DETECTION OF STRUCTURED QUERY LANGUAGE INJECTION ATTACKS USING MACHINE LEARNING TECHNIQUES

Taapopi John Angula and Valerianus Hashiyana

Department of Computing, Mathematical and Statistical Sciences, University of Namibia, Windhoek, Namibia

ABSTRACT

This paper presents a comparative analysis of various machine learning classification models for structured query language injection prevention. The objective is to identify the best-performing model in terms of accuracy on a given dataset. The study utilizes popular classifiers such as Logistic Regression, Naive Bayes, Decision Tree, Random Forest, K-Nearest Neighbors, and Support Vector Machine. Based on the tests used to evaluate the performance of the classifiers, the Naïve Bayes gets the highest level of accurate detection. The results show a 97.06% detection rate for the Naïve Bayes, followed by LogisticRegression (0.9610), Support Vector Machine (0.9586), RandomForest (0.9530), DecisionTree (0.9069), and K-Nearest Neighbor (0.6937). The code snippet provided demonstrates the implementation and evaluation of these models.

KEYWORDS

Classification models, SQL-I, Python, Machine learning, Evaluations

1. INTRODUCTION

SQL (structured query language) injection vulnerabilities encompass the measures inculcated to protect individual websites, network systems, and database infrastructures that host different applications. As viewed by [1], they cause sections of institution’s web pages to hang significantly, further increasing the spread of viruses, network paralysis, data privacy breaches, and remote control of the servers. According to [2], the straightforward nature of SQL injection vulnerabilities makes them a prime choice for network attacks aimed at infiltrating targeted systems. One of the common ways hackers intrude into databases is through an SQL injection attack [2]. This fact explains why programmers rely on using different modes to create codes that may be used in addressing these vulnerabilities [3]. However, several challenges have been exhibited that span from the limitations in accessing input data’s authenticity during the design and the development of codes, which poses security concerns [4]. To this effect, SQL-I is a web application security vulnerability that permits an attacker to execute mischievous SQL statements on a web application’s backend database hosted on an SQL server.

This paper focuses on SQL injection attack detection and prevention that specifically target user input fields in web server applications, such as XAMP. The objective of these attacks is to exploit vulnerabilities in the system by injecting malicious SQL statements. SQL injection is a type of web attack where an attacker inserts inputs that are executed as sequential queries. The targeted system is typically unprepared to handle such input, leading to potential data disclosure and unauthorized access. The consequences of a successful attack can be severe, impacting the security aspects of confidentiality, integrity, and data availability.

During an SQL injection, the malicious actor inserts an SQL statement into an input field on a client web application i.e., login page username field to access a database server. SQL is used to represent queries to database management systems (DBMSs). Maliciously injected SQL statement is designed to extract or modify data from the database server [5].

A successful injection can lead to verification and circumvention, as well as alterations to the database through the insertion, alteration, or removal of data, resulting in data loss and potential destruction of the entire database. Additionally, this type of attack has the capability to overwhelm and execute commands on DBMS, often leading to more severe consequences [5]. Consequently, SQL injection attacks pose significant risks to organizations. Numerous studies have been conducted to address this threat, introducing diverse artificial intelligence (AI) approaches for detecting SQL injection attacks using machine learning and deep learning models [5]. This approach includes threat detection by learning from past data that represents both attacks and normal data.

On that bases historical data has become valuable for teaching machine learning (ML) models to recognize attack patterns, comprehend detected injections, and even predict future attacks before they occur [5]. In light of this the malicious actor and security defender of SQL injection attacks must possess an understanding of how the SQL language operates in order to exploit its vulnerabilities [6]. To extract or modify data from a database, queries must be written using the SQL language and adhere to a standardized syntax, such as: "SELECT * FROM users WHERE username = 'John_Doe'".

The above query is employed to fetch data from a table named "users." In this query, the asterisk (*) denotes all columns, while the condition "WHERE username = 'John_Doe'" specifies that only rows with the username 'John_Doe' should be selected. This query allows user to retrieve user records from the database that match the specified username 'John_Doe'.

Now, let's assume a malicious actor wants to inject malicious SQL to the given query; i.e.,

$$\text{" SELECT}*\text{ FROM users WHERE}\quad\text{username} = '\text{John\_Doe}'\qquad\text{"}$$

What would most likely happen is that the malicious actor would add ' OR '1'='1 to the end of the above query. This would result in all the details in the table called users to be returned as a result of the that addition. The typical representation would be type like this:

$$\text{"SELECT} * \text{FROM users WHERE username} = '\text{John\_Doe}'\text{OR}'1' = '1'\quad\text{"}$$

All the data from the users table will be returned because the added string '1'='1' always equals true and adding OR informs the database that it should return all the data from the users table WHERE the username = 'John_Doe' or 1 = 1. Meaning this query would always return all the other users' details because 1 is aways equal to 1. This type of attack is sometimes referred to as a tautology type attack [7]. In addition, tautology type attacks aren't the only type of attacks that represent a threat to databases and web applications. According to [5], alternative techniques, apart from tautology, can be employed, like when a malevolent intruder purposely inserts a wrong query to force the database server into producing a typical error page. This page could potentially contain valuable insights that could assist the intruder in comprehending the database, thereby facilitating a more sophisticated assault. The term "UNION" can similarly be leveraged to extract data, in addition to numerous other approaches rooted in the same concept of exploiting SQL syntax to extract or modify information within the designated database. The below example shows this other method:

```
SELECT * FROM users WHERE username="
UNION
SELECT * FROM users WHERE 1=1;
```

Given the dangers identified from SQL injections, the question that remains is how an organization or developer can detect and prevent such attacks from affecting them?

As indicated earlier one method could be using ML, where a classifier is trained using a large dataset of labeled data then allowed to identify if a username contains any SQL injection or not. The outcome of which is to identify and reject requests to the database if suspicious activity is identified. The labeled dataset is used on a ML that employs a supervised learning method. This translates to bad data being labeled as bad and good data as good, this helps the ML learn and recognize the difference between good and bad data such that when it come cross any of the two it would be able to correctly assume and act.

[5], further elaborate on supervised learning, which involves using a labeled training dataset to train a classifier. In this approach, the input data is already labeled as normal or abnormal, enabling the algorithm to establish a straightforward mapping between the input and the known output. The algorithm iteratively modifies its weights until achieving an acceptable classification accuracy. Subsequently, a separate test dataset is employed to evaluate the classifier's performance. If the results fall within an acceptable accuracy range, the classifier becomes capable of detecting novel data that was not part of the training or testing process. However, generating and labeling the training and testing data can be time-consuming, especially for complex attacks. Supervised learning encompasses two main categories: classification and regression algorithms. Popular examples of supervised learning algorithms include Bayesian networks, decision trees, support vector machines, K-nearest neighbors, and neural networks. The foremost input of this paper is to provide a systematic review of machine learning techniques and input validation that are used to detect and prevent injections. Through this comprehensive review, our objective is to provide the insights and enhance the comprehension of the convergence of an SQL injection attack and artificial intelligence, thereby keeping researchers well-informed of the current developments in this field.

2. OBJECTIVES

The main objective of this research is to detect structured query language injection attacks (SQL-I) using machine learning techniques and achieve the following sub-objectives:

a) Evaluate the effectiveness of different machine learning classification models for SQLdetection.
b) Identify the model(s) that demonstrate the highest accuracy and performance in detecting SQL-I attacks.
c) Determine the most suitable model(s) for building a robust SQL-I detection system.
d) Compare the models based on metrics such as accuracy, precision, recall, F1-score, and computational efficiency.
e) Provide insights and recommendations for selecting and implementing machine learning models for SQL-I prevention in real-world applications.

3. Background and Related Work

[8] provide a foundational review of the types of SQL injection attacks being broadly categorized into In-band and Out-of-band. The former occurs when the attacker can receive the results of the injected SQL query through the same channel as the web application. On the other hand, out-ofband attacks involve the attacker receiving the results through a different channel, such as email or a separate website. They further suggest some counter measures such as input validation, parameterized queries, and stored procedures. Input validation involves checking the user input for malicious SQL code before it is sent to the database. While, parameterized queries and stored procedures use pre-defined SQL commands that do not allow for arbitrary user input.

[9] add to the foundation laid by stating that on top of these counter measures, machine learning techniques can be applied to detect SQL injection attacks.These methods include decision trees, neural networks, and support vector machines. By training the machine learning models with a large dataset of legitimate and malicious queries, the models can detect the malicious queries with high accuracy.

In conclusion, the combination of both methods provides an efficient way to detect and prevent SQL injection attacks.

4. Methodology

The research adopts a comparative analysis methodology to assess and identify machine learning classification models that are most effective in preventing SQL injection attacks on a given dataset. The primary objective is to evaluate the performance of different models and determine the optimal model(s) for SQL injection prevention.

4.1. Research Design

The research employed a quantitative research design to conduct a comparative analysis of machine learning classification models for SQL injection prevention on a given dataset. The study encompassed a review of existing research materials, including journal articles, conference papers, articles, blogs, books, webpages, and academic tools.The research project was divided into several stages to ensure a systematic and comprehensive approach. The stages followed in the study are as follows:

4.1.1. Dataset Selection

4.1.1.1. Identified a suitable dataset specifically designed for evaluating SQL injection prevention models.

4.1.1.2. Ensured the dataset encompassed a diverse range of SQL injection attack instances and benign queries, representing real-world scenarios.

4.1.2. Preprocessing and Feature Extraction

4.1.2.1. Performed necessary preprocessing steps on the dataset, such as data cleaning, handling missing values, and ensuring data quality.

4.1.2.2. Extracted meaningful features from the dataset using appropriate techniques, such as tokenization, vectorization, or n-gram representation, to represent the SQL queries effectively.

4.1.3. Model Selection

4.1.3.1. Evaluated various machine learning classification models suitable for SQL injection prevention, such as decision trees, logistic regression, support vector machines, neural networks, and others.
4.1.3.2. Considered factors such as model performance, interpretability, computational requirements, and scalability to select the most promising models for further evaluation.

4.1.4. Model Training and Evaluation

4.1.4.1. Trained the selected models using the preprocessed dataset, allowing the models to learn the patterns and characteristics of SQL injection attacks and normal queries.
4.1.4.2. Evaluated the trained models' performance using appropriate metrics, including accuracy, precision, recall, and F1-score, to assess their effectiveness in SQL injection prevention.

4.1.5. Comparative Analysis

4.1.5.1. Conducted a comprehensive comparative analysis of the trained models, considering their strengths, weaknesses, and suitability for SQL injection prevention.
4.1.5.2. Compared the models based on metrics, interpretability, and scalability to identify the optimal model(s) for SQL injection prevention on the given dataset.

4.1.6. Results and Conclusions

4.1.6.1. Analyzed and interpreted the results obtained from the comparative analysis.
4.1.6.2. Drew conclusions on the performance and effectiveness of the different machine learning classification models in SQL injection prevention.
4.1.6.3. Provided recommendations and insights based on the findings to guide future research and implementation of SQL injection prevention systems.

By following these stages, the research project aimed to provide valuable insights into selecting the most suitable machine learning classification models for SQL injection prevention, contributing to the development of effective security measures for database systems and web applications.

4.2. Dataset Selection

To conduct the study, A dataset containing both malicious and benign payloads was needed. Specifically the SQL statements for SQL injection (SQL-i). Our datasets primarily focused on payloads transmitted via HTTP requests to web applications. It's important to note that our research did not cover other types of XSS attacks, such as "reflected." The collection obtained encompasses a wide range of payloads, including obfuscated and non-obfuscated scripts, as well as both long and short scripts, covering SQL-i.

Table 1 displays all the harmful and benign scripts collected from this data, encompassing SQL injections, and regular text. It's worth noting that the datasets contain a limited number of SQL injections, which is significant given the presence of malicious payloads within financial institutions' systems. Web pages have a short lifespan in terms of appearance, making datasets containing such attack payloads scarce.

Table 1. Benign and Malicious Scrips

Type	**Benign**	**Malicious**	**Total**
SQL-i	1,050	3,150	4,200

The acquired data was used to construct datasets with the purpose of providing a simulation of intrusions in Namibia’s banking system. The dataset was split into two parts: one portion was used for training the models, while the other half was used for evaluating their performance. Once the classifiers were developed, the testing datasets were used in the evaluation process.

4.3. Preprocessing and Feature Extraction

Jupiter Notebook environment for python 3 was used in the entirety of this study. This environment was best suited for this study because it comes preinstalled with the vast majority of data analysis tools and machine learning libraries.

The pandas libraries was used to conduct the data analysis and manipulation via the following command:

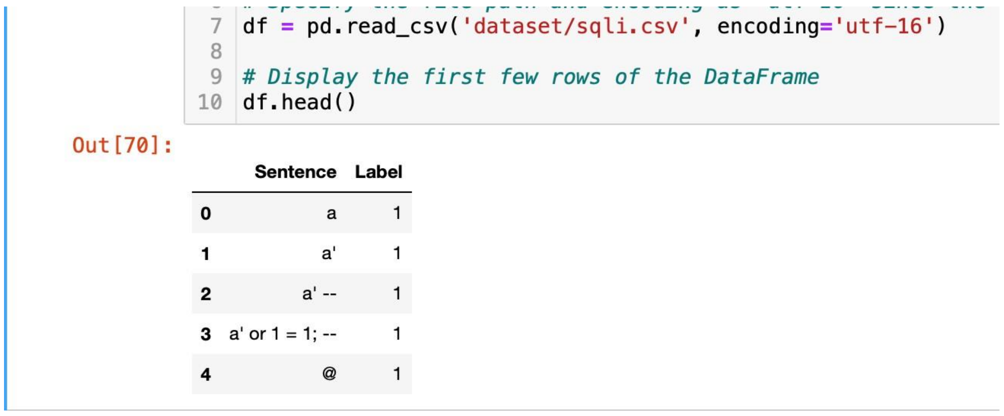
$$import\,pandas \quad as \quad pd \quad \# \quad Import\,the \quad pandas \quad library \quad for \quad data \quad manipulation\ and \quad analysis$$

In addition, the large dataset was imported into Jupyter Notebook as a Pandas dataframe using: $pd.read_csv$command. The actual implementation is:

$$df \quad = \quad pd.read\_csv('dataset/sqli.csv', encoding = 'utf-16')$$

The above simply reads the csv file into a pandasdataFrame, specify the file path and encode as 'utf-16' since the file was encoded in UTF-16 format.

Given the data frame being defined as df, a standard way to preview the it is by using the $hea()$ command. This command comes after the variable name of the dataframe, which in this case was df. Thus, to preview the dataframe command is: $df.hea()$ this resulted in the first first rows of the data frame being displayed:



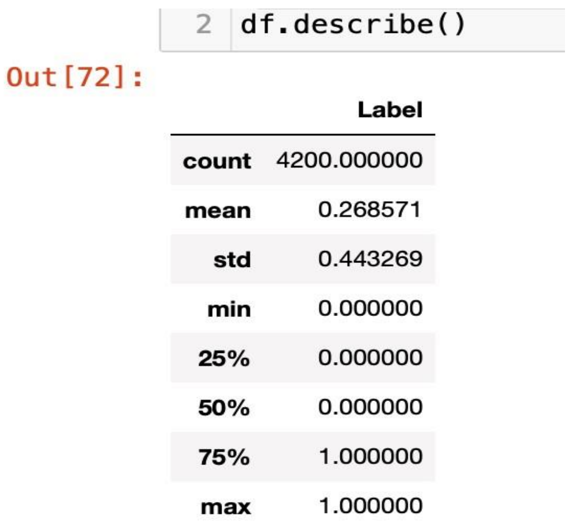
```
df = pd.read_csv('dataset/sqli.csv', encoding='utf-16')

# Display the first few rows of the DataFrame
df.head()
```

```
Out[70]:
         Sentence  Label
0               a      1
1              a'      1
2           a' --      1
3  a' or 1 = 1; --     1
4               @      1
```

Figure 1. Dataframe output

Also, the following command $df.\ describ()$ was used to generate a summary of the dataframe's mean, standard deviation and interquartile range:



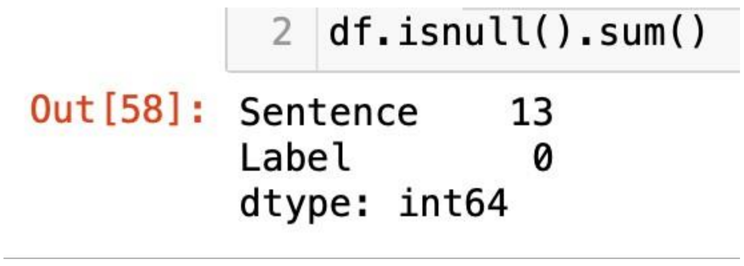
```
df.describe()
```

```
Out[72]:
```

	Label
count	4200.000000
mean	0.268571
std	0.443269
min	0.000000
25%	0.000000
50%	0.000000
75%	1.000000
max	1.000000

Figure 2. Dataframe summary

Furthermore, to fully understand the dataframe an inspection of the number of missing values per variable was needed and this was conducted using the $df.\ isnull().\ su()$ command. This command output the following:



```
df.isnull().sum()
```

```
Out[58]: Sentence    13
         Label        0
         dtype: int64
```

Figure 3. Inspecting missing values using isnull().sum()

The above indicated that there are 13 missing values in the column Sentence thus the need to use the $df.\ dropn()$ command to delete the rows with no values was valuable but consequently resulted in 13 few rows from our initial 4200 rows.

Equally important the need to perform split validation from the dataset was paramount. This step involved importing the necessary libraries through the following command:

$$from \quad sklearn.model\_selection \quad import\, train\_test\_split$$

This command simply imports train_test_split function for splitting data into training and testing sets.

This step was implemented using the following lines of code:

$X\ =\ d['Sentence']$, in essence is assigning the 'Sentence' column as the feature (X) variable and $y\ =\ df['Label']$, as the target (y) variable. Furthermore, given the two values, X = the input features and y = the output labels, creating the needed training and test data was optimal. Using the standard split validation parameters figure seen below, the split validation was created.

Table 2. Split validation parameters

Parameter	Argument	Explanation	Default
test_size	Float(between 0 and 1.0)	Proportion of test size, i.e. 0,3 = 30%	0.25 (if train_size is not specified)
	Integer	The number of test samples, i.e, 40 = for 40 test samples	
	None	Test value is automatically set to complement that of train_size.	
train_size (optional)	Float (between 0 and 1.0)	Proportion of test size, i.e. 0,7 # 70%	
	Integer	Number of test samples, i.e. 60 = for 60 test samples	
random_state (optional)	Integer	A seed number (integer) that can be reused to replicate the same random split. This ensures the model uses the same data split each time it's used.	
	None	A random seed number is used.	None
Shuffle (optional)	TRUE	Data is shuffled	TRUE
	FALSE	Data is not shuffled	

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 10 ,shuffle=True)

The above validation split consists of the following; X: The input features, y: The output labels, test_size: The proportion of the dataset to include in the testing set. Here, it is set to 0.3, which means 30% of the data will be used for testing, random_state: The random state used for shuffling the data before splitting. It ensures reproducibility of the split. In this case, it is set to 10 and lastly, shuffle: If True, the data will be shuffled before splitting. This helps in randomizing the data, which can be useful to avoid any bias in the splitting process.

With the validation completed, the data needed to be converted into numerical representation based on the frequency of words (or n-grams) in the text using the CountVectorizerclasss. This class was imported using the following command:

$from \quad \text{sklearn. feature\_extraction. text} \qquad \text{import CountVectorizer}$
Secondly an instance of CountVectorizer is created using:

$$vectorizer \quad = \quad CountVectorizer()$$

Then, the fit_transform method is applied to the training data X_train. This step fits the vectorizer to the training data and transforms the text into a numerical representation and thereafter, the result is stored in X_train_transformed.

$$X\_train\_transformed = \quad vectorizer.fit\_transform(X\_train)$$

Finally, the transform method is applied to the testing data X_test using the already fitted vectorizer. This step transforms the testing data into the same numerical representation as the training data. The result is stored in X_test_transformed.

$$X\_test\_transformed \;=\quad vectorizer.transform(X\_test)$$

After this transformation, one can use the transformed data X_train_transformed and X_test_transformed for training and evaluating machine learning models.

4.4. Model Selection

In order to select an appropriate machine learning model, a best suited model needed to be identified using training date and thus allow for a best accuracy to be utilized as performance measure and finally the actual model to be used.

The first step was to import classification models into the workbook. Classification models were used because the objective is to build a SQL-I detection system that would receive a query and determine if that query is normal SQL or SQL-I. The below models were imported:

$$from \quad sklearn.linear\_model import\, LogisticRegression$$
$$from \quad sklearn.naive\_bayes \;\; import\, MultinomialNB$$
$$from \quad sklearn.tree \;\; import\, DecisionTreeClassifier$$
$$from \quad sklearn.ensemble \quad import\, RandomForestClassifier$$
$$from \quad sklearn.neighbors \quad import\, KNeighborsClassifier$$
$$from \quad sklearn.svm \;\; import\, SVC$$

Secondly, an array initializing the models was implemented:

models = [LogisticRegression(), MultinomialNB(),DecisionTreeClassifier(),RandomForestClassifier(),KNeighborsClassifier(),SVC()]

Thirdly, a variable called best_model is initialized to None, and best_accuracy is set to 0.0 initially. A loop is used to calculate the accuracy for each model, compare it with the current best_accuracy value and if the new accuracy is higher, it updates best_accuracy and assign the corresponding model to best_model. Once the loop completes, the results are printed out showing the name of the best model along with its accuracy.

By using this method of model selection one can identify the model that achieved the highest accuracy on the dataset. The code below indicates how this was done:

```
best_model = None
best_accuracy = 0.0

for model in models:
```

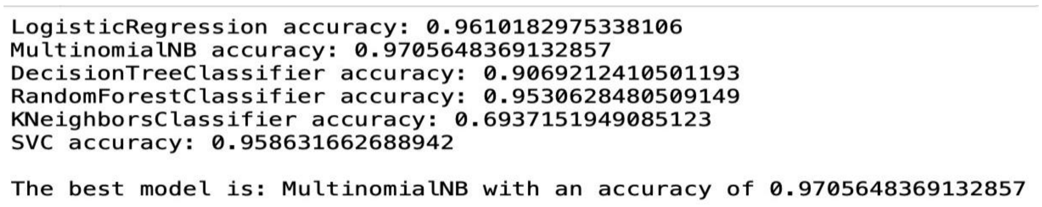
model.fit(X_train_transformed, y_train) y_pred = model.predict(X_test_transformed)
accuracy = accuracy_score(y_test, y_pred)
print(f"{model.__class__.__name__} accuracy: {accuracy}")

if accuracy >best_accuracy: best_accuracy = accuracy
best_model = model

print (f"\nThe best model is: {best_model.__class__.__name__} with an accuracy of {best_accuracy}")

4.5. Model Training and Evaluation

The results of the above model selection indicated that the best model for the dataset used was the MultinomialNB which is the Naive Bayes classifier for multinomial models. Below are the results of the model selection:



```
LogisticRegression accuracy: 0.9610182975338106
MultinomialNB accuracy: 0.9705648369132857
DecisionTreeClassifier accuracy: 0.9069212410501193
RandomForestClassifier accuracy: 0.9530628480509149
KNeighborsClassifier accuracy: 0.6937151949085123
SVC accuracy: 0.958631662688942

The best model is: MultinomialNB with an accuracy of 0.9705648369132857
```

Figure 4. Results of the model selection

The above figure indicates that the Naïve Bayes classifier was best suited for the dataset with an accuracy of 97 %.

Given this high accuracy rate, the MultinomialNB could then be set as the main model to train with the training data. This was done by initializing the model variable to MultinomialNB; $model = MultinomialN()$. Once the model was set, model needed to be fitted to the training data i.e., fitting the model to the input features X_train_transformed and the corresponding target labels y_train.;

$$model.fit(X\_train\_transformed, y\_train)$$

In addition, once the model was fitted, it allowed for an evaluation to be conducted by making predictions on the test data using the trained model. This is carried out by taking the input features X_test_transformed and returns the predicted labels for the corresponding test samples;

$$model\_predict = model.predict(X\_test\_transformed)$$

With the model having predicted a numerical value, additional steps needed to be taken to fully evaluate the model. These steps included using common evaluation methods called confusion matrix, classification report and lastly the accuracy score. According to [10] accuracy is a metric measuring how many cases the model classified correctly divided by the full number of cases;

$$Accuracy = \frac{Number \quad Predicted \quad Correctly}{Number \quad of \quad Cases}$$

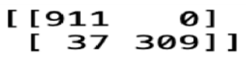
Furthermore, [10], states that if all predictions are correct, the accuracy score is 1.0, and 0 if all cases are predicted incorrectly.While accuracy alone is normally a reliable metric of performance, it may hide a lopsided number of false-positives or false-negatives. This isn't a problem if there's a balanced number of false-positives and false-negatives, but this isn't something we can ascertain using accuracy alone, which leads us to the following two evaluation methods [10]. The two-evaluation mentioned by Theobald, are confusion matrix and classification report. The two evaluation methods were imported via:

```
from  sklearn.metrics        import classification_report, confusion_matrix
```

With the above classes imported the trained model was evaluated by calling the following code:

```
print(confusion_matrix(y_test, model_predict))
```

Producing the following outcome:



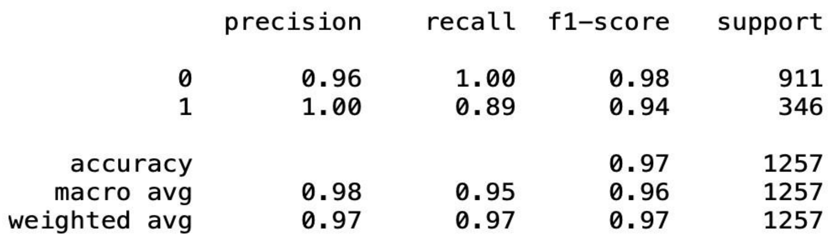
```
[[911   0]
 [ 37 309]]
```

Figure 5. Confusion matrix outcome

The confusion_matrix function from sklearn.metrics is used to compute the confusion matrix, which is a tabular representation of the predicted labels versus the true labels. It helps in evaluating the performance of a classification model by showing the counts of true positives, true negatives, false positives, and false negatives.

```
print(classification_report(y_test, model_predict))
```

Producing the following outcome:



```
              precision    recall  f1-score   support

           0       0.96      1.00      0.98       911
           1       1.00      0.89      0.94       346

    accuracy                           0.97      1257
   macro avg       0.98      0.95      0.96      1257
weighted avg       0.97      0.97      0.97      1257
```

Figure 6. Classification report outcome where 0 = Benign and 1 = Malicious

While the classification report function is used to generate a text report showing various classification metrics such as precision, recall, F1-score, and support. It provides a comprehensive summary of the model's performance for each class in the classification problem.

4.6. Comparative Analysis

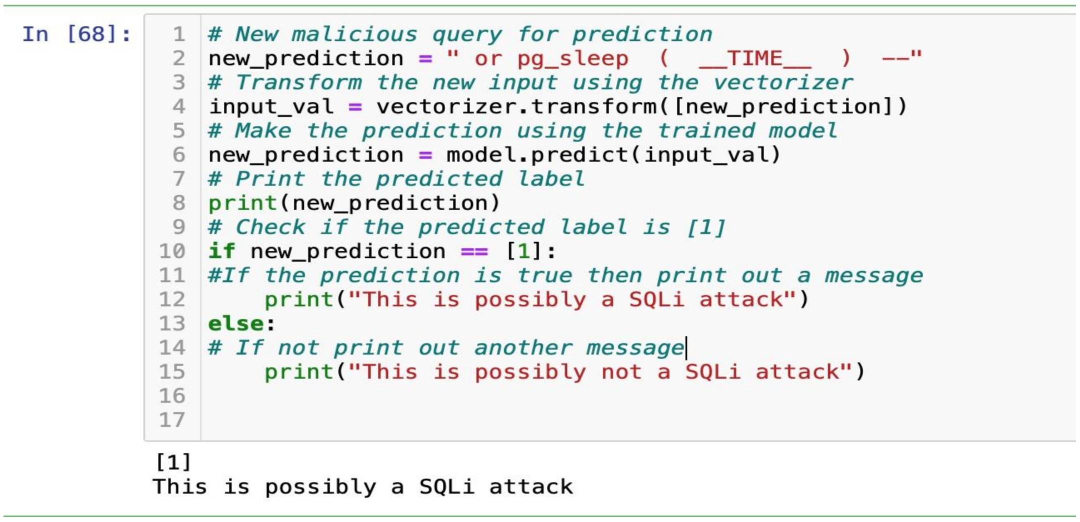
The outcome of the confusion matrix reveals that the model correctly classified 911 instances as negative (TN) and 309 instances as positive (TP). However, it misclassified 37 instances as negative (FN). In the classification report instance, few notable observations were made, these are precision, recall, f1-score and support. Precision measures the proportion of correctly predicted positive instances out of the total instances predicted as positive. For class 0, the precision is 0.96, indicating that 96% of the instances predicted as class 0 were correctly classified. For class 1, the precision is 1.00, indicating that all instances predicted as class 1 were correctly classified. Recall, also known as sensitivity or true positive rate, measures the proportion of correctly predicted positive instances out of the total actual positive instances. For class 0, the recall is 1.00, meaning that all instances of class 0 in the dataset were correctly identified. For class 1, the recall is 0.89, indicating that 89% of the instances of class 1 were correctly identified. The F1-score is the harmonic mean of precision and recall, providing a balanced measure of a model's performance. For class 0, the F1-score is 0.98, indicating a high balance between precision and recall. For class 1, the F1-score is 0.94, also indicating a good balance between precision and recall. Support refers to the number of instances in each class. In this case, class 0 has a support of 911 instances, while class 1 has a support of 346 instances. Accuracy measures the overall correctness of the model's predictions, considering both true positives and true negatives. In this case, the accuracy is 0.97, indicating that the model achieved an accuracy of 97% on the entire dataset. The macro average calculates the average of precision, recall, and F1score across all classes, without considering class imbalance. In this case, the macro average precision is 0.98, recall is 0.95, and F1-score is 0.96. Lastly, the weighted average calculates the average of precision, recall, and F1-score across all classes, considering class imbalance. The weight is proportional to the number of instances in each class. In this case, the weighted average precision, recall, and F1-score are all 0.97.

4.7. Results and Conclusions

Overall Performance:

The model achieves an accuracy of 0.97, indicating that it correctly classifies 97% of the instances in the dataset. This demonstrates the model's effectiveness in distinguishing between the two classes (0 = Benign and 1 = Malicious). The high precision and recall values for both classes indicate a strong ability to correctly identify instances of both positive and negative classes. The F1-scores for both classes also suggest a good balance between precision and recall.

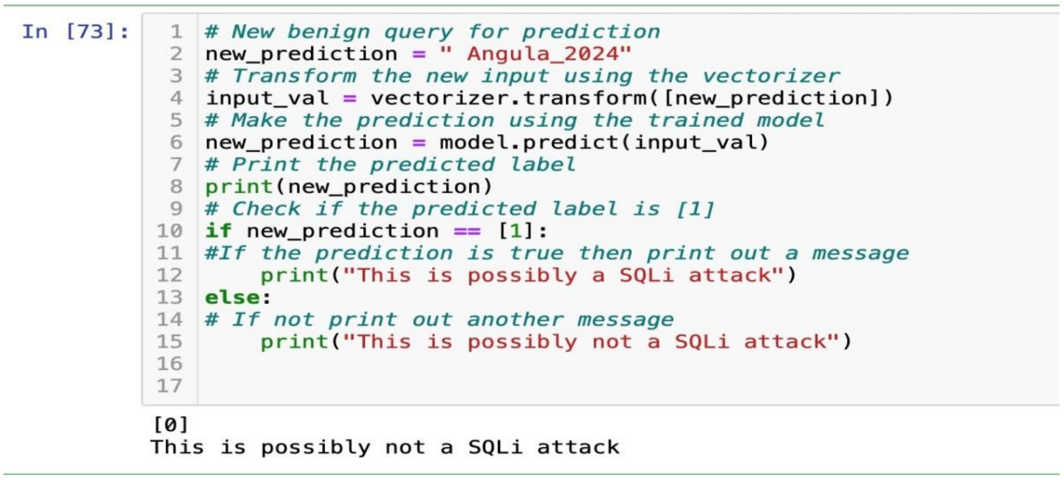
With the model achieving an accuracy f1-score of 97%, it was then tested on random benign and malicious queries to evaluate its prediction abilities given the high f1-score.



```
In [68]:
# New malicious query for prediction
new_prediction = " or pg_sleep  (  __TIME__  )  --"
# Transform the new input using the vectorizer
input_val = vectorizer.transform([new_prediction])
# Make the prediction using the trained model
new_prediction = model.predict(input_val)
# Print the predicted label
print(new_prediction)
# Check if the predicted label is [1]
if new_prediction == [1]:
#If the prediction is true then print out a message
    print("This is possibly a SQLi attack")
else:
# If not print out another message
    print("This is possibly not a SQLi attack")
```

```
[1]
This is possibly a SQLi attack
```

Figure 7. Results of model being used on malicious query



```
In [73]:
# New benign query for prediction
new_prediction = " Angula_2024"
# Transform the new input using the vectorizer
input_val = vectorizer.transform([new_prediction])
# Make the prediction using the trained model
new_prediction = model.predict(input_val)
# Print the predicted label
print(new_prediction)
# Check if the predicted label is [1]
if new_prediction == [1]:
#If the prediction is true then print out a message
    print("This is possibly a SQLi attack")
else:
# If not print out another message
    print("This is possibly not a SQLi attack")


[0]
This is possibly not a SQLi attack
```

Figure 8. Results of model being used on benign query

5. Conclusion

The results from the confusion matrix and classification report indicate that the classification model performs well in detecting and classifying instances of SQL injection attacks. The model shows high accuracy, precision, recall, and F1-scores for both the negative (class 0) and positive (class 1) classes. However, there were a few instances misclassified as negative, leading to false negatives. These results highlight the model's effectiveness in identifying potential SQL injection attacks and its ability to contribute to the prevention and security of web applications and databases.

6. Recommendations

Further analysis and fine-tuning of the model may be required to reduce the occurrence of false negatives and optimize its performance even further. Additionally, the model's performance should be validated on larger and more diverse datasets to ensure its robustness and generalizability in real-world scenarios.

References

[1] Lu, D., Fei, J., & Liu, L. (2023). A semantic learning-based SQL injection attack detection technology. *Electronics*, *12*(6), 1-22. https://doi.org/10.3390/electronics12061344

[2] Yunmar, R. A. (2018). Hybrid intrusion detection system using fuzzy logic inference engine for SQL injection attack. Kursor, 9(3), 83-93. https://doi.org/10.28961/kursor.v9i3.147

[3] Triloka, J., &Sutedi, H. (2022). Detection of SQL Injection Attack Using Machine Learning Based on Natural Language Processing. *International Journal of Artificial Intelligence Research, 6*(2).

[4] Demilie, W. B., &Deriba, F. G. (2022). Detection and prevention of SQLI attacks and developing compressive framework using machine learning and hybrid techniques. Journal of Big Data, 9(1), 1-30. https://doi.org/10.1186/s40537-022-00678-0

[5] Daniyal, A., Maha, A., &Suaad, A. (2022, 09). Detection of SQL Injection Attack Using Machine Learning Techniques: A Systematic Literature Review. *Journal of Cybersecurity and Privacy, 2*, 764-777.

[6] Vähäkainu, P., &Lehto, M. (2019). Artificial intelligence in the cyber security environment. *In Proceedings of the 14th International Conference on CyberWarfare and Security* (pp. 431-440). Stellenbosch: ICCWS 2019.

[7] Satapathy, S., Govardhan, A., Raju, K., & Mandal, J. (2015). SQL Injection Detection and Correction Using Machine Learning Techniques. *Advances in Intelligent Systems and Computing*, 435–442.

[8] Halfond, W. G. J., &Orso, A. (2005). AMNESIA: Analysis and monitoring for NEutralizing SQL-injection attacks. In ASE '05: Proceedings of the 20th IEEE/ACM international conference on automated software engineering (pp. 174-183). https://doi.org/10.1145/1101908.1101935

[9] Zhang, W., Yueqin, L., Xiaofeng, L., Shao, M., Mi, Y., Zhang, H., &Zhi, G. (2022). Deep neural network-based SQL injection detection method. Security and Communication Networks, 2022, 1-9. https://doi.org/10.1155/2022/4836289

[10] Theobald, O. (2019). Machine Learning with python. In O. Theobald. Scatterplot Press.

AUTHORS

Valerianus Hashiyana is a Senior Lecturer at Department of Computing, Mathematical & Statistical Sciences under Faculty of Agriculture, Engineering and Natural Sciences, University of Namibia. His areas of Research are Cybersecurity, Networking & Security, AI, IOT, E-Health, Next generation Computing and Educational Technologies.

Tel: +264 812830277, Email: vhashiyana@unam.na / vhashiyana@gmail.com.

Taapopi John Angula is currently a final year masters of information technology student his research work was based on the development of a structured query language injection prevention system for the banking sector in Namibia.