 |

**CHAPTER 1**

**SUMMARY OF THE BASE PAPER**

1. **Title:**  AE-Net : Novel Autoencoder-Based Deep Features for SQL Injection Attack Detection
2. **Publisher**: IEEE
3. **Year**: 2023
4. **Journal**: IEEE Access
5. **Indexing:**
6. **Base Paper URL:** <https://doi.org/10.1109/ACCESS.2023.3337645>
   1. **INTRODUCTION**

Structured Query Language (SQL) injection remains one of the most common and dangerous attacks against modern web applications and database systems. Through vulnerabilities in user input texts, attackers can inject SQL commands into database queries, gaining unauthorized access to observed data, altering it, and sometimes even taking over control of the system. The malicious consequences of these types of attacks represent enormous monetary losses, reputational damage, and legal implications for both people and organizations, among other things. While defensive measures exist, typical rule-based/pattern-based and/or signature-based detection and prevention systems have fallen short when it comes to detecting new or obfuscating injection schemes. With the intent of confronting this critical security threat, the paper introduces a new artificial intelligence (AI)-based approach—AE-Net, an autoencoding network specially trained to enhance SQL injection attack detection using advanced feature engineering.

AE-Net automatically extracts high-level deep features directly from SQL text data, enabling more timely and accurate attack detection using deep learning without human involvement. With more than 46,000 SQL queries, the study compares the performance of different machine learning and deep-learning models and highlights that by integrating the Extreme Gradient Boosting (XGB) classifier with AE-Net features, it achieved an unprecedented 0.99 accuracy level. This work on SQL injection detection provides a statistically substantiated and resilient answer to overcome the drawbacks of traditional feature extraction and machine learning approaches and raises the bar in SQL injection detection. AE-Net is demonstrated to be an efficient solution in helping protect web application security through extensive experimentation, hyperparameter optimization, and validation techniques such as k-fold cross-validation and t-test analysis.

* 1. **RELATED WORK**

Recent research on detecting SQL injection (SQLi) attacks is inspired by recent works from the machine learning (ML) and deep learning (DL) communities in search of advanced methods which are tailored around feature extraction in a more automated way than traditional rule-based approaches. In practice, supervised ML techniques have assessed mildly supervised ML methods that use CNNs, CNNs in combination with recurrent neural networks such as BiLSTM networks that have achieved up to 97% detection accuracy. Several of these studies also didn't extensively evaluate pragmatic metrics, such as precision and recall. There is also the possibility for semantic learning-based methods to provide alternatives for detecting SQL injection based on semantic representations for the contextual meaning associated with SQL queries using approaches such as synBERT and RNN-based autoencoders, but their accuracy remains moderate and low for complex or unknown attack patterns. However, regardless of this promise, most of the work completed to-date relies on traditional methods for feature extractions which has resulted in consistently high error rates with considerably less flexibility. These differences may offer a clearer picture of how different human shaped and tailored feature extraction methods are better suited for standard and varying configurations of automated work models for AI, particularly for automated feature extraction, demonstrated in AE-Net—a newer autoencoder-based model designed for extracting deep features from SQL queries and improving detection accuracy with minimal human intervention.

* 1. **PROPOSED SOLUTION AND SYSTEM ARCHITECTURE**
     1. **Study Area**

The study focuses on cybersecurity, particularly the detection of SQL Injection (SQLi) attacks, using artificial intelligence (AI) based methods. SQLi attacks present a serious vulnerability to web applications and database systems because they give attackers access to either alter or obtain sensitive data via authoring malicious SQL code into a field that is user-directed. Traditional detection methods are limited in their ability to handle complex and evasive attack patterns, hence intelligent, computerized detection systems need to be developed. The aim of this research is to improve SQLi detection through advanced deep learning-based feature engineering, thereby improving the system's potential and ability to identify malicious queries.

* + 1. **Methodology & Implementation**

The process begins with the acquisition and preprocessing of large dataset consisting of 46,392 SQL queries from open source, public repositories. Preprocessing includes: label encoding, removal of null or very short queries, and converting the SQL queries into strings. At the core of the approach is the proposed AE-Net (Autoencoder Network), a new deep learning architecture to automatically extract features. AE-Net takes the SQL query data and applies encoder-decoder modules to produce deep features that represent higher level, complex data patterns. The deep features are then used to train and test the resulting ensemble or pipeline of models (e.g, XGBoost, Autoencoder neural network) to know if a query is benign or malicious. The models' training and testing are completed using accuracy, precision, recall, and F1-score metrics. Furthermore, the method tests for robustness using k-fold cross validation, hyperparameter optimization, and statistical t-tests to validate and confirm the performance and generalized performance of the models.

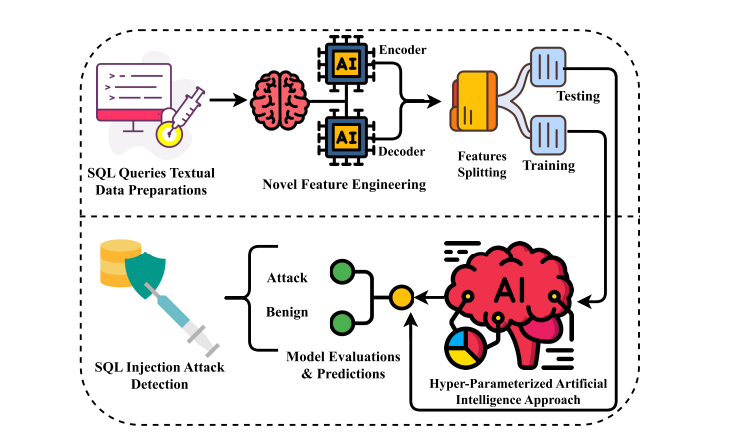


Fig 1 Sample Architecture

The architecture of the system under consideration is built around AE-Net, a new autoencoder neural network that can be used to extract high-level features from SQL query data. The whole detection pipeline is organized into the below-discussed components:

**Data Collection and Preprocessing**

* The system starts with collecting a public dataset of 46,392 SQL queries (from sqli.csv, sqliv2.csv, and SQLiV3.csv).
* Preprocessing involves:
  + Label encoding (Attack = 1, Benign = 0)
  + Deletion of null and extremely short queries
  + Text cleaning and formatting
  + Feature Engineering using AE-Net

**Feature Engineering with AE-Net**

* The AE-Net module conducts automatic feature extraction using an encoder-decoder architecture:
  + Encoder: Maps the input SQL query into a lower-dimensional latent space (deep feature representation)
  + Decoder: Tries to reconstruct the initial input from the latent vector
  + The input-reconstruction loss is minimized (autoencoder loss)
  + The latent space (deep features) is utilized for classification, and not the reconstruction

**Training and Testing Split**

* The data is divided into:
  + 80% Training set
  + 20% Testing set
* Deep features from AE-Net are drawn out from both sets

**Model Training (Classification Phase)**

* Machine Learning and Deep Learning models utilized:
  + Extreme Gradient Boosting (XGB)
  + Autoencoder neural network
* These models are trained with AE-Net deep features to predict queries as "Attack" or "Benign"

**Evaluation and Validation**

* Performance Metrics: Accuracy, Precision, Recall, F1-Score
* Validation Techniques:
  + K-Fold Cross-Validation (to validate model stability)
  + Statistical t-test (to verify performance significance)
  + Computational Complexity Analysis (to verify feasibility)
  + Confusion Matrix (to check classification performance)

**CHAPTER 2**

**MERITS AND DEMERITS OF BASE PAPER**

**2.1 LITERATURE SURVEY**

There are a number of research projects that share the use of machine learning and deep learning algorithms to detect SQL injection (SQLi) attacks. This may further the purposed interest of detection rates and avoidance of classic rule-based detection systems. There are few notable contributions:

* Miquitta and Asha (2023) also suggested a hybrid model and relied on supervised machine learning with Convolutional Neural Networks (CNN). They, too, were able to achieve very good outcomes with a model that had a high accuracy of 97%. However, they failed to confirm in a real-world application how robust the model would be.
* Crespo-martinex et al. (2023) examining SQLi attack flow data from various database engines employed logistic regression. This study achieved a detection rate that exceeded 97% and false alarms of under 0.07%. Crespo-martinex et al. (2023) failed both report the additional standard metrics, precision and recall. Therefore, we have no understanding of their actual performance or clarity surrounding their results.
* Zhang et al. (2023) introduced a feature ratio technique of finding SQLi payload injections found under a 96.29% with below-average F1-score descriptions relative to other examples.
* A CNN-BiLSTM Hybrid was presented by Gandhi et al. as an approach against SQLi attacks that had a precision of 98% or at least, but it underperformed in producing sufficient imp portions reporting under other performance metrics like recall and F1-score, and it was not demonstrated sufficiently that it generalized across data.
* Jothi et al. investigated input-pattern analysis for SQLi detection. Jothi et al.'s Multi-Pattern Learning model performed well with 98% accuracy, 98% precision and 97% recall. The model is also highly scalable and easy to use, due to its automated capacity to detect different variations of attacks.
* Alghawazi et al. executed Recurrent Neural Networks (RNN) using an open Kaggle SQLi dataset. The model was accurate at 94% with a 92% F1-score but less accurate than other new models.
* Lu et al. presented synBERT, a context-aware detection model using embedding vectors to obtain SQL query context. The model achieved, on a wide range of datasets, 90% accuracy, but its metrics did not benchmark with state-of-the-art approaches.
* Alotaibi and Rassam proposed CNN-based Intrusion Detection Systems (IDS) to enhance online security even more. The models began with increasing the accuracy up to a maximum of 97.51%, but became susceptible to attacks, which lowered the accuracy rate down to 78.12%. More alarming, there were no required measures such as precision and recall included in the research.

**RESEARCH GAP**

* What is learned is as follows after proper scrutiny of present work:
* All the present studies depend on standard feature extraction schemes that limit the capacity of the model to analyze deep structural patterns in SQL queries.
* All present models use traditional machine learning practices, which often cannot generalize towards fresh or obfuscated SQL injection attacks.
* High error rates and limited performance measures (more precisely, recall and F1-score) are predominant in disqualifying the reliability of existing detection models.

**2.2 MERITS**

**High Detection Accuracy**

* Developed AE-Net model, when used in conjunction with Extreme Gradient Boosting (XGB), boasts an outstanding detection accuracy of 99% that surpasses the detection accuracy of most conventional and cutting-edge solutions.

**Automatic Deep Feature Extraction:**

* AE-Net utilizes an autoencoder-based architecture that bypasses the tedious process of feature engineering by extracting deep-level high features automatically from SQL query data.

**Strong Model Assessment**

* The study utilizes k-fold cross-validation, confusion matrix, and statistical t-tests for rigorously testing model performance and reliability.

**Deep Performance Metrics**

* Performance metrics are accuracy, precision, recall, F1-score, and computational complexity, which provide a general idea of model effectiveness.

**Balanced Dataset**

* Utilization of a large, well-balanced dataset of 46,392 SQL queries with nearly equal split of attack and benign classes to train and test the model unbiased.

**Comparison with State-of-the-Art**

* The study compares AE-Net with the best 2023 methods and demonstrates improved performance and highlights its strength for real-world application.

**Flexible Implementation**

* It is coded in Python and follows trend libraries like Scikit-learn, Keras, and TensorFlow and thus reproducible as well as editable.

**2.3 DEMERITS**

**High Computational Cost**

* + The training of AE-Net and LSTM models is computationally costly, hence not as suitable for low-resource or real-time environments

**LSTM Underperformance**

* + Of the models utilized, LSTM underperformed with only 87% accuracy, suggesting limitations in sequential modeling for this application.

**Limited Dataset Variety**

* + Although the dataset is representative, it's not really exposed to live web application traffic encountered in the real world, and it could impact generalizability within the model.

**Lack of Interpretability**

* + AE-Net learned deep features that are hard to interpret, something which may restrict its utility to security analysts who need visibility into the behaviors of an attack.

**No Adversarial Testing**

* + The framework was not subjected to adversarial attack scenarios, which are a common occurrence during real-world cyber attacks.

**No Real-Time Interface**

* + The work does not propose or provide for a real-time intrusion detection device or monitoring board, which is essential for deployability in reality.

**Complex Hyperparameter Tuning**

* + The model's performance is based on rigorous hyperparameter tuning across different classifiers, and hence the optimization process is time-consuming and labor-intensive.

**CHAPTER 3**

**SOURCE CODE AND IMPLEMENTATION**

**3.1 METHODOLOGY**

**3.1.1 Autoencoder Feature Extraction**

Autoencoder feature extraction is a deep learning technique that can compress high-dimensional input data into a lower-dimensional and relevant representation through reconstruction of the initial input through learning. An autoencoder has two elements of importance: an encoder that encodes the input (such as SQL queries) into a lower-dimensional compressed representation that contains the patterns and structure pertinent to the input, and a decoder that reconstructs the input from the lower-dimensional representation. The important training objective of the model is to reduce the reconstruction loss-focusing on the loss of the reconstruction between the input and output- that allows the encoder to extract abstract and discriminative features. In the case of SQL injection detection, the AE-Net autoencoder was used to learn the deep features from inherently noisy/irrelevant raw SQL query text, which then serves as the input into classification models that classify (malicious/attack) and (non-)benign SQL queries, eliminating manual or guided feature engineering while also enhancing the ability to identify sophisticated and obfuscated attack patterns.

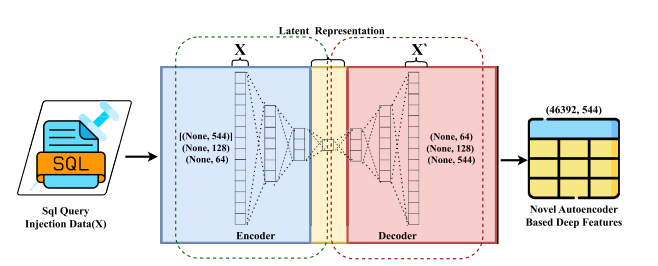


Fig 3.1.1 The architectural workflow analysis of novel proposed deep feature extraction approach.

**3.1.2 Extreme Gradient Boosting (XGB)**

Extreme Gradient Boosting (XGB) is a highly optimized and accelerated machine learning algorithm using the gradient boosting framework which is for high-velocity and high-efficiency classification and regression problems. XGB makes an ensemble of decision trees sequentially, where the second tree aims to fix the errors the first tree makes and so forth in the direction of minimizing some loss function. XGB uses the non-convex optimization algorithm gradient descent to update model predictions by giving extra emphasis on harder-to-predict examples at training time. What is a unique feature of XGB compared to boosting algorithms is it has regularization (L1 and L2) to mitigate overfitting, and the ability to parallel process trees which makes training much faster. It also supports handling missing values, tree pruning, and non-standard loss functions which makes it incredibly flexible and accurate. In the case of SQL injection classification based on AE-Net, XGB used the deep features the autoencoder had learned to classify SQL queries as either malicious or benign with high precision and an overwhelmingly high accuracy rates of up to 99%.

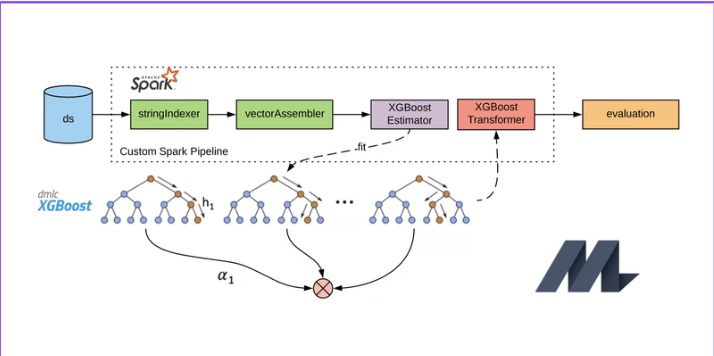


Fig 3.1.2 Extreme Gradient Boosting (XGB) Architecture

**3.2 MODULE 1: DATA PREPROCESSING AND EDA**

**3.2.1 Data Cleaning**

import pandas as pd #import pandas

#load the data

df1 = pd.read\_csv("sqli.csv")

df2 = pd.read\_csv("sqliv2.csv")

df3 = pd.read\_csv('data.csv')

df = pd.concat([df2, df1, df3], ignore\_index=True)

df.head()

df.dtypes

df.info()

#Duplicated datas

df.duplicated()

df.drop\_duplicates()

df.isnull().sum()

#Removing Null Values

df=df.dropna()

#Remove Empty Rows

df.isnull().sum()

**3.2.2 Imports and Dependencies**

import pandas as pd

import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.impute import KNNImputer

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense, BatchNormalization, ReLU

from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler, LabelEncoder

from xgboost import XGBClassifier

from sklearn.metrics import accuracy\_score

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics import confusion\_matrix, classification\_report, ConfusionMatrixDisplay

import matplotlib.pyplot as plt

from sklearn.model\_selection import KFold,cross\_val\_score

from scipy.stats import ttest\_rel

**Load Datasets**

# Load datasets

df1 = pd.read\_csv("sqli.csv")

df2 = pd.read\_csv("sqliv2.csv")

df3 = pd.read\_csv('data.csv')

# Concatenate datasets

df = pd.concat([df2, df1, df3], ignore\_index=True)

# Display basic info

df.info()

df.head()

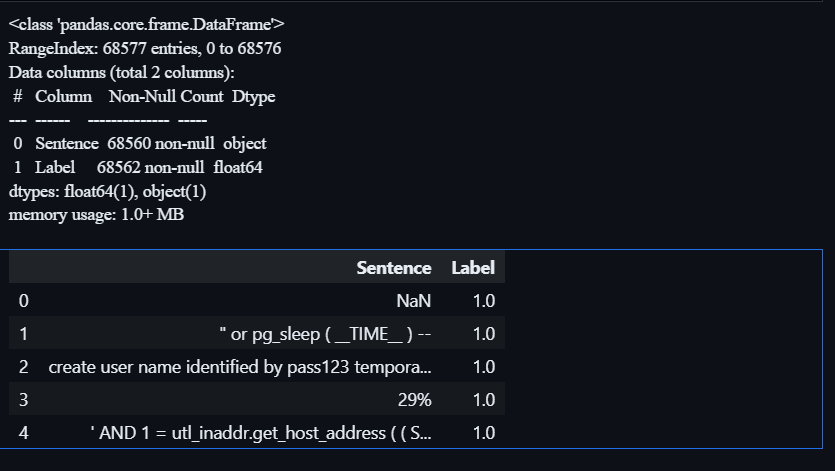


Fig 3.2.2 Dataset review

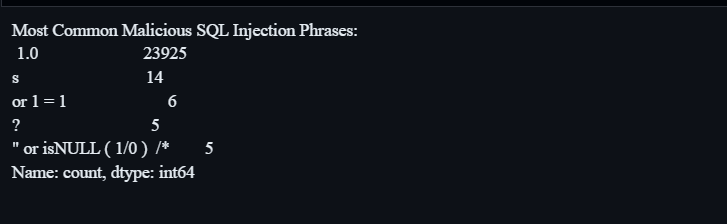
**3.2.3 Exploratory Data Analysis**

malicious\_queries = df[df['Label'] == 1]

common\_phrases = malicious\_queries.stack().value\_counts().head()

print("Most Common Malicious SQL Injection Phrases:\n", common\_phrases)

Identifying Common Malicious SQL Injection Phrases in the Dataset



**Class Distribution and Count Visualization before preprocessing**

plt.figure(figsize=(8, 4))

# Pie chart

plt.subplot(1, 2, 1)

df['Label'].value\_counts().plot.pie(autopct='%1.1f%%', shadow=True, explode=[0, 0.1])

plt.title('Class Distribution')

# Bar chart

plt.subplot(1, 2, 2)

sns.countplot(x='Label', data=df)

plt.title('Class Counts')

plt.tight\_layout()

plt.show()

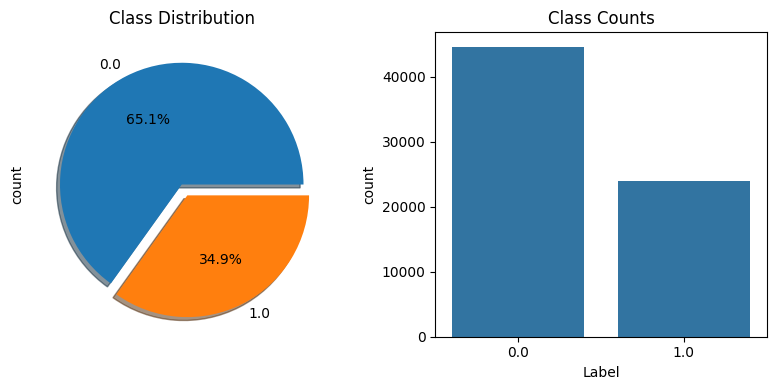


Fig 3.2.3 Class Distribution and Visualization before preprocessing

**3.2.4 Data Preprocessing**

**Encoding, Imputing, and Scaling for Model Training**

df['Sentence'] = df['Sentence'].astype(str)

df = df[df['Sentence'].str.split().str.len() > 2]

df.dropna(subset=['Sentence', 'Label'], inplace=True)

df['Label'] = df['Label'].astype(int)

X = df.drop(columns=['Label'])

y = df['Label']

categorical\_cols = X.select\_dtypes(include=['object']).columns

label\_encoder = LabelEncoder()

for col in categorical\_cols:

X[col] = label\_encoder.fit\_transform(X[col])

X = X.apply(pd.to\_numeric, errors='coerce')

imputer = KNNImputer(n\_neighbors=5)

X\_imputed = imputer.fit\_transform(X)

X = pd.DataFrame(X\_imputed, columns=X.columns)

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

**Class Distribution and Counts Visualization after preprocessing**

plt.figure(figsize=(8, 4))

# Pie chart

plt.subplot(1, 2, 1)

df['Label'].value\_counts().plot.pie(autopct='%1.1f%%', shadow=True, explode=[0, 0.1])

plt.title('Class Distribution')

# Bar chart

plt.subplot(1, 2, 2)

sns.countplot(x='Label', data=df)

plt.title('Class Counts')

plt.tight\_layout()

plt.show()

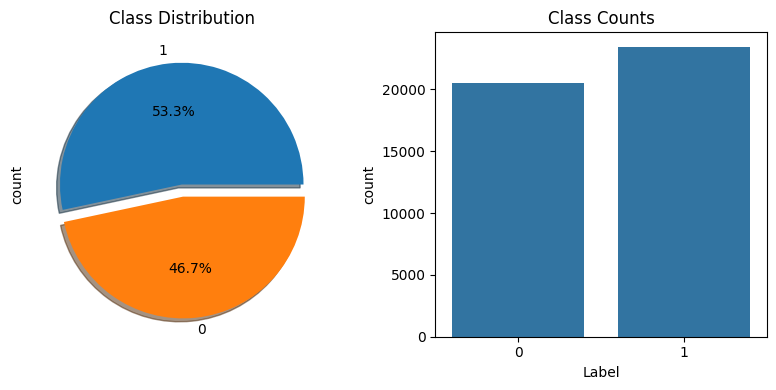
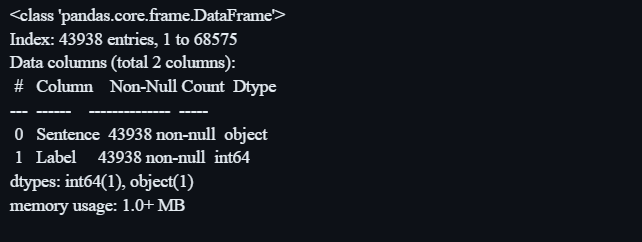


Fig 3.2.4 Class Distribution and Visualization after preprocessing

df.info()



**3.3 MODULE 2: FEATURE EXTRACTION AND MODEL TRAINING**

**3.3.1 TF-IDF Vectorization and Data Split**

vectorizer = TfidfVectorizer(max\_features=544)

X\_tfidf = vectorizer.fit\_transform(df['Sentence']).toarray()

# Show the 544 keywords

feature\_names = vectorizer.get\_feature\_names\_out()

print("Top 544 keywords extracted by TF-IDF:")

X\_scaled = X\_tfidf

print(X\_tfidf.shape)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.2, random\_state=42)



print(list(feature\_names))

['000', '01', '06', '07', '0x28', '0x2e', '0x4b774c75', '0x4c4d6142', '0x52515a50', '0x544d5a4c', '0x5c', '0x694a4745', '0x6d457153', '0x7171706a71', '0x717a767a71', '0x76555642', '0x78', '10', '100', '101', '102', '1022', '103', '1032', '1045', '105', '106', '107', '108', '109', '11', '111', '112', '113', '116', '1161', '117', '118', '119', '12', '120', '121', '1210', '122', '1297', '13', '1441', '15', '1570', '1808', '1996', '20', '2006', '2017', '21', '2367', '2388', '2633', '2716', '2724', '2853', '30', '3020', '3038', '3051', '31', '3114', '3202', '3393', '3440', '3484', '3580', '3623', '3702', '3707', '3715', '3754', '3785', '3824', '3931', '3select', '4144', '4232', '4240', '4241', '4249', '4386', '4411', '4493', '4587', '4595', '4747', '48', '49', '4906', '4915', '50', '5000000', '500000000', '5000000000', '5012', '5023', '5192', '5286', '5356', '5389', '5451', '5556', '5584', '58', '5840', '5873', '5903', '60', '6055', '62', '6237', '6240', '6270', '6272', '6414', '65', '6510', '6537', '66', '67', '6703', '6793', '68', '6872', '69', '6969', '6979', '70', '7158', '7185', '72', '7259', '7417', '7427', '7469', '75', '7533', '7552', '7562', '76', '7689', '77', '7756', '79', '7982', '80', '8113', '8148', '8156', '8189', '8190', '83', '8312', '8315', '8384', '8403', '8407', '8421', '8446744073709551610', '8459', '8466', '8488', '85', '8514', '8571', '8594', '8635', '8666', '87', '88', '8899', '90', '9067', '9173', '9198', '9254', '9255', '9323', '9354', '9627', '9643', '9660', '97', '99', '9981', '\_\_time\_\_', 'abcdefg', 'able', 'according', 'admin', 'age', 'ago', 'all', 'all\_users', 'almost', 'already', 'also', 'among', 'analyse', 'and', 'apos', 'area', 'around', 'as', 'asc', 'at', 'avg', 'back', 'banner', 'benchmark', 'better', 'between', 'boolean', 'building', 'business', 'but', 'by', 'call', 'calle', 'came', 'car', 'care', 'case', 'cast', 'cdata', 'cent', 'change', 'char', 'character\_sets', 'child', 'children', 'chr', 'church', 'city', 'clear', 'collations', 'column\_name', 'columns', 'come', 'company', 'concat', 'convert', 'could', 'count', 'country', 'court', 'crypt\_key', 'ctxsys', 'customername', 'customers', 'data', 'database', 'day', 'db', 'dbms\_pipe', 'dbms\_utility', 'de', 'del', 'delay', 'delete', 'desc', 'devices', 'distinct', 'doctor', 'domain', 'domains', 'drithsx', 'drop', 'dual', 'el', 'else', 'elt', 'employeeid', 'employees', 'end', 'europe', 'even', 'evening', 'every', 'exchange', 'exec', 'exp', 'extractvalue', 'face', 'fetch', 'field', 'fields', 'first', 'five', 'flight', 'floor', 'following', 'for', 'foreign', 'form', 'former', 'found', 'four', 'from', 'full', 'function', 'functions', 'fzno', 'gcrr', 'generate\_series', 'german', 'germany', 'get', 'get\_host\_address', 'good', 'government', 'great', 'group', 'half', 'having', 'he', 'health', 'hex', 'high', 'home', 'house', 'however', 'id', 'if', 'iif', 'in', 'including', 'information', 'information\_schema', 'inner', 'insert', 'int', 'into', 'is', 'it', 'join', 'juan', 'keep', 'know', 'la', 'last', 'law', 'least', 'left', 'life', 'like', 'limit', 'line', 'local', 'long', 'los', 'made', 'major', 'make', 'make\_set', 'man', 'many', 'market', 'master', 'may', 'md5', 'men', 'meta\_id', 'meta\_key', 'meta\_value', 'million', 'min', 'mode', 'money', 'month', 'mr', 'much', 'must', 'mysql', 'name', 'national', 'need', 'new', 'news', 'next', 'not', 'null', 'number', 'numeric', 'office', 'old', 'on', 'one', 'only', 'onlyselect', 'option\_name', 'option\_value', 'or', 'order', 'orders', 'ordersinner', 'ordersright', 'outer', 'part', 'party', 'passengers', 'password', 'past', 'pay', 'people', 'per', 'percent', 'pg\_sleep', 'place', 'police', 'possible', 'post\_id', 'president', 'price', 'procedure', 'public', 'qqpjq', 'quot', 'qzvzq', 'rand', 'randomblob', 'rdb', 'receive\_message', 'regexp\_substring', 'repeat', 'result', 'right', 'rlike', 'row', 'rownum', 'rows', 'safety', 'said', 'san', 'santa', 'say', 'says', 'school', 'sddo', 'second', 'select', 'service', 'set', 'several', 'sgvo', 'since', 'sleep', 'sn', 'snowden', 'something', 'south', 'speech', 'sqlid\_to\_sqlhash', 'srmq', 'state', 'statement', 'still', 'sum', 'sys', 'sys1', 'sys2', 'sys3', 'sys4', 'sys5', 'sys6', 'sys7', 'sysdatabases', 'sysibm', 'systables', 'system', 'system\_users', 'sysusers', 't1', 't2', 't3', 't4', 't5', 'table\_name', 'tables', 'take', 'taken', 'team', 'text', 'the', 'then', 'there', 'they', 'think', 'third', 'this', 'three', 'thursday', 'thus', 'time', 'to', 'today', 'told', 'top', 'town', 'traffic', 'travel', 'tt', 'two', 'types', 'union', 'update', 'updatexml', 'upper', 'us', 'use', 'username', 'users', 'utl\_inaddr', 'values', 'version', 'view', 'vwyq', 'waitfor', 'want', 'way', 'we', 'week', 'well', 'when', 'where', 'whether', 'within', 'without', 'work', 'world', 'would', 'wp\_options', 'wp\_postmeta', 'wp\_posts', 'xmltype', 'ydpu', 'year', 'years']

**3.3.2 Autoencoder Model for extraction viewing**

import numpy as np

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

# Your original 544 input feature names (replace with actual list)

#feature\_names = ['000', '01', '06', '07', '09', '10', '1', '1=1', '10=10', '1234', '20', '2000', '2002', '2003', '2004', '2005', '2006', '2007', '2008', '2009', '2010', '2011', '2012', '2013', '2014', '2015', '2016', '2017', '2018', '2019', '2020', '2021', '2022', '2023', '2024', '30', '31', '40', '50', '60', '70', '80', '81', '82', '83', '84', '85', '86', '87', '88', '89', '90', 'admin', 'after', 'all', 'alter', 'and', 'as', 'asc', 'backup', 'between', 'by', 'char', 'column', 'convert', 'create', 'current\_user', 'database', 'declare', 'default', 'delete', 'desc', 'distinct', 'drop', 'else', 'end', 'exec', 'exists', 'false', 'fetch', 'for', 'from', 'grant', 'group', 'having', 'if', 'in', 'insert', 'intersect', 'into', 'is', 'join', 'left', 'like', 'limit', 'login', 'mid', 'name', 'not', 'null', 'on', 'or', 'order', 'outer', 'password', 'procedure', 'rename', 'right', 'rollback', 'row', 'schema', 'select', 'set', 'sleep', 'sys', 'table', 'then', 'to', 'top', 'true', 'union', 'update', 'use', 'user', 'values', 'varchar', 'view', 'waitfor', 'when', 'where', 'with', 'year', ...] # Full 544

# === Define Autoencoder ===

input\_layer = Input(shape=(544,))

encoded = Dense(128, activation='relu')(input\_layer)

encoded = Dense(64, activation='relu')(encoded)

decoded = Dense(128, activation='relu')(encoded)

decoded = Dense(544, activation='linear')(decoded)

autoencoder = Model(input\_layer, decoded)

autoencoder.compile(optimizer=Adam(learning\_rate=0.001), loss='mean\_squared\_error')

encoder = Model(input\_layer, encoded)

# Callbacks

early\_stopping = EarlyStopping(monitor='val\_loss', patience=5, restore\_best\_weights=True)

lr\_scheduler = ReduceLROnPlateau(monitor='val\_loss', factor=0.5, patience=3, min\_lr=1e-6)

# === Extract Weights from Encoder Layers ===

# Layer 1: input → 128

weights\_input\_to\_128 = autoencoder.layers[1].get\_weights()[0] # shape (544, 128)

# Layer 2: 128 → 64

weights\_128\_to\_64 = autoencoder.layers[2].get\_weights()[0] # shape (128, 64)

# Combine weights to get input → encoded

combined\_weights = np.dot(weights\_input\_to\_128, weights\_128\_to\_64) # shape (544, 64)

# === Print Top 5 Keywords for Each Encoded Feature ===

for i in range(64):

feature\_weights = combined\_weights[:, i]

top\_indices = np.argsort(feature\_weights)[-5:][::-1]

top\_keywords = [feature\_names[j] for j in top\_indices]

print(f"Encoded Feature {i + 1}: {', '.join(top\_keywords)}")



input\_layer = Input(shape=(544,))

encoded = Dense(128, activation='relu')(input\_layer)

encoded = Dense(64, activation='relu')(encoded)

decoded = Dense(128, activation='relu')(encoded)

decoded = Dense(544, activation='linear')(decoded)

autoencoder = Model(input\_layer, decoded)

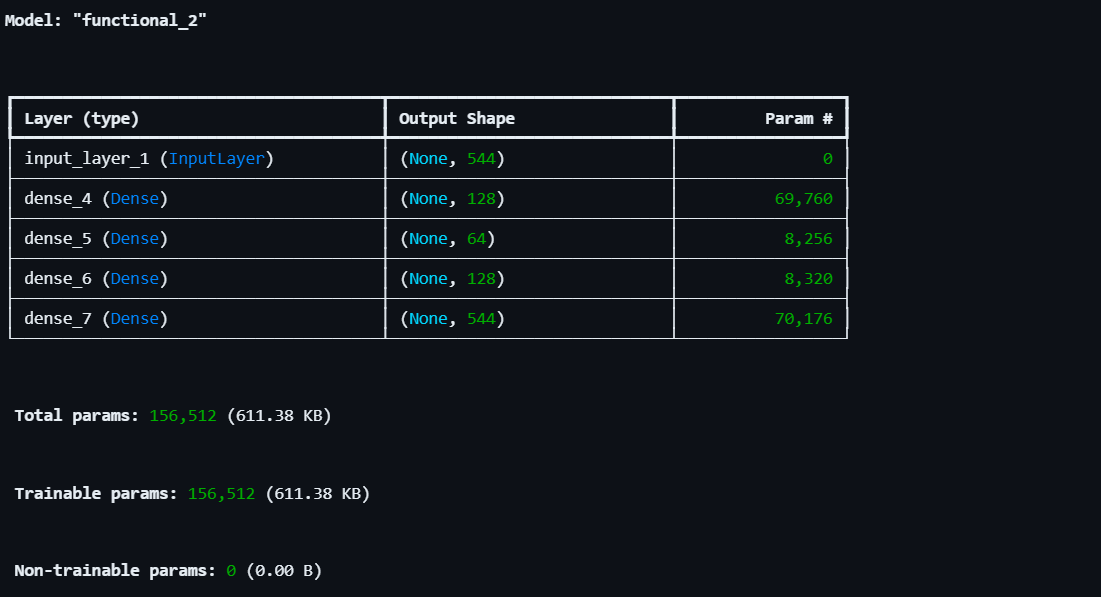
autoencoder.compile(optimizer=Adam(learning\_rate=0.001), loss='mean\_squared\_error')

encoder = Model(input\_layer, encoded)

autoencoder.summary()

early\_stopping = EarlyStopping(monitor='val\_loss', patience=5, restore\_best\_weights=True)

lr\_scheduler = ReduceLROnPlateau(monitor='val\_loss', factor=0.5, patience=3, min\_lr=1e-6)



**3.3.3 Training Autoencoder for Feature Encoding**

history = autoencoder.fit(X\_train, X\_train,

epochs=20, batch\_size=128,

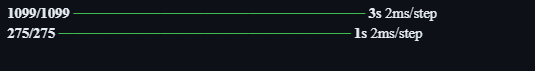
validation\_data=(X\_test, X\_test),

verbose=1,

callbacks=[early\_stopping, lr\_scheduler])

X\_train\_enc = encoder.predict(X\_train)

X\_val\_enc = encoder.predict(X\_test)



**3.3.4 XGBoost Model Training and Accuracy Evaluation**

xgb = XGBClassifier(n\_estimators=100, max\_depth=10, learning\_rate=0.05, random\_state=42)

xgb.fit(X\_train\_enc, y\_train)

y\_train\_pred = xgb.predict(X\_train\_enc)

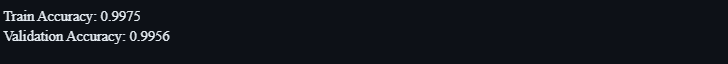
train\_accuracy = accuracy\_score(y\_train, y\_train\_pred)

y\_val\_pred = xgb.predict(X\_val\_enc)

val\_accuracy = accuracy\_score(y\_test, y\_val\_pred)

print(f"Train Accuracy: {train\_accuracy:.4f}")

print(f"Validation Accuracy: {val\_accuracy:.4f}")



**3.3.5 Autoencoder Training and Validation Loss Plot**

plt.figure(figsize=(6, 4))

plt.plot(history.history['loss'], label='Training Loss')

plt.plot(history.history['val\_loss'], label='Validation Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.title('Autoencoder Training Loss')

plt.show()

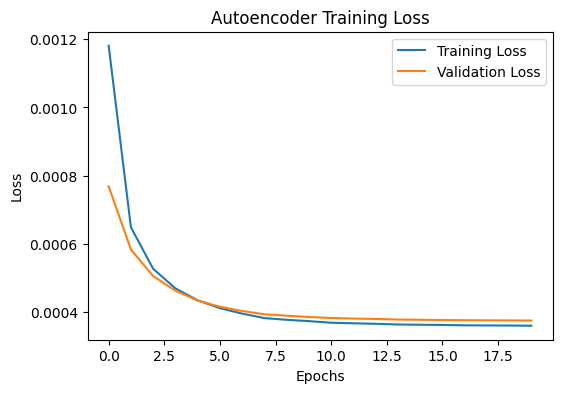


Fig 3.3.5 Autoencoder Training and Validation Loss Plot

**3.4 MODULE 3: MODEL PERFORMANCES**

**3.4.1 XGBoost Model Confusion Matrix**

# Get predictions on test set

y\_pred = xgb.predict(X\_val\_enc)

# Confusion Matrix Visualization

cm = confusion\_matrix(y\_test, y\_pred)

plt.figure(figsize=(5, 4))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', linewidths=1.5, linecolor='black')

plt.xlabel("Predicted")

plt.ylabel("Actual")

plt.title("XGBoost Confusion Matrix")

plt.show()

# Classification report with a smaller font size

report = classification\_report(y\_test, y\_pred, target\_names=["Normal", "Malicious"])

print("\nClassification Report:\n")

print(report)

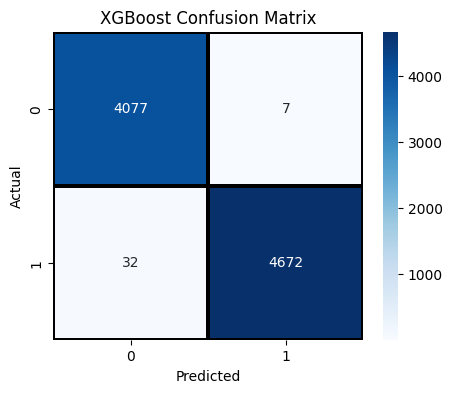


Fig 3.4.1 Confusion Matrix

**3.4.2 Classification Report**

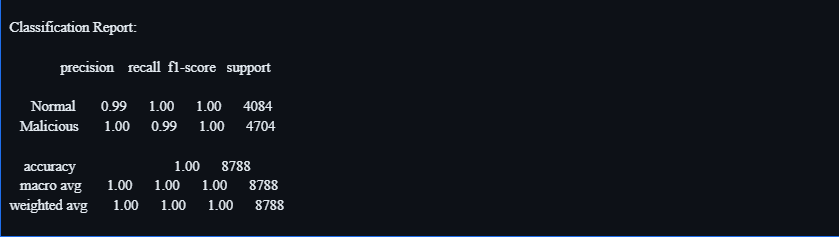


Fig 3.4.2 Classfication Report

**3.4.3 K-Fold Cross Validation and Statistical Significance Testing**

# K-Fold Cross Validation

kf = KFold(n\_splits=10, shuffle=True, random\_state=42)

kfold\_scores = cross\_val\_score(xgb, X\_scaled, y, cv=kf)

print(f'K-Fold Accuracy: {np.mean(kfold\_scores):.4f} ± {np.std(kfold\_scores):.4f}')

# Statistical T-Test (Requires Numeric Labels)

\_, p\_value = ttest\_rel(y\_test, y\_val\_pred)

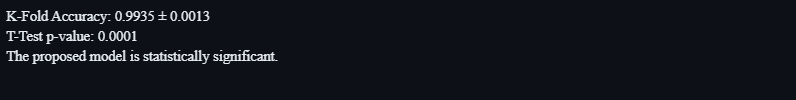
print(f'T-Test p-value: {p\_value:.4f}')

if p\_value < 0.05:

print("The proposed model is statistically significant.")

else:

print("No statistical significance found.")



**3.4.4 Model Performance Evaluation Metrics**

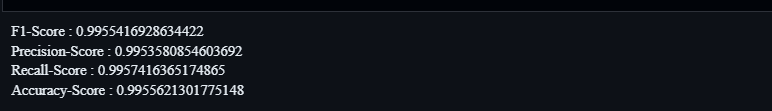
from sklearn.metrics import f1\_score,precision\_score,recall\_score,accuracy\_score

print("F1-Score :", f1\_score(y\_test, y\_val\_pred, average='macro'))

print("Precision-Score :", precision\_score(y\_test, y\_val\_pred, average='macro'))

print("Recall-Score :", recall\_score(y\_test, y\_val\_pred, average='macro'))

print("Accuracy-Score :", accuracy\_score(y\_test, y\_val\_pred))



**3.5 MODULE 4: PREDICTIONS**

**3.5.1 SQL Injection Prediction**

def predict\_query(query):

import numpy as np

# Preprocess query: convert to lowercase, remove extra spaces

query\_cleaned = query.strip().lower()

# Vectorize using the same TF-IDF vectorizer

query\_vector = vectorizer.transform([query\_cleaned]).toarray()

# Encode using the trained encoder

query\_encoded = encoder.predict(query\_vector)

# Predict using trained XGBoost model

prediction = xgb.predict(query\_encoded)

# Convert prediction to list and print

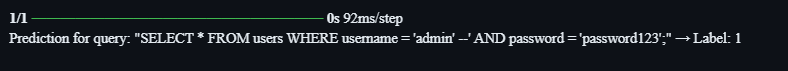
return prediction[0]

# Example usage:

test\_query = "SELECT \* FROM users WHERE username = 'admin' --' AND password = 'password123';"

predicted\_label = predict\_query(test\_query)

print(f"Prediction for query: \"{test\_query}\" → Label: {predicted\_label}")



**3.5.2 Gradio Interface for SQL Injection Detection**

import gradio as gr

import joblib

import numpy as np

# Load pre-trained models

xgb = joblib.load("C:\\Users\\koush\\OneDrive\\Desktop\\MiniProject\\xgb\_model.pkl")

vectorizer = joblib.load("C:\\Users\\koush\\OneDrive\\Desktop\\MiniProject\\tfidf\_vectorizer.pkl")

encoder = joblib.load("C:\\Users\\koush\\OneDrive\\Desktop\\MiniProject\\encoder.pkl")

# Prediction function with confidence and HTML output

def predict(query):

query\_cleaned = query.strip().lower()

query\_vector = vectorizer.transform([query\_cleaned]).toarray()

query\_encoded = encoder.predict(query\_vector)

prediction = xgb.predict(query\_encoded)

proba = xgb.predict\_proba(query\_encoded)[0][1] # probability of class 1 (Malicious)

# Confidence levels

confidence = round(proba \* 100, 2)

# Risk Level Based on Confidence

if prediction[0] == 1: # Malicious

label\_html = "<span style='color:red; font-weight:bold;'>🛑 Malicious</span>"

if confidence > 70:

risk = "🔴 High"

elif confidence > 40:

risk = "🟡 Medium"

else:

risk = "🟢 Low"

else: # Normal

label\_html = "<span style='color:lightgreen; font-weight:bold;'>✅ Normal</span>"

if confidence > 70:

risk = "🔴 High"

elif confidence > 40:

risk = "🟡 Medium"

else:

risk = "🟢 Low"

return label\_html, confidence, risk

# Build the UI

with gr.Blocks(title="SQL Injection Detection", theme=gr.themes.Base(primary\_hue="red", secondary\_hue="gray")) as demo:

gr.Markdown("""

<h1 style="color:#e60000;">🔍 SQL Injection Detection</h1>

<p style="color:#f2f2f2;">

Detect potentially malicious SQL input using a trained machine learning model.<br>

Helps safeguard applications from unauthorized database manipulation.

</p>

""")

with gr.Row():

with gr.Column():

input\_text = gr.Textbox(label="🧾 Enter SQL Query", placeholder="e.g. ' OR 1=1 --", lines=1)

example\_inputs = gr.Examples(

examples=[

"' OR 1=1 --",

"SELECT \* FROM users WHERE username='SASTRA'",

"1) or benchmark(10000000,MD5(1))#",

"Normal user input"

],

inputs=input\_text

)

submit\_btn = gr.Button("🚨 Detect")

clear\_btn = gr.Button("🧹 Clear")

with gr.Column():

result = gr.HTML(label="🔍 Detection Result")

confidence\_slider = gr.Slider(minimum=0, maximum=100, label="📊 Confidence (%)", interactive=False)

risk\_level = gr.Textbox(label="📊 Risk Level", interactive=False)

submit\_btn.click(fn=predict, inputs=input\_text, outputs=[result, confidence\_slider, risk\_level])

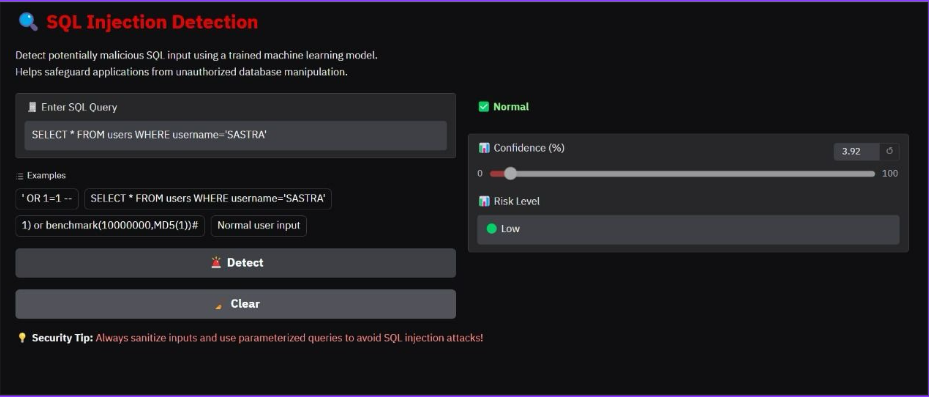
clear\_btn.click(fn=lambda: ("", 0, ""), inputs=None, outputs=[result, confidence\_slider, risk\_level])

gr.Markdown("<p style='color: #ff9999;'>💡 <strong>Security Tip:</strong> Always sanitize inputs and use parameterized queries to avoid SQL injection attacks!</p>")

# Launch

demo.launch(share=True)

**3.6 OUTPUT SNAPSHOTS**



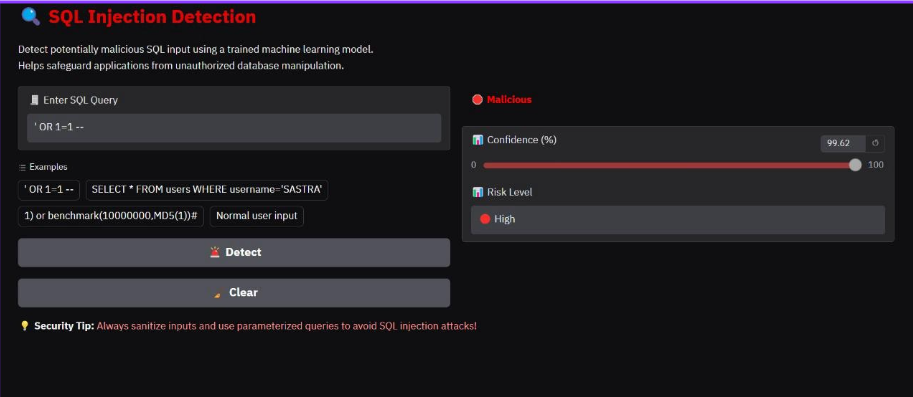


Fig 3.6 Prediction SQLi

**CHAPTER 4**

**4.1 CONCLUSION**

The paper suggested an automatic and timely process for SQL injection attack detection. We propose here a new concept named AE-Net to apply automatic feature engineering. AE-Net is capable of deriving high-level deep features from textual SQL data, which are then provided as inputs to machine learning algorithms for performance tests. Four new machine learning and deep learning-based methods are used for these performance tests. Comprehensive experimental testing shows that the extreme gradient boosting classifier performs better than previous research with a staggering K-fold accuracy value of 0.99 for SQL injection detection. Each learning method used is further improved with hyperparameter tuning and validated through K-fold cross-validation. Statistical t-test analysis is also used to test performance differences.

**CHAPTER 5**

**REFERENCES**

* M. Nasereddin, A. ALKhamaiseh, M. Qasaimeh, and R. Al-Qassas, ‘‘A systematic review of detection and prevention techniques of SQL injection attacks,’’ Inf. Secur. J., Global Perspective, vol. 32, no. 4, pp. 252–265, Jul. 2023.
* I. S. Crespo-Martínez, A. Campazas-Vega, Á. M. Guerrero-Higueras, C. Álvarez-Aparicio, and C. Fernández-Llamas, ‘‘Impact of the keep-alive parameter on SQL injection attack detection in network flow data,’’ in Proc. Comput. Intell. Secur. Inf. Syst. Conf. Cham, Switzerland: Springer, 2023, pp. 69–78.
* A. Arshad, M. Jabeen, S. Ubaid, A. Raza, L. Abualigah, K. Aldiabat, and H. Jia, ‘‘A novel ensemble method for enhancing Internet of Things device security against botnet attacks,’’ Decis. Anal. J., vol. 8, Sep. 2023, Art. no. 100307.
* F. Rustam, A. Raza, I. Ashraf, and A. D. Jurcut, ‘‘Deep ensemble-based efficient framework for network attack detection,’’ in Proc. 21st Medit. Commun. Comput. Netw. Conf. (MedComNet), Jun. 2023, pp. 1–10.
* R. Madhvan and M. F. Zolkipli, ‘‘An overview of malware injection attacks: Techniques, impacts, and countermeasures,’’ Borneo Int. J. eISSN, vol. 6, no. 3, pp. 22–30, 2023.
* T. Sheth, J. Anap, H. Patel, N. Singh, and R. R. B, ‘‘Detection of SQL injection attacks by giving a priori to Q-learning agents,’’ in Proc. IEEE IAS Global Conf. Emerg. Technol. (GlobConET), May 2023, pp. 1–6.
* M. Alghawazi, D. Alghazzawi, and S. Alarifi, ‘‘Detection of SQL injection attack using machine learning techniques: A systematic literature review,’’ J. Cybersecur. Privacy, vol. 2, no. 4, pp. 764–777, Sep. 2022.
* P. Roy, R. Kumar, and P. Rani, ‘‘SQL injection attack detection by machine learning classifier,’’ in Proc. Int. Conf. Appl. Artif. Intell. Comput. (ICAAIC), May 2022, pp. 394–400.
* A. M. A. Badri and S. Alouneh, ‘‘Detection of malicious requests to protect web applications and DNS servers against SQL injection using machine learning,’’ in Proc. Int. Conf. Intell. Comput., Commun., Netw. Services (ICCNS), Jun. 2023, pp. 5–11.
* K. Singh, S. Kokardekar, G. Khonde, P. Dekate, N. Badkas, and S. Lachure, ‘‘Cloud engineering-based on machine learning model for SQL injection attack,’’ in Proc. Int. Conf. Commun., Circuits, Syst. (IC3S), May 2023, pp. 1–6.
* J. Misquitta and S. Asha, ‘‘SQL injection detection using machine learning and convolutional neural networks,’’ in Proc. 5th Int. Conf. Smart Syst. Inventive Technol. (ICSSIT), Jan. 2023, pp. 1262–1266

**CHAPTER 6**

**APPENDIX – BASE PAPER**

1. **Title:**  AE-Net : Novel Autoencoder-Based Deep Features for SQL Injection Attack Detection
2. **Publisher**: IEEE
3. **Year**: 2023
4. **Journal**: IEEE Access
5. **Indexing:**
6. **Base Paper URL:** <https://doi.org/10.1109/ACCESS.2023.3337645>