DETECTION AND PREVENTION OF SQL INJECTION ATTACKS WITH MACHINE LEARNING

GAN YU EN

XIAMEN UNIVERSITY MALAYSIA

2023

XIAMEN UNIVERSITY MALAYSIA

厦門大學馬來西亞分校

FINAL YEAR PROJECT REPORT

DETECTION AND PREVENTION OF SQL INJECTION ATTACKS WITH MACHINE LEARNING

NAME OF STUDENT	:	GAN YU EN
STUDENT ID	:	CST1904699
SCHOOL/ FACULTY	:	SCHOOL OF COMPUTING AND DATA SCIENCE
PROGRAMME	:	BACHELOR OF ENGINEERING IN COMPUTER SCIENCE AND TECHNOLOGY (HONOURS)
INTAKE	:	2019/04
SUPERVISOR	:	DR MAHDI H. MIRAZ ASSOCIATE PROFESSOR

JANUARY 2023

DECLARATION

I hereby declare that this project report is based on my original work except for citations and quotations which have been duly acknowledged. I also declare that it has not been previously and concurrently submitted for any other degree or award at Xiamen University Malaysia or other institutions.

Signature : 

Name : Gan Yu En

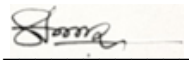
ID No. : CST1904699

Date : 09/12/2022

APPROVAL FOR SUBMISSION

I certify that this project report entitled "**DETECTION AND PREVENTION OF SQL INJECTION ATTACKS WITH MACHINE LEARNING**" that was prepared by GAN YU EN has met the required standard for submission in partial fulfilment of the requirements for the award of Bachelor of Engineering in Computer Science and Technology (Honours) at Xiamen University Malaysia.

Approved by,

Signature : 

Supervisor : Dr Mahdi H. Miraz

Date : 08/12/2022

ACKNOWLEDGEMENTS

I would like to thank all who have contributed to the successful completion of this project. I would like to express my gratitude to my research supervisor, Dr Mahdi H. Miraz for his invaluable advice, guidance, and his enormous patience throughout the development of the research.

In addition, I would also like to express my gratitude to my and friends who have helped and given me encouragement.

ABSTRACT

Structured Query Language (SQL) injection attack is one of the most vital web application security issues, as it can cause data loss, data breach and data modification which is not intended by the developers. An attacker can perform injection of SQL queries to trick the database to perform malicious requests, such as retrieving unauthorized data, or deleting data records. Web applications can be exposed to SQL injection vulnerabilities due to weak coding practices, or unintentional ignorance. Developers use many prevention and detection measures to minimize the risk of exposing SQL injection vulnerabilities, such as using parameterization and dynamic approach in detecting malicious queries. With the application of emerging machine learning technologies, predicting, and classifying data, including detecting SQL injected queries, has become very efficient. Hence, this project aims to design, develop, and evaluate approach to detect and prevent SQL injections using machine learning.

In this final year project, a SQL injection detection system has been integrated to an exploratory web application in order to evaluate how well the detection system can perform. To determine a machine learning model to be used for the detection system, different machine learning models including Multinomial NB, Random Forest Classifier, and AdaBoost Classifier have been evaluated. More than 3,900 datasets, taken from Kaggle have been used to train and evaluate the machine learning models. The evaluation process has also included the type of data vectorization method used, such as CountVectorizer and TfidfVectorizer, to understand how vectorization methods can affect the classifier's predicting ability. Based on the results achieved, Random Forest Classifier with TfidfVectorizer considering all tokens provides the best accuracy of 99.10%.

The best evaluated model identified, which is Random Forest Classifier, has then been used to develop a SQL injection detection system, which has also been integrated to a website application to determine its ability to detect and prevent SQL injections from the user inputs. The detection system also has a self-update function for unseen tokens, where it is possible to encounter unknown tokens from user inputs. Based on the evaluation results achieved through testing proposed ML based SQL injection

detection and prevention method on the exploratory website, it is able to identify injection statements in user log in, and sign up inputs. Attempts of SQL injection on the website are successfully identified and blocked from executing user requests. As for unseen inputs, they are saved temporarily and updated to dataset on system restart, without affecting the website functionalities.

Keywords – SQL injection, machine learning, web application vulnerabilities

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CMS	Content Management System
KNN	K-Nearest Neighbor
NB	Naïve Bayes
NN	Neural Network
SQL	Structure Query Language
SVM	Support Vector Machine

CHAPTER 1

INTRODUCTION

1.1 Background

Ever since the 1950s, digitization has never stopped transforming the world to a point where it almost completely changed the way humans live (Tarpey, 2020). The volume and diffusion of digital information almost exceeds non-digital form as it is more convenient and efficient. This has enforced digital transformation to many companies and businesses. Digital transformation unlocked many net networking possibilities, such as exchanging data and enabling cooperation between different people (Schallmo *et al*., 2017). Therefore, web applications play a very important role in these operations. However, this also leads to many possibilities of cyber-attacks, where attackers are able to exploit valuable information or maliciously alter information in the database.

Web-based attacks can also be considered as application layer intrusion, where application layer is the layer that users interact with (TrustNet, n.d.). In the application layer, it is able to assist in communication such as sending and receiving data, also, accessing data. As web applications become a common tool used by many, the number of application layer attacks are on the rise (Saha & Sanyal, 2014). These attacks pose a serious threat to individual users and especially companies that depend on web services and platforms to operate their businesses. For example, cloud computing is becoming a common trend to act as an internet-based platform for many businesses. These cloud-based services are able to greatly reduce maintenance and management cost as conventional on-site deployment is not needed (Tripathy *et al*., 2020). Yet, due to outsourced data in cloud services, it has a high risk of security intrusion.

One of the most vital application layer attacks include Structured Query Language (SQL) injection. SQL injection attack is a form of cyber-attack where it is able to pass malicious queries into input data of web applications as shown in Figure 1.1. Since

2017, SQL injection attack is still on the Open Web Application Security Project (OWASP) Top 10 list of website security vulnerabilities (OWASP, n.d.-b). SQL injection usually happens in interactive web applications, where it accepts user inputs and processes the user's request. Therefore, an attacker can perform an injection of SQL queries that is not intended by the developer, this will trick the database to perform malicious requests (Tasevski & Jakimoski, 2020). A successful SQL injection attack will allow the attacker to obtain or insert information to and from the database. This poses a serious threat, as the targeted application or organization will suffer data breach and loss of sensitive data, causing negative effects on business revenues and reputations.

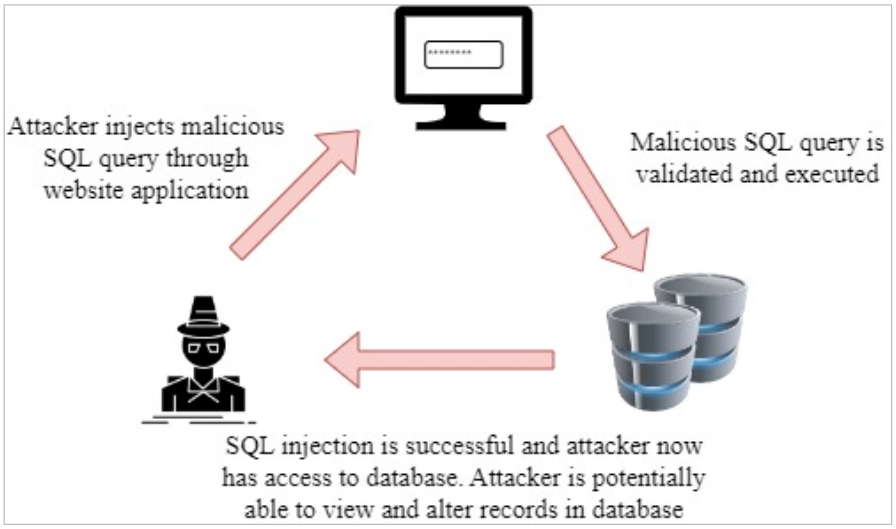


Figure 1.1: SQL Injection, adapted from[1]

Intrusion Detection Systems can be used to enhance the security of a web application (Katole *et al*., 2018). However, challenges arise in how to prevent and detect SQL injections, where the database is unable to differentiate a valid input query and a malicious input query. There will be a vulnerability when the user input is not properly filtered before executing it. SQL queries can be static or dynamic. Static query is where

[1] *SQL Injection Attacks (SQLi) — Web-based Application Security, Part 4*, by S. Oza, n.d., Spanning (https://spanning.com/blog/sql-injection-attacks-web-based-application-security-part-4/)

the processed details database query is retrieved from the application, while dynamic query is generated from static queries through user inputs (Das *et al*., 2019). Hence, one dynamic query is generated by each user depending on their input values. Yet, a user may be able to maliciously craft an input, and still be considered a valid dynamic query. This will result in the malfunctioning of an application, also affecting its information integrity.

As time goes on, malicious users tend to modify old attack tactics or injection methods. Hence, new attacking techniques will continue to emerge (Ross, 2018). This causes companies and security firms to put great effort in order to prevent and mitigate web application attacks. However, prevention measures can be very complicated. When developers implement controls to the source code, malicious users will always attempt to find alternatives to bypass the controls (Tripathy *et al*., 2020). Hence, as attacking methods evolve, prevention methods also have to be updated and modified so that the effectiveness of prevention measures will be maintained. Also, there can be a possibility where the SQL injection prevention or other security vulnerabilities prevention were left out (Roy *et al*., 2022). Due to the large source code, it can be difficult to spot the security defect in the program. Thus, SQL injection is still a potentially dangerous threat to web applications and application programs.

Various techniques were suggested in order to prevent SQL injection attacks (Mehta *et al*., 2015; Das *et al*., 2019; Jothi *et al*., 2021). The prevention approach can generally be grouped into three groups, which are static, dynamic, and hybrid (Hasan *et al*., 2019). For static approach, it compares the SQL query with a benign query, then determines if there are any mismatches within the query. As for dynamic approach, the system is able to dynamically determine if the query is an SQL injection, such as by using parse trees of SQL query structure (Bockermann *et al*., 2009). As for the hybrid approach, it is the combination of static and dynamic analysis. This is due to the static approach verifies the input parameter type, but unable to identify attacks with correct

input type; while dynamic approach is able to check for vulnerabilities, but unable to identify all types of SQL injection attacks (Minhas & Kumar, 2013). Hence, hybrid approach gives the best of both worlds.

Out of the many techniques recommended, machine learning is known to be a very effective approach to detect SQL injections (Hasan *et al*., 2019). Machine learning mainly focuses on the utilization of data and algorithms to mimic humans' learning process. This caused machine learning to become a significant component to make predictions or classifications of data using algorithms and through statistical methods (IBM Cloud Education, 2020). Hence, machine learning is able to notably improve detection and prevention systems by differentiating benign SQL queries with malicious queries. As the system will also be able to self-improve through learning, it will also be able to determine future unknown SQL injection attacks.

1.2 Aims and Objectives

This paper aims to propose a machine learning approach in detecting and preventing SQL injection attacks. By analysing user input from a simple login web application, the system will identify if the user input is classified as SQL injection or benign input. Also, the system has to be designed in a way, so that will able to learn new inputs whenever it encounters an unknown input. Once malicious input is detected, the program will not allow execution of the query to perform what the user has requested. Hence, SQL injection will be prevented. To achieve these aims, there are a few objectives including:

- Explore the theories of SQL injection and SQL queries to comprehend the state-of-the-art vulnerabilities in SQL-driven web applications.
- Evaluation and selection of appropriate SQL injection dataset in order to train the machine learning algorithm in predicting benign inputs as well as malicious inputs.
- Evaluation of performance and accuracy of different machine learning models to determine a suitable model for SQL injection detection system.
- Integrate the developed SQL injection detection and prevention system to a exploratory web application.

CHAPTER 2

LITERATURE REVIEW

The booming development of the Internet has caused web applications to become widely common. People use web applications almost daily to perform transactions, share data, or store data. To manage data in web application databases, Structured Query Language (SQL) is used. SQL is a command-based control language used to manage content, and serve as back-end of a web application (Hasan *et al*., 2019). SQL injection happens when malicious data are input or "injected" into SQL commands, dynamic queries, or stored procedures. This will enable attackers to go against the Database Management System (DBMS) by executing malicious SQL statements (Fioravanti & Mayron, 2014). An example of SQL injection can be shown using the example application in Figure 2.1. The application takes username of users and output their respective ID number. User can only access their data with the correct username input. However, a malicious user can use an input of " OR 1=1", to generate a SQL query of:

$$SELECT\ IDNumber\ FROM\ UserTable\ WHERE\ Username = OR\ 1 = 1$$

This query has a condition that is always true. Hence, it will select all records from the database, resulting in a successful SQL injection attack.

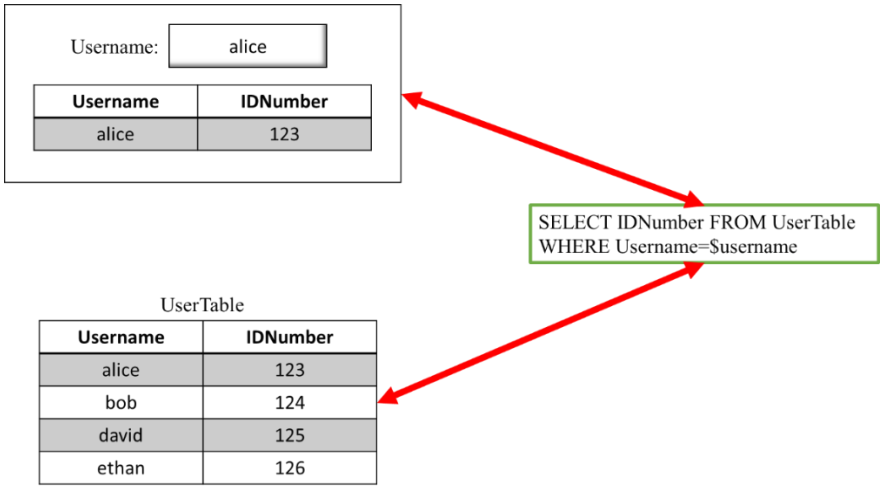


Figure 2.1: Example of User Interaction with Database through SQL

In order to mitigate SQL injection attacks, many studies have been done regarding the detection and prevention of SQL injections. This section reviews few of the related studies. Resources of articles and journals are mainly from IEEE (Institute of Electrical and Electronics Engineers) Xplore digital library (IEEE, n.d.) and ACM (Association for Computing Machinery) digital library (ACM, n.d.). Both sources provide trusted and comprehensive research journals, articles, and conference papers. They offer access to innovative ideas and relevant research papers. Also, equipped with user friendly search interface, and well-organized filtering functions to aid searching exact article efficiently. Key words of "SQL Injection" and "Machine Learning" are used for both IEEE Xplore and ACM digital library, with total of 80 and 294 results found respectively. Inclusion criteria to review the articles are related papers within the past 12 years, and related to detection and prevention of SQL injection attacks. Mainly focusing on machine learning detection methods, and some on basic principles of detection. As for exclusion criteria are outdated journal articles, and unrelated to SQL queries. Review and survey papers are also not included as more detailed and original papers is preferred.

One of the studies is proposed by Hasan *et al*. (2019), which is an approach to detect SQL injection attacks through machine learning. The system is developed using MATLAB classification algorithms (MathWorks, n.d.), in order to analyse between SQL injection query and non-SQL injection query. Before training the classifiers, a dataset of injected and non-injected inputs is collected. Then, the dataset file is processed to extract each of the statement's features. The features extracted includes presence of comment character, presence of always true conditions and presence of keywords. These features are the indicators of the classifiers, allowing it to make good predictions. Hasan *et al*. (2019) is able to obtain an accuracy of 93.8% using the Ensemble Boosted Tress algorithm. This algorithm is able to handle various predictor variable type and accommodate missing data, hence, capable of fitting complex non-linear relationships (Elith *et al*., 2008). The system has the potential to achieve more

than 99% accuracy in determining injected inputs. However, there are only relatively small number of non-injected statements in the dataset, causing classification rate for non-injected inputs to be 64%. Therefore, the overall accuracy is slightly reduced. Regardless, the system is able to assure almost no SQL injection attacks will be falsely classified as non-injected statements.

Zhang (2019) conducted a study to identify SQL injection vulnerabilities in files through machine learning classifiers. Dataset of vulnerable Hypertext Pre-processor (PHP) files are collected from different sources, which are National Institute of Standards and Technology's Software Assurance Reference Dataset Project (SARD) and National Vulnerability Database (NVD) (NIST, n.d.-a). From the sources, Zhang (2019) is able to gather a total of 8,800 non-vulnerable and 950 vulnerable samples. The dataset is then processed by a Python program to determine if a dataset sample file consists of functions to validate and sanitize input from users. The use of input validation and sanitization is able to greatly prevent use of characters that will initiate a SQL injection. This is due to input validation checks the received input from user with a standard defined input, which may include the use of parameters or regular expressions (Clarke, 2009). As for input sanitization, it can involve changing or refining input to avoid unwanted characters or incompatible characters being executed (Pollack, 2017). Hence, PHP files that contain more input validation and sanitization functions tend to be more secured from SQL injection attacks. Zhang (2019) used multiple machine learning algorithms from Scikit-Learn library (Scikit-learn, n.d.) for classical machine learning algorithms, and Keras library (Keras, n.d.) for deep learning algorithms. Among the algorithms used, Support Vector Machine (SVM) algorithm, achieved a high precision and accuracy of 98.6% and 95.4% respectively, but a low recall of 58.3%. On the other hand, Convolutional NN has precision, accuracy, and recall of 95.4%, 95.3%, and 59.9% respectively. Hence, deep learning algorithms provide a better overall performance for classifiers. Although, all models are only able to identify not more than 66% of overall vulnerable samples, they still have very low

false positive rates which still be able to provide some practicality for developers.

Tripathy *et al*. (2020) conducted a research on SQL injection detection for cloud Software as a Service (SaaS) using machine learning. Many critical or important applications are deployed to cloud computing environments by users or large organizations. A security vulnerability on these services may cause serious loss for businesses or individual users, consequently affecting the trust relationship between user and service provider. Hence, a detection system for SQL injection attacks is a significant component for cloud service providers and web application developers. Tripathy *et al*. (2020) proposed a solution to identify SQL injection using machine learning, which can be summarized in Figure 2.1. Few injection attack features are described to aid the machine learning algorithm, which includes SQL keywords, punctuation characters, mean number of bytes and length. The features ranking or weight are then evaluated by estimating chi-square between each feature. Using the features created, classifiers are trained with learning models. Results has shown that among the other classifiers, the Random Forest Classifier is able to achieve the highest accuracy of 99.8%. The Random Forest Classifier predicts the input by constructing decision trees with the subset of given training samples, then combining the output of multiple decision trees to produce a final prediction. With more features, the classifier models will also be able to produce higher accuracy predictions.

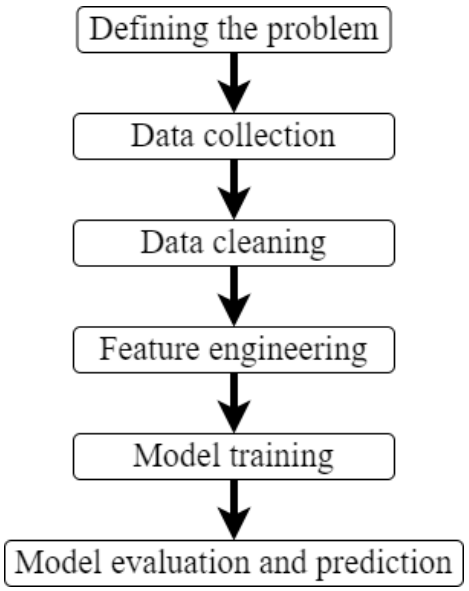


Figure 2.2: SQL Injection Detection Methodology, adapted from Tripathy *et al*. (2020)

Another machine learning SQL injection detection system proposed by Hosam *et al*. (2021) also uses machine learning models including Random Forest. After features of the dataset are extracted, machine learning models are trained and implemented by Scikit-learn library (Scikit-learn, n.d.). Notable classifier models used other than Random Forest, include Logistic Regression and Decision Tree. Logistic Regression is a machine learning approach to classify data or input by using logistic functions to determine its dependent variable. There will be usually two possible outcomes as the depedent variable usually involve two parts (Raj, 2020). For example, the given statement is an injected SQL query, or it is not an injected SQL query. However, it is possible to extend further for the classifier to predict more than two types of outcome, such as multinomial Logistic Regression or ordinal Logistic Regression. As for Decision Tree classifier, it makes predictions based on all given conditions and features, where the features are the internal nodes, conditions are the brances between the nodes, and the results are the leaf nodes of the tree (javaTpoint, n.d.). Comparing Decision Tree with Random Forest classifier, Decision Tree does not have a limit in tree depth, while Random Forest uses multiple decision trees and only have a depth of 2. Hosam

et al. (2021) is able to obtain 99.6% accuracy for Decision Tree classifier which is also the highest, followed by Logistic Regression classifier with 99.3% accuracy. However, Logistic Regression classifier is much recommended as it is able to make the highest accuracy predictions of 87.3% for unseen data, comparatively Decision Tree is only able to achieve accuracy of 85.0% for unseen data.

Sivasangari *et al*. (2021) proposed a machine learning algorithm to detect SQL injection attack using AdaBoost classifier. In 1996, Yoav Freund and Robert Schapire came out with AdaBoost classifier also known as Adaptive Boosting classifier. It is able to combine multiple low accuracy classifiers into one higher accuracy classifier, with a concept of setting weights and training the data in each iteration (Navlani, 2018). Each iteration will increase the accuracy of the classifier as shown in Figure 2.2.

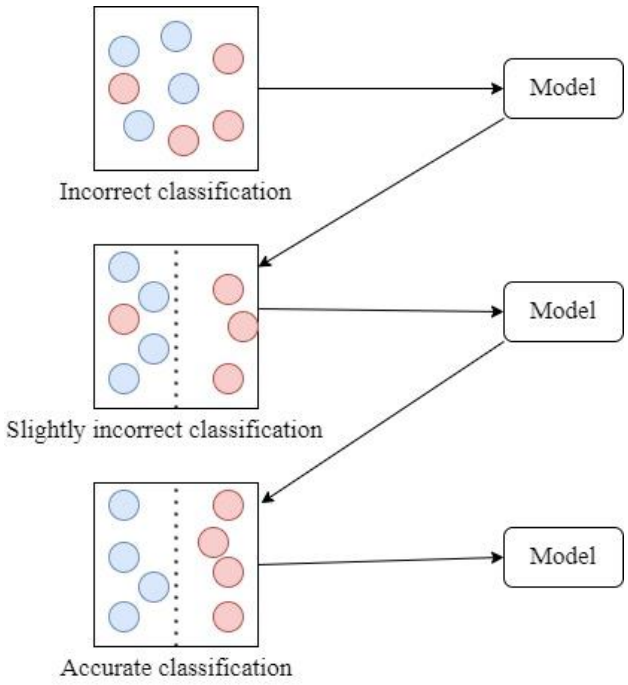


Figure 2.3: Boosting Algorithm, adapted from[2]

AdaBoost classifier is also less likely to be exposed to overfitting, hence it is selected

[2] *Boosting in Machine Learning | Boosting and AdaBoost*, by Raman, 2022, Geeks for Geeks (https://www.geeksforgeeks.org/boosting-in-machine-learning-boosting-and-adaboost/)

to be used in SQL injection detection machine learning algorithm (Sivasangari *et al*., 2021). When data is classified inaccurately, higher weights are given to the data that is classified incorrectly in order to correctly classify it in the next iteration. Hence, even with weak or low accuracy learners, it is able to produce a strong accuracy model through imrpoving during each step. After data collection and processing, Sivasangari *et al*. (2021) performed feature extraction using deep forest model to take in data of feature vectors. Through accuracy and precision analysis, it has shown that AdaBoost algorithm is able to surpass other classification algorithms such as K-Nearest Neighbor (KNN) and Support Vector Machine (SVM) algorithm. It is able to achieve both accuracy and precision more than 97%.

Kamuto and Soomlek (2016) conducted a study on SQL injection prediction on server-side scripting using machine learning. Research process includes validation of SQL syntax, extraction of dataset, train machine learning models, and verify prediction results. Data samples are collected from available Content Management System (CMS) applications, such as Drupal, Joomla, and WordPress. CMS applications helps in managing, storing, and modifying data on a website, hence, it consists of many HTML and SQL code suitable to be used as data samples. There are two types of variable, which include fixed variable that are written internally wihin the scripts and hardly involve SQL injection directly. Another type of variable is input variables, also known as dependent variable has a higher risk of being injected with malicious SQL statements. Collected samples are then given indicators of 0's and 1's representing the availbility of each feature. Few examples of the features include use of single line comment, use of double quote, and consist of number equals same number command (e.g. 1=1). The samples will be trained by selected machine learning models including Support Vector Machine (SVM), Boosted Decision Tree, and Decision Jungle. By diving the total data samples into ten parts, nine of them will be trained, while one part will be used of testing. This process is repeated for ten times, as it ensures all data are tested. Kamuto and Soomlek (2016) are able to conclude that Boosted Decision Tree

model is the best, achieving average accuracy of 99.68%. Furthure extension of this study will be on validating SQL queries on Integrated Development Environment (IDE).

SQLi-Fuzzer is a SQL injection vulnerability detection system proposed by Luo (2021). The proposed SQLi-Fuzzer uses Fuzzing which is a technique to discover vulnerabilities in software systems, which can include SQL injection attack vulnerability. Fuzzing can also be known as a Black-Box testing in order to identify defects through automatic irregular data injections (OWASP, n.d.-a). Hence, Fuzzing consists of a generator to generate data and identification tool to determine vulnerability. Due to Fuzzing highly depends on the testcase or input data quality, Luo (2021) propose to use Genetic algorithm to generate testcases with mutation or evolution factor. This includes replacement, addition, substraction and reversal of characters or bits. Then, machine learning is used to aid testcase generation and to identify SQL injection statements and non-SQL injection statements through Back Propagation Neural Network (NN). This is due to Back Propagation NN is able to remember state information and using it to modify the weights of neural nodes to improve prediction. When SQL statements are identified through machine learning, testcases will be generated based on the user requirements. At the same time the testcase generation regenerates other testcases with encoding or conversion changes through the Genetic algorithm. Results show that SQLi-Fuzzer is able to generate testcases with considerably high payload ratio of 53.9% after running 10 times. As for vulnerability discovery performance, SQLi-Fuzzer is able to discover 16 vulnerabilities in 72 hours. Comparing with other Fuzzers such as SPIKE and WFuzz, they only identified 11 and 8 vulnerabilities respectively.

Parashar *et al.* (2021) proposed an approach to identify SQL injection from an extracted dataset using machine learning models. Dataset is extracted from NVD, which is a source for software vulnerabilities data updated with CVE (Common

Vulnerabilities and Exposures) details such as version, data type, and format (NIST, n.d.-b). A total of 9,722 SQL injected samples and 10,215 random attack samples other than SQL injection are collected from NVD. Also, a total of 78,791 unrelated text are collected as benign or negative datasets from movie titles and descriptions. The collected samples are then processed to remove words such as verbs, adverbs, pronouns, and prepositions. Through a program, stop words and special characters are also removed leaving units that define SQL injections. For the unrelated benign text samples, a word count with 10 to 40 are extracted to match the word count of SQL injected samples from NVD. The dataset are then passed to machine learning models for training and evaluation, with splitted dataset of 80% training data and 20% testing data. Among the machine learning models, Random Forest achieved the best result with accuracy, precision, and recall of 78%, 75%, and 95% respectively. Runner ups are Logistic Regression and SVM with both accuracies of 76%. The detection model is also tested on web pages, summarized by TextRank, a Natural Language Processing (NLP) algorithm to summarize articles or documents (Joshi, 2022). It is able to predict SQL injection on web text with 73% accuracy. However, the accuracy result may be affected by the type of text summarization used which has to be further studied, such as comparing results with and without summarization.

Jothi *et al*. (2021) presented SQL injection detection using deep learning. In this approach, 3,692 plain text data and 5,928 SQL injection queries are collected to create their system. Plain text data are also collected as it will be used to differentiate a normal text against SQL injection queries. Then, artificial neural networks are used to detect any users are performing SQL injection attacks. The neural networks are formed layer by layers including embedding layer and Global Average Pooling Layer. In this model, it can be extended to not only for user input detection, but also for detection of inputs by force from URL. Few processing procedures has been undergone after data has been collected, including spliting collected sample statements into separate words, removing stop words, and converted to lower case for uniformity. Each word token are

also converted into their respective english root word. A word index is then formed, where it maps each unique token or word to an index number. Hence, each word in a sentence can be represented by an index number. To determine new or unknown words, a column is added to the word index to hold any token that is not available in the dataset. The dataset then goes through the neural network that consist of enbedding layer and dense layers. The embedding layer is first randomly allocate weights to each token, but the weights are updated through process of back-propagation. The dense layer is then responsible for allocating weights during the back-propagation process. Through the proposed model, it is able to achieve both accuracy and precision of 98%, also a recall of 97%. This indicates that it is able to correctly identify 97% of the SQL injection queries.

Bisht *et al*. (2010) came out with CANDID, also known as CANdidate evaluation for Discovering Intent Dynamically. It is a tool implemented to defend against SQL injection. As the name suggests, it will perform a technique of dynamic candidate evaluation that mines the structure of intended queries by programmer. CANDID first will add a candidate query which is also the intended query of programmer to the control path (Mehta *et al*., 2015). Then, it compares the candidate query with actual query input by user which may contain SQL injections. If the structure of input query does not match the candidate query, SQL injection attack is detected. There are two requirements for the candidate query to be effective in this program. First, candidate input must be a non-SQL injection input, which can be any benign input from user. Second, candidate input must be in the same program path as the intended input by programmer, where the intended input can be any statement the program expects to receive by users. By using a parse tree of an SQL syntax, the structure of SQL injected input and a benign input can be differentiated. Two queries can be said to have identical structure if they have similar parse trees. Therefore, to dynamically determine an SQL injected query, the query structure is determined exclusively by the program control path taken. This also indicates that any input that has a different structure from the one

generated by the control path is deemed as invalid. This technique performs the comparison dynamically during runtime. Hence, it is able to prevent the attacker from performing any malicious inputs. However, there are still limitations to CANDID, including external functions that might return exceptions even on non-injected inputs. Thus, it is important to handle external functions properly. Due to natural external functions do not require any transformation as they have preserved length, Bisht *et al*. (2010) is successful in implementing CANDID.

Table 2.1: Literature Review Synthesis Table

Author (s) (year)	Purpose of study	Proposed method/design	Limitation
Hasan *et al*. (2019)	To detect SQL injection attack using machine learning	Evaluate 23 MATLAB machine learning algorithms, and identify 5 most accurate algorithm. Based on the top 5 most accurate machine learning algorithm: Ensemble Boosted Trees, Ensemble Bagged Trees, Linear Discriminant algorithm, Cubic SVM and Fine Gaussian SVM, develop a user interface to classify user input.	Does not consider completely unseen data for user input
Zhang (2019)	To identify SQL injection vulnerabilities in PHP files using machine learning	Extract features of PHP file based on use of input validation and sanitization functions. Use classic machine learning algorithms from Scikit-Learn and deep learning algorithms Keras library.	Does not integrate system into applications for further evaluation
Tripathy *et al*. (2020)	SQL injection detection for cloud Software as a Service (SaaS) using machine learning	Describe different SQL injection attack features, and evaluate ranking of each feature using chi-square. Then, train the machine learning models including Random Forest, AdaBoost, Deep Artificial NN, Decision Tree, Tensor Flow.	Does not consider evaluating in real-life web application environment

Hosam *et al.* (2021)	To detect SQL injection attack using machine learning	Identify 13 features of SQL injection statements. Evaluate 6 machine learning models based on full feature selection, reduced feature set, and unseen data	Does not integrate system in web applications and update dataset for unseen data
Sivasangari *et al.* (2021)	To propose an algorithm to detect SQL injection attacks	Gather SQL query samples and train machine learning models using dataset collected. Notably AdaBoost Classifier from Sklearn library, as it provides a better performance relatively with other classifiers.	Evaluation of model in real time environment is not considered
Kamuto and Soomlek (2016)	To prevent SQL injection on Server-Side Scripting using machine learning	Identify features of vulnerable SQL queries, and extract features from collected data samples. Then, train and evaluate machine learning models used.	Research is to be extended to implement in compiler platform, but limited to detecting SQL vulnerability in development phase
Luo (2021)	To detect SQL injection vulnerabilities through Fuzzing technique	Convert normal SQL statements and injected SQL statements into general structure as initial data. Back Propagation NN will analyse data and generate testcase for Fuzzing.	Unable to make adjustments directly when Fuzzing mutates to a different direction
Parashar *et al.* (2021)	To identify SQL injection vulnerability through text	Extract datasets from National Vulnerability Database (NVD) and benign samples from movie titles and descriptions. Train dataset using models including Random Forest, Logistic Regression, and SVM. Summarize web text using TextRank, and test on proposed model.	Does not evaluate system integration to web applications, and text summarization method requires further studies

Jothi *et al.* (2021)	To identify SQL injection using deep learning	Process data including remove stop words, tokenization, and uniform word case. Then pass the data to a NN consisting of Embedding Layer and Dense Layer.	Feature extraction for deep learning is not further specified
Bisht *et al.* (2010)	To defend against SQL injection through CANdidate evaluation	Initialize a candidate query and compare with input query using parse tree of SQL syntax. If the structure of both query matches, the input query is a benign query.	Unable to predict new SQL injection types, unless candidate query updated by developer

Based on the literature review, various studies are made to identify and prevent the attacks of SQL injection in web applications. Table 2.1 above shows the summary of research articles reviewed. This review mainly focuses on SQL injection detection using machine learning, but also includes other concepts of detecting SQL injections, such as CANdidate evaluation from Bisht *et al.* (2010) which is not related to machine learning. This to grasp a different view of SQL injection detection. From the overall articles, effective machine learning models include Random Forest and AdaBoost Classifier, where it has achieved relatively high accuracy within the individual studies. As for the overall limitation of the reviewed studies, integration of detection system to web application is mostly not considered. Also, the research articles do not consider measures or methods when encountering an unseen input, such as self-update to dataset. Hence, the next chapter will discuss more on machine learning models and integration to web application with self-updating dataset capability.

CHAPTER 3

RESEARCH METHODLOGY

The framework design of detecting SQL injection through machine learning consists of collecting SQL query datasets and processing the data samples. Processing includes defining the features of SQL injection query and categorizing them based on the features into tokens. Each data sample will be split into multiple tokens and saved as a vector. The machine learning models will be trained using the dataset vector, hence, being able to make predictions of input statements if they contain any SQL injection queries. The framework can be summarized as shown in Figure 3.1.

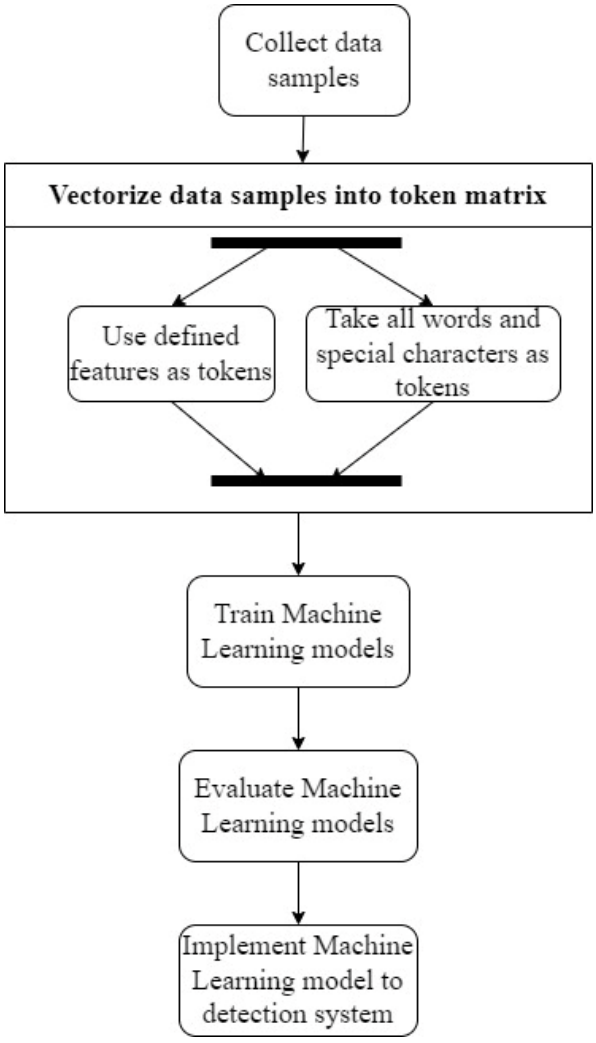


Figure 3.1: Framework of SQL Injection Detection using Machine Learning

3.1 Data Collection

SQL injection query dataset is collected from Kaggle, an online platform that is able to provide over 50,000 public datasets, and for users who are interested in data science to explore and build models (Kaggle, n.d.). Kaggle is also a popular data science competition platform, with some competitions are sponsored by organizations or companies to solve challenging problems (Ali, 2022). Hence, datasets provided by Kaggle are more reliable and realistic, compared to other available ones, to support participants in solving the most efficient and strong solution. Solutions of Kaggle competition are shared publicly to promote learning new ideas and spark inspirations. For this project, more than 3,900 of input samples including SQL injected statements are collected from Kaggle. Datasets are downloaded in the form of CSV (Comma Separated Values) file, where there are two columns which are "Sentence" and "Label". The "Sentence" column consists of the input statement itself, which can be a benign input or a SQL injected input. As for the "Label" column, it will indicate if the statement is benign or an SQL injection attack using 0's and 1's. For example, the statement:

$$'\,OR\ 1 = 1 - -, 1$$

is an SQL injection attack, therefore it has a label of 1 after the comma to indicate it is a malicious input.

The original CSV file of dataset obtained is in UTF-16 (16-bit Unicode Transformation Format). However, the default encoding type for the text editing software used in this project is in UTF-8 (8-bit Unicode Transformation Format). Hence, to reduce the complexity, collected data samples are converted to UTF-8 encoding CSV file. This will ease file read and write access in the implementation phase. The first 10 samples of the dataset are shown in Figure 3.2, where the CSV file is read and output as a Data Frame in Jupyter Notebook.

	Sentence	Label
0	a	0
1	a'	1
2	a' --	1
3	a' or 1 = 1; --	1
4	@	1
5	?	1
6	' and 1 = 0) union all	1
7	? or 1 = 1 --	1
8	x' and userid is NULL; --	1
9	x' and email is NULL; --	1

Figure 3.2: First 10 Samples in Dataset

3.2 Data Vectorization

With the dataset of input statement and their labels to indicate benign and malicious statement, the machine is still yet able to predict SQL injection attacks. This is because the machine is unable to comprehend the input statement as a text. However, if the text is broken down and convert into a numerical information, the machine will be able to process the data (Jain, 2021). Hence, data vectorization process is to break down strings of text into a matrix of token counts. This is also known as feature extraction, where the feature is the token or unique word in the dataset. Two vectorizer functions will be used which are CountVectorizer and TfidfVectorizer from Sklearn library.

CountVectorizer calculates the number of occurrences of a unique word in the dataset, and put all counts into a matrix. The row index of matrix indicates the row index of dataset, while each column represents a unique word. For example, a text data of:

$$text = [\text{'}Hello, I\ am\ YuEn\text{'}, \quad \text{'}Hello\ World\text{'}]$$

In the text array, it has two strings of words and 5 unique words. Hence, CountVectorizer will generate a matrix of two rows and five columns as shown in Table 3.1.

Table 3.1: Vectorization of CountVectorizer

	hello	i	am	yuen	world
0	2	1	1	1	1

Note that default CountVectorizer converts all characters into lowercase, and only tokenize words excluding any special characters.

For TfidfVectorizer, it also extracts features or unique words similar to CountVectorizer. However, TfidfVectorizer has a different way of calculating the matrix values. It uses the product of Term Frequency (TF) and Inverse Document Frequency (IDF) to generate TF-IDF values as matrix entries (Unnikrishnan, 2021). TfidVectorizer from Sklearn library calculate TF and IDF using the functions as shown below:

$$tf(t) = no.of\ times\ term\ 't'\ occurs\ in\ a\ document$$

$$idf(t) = \ln\left[\frac{(1+n)}{(1+df(t))}\right] + 1$$

Where:

$$n = total\ number\ of\ documents$$

$$t = term\ or\ unique\ word\ to\ be\ calculated$$

$$df(t) = number\ of\ documents\ term\ 't'\ appears$$

In the case of SQL injection dataset, each statement will be considered a document. As the normalized TF-IDF value of a term approach to 1, indicates that term is rare and having greater importance. Figure 3.3 shows a snippet of a matrix generated by TfidfVectorizer using the collected dataset.

```
[[0.         0.         0.         0.         0.        ]
 [0.         1.         0.         0.         0.        ]
 [0.         0.75144983 0.65979023 0.         0.        ]
 [0.         0.51367039 0.45101442 0.         0.        ]
 [0.         0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.        ]
 [0.         0.33390259 0.         0.         0.79778888]
 [0.         0.         0.58143282 0.         0.        ]
 [0.         0.55023142 0.48311584 0.         0.        ]
 [0.         0.55023142 0.48311584 0.         0.        ]]
```

Figure 3.3: Sample 10 x 5 Matrix Generated using TfidfVectorizer

Both CountVectorizer and TfidfVectorizer only tokenize words by default. However, SQL injection attacks includes many special characters, such as the single line comment ("--") and semicolon (";"). Hence, the vectorizer functions are modified to be able to tokenize all words and special characters using a custom function. The function will input a whitespace before and after special characters, then split all words and special characters into individual tokens illustrated in Figure 3.4.

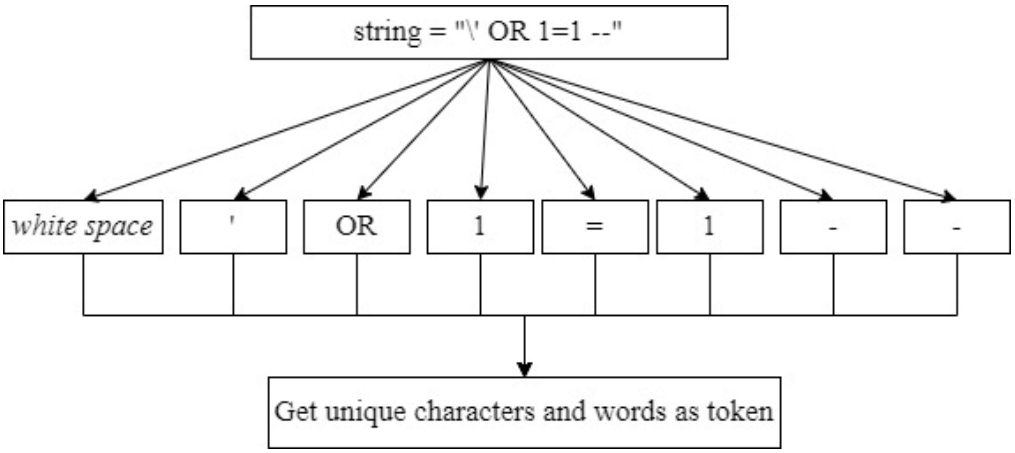


Figure 3.4: Get all words and special characters as tokens

For comparison purposes, another separate function to describe SQL injection specific features will also be used to generate tokens. Hence, instead of using all unique characters and words in the data set as tokens, only specified features are used as tokens. The specified features include use of SQL keywords, single line comment, semicolon, quotations, and always true conditions (e.g. 1=1, 'a'='a').

3.3 Machine Learning Models

In order to make predictions on a statement is benign or malicious, machine learning models will be trained using the vectorized data samples. Three machine learning models from Sklearn library will be used, which are Multinomial NB (Naïve Bayes), Random Forest Classifier, and AdaBoost Classifier. All three machine learning models are common and effective models to predict and classify text data. Multinomial NB is commonly used to classify documents based on their contents (Ratz, 2021). It is able

to understand the content of a document with the terms in the document, which can be applied to predicting SQL injected queries in this project. As for Random Forest Classifier and AdaBoost Classifier, they are strong machine learning models in classifying text documents, which was proven in some reviewed articles (Parashar *et al.*, 2021; Sivasangari *et al*., 2021; Tripathy *et al*., 2020). Through the machine learning models, the machine will be able to make observations on each statement in the dataset, and to classify each of them if they contain any SQL injection or not. Due to different models have different methods of assessing the data, their accuracy and performance will be assessed to determine the best model to be used for website implementation. Each model used will be discussed in detail below.

3.3.1 Multinomial NB

Multinomial NB classifier is based on the probability concept of Bayes' rule, and treats each token or feature in the data samples as mutually independent (Solanki, 2022). This will avoid respective probabilities of two features appearing in a document affecting each other. The formula of the probability model is as shown:

$$Posterior\ Probability = \frac{Conditional\ Probability * Prior\ Probability}{Predictor\ Prior\ Probability}$$

$$P(A|B) = \frac{P(A) * P(B|A)}{P(B)}$$

Based on the probability value obtained, Multinomial NB is able to determine the probability of feature A occurring, while knowing that is from class B. Hence, the probability score will become the decision factor of predicting is a statement contains any SQL injection. Default parameters will be used when applying Multinomial NB, with:

- Additive smoothing parameter = 1.0
- Learn class prior probabilities = True

- Class prior probabilities = None

3.3.2 Random Forest Classifier

Random Forest model are made up of multiple decision trees, where the decision trees will split the data sample into groups that are as distinct as possible from each other based on the features (Yiu, 2019). As the Random Forest will consist of many decision trees, each tree will generate a prediction. Therefore, with all prediction from each tree collected, the most predicted class will be the final prediction. Figure 3.5 shows an example of a Random Forest model with each decision tree giving individual predictions.

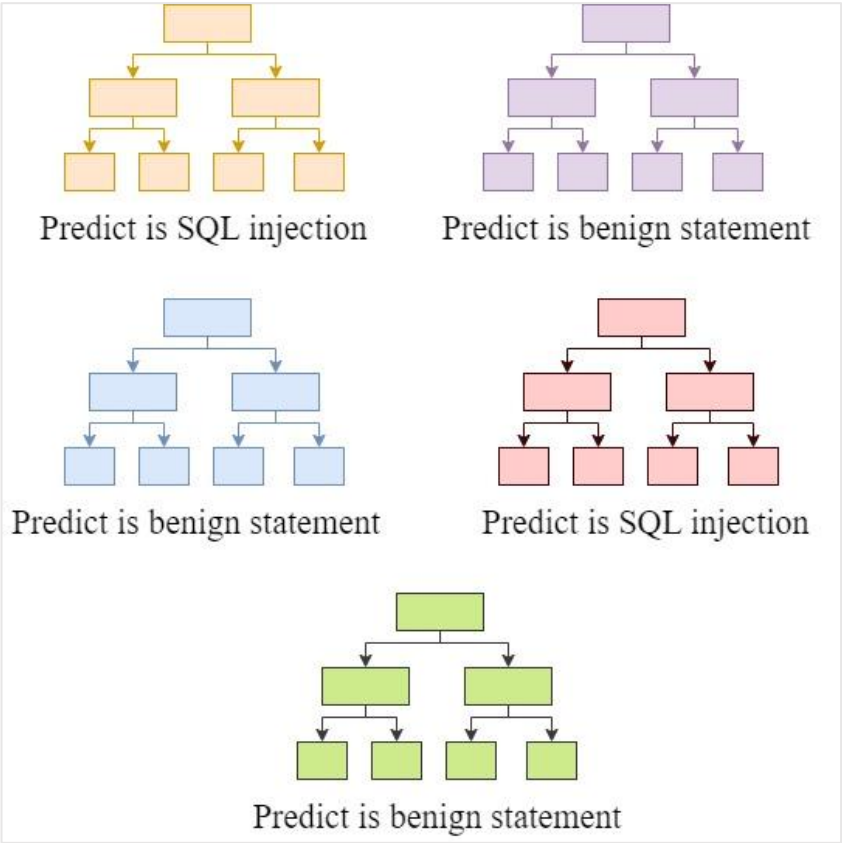


Figure 3.5: Random Forest Model

Based on Figure 3.5, three out of five decision trees give the prediction that a statement is benign. Hence, the overall prediction of the Random Forest classifier will conclude the statement is benign. In this paper, parameters of the Random Forest classifier will

remain as default with values:

- Number of trees in forest = 100
- Function to measure quality of split = Gini Impurity
- Minimum samples needed to split node = 2

3.3.3 AdaBoost Classifier

AdaBoost classifier also known as adaptive boosting classifier, where it uses a base model to make a prediction while iteratively improving the model to produce a more accurate prediction in the next iteration (Veronica, 2020). Every iteration the model will go through training and testing to identify any errors made. Mistakes made in the previous iteration will be amended in the next repetition leading up to a strong model making accurate predictions. The work flow of AdaBoost classifier is as shown in Figure 3.6.

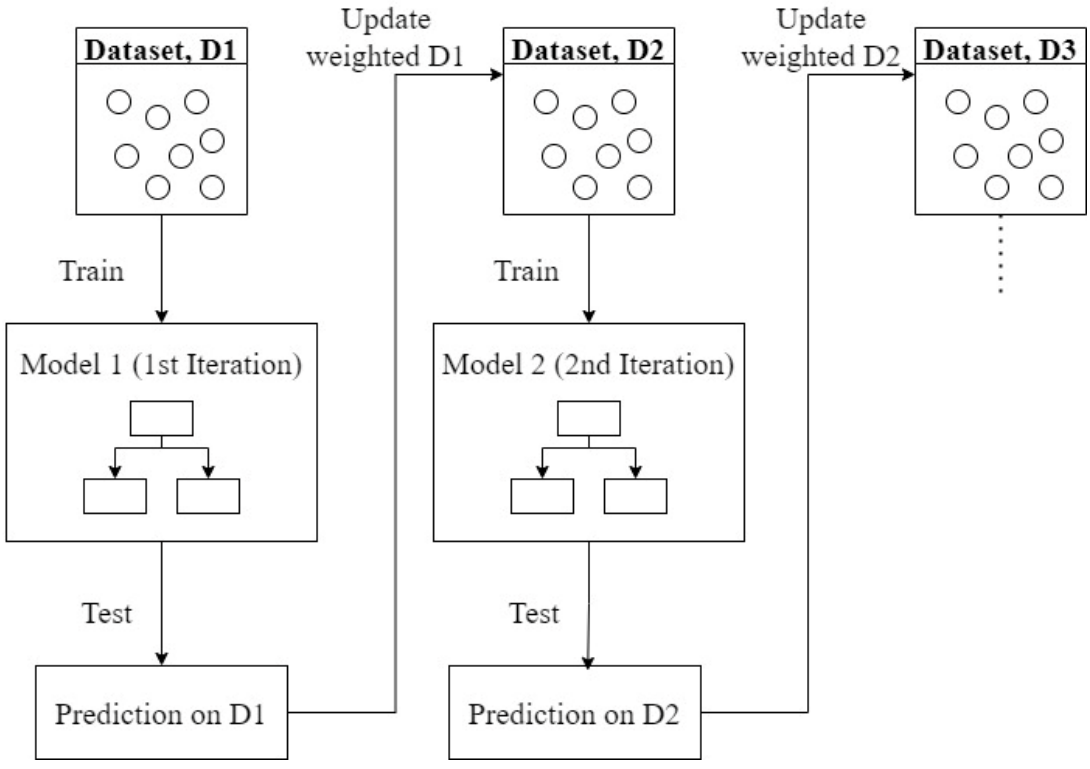


Figure 3.6: AdaBoost Model

AdaBoost classifier selects the training set according to the last iteration's accurate

predictions. In Figure 3.6, predictions on dataset D1 will be weighted and updated into dataset D2, passing on to the next iteration as training data. Higher weight will be assigned to incorrect predictions, in order to get higher probability of making the correct prediction in the next iteration. Default parameters of AdaBoost classifier will be used with the values as follows:

- Base estimator = Decision Tree Classifier
- Maximum number of estimators = 50
- Boosting algorithm = SAMME.R (Real-Valued Stagewise Additive Modelling)

3.4 Website Implementation

With the SQL injection detection built, it will be integrated into a simple website. The website will have simple functions including logging in to existing profile, signing up for a new profile, and delete existing profile. Figure 3.7 below shows the use case diagram of the simple website.

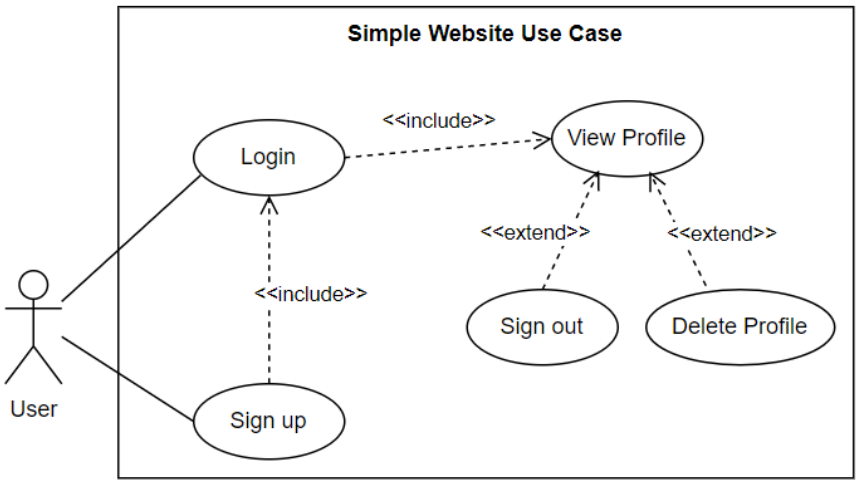


Figure 3.7: Use Case Diagram of Simple Demo Website

Through the website, user will be able to create a profile that is password protected through sign up function. This means that other users are not allowed to view profiles

other than their own profiles. After signing up, the users are required to login to their profiles, using the identity (ID) and password they have created. With the correct ID and password, the users will be successfully logged in and brought to a new page that shows their profile details. At this page, the users have the option to delete their profile or to sign out of their profile.

The website will use many text boxes and page redirecting as the functionality heavily depends on the user input and validating user input. This will create a high chance for SQL injection attack. However, to emphasize the implementation of the SQL injection detection system, the website itself is designed to be vulnerable to SQL injection attacks. Only minimal or hardly any SQL injection prevention will be done on the website coding. For example, a dynamic SQL query to select count of user profile with given ID and password as shown:

$$\text{"SELECT COUNT}(*)\text{ FROM Profile WHERE ID} = '\text{"} + \text{TextBox1.Text} + \text{"'AND Password} = '\text{"} + \text{TextBox2.Text} + \text{"'"}$$

The SQL query will take the direct input from user instead of using parameterization, which leaves malicious users a chance to take advantage of SQL injection.

The website alone will be weak to SQL injection attacks, and here is where the SQL injection detection system takes place. The machine learning detection system will be able to provide a form of security protection to a vulnerable demo website. To illustrate the functions and activities on how the detection and prevention of SQL injection attacks takes place, the login activity diagram for the web application is as shown in Figure 3.8.

Based on Figure 3.8, the web application and SQL injection detection system will work in different environment. The web application mainly focuses on the front-end website functions and design, while the detection system is a back-end system solely focusing

on determining if given input is a SQL injection or not. The SQL injection detection system will interpret a given input through machine learning and provide its prediction. Hence, based on the prediction, the web application will only run the SQL query if the input by user is determined as safe.

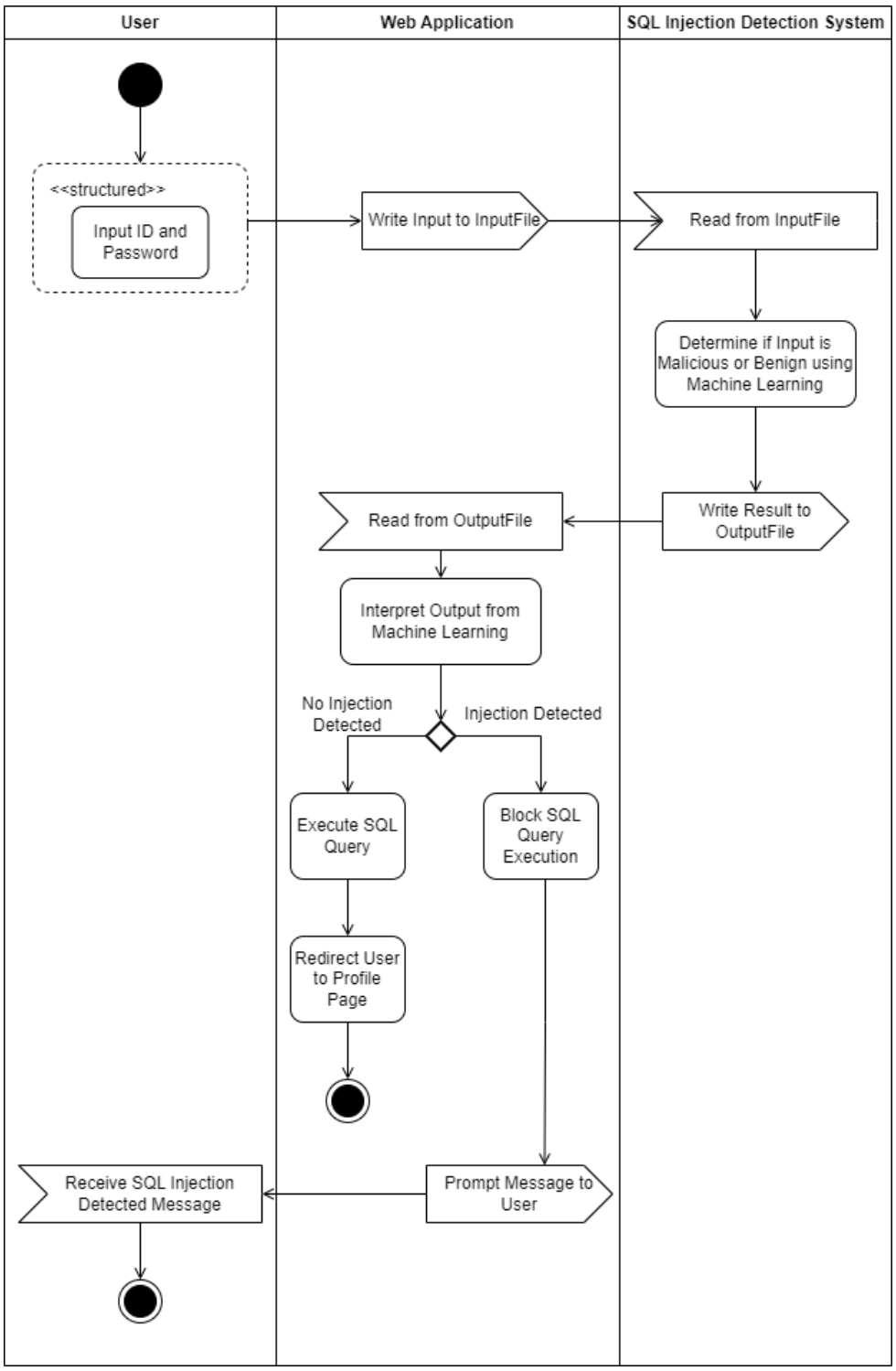


Figure 3.8: Activity Diagram of Website Login Function

For the front-end website and back-end detection system to communicate, two temporary files will be used, namely “InputFile” and “OutputFile”. As the name suggests, the “InputFile” will contain the raw input from users, while the “OutputFile” will contain the prediction result from the SQL injection detection system.

However, the system is still incomplete, as the input of website users can be very dynamic, and may cause the back-end detection system to encounter inputs that are entirely new. This is due to the limitation of the dataset collected to train the machine learning model. The SQL injection statement dataset only has a limited token variation. Hence, there is a probability that the system will encounter a new user-name that is not a part of its token vectorization matrix. Figure 3.9 shows an example of the machine learning model encountering an unseen input.

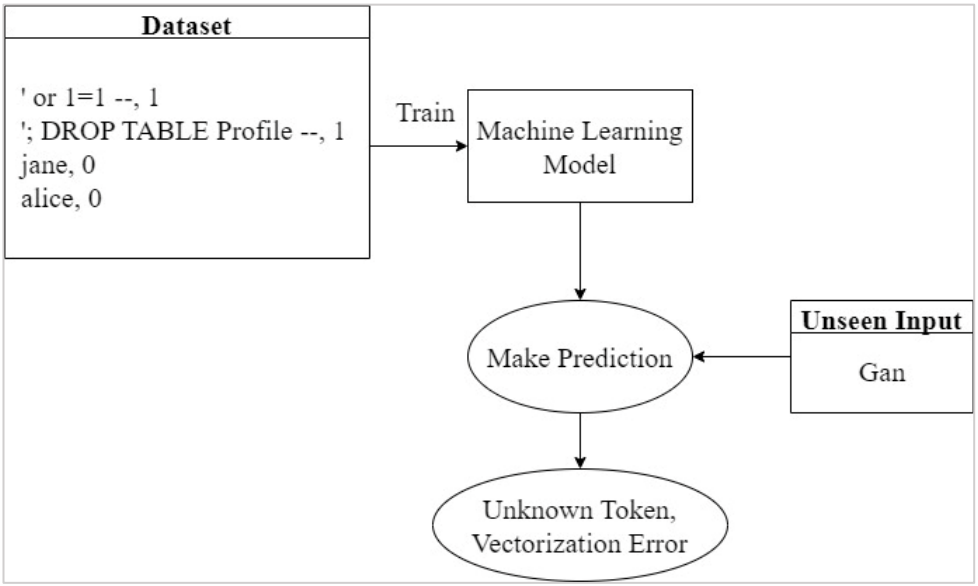


Figure 3.9: Machine Learning Model Encountering New Token From Input

To take measures of encountering new tokens from user input, unseen inputs will be saved into a temporary file. When the detection system reboots, the temporary file with unseen inputs will be combined with the existing training dataset and the machine learning model will be trained again with the updated dataset. This will be able to

improve the machine learning model's vocabulary and predicting ability. Hence, the overall detection and prevention of SQL injection system will be able to evolve and grow when it is used regularly.

The website implementation will be assessed through manual testing of each function including attempt to perform SQL injection on the website, perform normal profile login, input unseen token and verify if dataset is updated. Table 3.2 below shows the test cases and expected result for the website implementation. Tested cases with expected result will indicate that the website implementation is successful and has fulfil its requirements.

Table 3.2: Test Cases for Website Implementation

Page	Function	Test Scenario	Expected Result
Sign Up	Normal Sign Up	1. Fill in normal inputs: ID: abc123 Name: abc Secret Phrase: abc Password: abc 2. Click "Sign Up" button	1. Profile successfully signed up 2. Able to login to new profile
	Detect and Prevent SQL Injection	1. Input SQL injected statement: ', ' ', ' ', ' '); DELETE FROM Profile WHERE ID='admin' -- 2. Click "Sign Up" button	1. Prompt message "SQL Injection Detected" 2. Profile sign up unsuccessful
	Sign Up with unseen tokens	1. Input unseen token: ID: ethan123 Name: Ethan Secret Phrase: I like apples Password: ethanpass321 2. Click "Sign Up" button	1. Profile sign up successful 2. Able to login to new profile
Login	Normal Login	1. Fill in normal inputs: ID: abc123 Password: abc 2. Click "Login" button	1. Successfully login to profile

	Detect and Prevent SQL Injection	1. Input SQL injected statement: ' OR 1 =1 -- 2. Click “Login” button	1. Prompt message "SQL Injection Detected" 2. Login unsuccessful
	Update unseen tokens	1. Input unseen token: ID: ethan123 Password: ethanpass321 2. Click “Login” button	1. Back-end detection system output "Unknown Input" 2. Profile Login successful 3. Unseen token is updated to current dataset after system reset
	Attempt SQL Injection with inactive detection system	1. Input SQL injected statement: ' OR 1 =1 -- 2. Click “Login” button	1. SQL injection successful 2. Successfully login to profile, with all users' profile displayed

CHAPTER 4

CODE IMPLEMENTATION

4.1 SQL Injection Detection System

The machine learning detection system is built using Python language, with Sklearn library. The library contains various efficient tools and functions for machine learning which involved predictive data analysis, and it is easily accessible. The built SQL injection detection system is compiled and run on Jupyter Notebook, an interactive web-based computing platform. Figure 4.1 below shows the version Python libraries and Sklearn.

```
Python:3.8.3 (default, Jul  2 2020, 17:30:36) [MSC v.1916 64 bit (AMD64)]
scipy:1.5.0
numpy:1.18.5
matplotlib:3.2.2
pandas:1.0.5
sklearn:0.23.1
```

Figure 4.1: Python Libraries and Sklearn Version

4.1.1 Load Dataset

The program first starts with loading the dataset. The prepared dataset is initially saved in a CSV file, it is first loaded into the Python program with a read CSV file function. The dataset is loaded into a data frame which includes two columns, namely “Sentence” which is the input samples, and “Label” which is the indicator of SQL injection. The values of the two columns will be stored separately into two variables to ease training and testing later in the program. Figure 4.2 below shows the pseudocode for loading dataset.

```
Load “sqli.csv” into DataFrame df
X=df[“Sentence”]
Y=df[“Label”]
```

Figure 4.2: Loading Dataset Pseudocode

However, the loaded data is still unable to act as training data for the machine learning model. This is due to the machine learning model is unable to comprehend full sentences, hence, the sentences have to be broken down into individual words.

4.1.2 Declare Vectorization Functions

To break down sentences into separate tokens or words, vectorizer functions are used. For evaluation purposes, two vectorizer functions namely Count Vectorizer and Tfidf Vectorizer are used. The vectorizers are also customized to have different tokenization processing, where it determines what to take as a token. For example, only take specific word or characters as a token, and ignore others.

Two customized functions are used as the tokenizer, which are "specificSQLTokens" and "includeAllTokens". The first will only take features of SQL injection as tokens, such as keywords "and", "union", "or", and special characters like semicolon and quotation marks. Also, always true conditions such as "1 == 1" will also be taken as a token for specific SQL injection features. As for the latter function, "includeAllTokens" will take all words and special characters as a token. The pseudocode is as shown in Figure 4.3 below.

```
#Custom Tokenization Functions
procedure specificSQLTokens:
    tokenize "and", "union", "or", "delete", "drop table",
            "--", ";", "literal single quote", "literal double quote",
            "[0-9]=[0-9]", "[a-z]=[a-z]"

procedure includeAllTokens:
    tokenize all words and special characters

#Call Count Vectorizer Function
fixTokenCountVectorizer = CountVectorizer(tokenizer=specificSQLTokens)
countVectorizer = CountVectorizer(tokenizer=includeAllTokens)

#Call Tfidf Vectorizer Function
fixTokentfidfVectorizer = TfidfVectorizer(tokenizer=specificSQLTokens)
tfidfVectorizer = TfidfVectorizer(tokenizer=includeAllTokens)
```

Figure 4.3: Tokenization and Vectorization Pseudocode

4.1.3 Train Machine Learning Models for Evaluation

With the vectorizer functions defined, the data samples with input statements are passed to the vectorizer to generate a token matrix. Then the matrix is ready to be put into the machine learning models as training data. To evaluate each machine learning models, the vectorized data is randomly split 10 times using Stratified K Fold function as shown below (Lendave, 2021). With this, it can ensure all data samples are tested on the model. Then, cross-validation results are obtained from the machine learning models of MultinomialNB, Random Forest Classifier, and AdaBoostClassifier. The pseudocode for training machine learning models using data vectorized by Count Vectorizer with "specificSQLTokens" is shown in Figure 4.4 below.

```
X=df["Sentence"]
kfold = StratifiedKFold(n_splits=10, random_state=1, shuffle=True)
fit transform X using fixTokenCountVectorizer
split X into X_train, X_test, Y_train, Y_test

nb_clf = MultinomialNB()
nb_clf.fit(X_train, Y_train)
get kfold cross validation score for accuracy, precision, recall, and F1

randomForest_clf = RandomForestClassifier()
randomForest_clf.fit(X_train, Y_train)
get kfold cross validation score for accuracy, precision, recall, and F1

adaBoost_clf = AdaBoostClassifier()
adaBoost_clf.fit(X_train, Y_train)
get kfold cross validation score for accuracy, precision, recall, and F1
```

Figure 4.4: Machine Learning Models Evaluation Pseudocode

Similar code is used with other combination of vectorizer functions and its customized tokenizer, where this includes Count Vectorizer with "includeAllTokens", Tfidf Vectorizer with "specificSQLTokens", and Tfidf Vectorizer with "includeAllTokens". Evaluation of accuracy score, precision, recall and F1-score are obtained from each of the trained machine learning models.

4.1.4 Machine Learning Model on SQL Injection Detection for Web Implementation

To implement the SQL injection detection system on the demo web application, the most efficient machine learning model is determined among the three machine learning models evaluated (Multinomial NB, Random Forest, AdaBoost Classifier). Then, it is trained with 100% of the dataset instead of splitting into training and testing data. This is due to the testing data is now replaced with the input of the website user. Hence, the detection system will take the input from user and predict it using the trained machine learning model.

The detection system first will have to obtain the user input from a file, which is written by the front-end web application. Therefore, the file modification time of the input file is constantly verified to determine if the file is updated or not. If the file is updated, it indicates that the user has sent an input in real time. The user input will then be read and processed by the detection system.

The user input will be vectorized into tokens before passing it to the machine learning model for predictions. However, vectorizing the user input may result in creating a new token, apart from the existing token list from the training data. This is due to the user input may contain new words that the machine learning model has never seen previously from training. Hence, the predicting result will have three outcomes, which are “0” for non-injected, “1” for injected, and “-1” for unknown.

Once the prediction is generated, the detection system will write the results into an output file, which will be read by the front-end web application to further decide the next execution. The pseudocode for the website SQL injection detection system is as shown in Figure 4.5.

```
initialize new_timestamp = time now
initialize old_timestamp = new_timestamp

while True:
    if new_timestamp > old_timestamp:
        print("file modified")
        old_timestamp = new_timestamp
        load "input.csv" into df1
        dfAppended = append df1 to original dataset

        vector = vectorized dfAppended

        if vector has more columns than original vectorized dataset:
            print("Unknown Input")
            determine unknown input
            write("-1") to output file "output.txt"

        else:
            predict input using machine learning model
            write result to output file "output.txt"
```

Figure 4.5: SQL Injection Detection System Pseudocode

4.2 Website Code Implementation

The exploratory web application is implemented using ASP.NET in Visual Studio, which is a web framework that is open sourced and able to build web applications easily. Button functions are scripted in C# language and user-friendly design view of website HTML layout. Web forms in Visual Studio also allows integration of SQL database within the application. Hence, it eases the use of SQL query command and directly links to the database. In this project, the website consists of main functions including to login, sign up, sign out, and view profile. Hence, it has three main web pages, namely login, sign up, and profile page. Any page that requires user input will first go through the SQL injection detection system before executing user request.

4.2.1 Login Page

The login page will get user input of their ID and password using a text box. User will be able to confirm their input and proceed to login by clicking the "Login" button, or

clicking the “Sign Up” button if the user does not have a profile created. Figure 4.6 below shows the layout of the Login page.

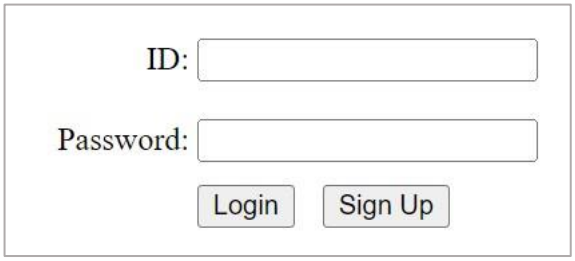


Figure 4.6: Login Page Layout

After clicking the “Login” button, the user input will be written into a file. Subsequently, the file will be read by the back-end detection system for processing. Based on the result of the detection system, the function will redirect the page to the user’s profile page, or prompt a message to user. As for the “Sign Up” button, it will simply redirect to the sign up page. Figure 4.7 below shows the pseudocode of the “Login” button function.

```
function loginButton_Click {
    if input text box is empty {
        prompt user to fill in all fields
        end
    }

    get user input[]
    write input[] to “input.csv” file

    declare SQL command string cmdStr = "SELECT COUNT(*) FROM Profile WHERE ID='"
                                    + TextBox1.Text + "'AND Password='"
                                    + TextBox2.Text + "'";

    call function smartQueryExecution(input, cmdStr)
    end
}
```

Figure 4.7: "Login" Button Function Pseudocode

In the pseudocode, a function called smartQueryExecution() is called to execute the user request after interpreting the output of detection system. This function will determine if the SQL injection detection result is no injection (0), injected (1), or

unknown token found (-1). The function will execute the query if it is identified as no injection, and block the execution if injection is detected.

As for inputs with unseen tokens by the machine learning model, the function will execute the query and verify the item count retrieved after execution. The unseen inputs will then be saved into a temporary file, and update to the existing machine learning dataset during system reboot. Figure 4.8 below shows the pseudocode for the smartQueryExecution() function for login.

```
function smartQueryExecution {
    wait for back-end detection system
        to write result to "output.txt" file

    read "output.txt" file

    if output == 0
        execute SQL query

    else if output == 1
        prompt "SQL Injection detected"

    else if output == -1 {
        execute SQL query
        if multiple rows found
            learn input as SQL injection attack
        else
            learn input as benign statement
    }
}
```

Figure 4.8: smartQueryExecution() Function Pseudocode for Login

4.2.2 Sign Up Page

The sign up page is for users to create their own profile. The user will have to input their ID, name, a secret phrase, and password. This information is only for the individual user and other users shall not know each other's' profiles. Figure 4.9 shows the website layout for the sign up page.

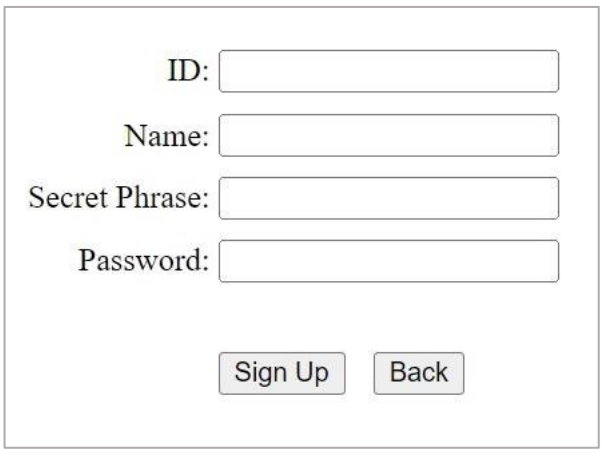


Figure 4.9: Sign Up Page Layout

By clicking the “Sign Up” button, it will go through a verification process of SQL injection detection similar to “Login” button. However, “Sign Up” button will have a different query command. The pseudocode for sign up function is as shown in Figure 4.10 below.

```
function signUpButton_Click {
    if input text box is empty {
        prompt user to fill in all fields
        end
    }

    get user input[]
    write input[] to “input.csv” file

    declare SQL command string cmdStr = "INSERT INTO Profile VALUES('"
                    + IDSignUp.Text + "','" + NameSignUp.Text + "','"
                    + SecretPhraseSignUp.Text + "','"
                    + PasswordSignUp.Text + "')";

    call function smartQueryExecution(input, cmdStr)
    end
}
```

Figure 4.10: Sign Up Function Pseudocode

Due to the different SQL command string in “Sign Up” button click function, the function of smartQueryExecution() will be unable to process the unseen tokens (-1) by verifying the row counts as nothing is selected from the database. Hence, the

smartQueryExecution() for sign up page is slightly modified, where the unknown inputs will not be learnt by the detection system. The modified smartQueryExecution() for sign up page is as shown in Figure 4.11.

```
function smartQueryExecution {
    wait for back-end detection system
        to write result to "output.txt" file

    read "output.txt" file

    if output == 0 or output == -1
        execute SQL query

    else if output == 1
        prompt "SQL Injection detected"
}
```

Figure 4.11: smartQueryExecution() Function Pseudocode for Sign Up

Although the unknown inputs are treated as normal inputs, the sign up page will require user to login to their profile in login page. Hence, the unknown inputs in sign up page will eventually be saved during the login process and verified with row count retrieved.

4.2.3 Profile Page

Users will be redirected to their respective profile page when the login inputs are successfully verified as non-injected, also with correct ID and password combination. The profile page simply displays the user's information, and allow user to delete their profile or to sign out of their profile. Figure 4.12 shows the layout of profile page.

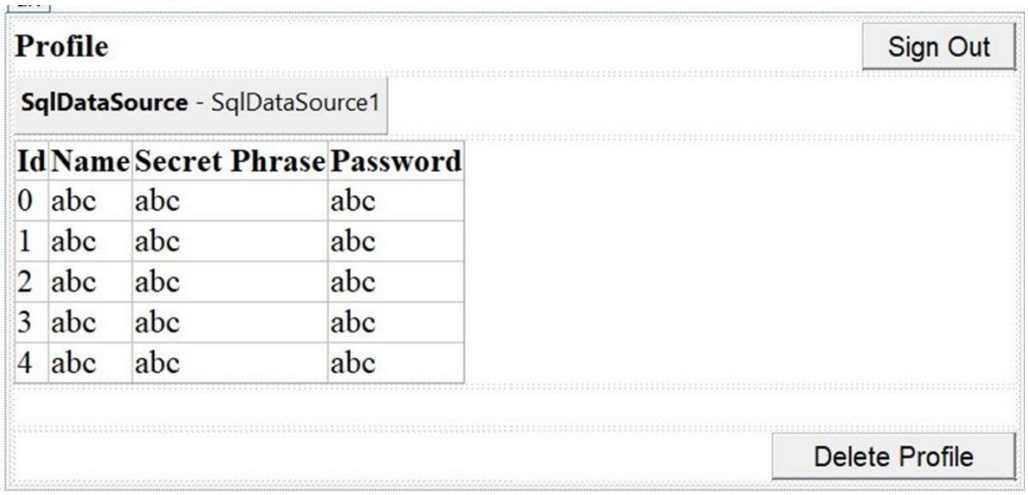


Figure 4.12: Profile Page Layout

As the profile page does not take any user input, no input will be passed to the SQL injection detection system for processing. The page will display the retrieved profile from the login function, indicating the user has successfully logged into his or her profile. The pseudocode for profile page is as shown in Figure 4.13 below.

```
function pageLoad {
    execute SQL command "SELECT * FROM Profile WHERE
                ID='"+ Session["Id"].ToString() +"'";
    bind result to grid table
}
function signOutButton_Click {
    clear session
    redirect to Login page
}
function deleteProfileButton_Click {
    execute SQL command "DELETE FROM Profile WHERE
            ID='" + Session["Id"].ToString()+ "'";
}
```

Figure 4.13: Profile Page Pseudocode

CHAPTER 5

RESULTS AND DISCUSSIONS

5.1 Results

Cross-validation results of Stratified K Fold function are obtained from all three models used, which are Multinomial NB, Random Forest Classifier, AdaBoost Classifier. The Stratified K Fold function is set to randomly split the dataset 10 times, to ensure all data samples are tested. Each split generates a score for the evaluation of training and testing values, and the mean score is generated. Table 5.1 below shows the average accuracy, precision, recall and F1-score of the cross-validation results for all three machine learning models, with each combination of vectorizer and features used.

Table 5.1: 10-Fold Cross-validation Result

Model	Vectorizer	Accuracy	Precision	Recall	F1-Score
Multinomial NB	CountVectorizer (specified features)	93.95%	98.53%	78.60%	93.70%
	CountVectorizer (consider all tokens)	97.17%	95.95%	93.43%	97.15%
	TfidfVectorizer (specified features)	94.31%	98.13%	80.29%	94.10%
	TfidfVectorizer (consider all tokens)	97.98%	97.99%	94.41%	97.96%
Random Forest Classifier	CountVectorizer (specified features)	94.45%	98.88%	80.20%	94.26%
	CountVectorizer (consider all tokens)	98.60%	96.29%	98.31%	98.60%
	TfidfVectorizer (specified features)	94.31%	98.04%	80.38%	94.07%
	TfidfVectorizer (consider all tokens)	99.10%	97.98%	98.13%	99.02%
AdaBoost Classifier	CountVectorizer (specified features)	94.40%	98.45%	80.38%	94.19%

	CountVectorizer (consider all tokens)	96.74%	96.01%	91.74%	96.71%
	TfidfVectorizer (specified features)	94.21%	97.82%	80.20%	94.00%
	TfidfVectorizer (consider all tokens)	97.90%	96.64%	95.56%	97.90%

Based on Table 5.1, all models are able to obtain accuracy and F1-score above than 90%. For each machine learning model, vectorizer and feature combination of TfidfVectorizer with all tokens considered has obtained the best result among other vectorizer combinations. As for the overall machine learning models, Random Forest Classifier with TfidfVectorizer considering all tokens got the highest score, with accuracy, precision, recall, and F1-score of 99.10%, 97.98%, 98.13%, and 99.02% respectively. Hence, Random Forest Classifier using TfidfVectorizer (consider all tokens) is chosen as the machine learning model used for the SQL injection detection system for website implementation.

The website implementation of SQL injection detection system is tested based on the test cases in Table 3.2. Starting with normal sign up and login functions, the website successfully produced an expected result. Figure 5.1 shows a before and after screenshot of signing up a new user profile, and message indicating the sign up is successful is prompted after clicking the sign up button. To test if the newly created profile is usable, a login function is tested. Figure 5.2 below shows the created profile is usable and successfully logged into the profile showing user information. Before executing the query, the user inputs are passed to the back-end detection system with outputs as shown in Figure 5.3. The detection system has outputs of 0 to indicate they are safe inputs, and allowing the website to execute the query.

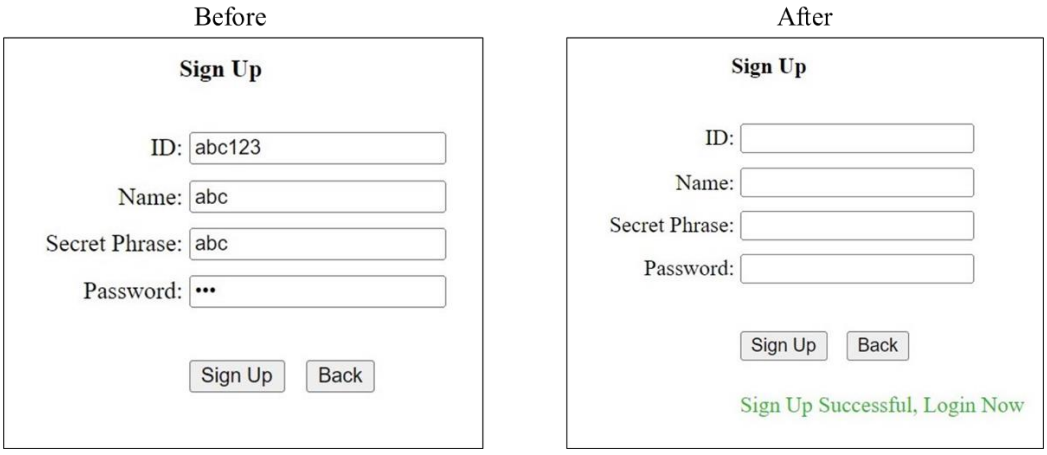


Figure 5.1: Normal Sign Up

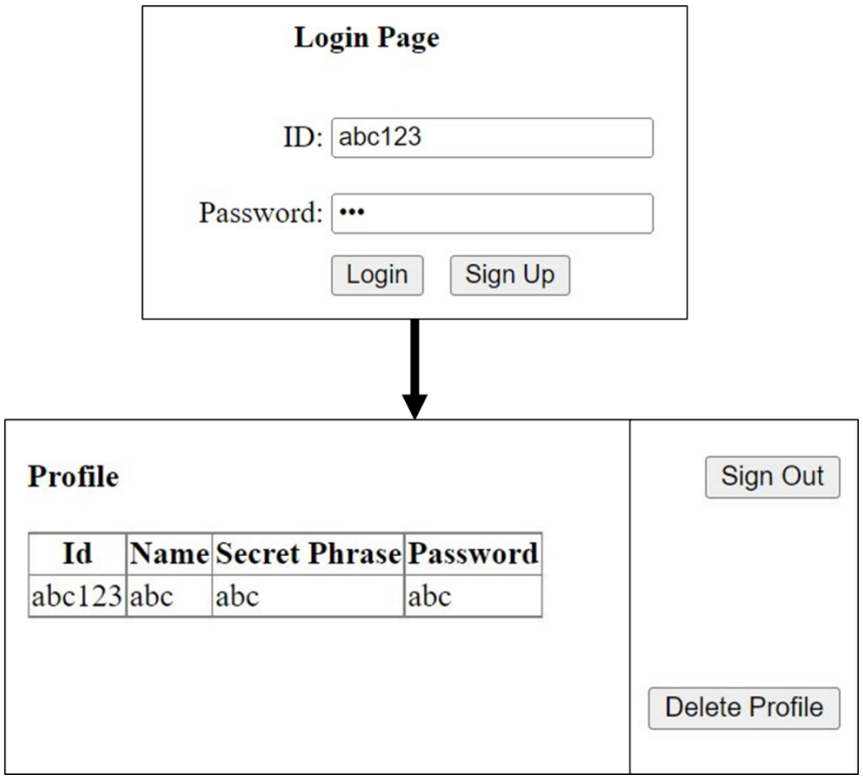


Figure 5.2: Normal Login

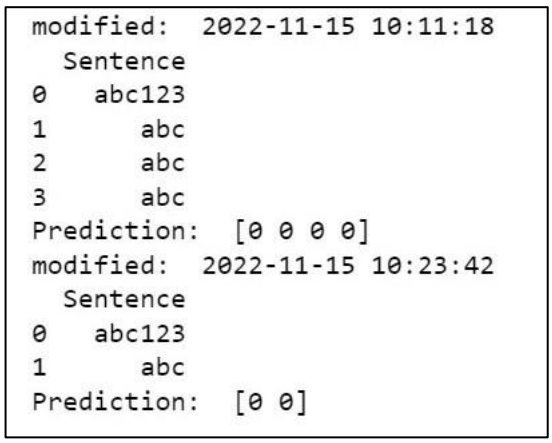


Figure 5.3: SQL Injection Detection System Output for Normal Sign Up and Login

The next test case includes testing on unseen tokens, where they usually are rare names that are not included within the existing dataset. In this test, the name of “ethan” is used as unseen token. It is also proven to be an unseen token based on the output of detection system shown in Figure 5.6, where it is unable to output a clear prediction of 0s or 1s. Regardless, the website is still able to successfully sign up an account of the user as shown in Figure 5.4. The created profile is also proven usable when login function is performed as shown in Figure 5.5. To test if unseen token is updated to the existing dataset, the detection system is restarted. Figure 5.7 below shows the result of dataset after system is restarted, where the unseen token is successfully updated according to expected result. Figure 5.7 also shows that the detection system is able to predict the input correctly after performing a login function again. This indicates that the demo website and detection system is able to successfully handle unseen tokens and update to existing dataset.

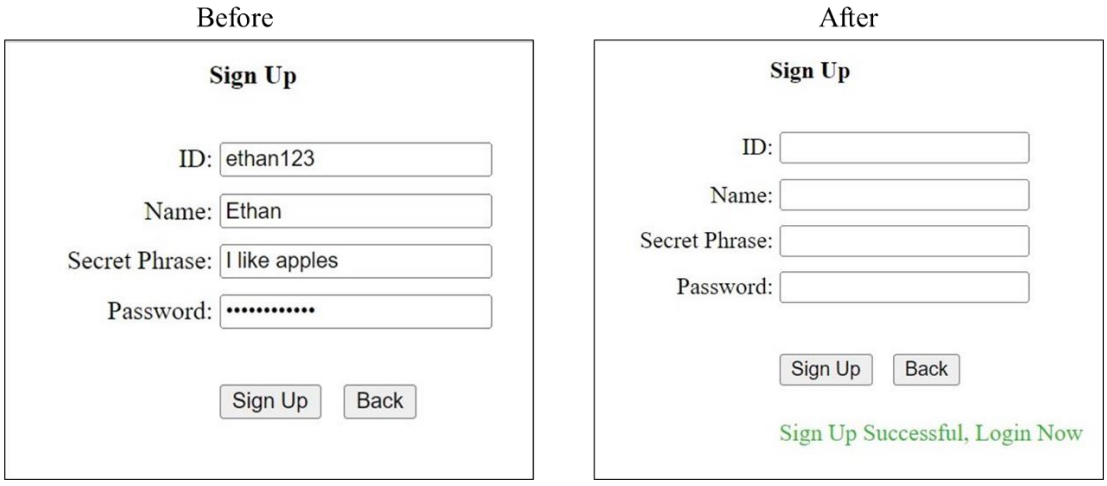


Figure 5.4: Sign Up with Unseen Tokens

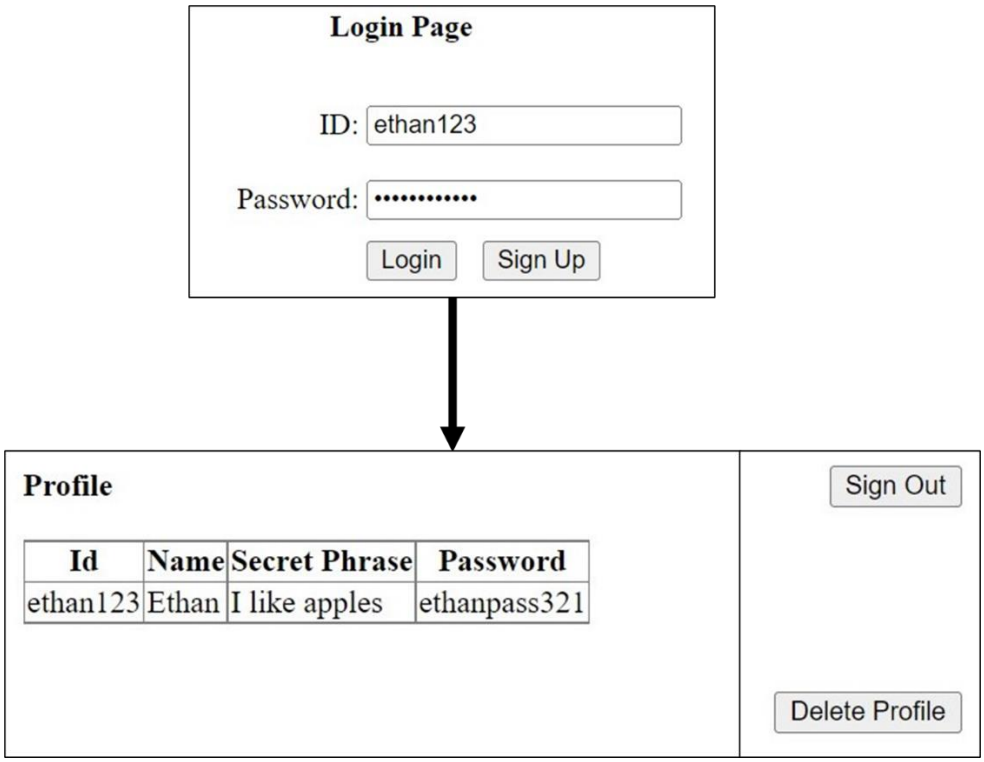


Figure 5.5: Login with Unseen Tokens

```
modified:  2022-11-15 10:31:50
        Sentence
0       ethan123
1          Ethan
2  I like apples
3   ethanpass321
Unknown Input
```

```
modified:  2022-11-15 10:35:12
       Sentence
0      ethan123
1  ethanpass321
Unknown Input
```

Figure 5.6: SQL Injection Detection System Output for Unseen Tokens

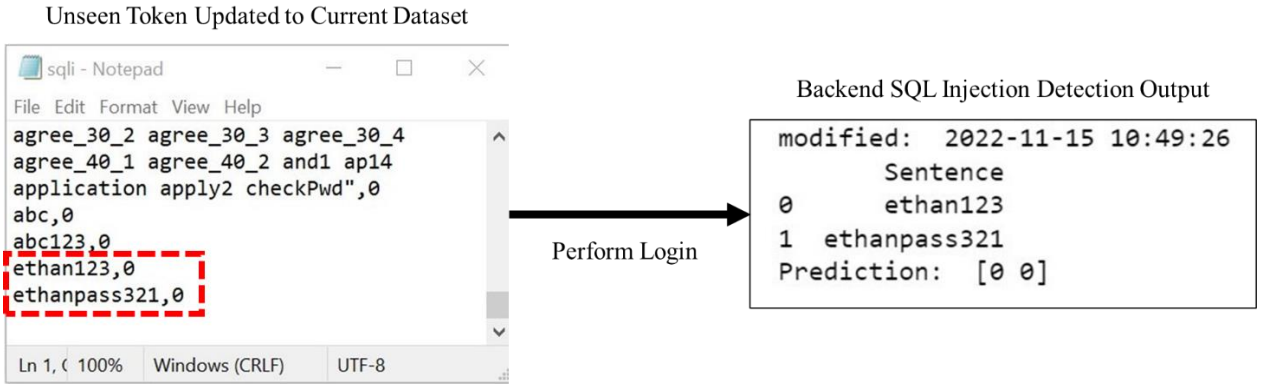


Figure 5.7: Unseen Token Successfully Updated to Dataset after System Restart

SQL injected inputs are also tested on the sign up and login functions as per the test cases in Table 3.2. In Figure 5.8 below shows result of the attempt to inject SQL query to the input with intention of deleting a user profile record. However, it is shown that the SQL injection has failed with the message of SQL injection detected is prompted. The sign up of the profile is also not executed. As for the login function in Figure 5.9 also resulted in the same outcome after attempting SQL injection. This is due to the back-end SQL injection detection system has successfully detected the injection with output shown in Figure 5.10. The detection system has output of 1 for the injected statement indicating it is a malicious input. Hence, the web application with implemented detection system has successfully detected and prevented the execution of SQL injected statement.

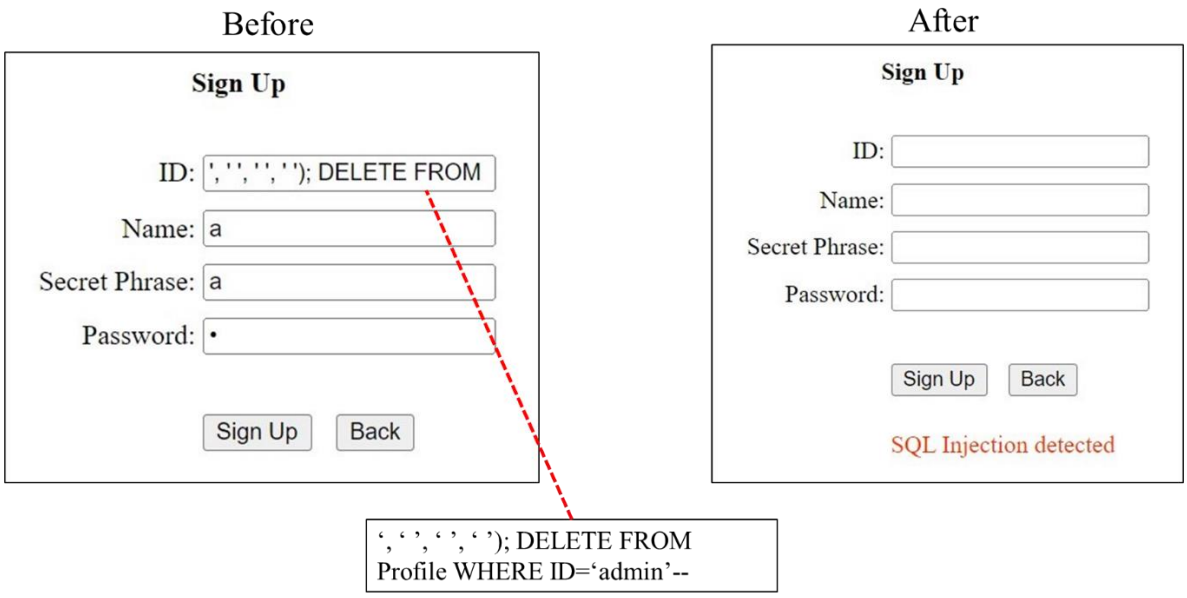


Figure 5.8: Attempt SQL Injection in Sign Up Page

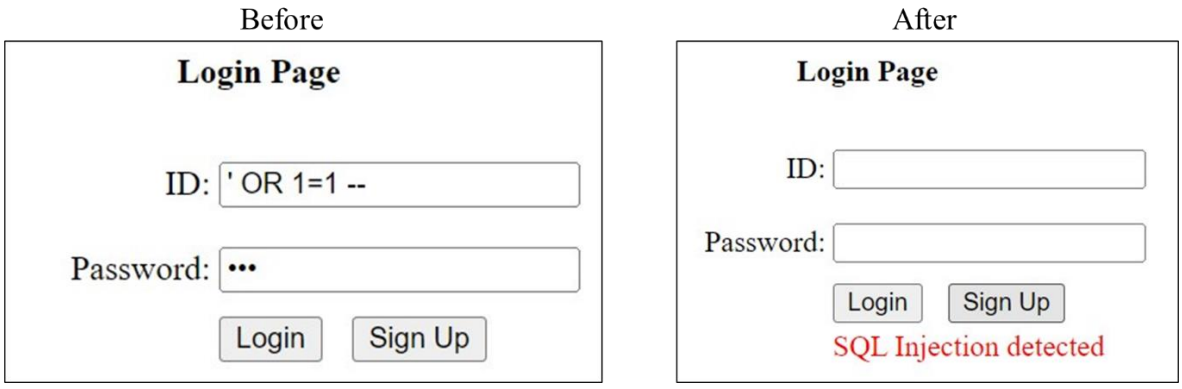
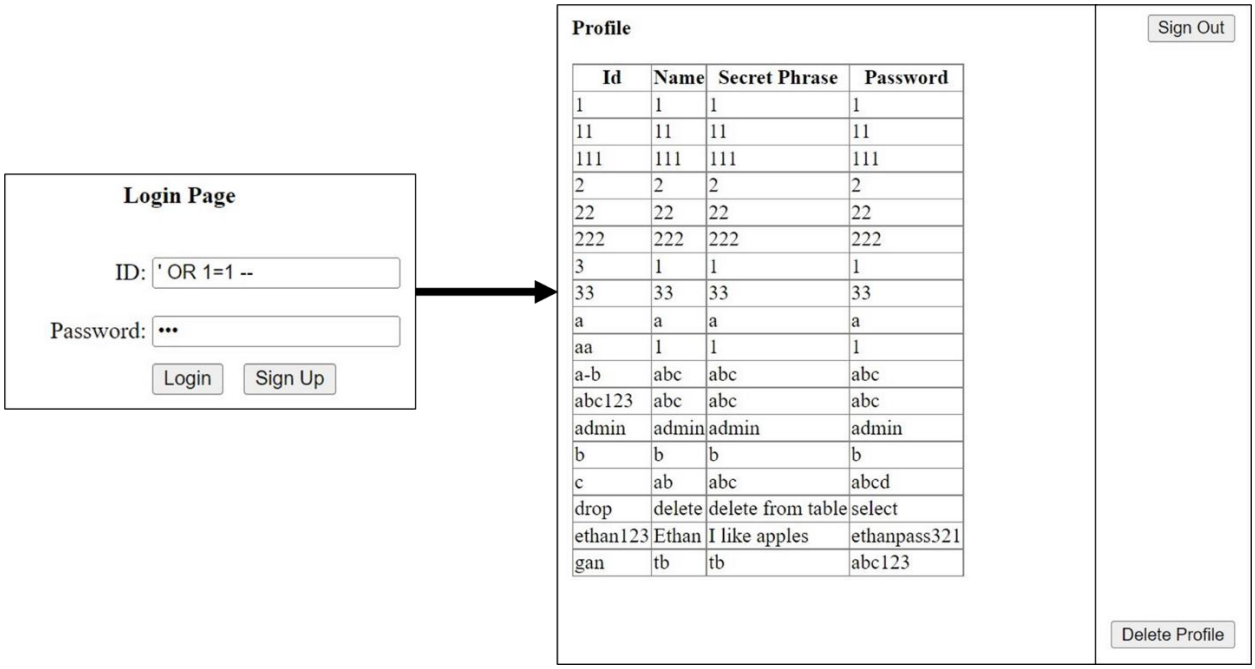


Figure 5.9: Attempt SQL Injection in Login Page

```
modified:  2022-11-15 10:59:34
                                                    Sentence
'  ' '  ' '   ' '); DELETE FROM Profile WHERE ID='admin'--
a NaN  NaN                                                NaN
       NaN                                                NaN
       NaN                                                NaN
Prediction:  [1 0 0 0]
modified:  2022-11-15 11:09:02
      Sentence
0  ' OR 1=1 --
1          abc
Prediction:  [1 0]
```

Figure 5.10: SQL Injection Detection System Output for Attempted SQL Injection

Table 3.2 test case also included a test to attempt SQL injection with the detection system in-active. This is to further show a comparison of the demo website with and without the SQL injection detection system. Figure 5.11 shows the result of SQL injection attempt on the login page to select all user profile information with no detection system. The SQL injection was a success as all records were retrieved in the profile page. This is due to without the detection system the website itself will treat all inputs as normal inputs and execute the query. Hence, this shows that the demo website is vulnerable to SQL injection, but with the back-end detection system, it is able to defend the website for its SQL injection vulnerabilities.



Id	Name	Secret Phrase	Password
1	1	1	1
11	11	11	11
111	111	111	111
2	2	2	2
22	22	22	22
222	222	222	222
3	1	1	1
33	33	33	33
a	a	a	a
aa	1	1	1
a-b	abc	abc	abc
abc123	abc	abc	abc
admin	admin	admin	admin
b	b	b	b
c	ab	abc	abcd
drop	delete	delete from table	select
ethan123	Ethan	I like apples	ethanpass321
gan	tb	tb	abc123

Figure 5.11: Attempt SQL Injection in Login with Inactive Detection System

5.2 Discussions

Analysis has been done on the obtained results of Table 5.1. Each machine learning model with different vectorizers used has their prediction quality measured using accuracy, precision, recall, and F1-score. Prediction of a machine learning model can be correct or incorrect, which generally categorized into true negative (TN), true positive (TP), false negative (FN), and false positive (FP) (Krishnan, 2018). True negative and true positive are predictions where a case is correctly predicted to be negative and positive respectively. As for false negative, is when the machine predicted a negative case, but it is actually a positive case. While false positive is when a positive case is predicted, but the correct answer is a negative. The four category of predictions is used to calculate each quality of overall prediction, summarized in Table 5.2 below.

Table 5.2: Classification Metrics Formula, adapted from Krishnan (2018)

Quality	Formula
Accuracy =	(TP+TN)/(TP+TN+FP+FN)
Precision =	TP/(TP+FP)
Recall =	TP/(TP+FN)
F1-score =	2*(Recall*Precision)/(Recall+Precision)

Based on Table 5.2, accuracy is the overall percentage of correct predictions made, and precision is the percentage of correct positive predictions. As for recall is the percentage of correctly predicted positive cases, while F1-score is the weighted harmonic mean of precision and recall. By understanding the qualities of measure in machine learning classification, it can be concluded that the best model is Random Forest Classifier with TfidfVectorizer (consider all tokens), where it is able to make 99.10% correct predictions, with 97.98% correct positive predictions, and 98.13% correctly predicted positive cases among all positive cases.

Random Forest Classifier is able to stand out among other classifiers is due to it uses concept of multiple decision trees and taking the majority prediction. Comparing Random Forest with Multinomial NB, the latter model uses concept of probability which is also a strong classification model achieving more than 90% accuracy. However, Multinomial NB only make the prediction once, unlike Random Forest which is able to make multiple predictions before concluding a final prediction among its decision trees. This has slightly reduced the performance of Multinomial NB comparing with Random Forest Classifier.

AdaBoost Classifier is also able to obtain overall accuracy more than 90%, but it is still not better than Random Forest Classifier. In this project, AdaBoost Classifier uses a base estimator of Decision Tree Classifier and undergo multiple iterations before concluding its prediction. However, multiple iterations are still unable to outperform predictions of multiple individual decision trees. This is due to the strong concept of Random Forest using the wisdom of crowds, where collective result of many independent models is able to surpass one individual strong model (Yiu, 2019). Multiple independent models are also able to defend each other's errors, even with one wrong prediction, other models are still able to make a correct prediction.

Among the vectorizers used, TfidfVectorizer is able to achieve better results than CountVectorizer when considering all tokens. For example, Random Forest model using TfidfVectorizer (consider all tokens) achieved 99.10% accuracy, while Random Forest model using CountVectorizer (consider all tokens) achieved 98.60% accuracy. This is due to TfidfVectorizer considers overall weightage of a token in a document, and able to penalize tokens that appears too frequent or too rare. As for CountVectorizer, it only considers the frequency of the token, and unable to identify significance of a token causing bias towards the most frequently appeared token (Saket, 2020). Hence, when a large number of tokens are considered, TfidfVectorizer performs

better as it does not simply rely on the frequency of a token, rather it also considers the importance of the token.

As a contrast, CountVectorizer performs slightly better than TfidfVectorizer when only specified tokens are considered. This can be showed where Random Forest using CountVectorizer (specified features) obtained 94.45% accuracy, while the same classifier with TfidfVectorizer (specified features) obtained 94.31%. AdaBoost Classifier also showed similar results with 94.40% and 94.21% accuracy for CountVectorizer (specified features) and TfidfVectorizer (specified features) respectively. This may be due to there are only limited token or features considered, where the features are only SQL injection characteristics. Hence, the significance of considering token importance is low when there are only a few tokens to be considered, and token frequency plays a slightly more important role.

When comparing considering all tokens and specified features, taking all tokens generally gives a better performance. For example, Random Forest using TfidfVectorizer (consider all tokens) is able to achieve 99.10% accuracy, while with the same model but only taking specified features only gives 94.31% accuracy. This shows that as more tokens are considered, the classifier is able to give a better prediction. It is also logical to conclude that the more the machine learning model learns, the better it can predict. However, when more tokens are considered, more processing power will be needed, which is a sacrifice for better prediction. This factor also has the potential to be further studied in future projects.

Based on the results in Table 5.1, Random Forest Classifier with TfidfVectorizer (consider all tokens) has the best performance result. Hence, the model is chosen to be used in the website SQL injection detection system. The demo website with integrated SQL injection detection system is able to successfully detect and prevent malicious

users from performing a SQL injection attack. It is also able to successfully perform normal functions with non-malicious users. Furthermore, to have self-evolving properties, the detection system is successful in identifying unseen tokens and update to its existing dataset on system restart. This is able to overcome the limitation of existing dataset and improve its predicting ability base on the respective group of website users.

However, the demo website with integrated SQL injection detection system requires to save user input including passwords to temporary files. This can pose a security risk if the files are exposed to public users, as the user passwords, ID and other information are recorded in the files. Security measures such as apply strong encryption to the files to avoid information being leaked can be imposed. Other methods such as imposing a authorization to only certain administrative users to access the files can also be done. Methods on improving security access of files and information transfer between the back-end detection system and front-end website should be further studied in future projects. This is to further refine the integration of SQL injection detection system on web applications.

CHAPTER 6

CONCLUSION

6.1 Critical Evaluation

Throughout this project, the established aims and objectives are achieved successfully during the process. The fundamental purpose of this project is to develop, design, and evaluate an enhanced machine learning model to detect and prevent SQL injections in web applications. This is achieved by successfully integrating a SQL injection detection system to a prototype web application. The integrated system is also successful in learning unseen inputs by updating unseen tokens to existing dataset. As for objectives in this project, it is achieved through understanding theories of SQL and SQL vulnerabilities in web applications, and training machine learning models for accuracy evaluation. Hence, in terms of completion, this project is successful in accomplishing intended aims and objectives throughout the process.

SQL vulnerabilities in web applications and capabilities of machine learning are important knowledge learnt during the project. SQL is a common database management language used in many web applications. It is important to understand its potential vulnerabilities to develop stronger and better web applications. As for machine learning, it is becoming a popular technology helping businesses and organizations in making better decisions in various domains (Singh, 2022). Understanding more on machine learning models and datasets gives a large advantage for future technology, as it can be extended to many different fields and solving different problems.

Challenges faced in this project include integration of systems between two different platforms. The back-end SQL injection detection system uses Jupyter Notebook, while the demo web application uses Visual Studio. Data and parameters are shared between the two platforms using shared file access, where the challenge is to avoid dead-lock

situations between file accesses. Another challenge is to select specific machine learning models for evaluations. There are many different types of machine learning models that can be evaluated to further understand their differences and advantages. However, due to time constraints, three machine learning models that are suitable for text classification are chosen for evaluation.

If this project can be repeated, several factors can be further improved including increasing the number of data samples, as larger dataset can improve the model's accuracy. Also, the dataset can be further balanced with almost an equal number of SQL injected statements and non-SQL injected statements. This can expose the machine learning model to a wider range of token types and improve its ability to identify specific features among the data samples.

6.2 Concluding Discussions

This project report evaluated three different machine learning models to detect SQL injection attacks. The three machine learning models include Multinomial NB, Random Forest Classifier, and AdaBoost Classifier. Evaluation of each machine learning model used a different combination of vectorizer and token features. Based on the evaluation result, Random Forest Classifier using TfidfVectorizer (consider all tokens) has the best performance metrics, which achieved 99.10% accuracy, 97.98% precision, 98.13% recall, and 99.02% F1-score. A SQL injection detection system is then built using the classifier with the best performance. The detection system is integrated into a simple web application, where the website allows users to perform simple login, sign up and view profile functions.

The SQL injection detection system is successfully integrated to the demo web application as a back-end system. The detection system obtains and processes user inputs from the website in order to determine if the user input is a SQL injection attack. Based on the detection system output, the front-end website will prevent the execution

of the user query if SQL injection is detected. If user input is determined as safe input, the website will execute the user request as intended. Through the website testing results, the website with SQL injection detection system integrated is able to successfully prevent execution of SQL injected inputs. As a comparison, the website is vulnerable to SQL injection if the back-end detection system is inactive.

The integrated SQL injection detection system has also successfully handled unseen token cases, where the existing dataset does not consist of such tokens. This can happen due there are many different unique names and ID combinations that the existing dataset may not have a sample of, causing a new token to be added. The web application records all unseen tokens to a temporary file, while the back-end system updates the existing dataset on system restart. Hence, when the detection system encounters the same token, it will successfully identify the token and generate a definite output of the web application.

6.3 Future Research Directions

This project still has room for further research and improvement, where the integration of the detection system to the web application uses temporary files to transfer data and information. This may raise some security issues where the user inputs including passwords and ID are recorded to temporary files for back-end system processing. Therefore, information transfer between websites and back-end detection systems has to be further studied to determine the most efficient and safe method for data transfer. For example, implementing strong encryption or user authorization. Future research on how to securely store files and access them confidentially should also be done, as the training dataset has to be stored securely with a lot of user information stored in the file.

Further extending this project will also lead to identifying more efficient ways of file accessing, where two different systems have to access the same file to share

information. For example, the back-end python program and the front-end C# program reads and writes to the same file for communication between the two programs. Further research will explore better methods or alternatives to access shared files between programs. There also can be specific programs or functions to assist programs on different platforms to share parameters that are worth researching.

Another future research direction is on the machine learning aspect, where more machine learning models can be explored. This is to determine the types and conditions of each model for it to have maximum performance by exploiting their characteristics. A specific case can be finding the most suitable specifications and parameters for a classifier to outperform other classifiers. Furthermore, with a machine learning approach, prevention and detection of other web application attacks can be expanded beyond SQL injection. Prevention of other web application vulnerabilities such as broken access control or cryptographic failures can be further studied using machine learning techniques.

REFERENCES

ACM. (n.d.). *ACM Digital Library*. Retrieved from https://dl.acm.org/

Ali, Z. (2022, July 3). *The Beginner's Guide to Kaggle*. Retrieved from ELITE Data Science: https://elitedatascience.com/beginner-kaggle

Bisht, P., Madhusudan, P., & Venkatakrishnan, V. N. (2010). CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks. *ACM Transactions on Information and System Security, 13*(2), 1–39. Retrieved from https://doi.org/10.1145/1698750.1698754

Bockermann, C., Apel, M., & Meier, M. (2009). Learning SQL for Database Intrusion Detection Using Context-Sensitive Modelling (Extended Abstract). In U. Flegel, & D. Bruschi (Ed.), *Detection of Intrusions and Malware, and Vulnerability Assessment* (pp. 196–205). Springer, Berlin, Heidelberg. doi:https://doi.org/10.1007/978-3-642-02918-9_12

Clarke, J. (2009). Chapter 8 - Code-Level Defenses. In *SQL Injection Attacks and Defense* (pp. 341-376). Elsevier Inc. Retrieved from https://doi.org/10.1016/B978-1-59749-424-3.00008-6

Das, D., Sharma, U., & Bhattacharyya, D. K. (2019). Defeating SQL Injection Attack in Authentication Security: An Experimental Study. *International Journal of Information Security, 18*(1), 1-22. Retrieved from https://doi.org/10.1007/s10207-017-0393-x

Elith, J., Leathwick, J. R., & Hastie, T. (2008). A working guide to boosted regression trees. *Journal of Animal Ecology, 77*(4), 802-813. Retrieved from https://doi.org/10.1111/j.1365-2656.2008.01390.x

Fioravanti, M. E., & Mayron, L. M. (2014). Measuring the Effectiveness of Output Filtering against SQL Injection Attacks. *Proceedings of the 2014 ACM Southeast Regional Conference* (pp. 1-6). Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2638404.2638457

Hasan, M., Balbahaith, Z., & Tarique, M. (2019). Detection of SQL Injection Attacks:

A Machine Learning Approach. *2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA)*, (pp. 1-6). doi:10.1109/ICECTA48151.2019.8959617

Hosam, E., Hosny, H., Ashraf, W., & Kaseb, A. S. (2021). SQL Injection Detection Using Machine Learning Techniques. *2021 8th International Conference on Soft Computing & Machine Intelligence (ISCMI)*, (pp. 15-20). doi:10.1109/ISCMI53840.2021.9654820

IBM Cloud Education. (2020, July 15). *Machine Learning*. Retrieved from IBM: https://www.ibm.com/my-en/cloud/learn/machine-learning

IEEE. (n.d.). *IEEE Xplore*. Retrieved from https://ieeexplore.ieee.org/Xplore/home.jsp

Jain, P. (2021, May 2). *Basics of CountVectorizer*. Retrieved from Towards Data Science: https://towardsdatascience.com/basics-of-countvectorizer-e26677900f9c

javaTpoint. (n.d.). *Decision Tree Classification Algorithm*. Retrieved from javaTpoint: https://www.javatpoint.com/machine-learning-decision-tree-classification-algorithm

Joshi, P. (2022). *An Introduction to Text Summarization using the TextRank Algorithm (with Python implementation)*. Retrieved from Analytics Vidhya: https://www.analyticsvidhya.com/blog/2018/11/introduction-text-summarization-textrank-python/

Jothi, K. R., Saravana, B. B., Pandey, N., Beriwal, P., & Amarajan, A. (2021). An Efficient SQL Injection Detection System Using Deep Learning. *2021 International Conference on Computational Intelligence and Knowledge Economy (ICCIKE)* (pp. 442-445). IEEE. doi:10.1109/ICCIKE51210.2021.9410674

Kaggle. (n.d.). *Start with more than a blinking cursor*. Retrieved from Kaggle: https://www.kaggle.com/

Kamtuo, K., & Soomlek, C. (2016). Machine Learning for SQL injection prevention on server-side scripting. *2016 International Computer Science and*

Engineering Conference (ICSEC) (pp. 1-6). IEEE. doi:10.1109/ICSEC.2016.7859950

Katole, R. A., Sherekar, S. S., & Thakare, V. M. (2018). Detection of SQL injection attacks by removing the parameter values of SQL query. *2018 2nd International Conference on Inventive Systems and Control (ICISC)* (pp. 736-741). IEEE. doi:10.1109/ICISC.2

Keras. (n.d.). *Keras*. Retrieved from https://keras.io/

Krishnan, M. (2018, July 7). *Understanding the Classification report through sklearn*. Retrieved from Muthukrishnan: https://muthu.co/understanding-the-classification-report-in-sklearn/

Lendave, V. (2021, August 7). *Hands-On Tutorial on Performance Measure of Stratified K-Fold Cross-Validation*. Retrieved from Analytics in Diamag: https://analyticsindiamag.com/hands-on-tutorial-on-performance-measure-of-stratified-k-fold-cross-validation/

Li, B., & Lu, P. (2021). *Two-Class Support Vector Machine component*. Retrieved from Microsoft: https://learn.microsoft.com/en-us/azure/machine-learning/component-reference/two-class-support-vector-machine

Luo, Y. (2021). SQLi-Fuzzer: A SQL Injection Vulnerability Discovery Framework Based on Machine Learning. *2021 IEEE 21st International Conference on Communication Technology (ICCT)*, (pp. 846-851). doi:10.1109/ICCT52962.2021.9657925

MathWorks. (n.d.). *Classification*. Retrieved from MathWorks: https://www.mathworks.com/help/stats/classification.html

Mehta, P., Sharda, J., & Das, M. L. (2015). SQLshield: Preventing SQL Injection Attacks by Modifying User Input Data. *International Conference on Information Systems Security* (pp. 192–206). Springer, Cham. Retrieved from https://doi.org/10.1007/978-3-319-26961-0_12

Minhas, J., & Kumar, R. (2013). Blocking of SQL Injection Attacks by Comparing Static and Dynamic Queries. *International Journal of Computer Network and*

Information Security, 5(2), 1-9. doi:10.5815/ijcnis.2013.02.01

Navlani, A. (2018, November). *AdaBoost Classifier in Python*. Retrieved from datacamp: https://www.datacamp.com/tutorial/adaboost-classifier-python

NIST. (n.d.-a). *NIST SOFTWARE ASSURANCE REFERENCE DATASET*. Retrieved from https://samate.nist.gov/SARD/

NIST. (n.d.-b). *NVD*. Retrieved from https://nvd.nist.gov/vuln

OWASP. (n.d.-a). *Fuzzing*. Retrieved from OWASP: https://owasp.org/www-community/Fuzzing

OWASP. (n.d.-b). *OWASP Top 10: 2021*. Retrieved from https://owasp.org/Top10/.

Oza, S. (n.d.). *SQL Injection Attacks (SQLi) — Web-based Application Security, Part 4*. Retrieved from Spanning: https://spanning.com/blog/sql-injection-attacks-web-based-application-security-part-4/

Parashar, D., Sanagavarapu, L. M., & Reddy, Y. R. (2021). SQL Injection Vulnerability Identification from Text. *14th Innovations in Software Engineering Conference (Formerly Known as India Software Engineering Conference)* (pp. 1-5). Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3452383.3452405

Pollack, E. (2017, February 17). *Sanitizing Inputs: Avoiding Security and Usability Disasters*. Retrieved from SQLShack: https://www.sqlshack.com/sanitizing-inputs-avoiding-security-usability-disasters/

Raj, A. (2020, November 7). *Perfect Recipe for Classification Using Logistic Regression*. Retrieved from Towards Data Science: https://towardsdatascience.com/the-perfect-recipe-for-classification-using-logistic-regression-f8648e267592#:~:text=Logistic%20Regression%20is%20a%20classification%20technique%20used%20in%20machine%20learning,cancer%20is%20malignant%20or%20not).

Ratz, A. V. (2021, May 17). *Multinomial Naïve Bayes' For Documents Classification and Natural Language Processing (NLP)*. Retrieved from Towards Data

Science: https://towardsdatascience.com/multinomial-na%C3%AFve-bayes-for-documents-classification-and-natural-language-processing-nlp-e08cc848ce6#:~:text=The%20multinomial%20na%C3%AFve%20Bayes%20is,drastically%20simplifies%20textual%20data%20classification.

Ross, K. (2018). SQL Injection Detection Using Machine Learning Techniques and Multiple Data Source. *Master's Projects*, 650. doi:https://doi.org/10.31979/etd.zknb-4z36

Roy, P., Kumar, R., & Rani, P. (2022). SQL Injection Attack Detection by Machine Learning Classifier. *2022 International Conference on Applied Artificial Intelligence and Computing (ICAAIC)*, (pp. 394-400). doi:10.1109/ICAAIC53929.2022.9792964

Saha, A., & Sanyal, S. (2014). Application Layer Intrusion Detection with Combination of Explicit-RuleBased and Machine Learning Algorithms and Deployment in CyberDefence Program. *Computing Research Repository*. Retrieved from http://arxiv.org/abs/1411.3089

Saket, S. (2020, January 13). *Count Vectorizer vs TFIDF Vectorizer | Natural Language Processing*. Retrieved from LinkedIn: https://www.linkedin.com/pulse/count-vectorizers-vs-tfidf-natural-language-processing-sheel-saket/

Schallmo, D., Williams, C. A., & Boardman, L. (2017). Digital Transformation of Business Models-Best Practice, Enablers, and Roadmap. *International Journal of Innovation Management, 21*(8), 1740014. Retrieved from https://doi.org/10.1142/S136391961740014X

Scikit-learn. (n.d.). *scikit-learn*. Retrieved from https://scikit-learn.org/stable/

Singh, P. (2022, May 30). *What is the Future of Machine Learning?* Retrieved from naukri learning: https://www.naukri.com/learning/articles/future-of-machine-learning/

Sivasangari, A., Jyotsna, J., & Pravalika, K. (2021). SQL Injection Attack Detection using Machine Learning Algorithm. *2021 5th International Conference on*

Trends in Electronics and Informatics (ICOEI), (pp. 1166-1169). doi:10.1109/ICOEI51242.2021.9452914

Solanki, G. (2022, September 9). *Multinomial Naive Bayes Explained*. Retrieved from Great Learning Team: https://www.mygreatlearning.com/blog/multinomial-naive-bayes-explained/

Tarpey, M. (2020). *A Brief History of Digitization*. Retrieved from Exela Technologies: https://www.exelatech.com/blog/brief-history-digitization?language_content_entity=en

Tasevski, I., & Jakimoski, K. (2020). Overview of SQL Injection Defense Mechanisms. *2020 28th Telecommunications Forum (TELFOR)* (pp. 1-4). IEEE. doi:10.1109/TELFOR51502.2020.9306676

Tripathy, D., Gohil, R., & Halabi, T. (2020). Detecting SQL Injection Attacks in Cloud SaaS using Machine Learning. *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)* (pp. 145-150). IEEE. doi:10.1109/BigDataSecurity-HPSC-IDS49724.2020.00035

TrustNet. (n.d.). *Common Web Application Attacks*. Retrieved from TrustNet: https://www.trustnetinc.com/web-application-attacks/#:~:text=Web%2DBased%20Attacks%20Defined,be%20kept%20private%20and%20safe.

Unnikrishnan, C. S. (2021, November 3). *How sklearn's Tfidfvectorizer Calculates tf-idf Values*. Retrieved from Analytics Vidhya: https://www.analyticsvidhya.com/blog/2021/11/how-sklearns-tfidfvectorizer-calculates-tf-idf-values/

Veronica, A. (2020, May 16). *Understanding Adaboost and Scikit-learn's algorithm:*. Retrieved from Data Driven Investor: https://medium.datadriveninvestor.com/understanding-adaboost-and-scikit-learns-algorithm-c8d8af5ace10

Yiu, T. (2019, June 12). *Understanding Random Forest*. Retrieved from Towards Data Science: https://towardsdatascience.com/understanding-random-forest-58381e0602d2

Zhang, K. (2019). A Machine Learning based Approach to Identify SQL Injection Vulnerabilities. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 1286-1288). IEEE. doi:10.1109/ASE.2019.00164

APPENDICES

APPENDIX A

Jupyter Notebook Python Machine Learning Source Code

```
#Load libraries
from pandas import read_csv
from pandas.plotting import scatter_matrix
from matplotlib import pyplot
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.svm import SVC
```

Update Dataset (unseen inputs)

```
import csv
#Append unknown/new input to existing dataset
#remove duplicates from unknowns dataset
tempList=[]
with open(r"D:\Machine Learning\SQLInjection\unknowns.csv", mode="r", newline="") as file:
    reader=csv.reader(file)
    for line in reader:
        tempList.append(line)

# only take distinct values
distinctList=[]
for item in tempList:
    if item not in distinctList:
        distinctList.append(item)

print(tempList)
print(distinctList)

if distinctList: #if list is not empty
    #write into existing dataset
    with open(r"D:\Machine Learning\SQLInjection\sqli.csv", mode="a", newline="") as file:
        writer=csv.writer(file)
        for item in distinctList:
            writer.writerow(item)

# clear unknown file, remove all contents
open(r"D:\Machine Learning\SQLInjection\unknowns.csv", mode="w").close()
```

Load Dataset and Vectorize Dataset

```
#Load dataset
df=read_csv("D:\Machine Learning\SQLInjection\sqli.csv", encoding="utf-8")
#pandas.read_csv(), reads comma separated csv file and returns to DataFrame

X=df["Sentence"]
Y=df["Label"]
```

```
#View dataset

#view shape (dimension) of dataset
print(df.shape) #pandas.DataFrame.shape, returns (rows, col)

#view first 5 rows of data
print(df.head(10)) #pandas.DataFrame.head

#view row 2 to 4 of data
print("\n",df.loc[2:5,"Sentence"])

print(df.loc[39, "Sentence"])
```

```
import pandas as pd
import re #regex
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
import nltk
from nltk.corpus import stopwords #stopwords e.g. and, to, in, the...
from nltk.tokenize import RegexpTokenizer
import numpy as np
import warnings

def specificSQLTokens(text):
    # get features of SQL injection query
    tokenizer=RegexpTokenizer("and|union|or|delete|drop table|--|\'|\"|;|[0-9] += [0-9]|[0-9]+=[0-9]|[a-z] += [a-z]|[a-z]+=[a
    return tokenizer.tokenize(text)

def includeAllTokens(text):
    # create a space between special characters
    text=re.sub("(\\W)", " \\1 ", text)
    # split based on whitespace
    return re.split("\\s+",text)

temp=includeAllTokens("\'A,B,c--")
print(temp)

fixTokenCountVectorizer=CountVectorizer(tokenizer=specificSQLTokens)
countVectorizer=CountVectorizer(tokenizer=includeAllTokens) #token include all special characters
fixTokentfidfVectorizer=TfidfVectorizer(tokenizer=specificSQLTokens)
tfidfVectorizer=TfidfVectorizer(tokenizer=includeAllTokens) #token include all special characters

kfold=StratifiedKFold(n_splits=10, random_state=1, shuffle=True)
```

Note: full text of RegexpTokenizer() is

RegexpTokenizer("and|union|or|delete|drop table|--|\'|\"|;|[0-9] += [0-9]|[0-9]+=[0-9]|[a-z] += [a-z]|[a-z]+=[a-z]")

CountVectorizer (fixed SQL injection tokens)

```
X=df["Sentence"]
X = fixTokenCountVectorizer.fit_transform(X.values.astype('U')).toarray() #convert to U -> Unicode
#print(X)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.2, random_state=1)
print(X_train.shape)
print(Y_train.shape)
print(X_test.shape)
print(Y_test.shape)

warnings.filterwarnings('ignore')

nb_clf = MultinomialNB()
nb_clf.fit(X_train, Y_train)
Y_pred = nb_clf.predict(X_test)
print(f"Classification Report of Multinomial Naive Bayes on test set: \n{classification_report(Y_test, Y_pred, digits=3)}")
print(f"10 fold Cross Validation: ")
cv_results=cross_val_score(nb_clf, X, Y, cv=kfold, scoring='accuracy')
print(cv_results)
print('Multinomial Naive Bayes Accuracy: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(nb_clf, X, Y, cv=kfold, scoring='precision')
print('Multinomial Naive Bayes Precision: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(nb_clf, X, Y, cv=kfold, scoring='recall')
print('Multinomial Naive Bayes Recall: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(nb_clf, X, Y, cv=kfold, scoring='f1_weighted')
print('Multinomial Naive Bayes F1 score: %f (%f)\n' %(cv_results.mean(),cv_results.std()))
```

```
randomForest_clf = RandomForestClassifier()
randomForest_clf.fit(X_train, Y_train)
Y_pred = randomForest_clf.predict(X_test)
print(f"Classification Report of Random Forest Classifier on test set: \n{classification_report(Y_test, Y_pred, digits=3)}")
print(f"10 fold Cross Validation: ")
cv_results=cross_val_score(randomForest_clf, X, Y, cv=kfold, scoring='accuracy')
print('Random Forest Classifier Accuracy: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(randomForest_clf, X, Y, cv=kfold, scoring='precision')
print('Random Forest Classifier Precision: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(randomForest_clf, X, Y, cv=kfold, scoring='recall')
print('Random Forest Classifier Recall: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(randomForest_clf, X, Y, cv=kfold, scoring='f1_weighted')
print('Random Forest Classifier F1 score: %f (%f)\n' %(cv_results.mean(),cv_results.std()))

adaBoost_clf = AdaBoostClassifier()
adaBoost_clf.fit(X_train, Y_train)
Y_pred = adaBoost_clf.predict(X_test)
print(f"Classification Report of AdaBoost Classifier on test set: \n{classification_report(Y_test, Y_pred, digits=3)}")
print(f"10 fold Cross Validation: ")
cv_results=cross_val_score(adaBoost_clf, X, Y, cv=kfold, scoring='accuracy')
print('AdaBoost Classifier Accuracy: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(adaBoost_clf, X, Y, cv=kfold, scoring='precision')
print('AdaBoost Classifier Precision: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(adaBoost_clf, X, Y, cv=kfold, scoring='recall')
print('AdaBoost Classifier Recall: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(adaBoost_clf, X, Y, cv=kfold, scoring='f1_weighted')
print('AdaBoost Classifier F1 score: %f (%f)\n' %(cv_results.mean(),cv_results.std()))
```

CountVectorizer (take all as tokens including special characters)

```
X=df["Sentence"]
X = countVectorizer.fit_transform(X.values.astype('U')).toarray() #convert to U -> Unicode

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.2, random_state=1)
print(X_train.shape)
print(Y_train.shape)
print(X_test.shape)
print(Y_test.shape)

warnings.filterwarnings('ignore')

nb_clf = MultinomialNB()
nb_clf.fit(X_train, Y_train)
Y_pred = nb_clf.predict(X_test)
print(f"Classification Report of Multinomial Naive Bayes on test set: \n{classification_report(Y_test, Y_pred, digits=3)}")
print(f"10 fold Cross Validation: ")
cv_results=cross_val_score(nb_clf, X, Y, cv=kfold, scoring='accuracy')
print('Multinomial Naive Bayes Accuracy: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(nb_clf, X, Y, cv=kfold, scoring='precision')
print('Multinomial Naive Bayes Precision: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(nb_clf, X, Y, cv=kfold, scoring='recall')
print('Multinomial Naive Bayes Recall: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(nb_clf, X, Y, cv=kfold, scoring='f1_weighted')
print('Multinomial Naive Bayes F1 score: %f (%f)\n' %(cv_results.mean(),cv_results.std()))
```

```
randomForest_clf = RandomForestClassifier()
randomForest_clf.fit(X_train, Y_train)
Y_pred = randomForest_clf.predict(X_test)
print(f"Classification Report of Random Forest Classifier on test set: \n{classification_report(Y_test, Y_pred, digits=3)}")
print(f"10 fold Cross Validation: ")
cv_results=cross_val_score(randomForest_clf, X, Y, cv=kfold, scoring='accuracy')
print('Random Forest Classifier Accuracy: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(randomForest_clf, X, Y, cv=kfold, scoring='precision')
print('Random Forest Classifier Precision: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(randomForest_clf, X, Y, cv=kfold, scoring='recall')
print('Random Forest Classifier Recall: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(randomForest_clf, X, Y, cv=kfold, scoring='f1_weighted')
print('Random Forest Classifier F1 score: %f (%f)\n' %(cv_results.mean(),cv_results.std()))

adaBoost_clf = AdaBoostClassifier()
adaBoost_clf.fit(X_train, Y_train)
Y_pred = adaBoost_clf.predict(X_test)
print(f"Classification Report of AdaBoost Classifier on test set: \n{classification_report(Y_test, Y_pred, digits=3)}")
print(f"10 fold Cross Validation: ")
cv_results=cross_val_score(adaBoost_clf, X, Y, cv=kfold, scoring='accuracy')
print('AdaBoost Classifier Accuracy: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(adaBoost_clf, X, Y, cv=kfold, scoring='precision')
print('AdaBoost Classifier Precision: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(adaBoost_clf, X, Y, cv=kfold, scoring='recall')
print('AdaBoost Classifier Recall: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(adaBoost_clf, X, Y, cv=kfold, scoring='f1_weighted')
print('AdaBoost Classifier F1 score: %f (%f)\n' %(cv_results.mean(),cv_results.std()))
```

TfidfVectorizer (fixed SQL injection tokens)

```
X=df["Sentence"]
X = fixTokentfidfVectorizer.fit_transform(X.values.astype('U')).toarray() #convert to U -> Unicode
#print(X[0:10, :5])

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.2, random_state=1)
print(X_train.shape)
print(Y_train.shape)
print(X_test.shape)
print(Y_test.shape)

warnings.filterwarnings('ignore')

nb_clf = MultinomialNB()
nb_clf.fit(X_train, Y_train)
Y_pred = nb_clf.predict(X_test)
print(f"Classification Report of Multinomial Naive Bayes on test set: \n{classification_report(Y_test, Y_pred, digits=3)}")
print(f"10 fold Cross Validation: ")
cv_results=cross_val_score(nb_clf, X, Y, cv=kfold, scoring='accuracy')
print('Multinomial Naive Bayes Accuracy: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(nb_clf, X, Y, cv=kfold, scoring='precision')
print('Multinomial Naive Bayes Precision: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(nb_clf, X, Y, cv=kfold, scoring='recall')
print('Multinomial Naive Bayes Recall: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(nb_clf, X, Y, cv=kfold, scoring='f1_weighted')
print('Multinomial Naive Bayes F1 score: %f (%f)\n' %(cv_results.mean(),cv_results.std()))
```

```
randomForest_clf = RandomForestClassifier()
randomForest_clf.fit(X_train, Y_train)
Y_pred = randomForest_clf.predict(X_test)
print(f"Classification Report of Random Forest Classifier on test set: \n{classification_report(Y_test, Y_pred, digits=3)}")
print(f"10 fold Cross Validation: ")
cv_results=cross_val_score(randomForest_clf, X, Y, cv=kfold, scoring='accuracy')
print('Random Forest Classifier Accuracy: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(randomForest_clf, X, Y, cv=kfold, scoring='precision')
print('Random Forest Classifier Precision: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(randomForest_clf, X, Y, cv=kfold, scoring='recall')
print('Random Forest Classifier Recall: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(randomForest_clf, X, Y, cv=kfold, scoring='f1_weighted')
print('Random Forest Classifier F1 score: %f (%f)\n' %(cv_results.mean(),cv_results.std()))

adaBoost_clf = AdaBoostClassifier()
adaBoost_clf.fit(X_train, Y_train)
Y_pred = adaBoost_clf.predict(X_test)
print(f"Classification Report of AdaBoost Classifier on test set: \n{classification_report(Y_test, Y_pred, digits=3)}")
print(f"10 fold Cross Validation: ")
cv_results=cross_val_score(adaBoost_clf, X, Y, cv=kfold, scoring='accuracy')
print('AdaBoost Classifier Accuracy: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(adaBoost_clf, X, Y, cv=kfold, scoring='precision')
print('AdaBoost Classifier Precision: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(adaBoost_clf, X, Y, cv=kfold, scoring='recall')
print('AdaBoost Classifier Recall: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(adaBoost_clf, X, Y, cv=kfold, scoring='f1_weighted')
print('AdaBoost Classifier F1 score: %f (%f)\n' %(cv_results.mean(),cv_results.std()))
```

TfidfVectorizer (take all as tokens including special characters)

```
X=df["Sentence"]
X = tfidfVectorizer.fit_transform(X.values.astype('U')).toarray() #convert to U -> Unicode

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.2, random_state=1)
print(X_train.shape)
print(Y_train.shape)
print(X_test.shape)
print(Y_test.shape)

warnings.filterwarnings('ignore')

nb_clf = MultinomialNB()
nb_clf.fit(X_train, Y_train)
Y_pred = nb_clf.predict(X_test)
print(f"Classification Report of Multinomial Naive Bayes on test set: \n{classification_report(Y_test, Y_pred, digits=3)}")
print(f"10 fold Cross Validation: ")
cv_results=cross_val_score(nb_clf, X, Y, cv=kfold, scoring='accuracy')
print('Multinomial Naive Bayes Accuracy: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(nb_clf, X, Y, cv=kfold, scoring='precision')
print('Multinomial Naive Bayes Precision: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(nb_clf, X, Y, cv=kfold, scoring='recall')
print('Multinomial Naive Bayes Recall: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(nb_clf, X, Y, cv=kfold, scoring='f1_weighted')
print('Multinomial Naive Bayes F1 score: %f (%f)\n' %(cv_results.mean(),cv_results.std()))
```

```
randomForest_clf = RandomForestClassifier()
randomForest_clf.fit(X_train, Y_train)
Y_pred = randomForest_clf.predict(X_test)
print(f"Classification Report of Random Forest Classifier on test set: \n{classification_report(Y_test, Y_pred, digits=3)}")
print(f"10 fold Cross Validation: ")
cv_results=cross_val_score(randomForest_clf, X, Y, cv=kfold, scoring='accuracy')
print('Random Forest Classifier Accuracy: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(randomForest_clf, X, Y, cv=kfold, scoring='precision')
print('Random Forest Classifier Precision: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(randomForest_clf, X, Y, cv=kfold, scoring='recall')
print('Random Forest Classifier Recall: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(randomForest_clf, X, Y, cv=kfold, scoring='f1_weighted')
print('Random Forest Classifier F1 score: %f (%f)\n' %(cv_results.mean(),cv_results.std()))

adaBoost_clf = AdaBoostClassifier()
adaBoost_clf.fit(X_train, Y_train)
Y_pred = adaBoost_clf.predict(X_test)
print(f"Classification Report of AdaBoost Classifier on test set: \n{classification_report(Y_test, Y_pred, digits=3)}")
print(f"10 fold Cross Validation: ")
cv_results=cross_val_score(adaBoost_clf, X, Y, cv=kfold, scoring='accuracy')
print('AdaBoost Classifier Accuracy: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(adaBoost_clf, X, Y, cv=kfold, scoring='precision')
print('AdaBoost Classifier Precision: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(adaBoost_clf, X, Y, cv=kfold, scoring='recall')
print('AdaBoost Classifier Recall: %f (%f)' %(cv_results.mean(),cv_results.std()))
cv_results=cross_val_score(adaBoost_clf, X, Y, cv=kfold, scoring='f1_weighted')
print('AdaBoost Classifier F1 score: %f (%f)\n' %(cv_results.mean(),cv_results.std()))
```

Train Random Forest Model using all of dataset ¶

```
X=df["Sentence"]
X = tfidfVectorizer.fit_transform(X.values.astype('U')).toarray() #convert to U -> Unicode

randomForest_clf_all = RandomForestClassifier()
randomForest_clf_all.fit(X, Y)
print(X.shape)
print(Y.shape)
```

```
import pathlib
import datetime

#predict input
print("running")
inputCSVFile = "D:\Machine Learning\SQLInjection\input.csv"
f_name = pathlib.Path(inputCSVFile)
new_timestamp=f_name.stat().st_mtime
old_timestamp=new_timestamp

#keep program running
while True:
    new_timestamp=f_name.stat().st_mtime

    if new_timestamp>old_timestamp: #if inputCSVFile is modified
        print("modified: ",datetime.datetime.fromtimestamp(new_timestamp))
        old_timestamp=new_timestamp

        df1=read_csv(inputCSVFile)
        print(df1)

        if not df1.empty:
            #for i in range(len(df1)):

            df2=df[["Sentence"]] #original data set (double brackets: returns a data frame)
            df3=df2.append(df1, ignore_index=True) #append input sentence to original data set
            dfAppended=df3["Sentence"] #single bracket: returns a series

            #vectorize into array of "key words"
            tfidfVectorizer=TfidfVectorizer(tokenizer=includeAllTokens) #token include all special characters
            vector=tfidfVectorizer.fit_transform(dfAppended.values.astype('U')).toarray()
            #print(len(vector))
            #print(len(vector[0]))
            #print(len(X_train[0]))
```

```
#output file
outputFile = open("D:\Machine Learning\SQLInjection\output.txt", "w")

#if len(vector[0])>len(X_train[0]): #increase in column number indicates new key words are added
if len(vector[0])>len(X[0]):
    print("Unknown Input")

    X=df["Sentence"]
    X = tfidfVectorizer.fit_transform(X.values.astype('U')).toarray() #convert to U -> Unicode

    for i in range(len(df1)):#check each input
        unknownFlag=0
        tokenList = includeAllTokens(df1.iloc[i]["Sentence"])
        for k in tokenList:
            if(tfidfVectorizer.vocabulary_.get(k)==None): #if is new/unseen token
                unknownFlag=1
                break
            # else do nothing
        if(unknownFlag==1):
            outputFile.write("-1\n")
        else:
            outputFile.write("0\n")

else:
    prediction = randomForest_clf_all.predict(vector[len(vector)-len(df1):len(vector)])
    #predict input (appended to last, len(df1)=number of inputs)
    #prediction = randomForest_clf.predict(vector[len(vector)-len(df1):len(vector)])
    print("Prediction: ", prediction)
    for i in prediction:
        outputFile.write(repr(i)+"\n")

outputFile.close()
```

APPENDIX B

Visual Studio Web Application C# Source Code

Login.aspx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Data.SqlClient;
using System.Configuration;
using System.Data;
using System.IO;
using System.Text;
using System.Text.RegularExpressions;

namespace Website_Login
{
    public partial class Login : System.Web.UI.Page
    {
        //create connection
        SqlConnection con = new
SqlConnection(ConfigurationManager.ConnectionStrings["ConnectionString"].Conn
ectionString);
        protected void Page_Load(object sender, EventArgs e)
        {

        }

        //Login Button
        protected void Button1_Click(object sender, EventArgs e)
        {
            //prevent NULL or blank input
            if (TextBox1.Text == "" || TextBox2.Text == "")
            {
                Label1.Text = "Please Fill in All Fields";
                return;
            }
```

```
                //write input to temporary file
                string[] input = { "Sentence", TextBox1.Text, TextBox2.Text };
                writeInputToFile(input);

                //declare SQL command
                string cmdStr = "SELECT COUNT(*) FROM Profile WHERE ID='"
+ TextBox1.Text + "'AND Password='" + TextBox2.Text + "'";

                //function linking to machine learning system to detect injection
before executing the query
                smartQueryExecution(input, cmdStr);

                GridView1.DataBind();

                con.Close();
                TextBox1.Text = "";
                TextBox2.Text = "";
            }

            //Sign up Button
            protected void Button2_Click(object sender, EventArgs e)
            {
                Response.Redirect("SignUp.aspx");
            }

            //write to file function
            protected void writeInputToFile(string[] lines)
            {
                File.WriteAllLines("D:/Machine Learning/SQLInjection/input.csv",
lines);
            }

            protected void learnInput(string[] lines)
            {
                File.AppendAllLines("D:/Machine
Learning/SQLInjection/unknowns.csv", lines);
            }

            protected void smartQueryExecution (string[] input, string cmdStr)
            {
                //read output file to determine result
                var outputFile = "D:/Machine Learning/SQLInjection/output.txt";
                var new_timeStamp = File.GetLastWriteTime(outputFile);
```

```
                var old_timeStamp = new_timeStamp;

                int timeoutFlag = 0, counter = 0; //timeout counter for "infinite" while
loop
                string[] output = File.ReadAllLines(outputFile);
                int injectionFlag = 0;

                while (true) //loop until output file is updated
                {
                    new_timeStamp = File.GetLastWriteTime(outputFile); //get
modified time

                    if (new_timeStamp > old_timeStamp) //if output file is modified,
meaning detection program has completed
                    {

System.Diagnostics.Debug.WriteLine("outputFile_old_timeStamp: " +
old_timeStamp.ToString());

System.Diagnostics.Debug.WriteLine("outputFile_new_timeStamp: " +
new_timeStamp.ToString());
                        old_timeStamp = new_timeStamp;
                        counter = 0;

                        while (true) // loop until output file is read
                        {
                            try
                            {
                                //read output file
                                output = File.ReadAllLines(outputFile);
                                break;
                            }

                            catch (IOException ex)
                            {
                                // wait for 50ms, before reading file again
                                System.Threading.Thread.Sleep(50);
                            }
                        }

                        //stop inifinite loop
                        break;
                    }
```

```
                else
                {
                    counter++; //counter if file is not modified
                }

                if (counter == 100000) //if file is not modified after 1000 loops,
break loop
                {
                    System.Diagnostics.Debug.WriteLine("SQL Injection
Detection System Timeout");
                    counter = 0;
                    timeoutFlag = 1;
                    break;
                }
            }

            Label1.Text = "";
            if (timeoutFlag == 1) //timeout, detection program run unsuccessful
            {
                executeQuery(); //assuming there is no injection, run the query
regardless
            }

            else //no timeout, detection program running successfully
            {
                injectionFlag = detectInjection();

                executeQuery();
            }

            //local function definition:

            //function to detect injection among all input fields,
            //at least 1 field with injection detected will consider entire query
unusable
            int detectInjection()
            {
                for (int i = 0; i < output.Length; i++)
                {
                    if (output[i] == "0") //no injection detected
                    {
                        //do nothing
                    }
```

```
                else if (output[i] == "1") //injection detected (1)
                {
                    if (!Regex.Match(input[i + 1], "\'|--|;").Success)
//possibility of incorrect prediction
                    {
                        //temporary string array
                        string[] temp = { input[i + 1] + ",0" };
                        learnInput(temp);
                    }
                    else
                    {
                        System.Diagnostics.Debug.WriteLine("SQL
Injection detected");
                        System.Diagnostics.Debug.WriteLine(cmdStr);
                        Label1.Text = "SQL Injection detected";
                        injectionFlag = 1;
                        break;
                    }
                }

                else //unknown (-1)
                {
                    if (Regex.Match(input[i + 1], "\'|--|;").Success) //very
probable is an injection
                    {
                        //temporary string array
                        string[] temp = { input[i + 1] + ",1" };
                        learnInput(temp);

                        System.Diagnostics.Debug.WriteLine("SQL
Injection detected");
                        System.Diagnostics.Debug.WriteLine(cmdStr);
                        Label1.Text = "SQL Injection detected";
                        injectionFlag = 1;
                        break;
                    }

                    injectionFlag = -1;
                }
            }

            return injectionFlag;
```

```
        }

        //function to execute SQL command query, based on injection flag
from detection
        //also to determine if unseen input is benign or malicious, then learn
input
        void executeQuery()
        {
            con.Open();
            SqlCommand cmd = new SqlCommand(cmdStr, con);

            int numberOfRows = 0;

            if (injectionFlag != 1) //if no injection detected
            {
                try
                {
                    SqlDataReader reader = cmd.ExecuteReader();
                    while (reader.Read())
                    {
                        numberOfRows = reader.GetInt32(0);
                    }
                    con.Close();
                }
                catch (SqlException ex)
                { }


                if (numberOfRows == 0) //if no rows are found
                {
                    Label1.Text = "Wrong ID or Password";
                }

                else if (numberOfRows == 1) //exactly one row is found
                {
                    if (injectionFlag == -1)
                    {
                        learnUnseenInput(",0");
                    }

                    Session["Id"] = TextBox1.Text;
                    Response.Redirect("ProfilePage.aspx");
```

```
                    }

                    else //more than one row is found (highly being injected)
                    {
                        if (injectionFlag == -1)
                        {
                            learnUnseenInput(",1");
                        }

                        if (timeoutFlag == 1) //if detection system is down,
redirect page regardless
                        {
                            Session["Id"] = TextBox1.Text;
                            Response.Redirect("ProfilePage.aspx");
                        }
                    }
                }

                    //query is not executed when injectionFlag == 1

            }

            void learnUnseenInput(string flagValue)
            {
                for (int i = 0; i < output.Length; i++)
                {
                    if (output[i] == "-1")
                    {
                        string[] temp = { input[i + 1] + flagValue };
                        learnInput(temp);
                    }
                }
            }
        }

    }
}
```

SignUp.aspx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Data.SqlClient;
using System.Configuration;
using System.Data;
using System.IO;
using System.Text;
using System.Text.RegularExpressions;

namespace Website_Login
{
    public partial class SignUp : System.Web.UI.Page
    {
        //create connection
        SqlConnection con = new
SqlConnection(ConfigurationManager.ConnectionStrings["ConnectionString"].Conn
ectionString);
        protected void Page_Load(object sender, EventArgs e)
        {
            con.Open();
        }

        //sign up button
        protected void Button1_Click(object sender, EventArgs e)
        {
            //prevent NULL or blank input
            if (IDSignUp.Text == "" || NameSignUp.Text == "" ||
SecretPhraseSignUp.Text == "" || PasswordSignUp.Text == "")
            {
                Label1.Text = "Please Fill in All Fields";
                return;
            }

            //write input to temporary file
            string[] input = { "Sentence", IDSignUp.Text, NameSignUp.Text,
SecretPhraseSignUp.Text, PasswordSignUp.Text};
            writeInputToFile(input);
```

```
                //declare SQL command
                string cmdStr = "INSERT INTO Profile VALUES('" + IDSignUp.Text
+ "','" + NameSignUp.Text + "','" + SecretPhraseSignUp.Text + "','" +
PasswordSignUp.Text + "')";
                System.Diagnostics.Debug.WriteLine(cmdStr);

                //function linking to machine learning system to detect injection
before executing the query
                smartQueryExecution(input, cmdStr);

                con.Close();
                GridView1.DataBind();
                IDSignUp.Text = "";
                NameSignUp.Text = "";
                SecretPhraseSignUp.Text = "";
                PasswordSignUp.Text = "";
            }

            //Back button
            protected void Button2_Click(object sender, EventArgs e)
            {
                Response.Redirect("Login.aspx");
            }

            //write to file function
            protected void writeInputToFile(string[] lines)
            {
                File.WriteAllLines("D:/Machine Learning/SQLInjection/input.csv",
lines);
            }

            protected void learnInput(string[] lines)
            {
                File.AppendAllLines("D:/Machine
Learning/SQLInjection/unknowns.csv", lines);
            }

            protected void smartQueryExecution(string[] input, string cmdStr)
            {
                //read output file to determine result
                var outputFile = "D:/Machine Learning/SQLInjection/output.txt";
                var new_timeStamp = File.GetLastWriteTime(outputFile);
                var old_timeStamp = new_timeStamp;
```

```
                int timeoutFlag = 0, counter = 0; //timeout counter for "infinite" while
loop
                string[] output = File.ReadAllLines(outputFile);
                int injectionFlag = 0;

                while (true) //loop until output file is updated
                {
                    new_timeStamp = File.GetLastWriteTime(outputFile); //get
modified time

                    if (new_timeStamp > old_timeStamp) //if output file is modified,
meaning detection program has completed
                    {

System.Diagnostics.Debug.WriteLine("outputFile_old_timeStamp: " +
old_timeStamp.ToString());

System.Diagnostics.Debug.WriteLine("outputFile_new_timeStamp: " +
new_timeStamp.ToString());
                        old_timeStamp = new_timeStamp;
                        counter = 0;

                        while (true) // loop until output file is read
                        {
                            try
                            {
                                //read output file
                                output = File.ReadAllLines(outputFile);
                                break;
                            }

                            catch (IOException ex)
                            {
                                // wait for 50ms, before reading file again
                                System.Threading.Thread.Sleep(50);
                            }
                        }

                        //stop inifinite loop
                        break;
                    }
                    else
```

```
                {
                    counter++; //counter if file is not modified
                }

                if (counter == 100000) //if file is not modified after 1000 loops,
break loop
                {
                    System.Diagnostics.Debug.WriteLine("SQL Injection
Detection System Timeout");
                    counter = 0;
                    timeoutFlag = 1;
                    break;
                }
            }

            Label1.Text = "";
            if (timeoutFlag == 1) //timeout, detection program run unsuccessful
            {
                executeQuery(); //assuming there is no injection, run the query
regardless
            }

            else //no timeout, detection program running successfully
            {
                injectionFlag = detectInjection();

                executeQuery();
            }

            //local function definition:

            //function to detect injection among all input fields,
            //at least 1 field with injection detected will consider entire query
unusable
            int detectInjection()
            {
                for (int i = 0; i < output.Length; i++)
                {
                    if (output[i] == "0") //no injection detected
                    {
                        //do nothing
                    }
```

```
                else if (output[i] == "1") //injection detected (1)
                {
                    if (!Regex.Match(input[i + 1], "\'|--|;").Success)
//possibility of incorrect prediction
                    {
                        //temporary string array
                        string[] temp = { input[i + 1] + ",0" };
                        learnInput(temp);
                    }
                    else
                    {
                        System.Diagnostics.Debug.WriteLine("SQL
Injection detected");
                        Label1.Text = "SQL Injection detected";
                        injectionFlag = 1;
                        break;
                    }
                }

                else //unknown (-1)
                {
                    if (Regex.Match(input[i + 1], "\'|--|;").Success) //very
probable is an injection
                    {
                        //temporary string array
                        string[] temp = { input[i + 1] + ",1" };
                        learnInput(temp);

                        System.Diagnostics.Debug.WriteLine("SQL
Injection detected");
                        Label1.Text = "SQL Injection detected";
                        injectionFlag = 1;
                        break;
                    }

                    injectionFlag = -1;
                }
            }

            return injectionFlag;
        }

        //function to execute SQL command query, based on injection flag
```

```
from detection
                //also to determine if unseen input is benign or malicious, then learn
input
                void executeQuery()
                {
                    SqlCommand cmd = new SqlCommand(cmdStr, con);

                    if (injectionFlag != 1) //if no injection detected
                    {
                        try
                        {
                            cmd.ExecuteNonQuery();
                            Label1.ForeColor =
System.Drawing.ColorTranslator.FromHtml("#2eb82e");
                            Label1.Text = "Sign Up Successful, Login Now";
                        }

                        catch (SqlException ex)
                        {
                            Label1.ForeColor =
System.Drawing.ColorTranslator.FromHtml("#ff3300");
                            Label1.Text = "Duplicate ID or Invalid Input";
                        }

                    }

                    //query is not executed when injectionFlag == 1

                }

                /*void learnUnseenInput(string flagValue)
                {
                    for (int i = 0; i < output.Length; i++)
                    {
                        if (output[i] == "-1")
                        {
                            string[] temp = { input[i + 1] + flagValue };
                            learnInput(temp);
                        }
                    }
                }*/
            }
        }
```

```
}
```

Profile.aspx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Data.SqlClient;
using System.Configuration;
using System.Data;

namespace Website_Login
{
    public partial class ProfileList : System.Web.UI.Page
    {
        //create connection
        SqlConnection con = new
SqlConnection(ConfigurationManager.ConnectionStrings["ConnectionString"].Conn
ectionString);

        protected void Page_Load(object sender, EventArgs e)
        {
            if (!Page.IsPostBack)
            {
                if (Session["Id"] == null)
                {
                    System.Diagnostics.Debug.WriteLine("Browser Back
Button Clicked");
                    Response.Redirect("Login.aspx"); //Session is logged out,
unable to redirect back to profile page
                }

                else
                {
                    con.Open();

                    System.Diagnostics.Debug.WriteLine("Session ID:
"+Session["Id"].ToString());

                    string cmdStr = "SELECT * FROM Profile WHERE ID='"+
```

```
Session["Id"].ToString() +"'";
                        SqlCommand cmd = new SqlCommand(cmdStr, con);

                        System.Diagnostics.Debug.WriteLine(cmdStr);
                        cmd.ExecuteNonQuery();

                        SqlDataAdapter da = new SqlDataAdapter();
                        da.SelectCommand = cmd;
                        DataSet ds = new DataSet();
                        da.Fill(ds, "Id"); //show row from "Id" table
                        GridView1.DataSourceID = null;
                        GridView1.DataSource = ds;
                        GridView1.DataBind();

                        con.Close();
                    }

                }

            }

            //sign out button
            protected void Button1_Click(object sender, EventArgs e)
            {
                Session.Abandon();
                Session.Clear();
                Session["Id"] = null;
                System.Diagnostics.Debug.WriteLine("Logged out");
                Session.RemoveAll();

                Response.Redirect("Login.aspx");
            }

            //delete profile button
            protected void Button3_Click(object sender, EventArgs e)
            {
                con.Open();

                string cmdStr = "DELETE FROM Profile WHERE ID='" +
Session["Id"].ToString()+ "'";
                SqlCommand cmd = new SqlCommand(cmdStr, con);
                System.Diagnostics.Debug.WriteLine(cmdStr);
                cmd.ExecuteNonQuery();
```

```
                con.Close();

                Response.Redirect("Login.aspx");
            }
        }
    }
```