

Spring 5-23-2019

SQL Injection Detection Using Machine Learning

Sonali Mishra
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the [Artificial Intelligence and Robotics Commons](#), and the [Information Security Commons](#)

Recommended Citation

Mishra, Sonali, "SQL Injection Detection Using Machine Learning" (2019). *Master's Projects*. 727.
DOI: <https://doi.org/10.31979/etd.jsdj-ngvb>
https://scholarworks.sjsu.edu/etd_projects/727

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

SQL Injection Detection Using Machine Learning

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Sonali Mishra

May 2019

© 2019

Sonali Mishra

ALL RIGHTS RESERVED

The Designated Thesis Committee Approves the Thesis Titled

SQL Injection Detection Using Machine Learning

by

Sonali Mishra

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2019

Dr. Robert Chun Department of Computer Science

Dr. Nada Attar Department of Computer Science

Terence Runge Illumio

ABSTRACT

Sharing information over the Internet over multiple platforms and web-applications has become a quite common phenomenon in the recent times. The web-based applications that accept critical information from users store this information in databases. These applications and the databases connected to them are susceptible to all kinds of information security threats due to being accessible through the Internet. The threats include attacks such as Cross Side Scripting (CSS), Denial of Service Attack (DoS), and Structured Query Language (SQL) Injection attacks. SQL Injection attacks fall under the top ten vulnerabilities when we talk about web-based applications. Through this kind of attack, the attacker can steal critical and confidential information and hence it could have damaging effects on a business or organization. The effects could range from monetary loss, leaking confidential business information, decrease in company's stock market value or any combination of these. In this paper we have used an algorithm called Gradient Boosting Classifier from ensemble machine learning approaches to classify and detect SQL Injection attacks.

ACKNOWLEDGEMENT

I feel fortunate to get continuous guidance, support and encouragement from my advisor Dr. Robert Chun. His experience in Artificial Intelligence and Machine Learning helped me to develop and implement my project and overcome the technical challenges that I faced during implementation. I would like to thank him for his constant guidance throughout the two semesters that helped me in the research and implementation related to my project. I would also like to thank my committee members Dr. Nada Attar and Terence Runge for taking time to review my project.

Table of Contents

I.	INTRODUCTION.....	3
II.	UNDERSTANDING SQL INJECTION.....	5
i.	Union Based SQL Injection	7
ii.	Error Based SQL Injection.....	8
iii.	Blind SQL Injection	8
A.	Boolean Based SQL Injection.....	9
B.	Time Based SQL Injection Attacks	9
III.	Related Work	10
IV.	Supervised Learning.....	14
i.	Naïve Bayes.....	15
ii.	Ensemble Learning.....	17
iii.	Bagging and Boosting.....	19
V.	Methodology	21
VI.	Dataset.....	22
i.	Plain-Text Dataset.....	22
A.	Diversity	23
B.	Size.....	23
C.	Source	23

ii.	SQL Injection Dataset.....	23
A.	Categories	24
B.	Size.....	24
VII.	Tokenization:	25
i.	Regular Expressions	26
VIII.	Feature Extraction	28
i.	Step 1 : Calculate G-test Score	28
ii.	Step 2 : Calculate Entropy	29
iii.	Step 3 : Calculate G-test score mean.....	30
IX.	Experiments	31
i.	Experiment 1 : Naïve Bayes.....	31
ii.	Experiment 2 : Gradient Boosting Classifier	32
A.	Parameter Tuning	33
X.	Results and Analysis	35
XI.	Conclusion and Future Work.....	39
	REFERENCES	41

LIST OF FIGURES

- Figure 1. Example Login Page in Browser
- Figure 2. Types of SQL Injection
- Figure 3. Supervised Machine Learning
- Figure 4. Equation of Bayes Theorem
- Figure 5. Bias Variance Trade-Off
- Figure 6. Ensemble Learning – Bagging vs Boosting
- Figure 7. Multiple Decision Tree Classifiers used for Gradient Boosting
- Figure 8. Sample Plain-text Dataset
- Figure 9. Sample SQL Injection Dataset
- Figure 10. Tokenization in NLP
- Figure 11. `re.compile()` for Tokenization
- Figure 12. Entropy Formula
- Figure 13. Dataset with Extracted Features
- Figure 14. Naïve Bayes Classifier Results
- Figure 15. Algorithms Mean Ranking
- Figure 16. Gradient Boosting Classifier Results
- Figure 17. Accuracy Naïve Bayes vs Gradient Boosting

Figure 18. Identifying Union Based SQL Injection

Figure 19. Identifying Error Based SQL Injection

Figure 20. Identifying Boolean Based SQL Injection

Figure 21. Identifying Time Based SQL Injection

Figure 22. Identifying Plain-Text

Figure 23. Identifying Plain-Text

I. INTRODUCTION

Most of the applications that we use every day are web-based applications. Organizations choose to make the applications accessible over the Internet to increase the exposure they gain. Being exposed to Internet increases the security challenges that come along with uncontrolled access. With the growth of Internet, we are used to performing various kinds of transactions online. All the data entered by the users during these transactions on web applications or websites is stored in some kind of a database. Relational Databases can be communicated with a language called Structured Query Language, i.e. SQL. Using SQL to launch attacks on databases and manipulate them to do what the user wants is a form of a web hacking technique called SQL Injection Attack. SQL Injection Attacks have become an increasing cause of worry for the cyber defenders. In the previous year alone, SQL Injection and Remote Code execution attacks contributed to more than four-fifths of the detected web-based attacks. SQL Injection attacks remain one of the most pervasive cyber-attacks. Many techniques have been developed to deal with such attacks, however cyber hackers still seem to successfully get through the various defense mechanisms in place to deal with SQL Injection attacks.

Lately, the use of machine learning algorithms to detect and prevent various cyber security threats is being debated largely. While the power of using supervised and unsupervised learning techniques to detect security threats cannot be questioned, the computing resources and time required to execute such complex algorithms remains a major concern for the ever advancing cyber security community. Tremendous research work has been done on using various machine learning algorithms to detect SQL Injection attacks. There

is no single perfect algorithm or technique in machine learning that can be applied to a particular problem. A problem needs to be tested against various algorithms falling under classification or regression techniques, and the results need to be compared, before finalizing a particular approach, for maximum accuracy. SQL Injection detection using Naïve Bayes algorithm has been implemented in previous researches. In this paper we use an approach called Gradient Boosting algorithm to detect and prevent SQL Injection attacks. We also implemented the Naïve Bayes algorithm and compared the results against Gradient Boosting for this particular problem, details of which are discussed in later parts of this paper.

In this paper we begin with an introduction to SQL Injection attacks and the need and motivation to build a better SQL Injection detection system. We then understand the SQL Injection attacks and various types in detail in section II. Section III describes the related work done in this area so far. All the significant implementations and research work done so far provides enough literature review to learn from and improve on the problem. Section IV gives an introduction to supervised learning, which is the generic approach we are using to solve this problem. This section also explains and delves a bit into the two algorithms considered for this experiment, i.e.

1. Naïve Bayes
2. Gradient Boosting

Section V goes into the details of the dataset used, the implementation of the two algorithms, Naïve Bayes and Gradient Boosting, and comparing both the results. Finally, we conclude the paper with conclusion and future works in section VI.

II. UNDERSTANDING SQL INJECTION

SQL Injection is an attack that tries to get unauthorized access to a database by injecting a code and exploiting the SQL query [1]. Let us understand this through a simple example. Say there is a banking website that lets users login by entering their username and password. When the user enters a valid username and password, the authentication will pass, and the user will be allowed to login.

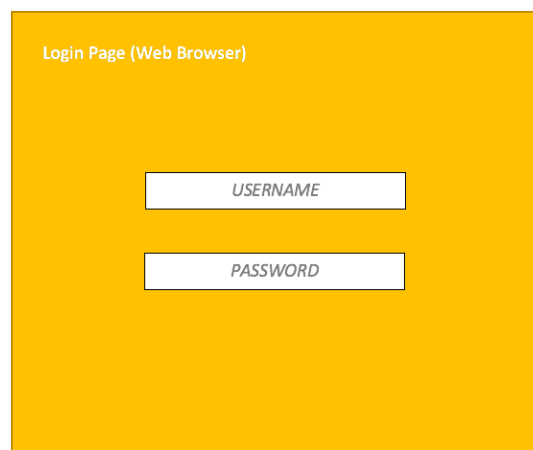


Fig. 1. Example login page in browser

Following will be the query constructed in case of an authorized login attempt where:

Username = usr

Password = usr123

SQL Query: *SELECT * FROM users WHERE name = 'usr' and password = 'usr123'*

However, it is also possible that a user with malicious intent enters the following input in the username and password fields of the website where:

Username = usr

Password = ' or '1' = '1

The SQL Query constructed in this case will be.

SQL Query: *SELECT * FROM users WHERE name = 'usr' and password = " or '1' = '1'*

Since 1=1 will always be true, this user will always be allowed to login to the website. The user gets unauthorized access to someone else's account details and the possession of this information could result in serious consequences for the person whose account information was stolen. This is a case of theft and a violation of data privacy.

This was a very simple example of a SQL Injection attack just for understanding, and most of the websites and web applications today would easily prevent this kind of attack. But there are various and more complex forms of SQL Injection attacks, some of which are described later in detail. The aim of the attackers using SQL Injection is to exploit the database that is connected to a website or a web application. It is extremely important to protect such databases against SQL Injection attacks in order to protect the important data stored in them. Letting an unauthorized user get access to a database can result in many unauthorized actions on the database like deleting tables, retrieving important information and many more terrifying things, and SQL Injection attacks make all of this possible.

SQL Injection Attacks can be broadly classified into the following three categories:

1. Union Based SQL Injection
2. Error Based SQL Injection

3. Blind SQL Injection

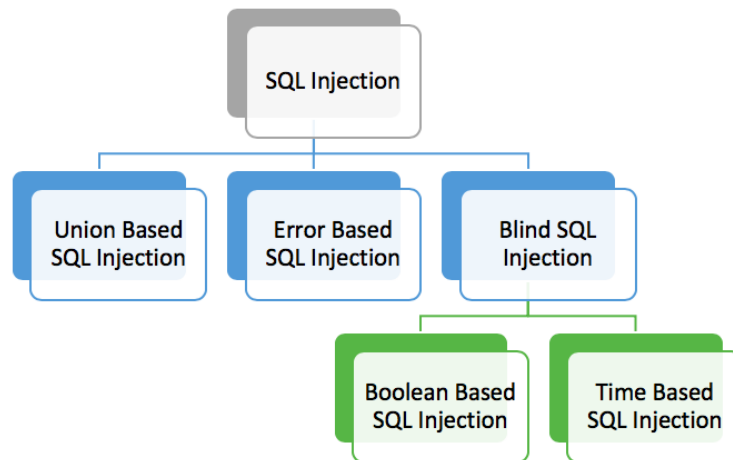


Fig. 2. Types of SQL Injection

i. Union Based SQL Injection

In SQL, UNION operator is used to join two SQL statements or queries. Union Based SQL Injection takes advantage of this feature to make the database return desired results in addition to the intended results. This is achieved by injecting another query in place of plain text and using UNION keyword at the beginning of the query.

A simple example would be searching for a song in a database. When we enter the name of the song in the search field, following query is formed.

Value Entered: Magic

SQL Query: *SELECT * FROM songs WHERE name = 'magic'*

However, a malicious user might enter the following in the song search field to exploit the database.

Value Entered: Magic' UNION DROP TABLE songs

SQL QUERY: *SELECT * FROM songs WHERE name = 'magic' UNION DROP TABLE songs*

This might end up in deleting the entire songs table. Here the user is just trying to run two queries at one time and has used UNION keyword to combine both the queries. Using this approach, the second part of the query can be used to perform any desired unauthorized action on the database.

ii. Error Based SQL Injection

Error based SQL Injection approach works by passing an invalid input in the query and thereby triggering an error in the database. This is achieved by forcing the database to perform an action that will lead to an error. The user can then look for the errors generated by the database and use those errors to gain information on how to further manipulate the database by exploiting the SQL query.

iii. Blind SQL Injection

Blind SQL Injection attack is a technique where the malicious user asks questions to the database and decides on further course of action based on the returned answers. This is the most difficult type of SQL Injection attack since no information is known about the database. This type of approach is used when the database returns generic errors like

'Syntax Error'. Blind SQL Injection attacks are further classified into Boolean Based SQL Injection attacks and Time-Based SQL Injection attacks.

A. Boolean Based SQL Injection

Boolean-based SQL Injection is an inferential SQL Injection technique that relies on sending an SQL query to the database which forces the application to return a different result depending on whether the query returns a TRUE or FALSE result. Depending on the result, the content within the HTTP response will change, or remain the same. This allows an attacker to infer if the payload used returned true or false, even though no data from the database is returned. This attack is typically slow (especially on large databases) since an attacker would need to enumerate a database, character by character.

B. Time Based SQL Injection Attacks

Time-based SQL Injection is an inferential SQL Injection technique that relies on sending a SQL query to the database which forces the database to wait for a specified amount of time (in seconds) before responding. The response time will indicate to the attacker whether the result of the query is TRUE or FALSE.

Depending on the result, an HTTP response will be returned with a delay, or returned immediately. On the basis of the returned response, with or without the time delay, an attacker can infer information about the database and application.

III. Related Work

This section briefly describes the research and work done so far on detecting SQL Injection attacks and preventing them effectively. The research work done so far on detecting SQL Injections can be broadly classified into two types of approaches. The first approach is to securely write the source code itself to enforce enough input validation for SQL queries. The second approach is to deploy additional software to verify the SQL queries being passed through web applications to be executed across the database. Lately some researchers have also highlighted the importance of using these two approaches in combination to achieve a more reliable SQL Injection detection model. In this section, both the approaches and the research work done in those fields is discussed.

Goud et al. developed a JDBC checker that is a static analysis tool to check for errors in SQL strings and verify them for potential malicious queries [3]. It verifies the SQL strings for correctness and promises to identify and indicate potential errors in SQL queries. The way it works is instead of dynamically checking each query while it is generated at runtime, it statically creates a list of all potential SQL strings that could be executed across a particular application and then analyses all those potential SQL strings for malicious content and semantic errors. The problem with this approach could be multiple, including the high storage that will be required for storing all the potential SQL queries, and how could a tool generate all possible potential queries for an application. There is a huge possibility that it would miss out on predicting SQL query statements that were actually executed. To overcome this limitation of static analysis, a tool named CANDID was developed, that stands for 'Candidate Evaluation for Discovering Intent Dynamically', [4]. It works on the

concept of programmer intent. The concept of programmer intent describes how a SQL query structure should look like if it is formed exactly as was intended by the programmer of the application. There is a pattern observed in SQL injection attacks that the malicious SQL query executed across database always has a different structure than the one that was intended by the programmer. Authors of this paper believed that identifying this difference in structure could be a significant step towards successfully identifying and preventing SQL injection attacks. The way this tool achieves its goal is that it dynamically creates the programmer intended SQL query structure when the program reaches a location where it will generate and execute a SQL query. This generated programmer intended SQL query is then compared with the actual SQL query that was passed by the user, to identify and prevent the malicious queries from being executed. Later a few researchers came up with the idea of combining the use of static analysis and dynamic monitoring to efficiently prevent SQL Injection attacks. AMNESIA is a tool that was developed on this idea of using both static and dynamic approaches [5]. In this approach, the application code itself contains information to produce all the possible SQL queries that could be generated by the application. All these possible legitimate queries are generated and stored for comparison. At the same time dynamic monitoring is done of SQL queries generated at runtime, and each dynamically generated SQL query is compared to the list of possible SQL queries. If no match is found for a SQL query generated as a result of some input from a user, in the list of possible legitimate SQL queries, the query is classified as malicious and is not allowed to be executed on the database. This method has its own set of drawbacks, biggest one being that it is not a hundred percent accurate and could generate a lot of false positives. Buehrer et al. used similar approach of comparing

the actually generated queries with the one that should have been generated (programmer intended) [6]. The only difference in this approach is that it achieves the results by using Parse Trees. Parse trees are generated by parsing the statements by using the syntax of the language that statement is written in. This is a dynamic approach only, that is, it verifies the SQL queries generated at runtime and has no previously stored set of possible legitimate queries. In this approach two parse trees are generated, one is generated by using the input from the user and generating a tree that would be generated if this was a legitimate query, and the other tree is generated by processing the actual input and seeing how the query actually will be executed. Now there is one thing about malicious SQL queries, that is also discussed previously, that the behaviour of such queries is always different than the programmer intended behaviour. Hence in case of a malicious query, this paper promises that the two parse trees generated will be different, and hence the SQL Injection attacks can be identified more efficiently.

With the development of AI and Machine Learning, some researchers proposed using the machine learning algorithms to prevent SQL Injection attacks [7]. This paper detects SQL Injection attacks using a machine learning algorithm called Naïve Bayes. Naïve Bayes is a classification machine learning algorithm that assumes that a particular incident is unrelated to and is independent of other all other incidents. In this paper Naïve Bayes classifier is used to classify between malicious and non-malicious SQL queries. To train the model they have used a training dataset that consists of both malicious and non-malicious SQL queries and also every query in this training data is labelled. Labelling the data helps the model to learn what is malicious and what is non-malicious. This type of model is called a supervised machine learning model. Once the model has been trained it is then used on

the test dataset to verify if the model is classifying the SQL queries correctly. The model suggested in this paper promises to even detect those SQL Injection attacks that are new and whose signatures are not known. We will also use machine learning to detect SQL Injection attacks but with a different machine learning algorithm.

IV. Supervised Learning

Machine learning algorithms can be broadly classified as Supervised Learning algorithms and Unsupervised Learning algorithms. Supervised learning is a type of machine learning that in its simplest form, works in the following manner. We have a dataset called as training dataset and each individual component of this dataset is labelled. The supervised learning model basically learns the relationship between the data and the label and then uses this learnt information to classify new data that it has never seen before. This new data is called as the test dataset. We use test dataset to determine the accuracy of a supervised learning algorithm. This is how we predict the values or classify never before seen data using supervised machine learning. Supervised learning algorithms can further be broadly classified as Regression algorithms and Classification algorithms.

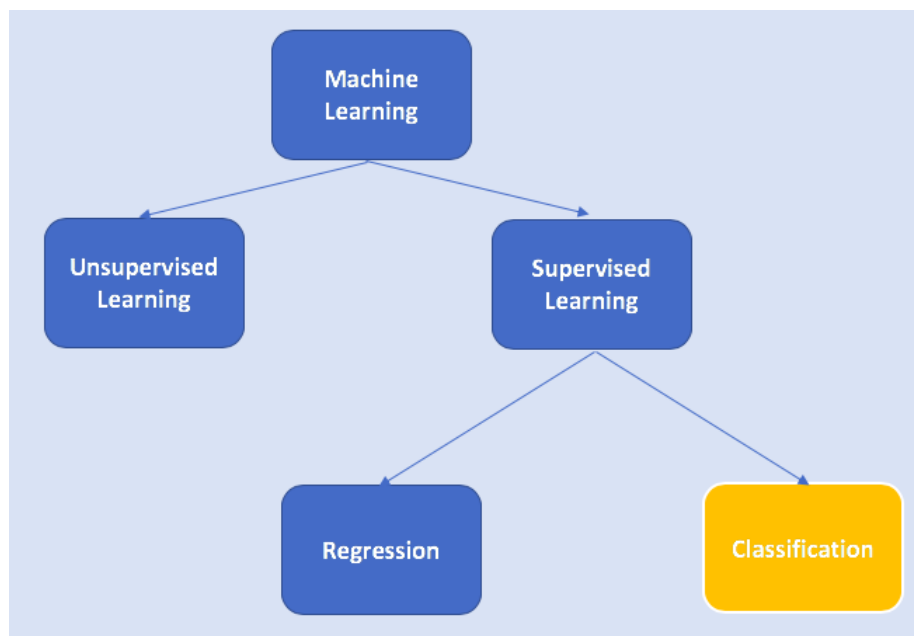


Fig. 3 Supervised Machine Learning

Regression algorithms are used for predicting a value for an individual data component, for example, predicting the value of a house, or predicting stocks. Usually the values predicted by the Regression algorithms are quantitative or numerical. Classification algorithms are used for classifying individual data components. For example, classifying if the vehicle is a truck or a car, or predicting if it will rain or not on a given day. Classification algorithms are used to predict qualitative values. Figure 3 shows the derived hierarchy of Classification and Regression algorithms. In this section we will focus on Classification algorithms in Machine Learning, and more specifically the two classification models that are Naïve Bayes and Gradient Boosting.

i. Naïve Bayes

Naïve Bayes algorithm has already been implemented for detecting SQL Injections [7]. Naïve Bayes is a classification model in supervised learning that is based on Bayes Theorem. The essence to Naïve Bayes is that it assumes that the presence of a feature in a data model is unrelated to the presence of other features. In short it assumes that all the features in a data are conditionally independent of each other, hence it gets its name 'Naïve Bayes'. Figure 4 shows the equation for Bayes Theorem.

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

Fig. 4 Equation of Bayes Theorem

Where:

$P(A|B)$ = Probability of A being true given that B is true.

$P(B|A)$ = Probability of B being true given that A is true.

$P(A)$ = Probability of A regardless of the other data.

$P(B)$ = Probability of B regardless of other data.

There are two types of probabilities in this equation. Prior probability, that is $P(A)$ and $P(B)$ and posterior probability, that is $P(A|B)$ and $P(B|A)$.

$P(A|B)$ and $P(B|A)$ are also called conditional probabilities since they are condition to something.

The benefits of using Naïve Bayes model could be many including the following:

1. It can be trained on a small dataset.
2. It is easier to compute and requires less computational resources.

However, the Naïve Bayes classifier, being extremely simple to implement, could also result in missing to detect a few SQL injection attacks, especially when a particular type of SQL Injection is being used for the first time. To deal with the issues in Naïve Bayes

algorithm, there are Ensemble methods in machine learning that can be used to improve accuracy of models.

ii. Ensemble Learning

Ensemble models are multiple supervised learning models used together to predict a value. In an ensemble model, individual supervised learning models are trained independently, and then the results obtained from each model are either averaged or voted to provide a single result. This obtained result obviously provides much better predictive accuracy than the individual models. Machine Learning models are prone to various errors including Bias Error and Variance Error. Bias and Variance Errors can be defined as follows.

Bias Error: This error signifies how much the predicted value is different from the actual value.

Variance Error: This error signifies how much a model's behavior will change if we change the training dataset. High Variance Error indicates overfitting issue. Understandably the outcome of a function will depend on the training data, since it was used to train the function in the first place. Hence some amount of variance is expected in every supervised learning model. But it should not be the case that the model significantly changes if the training data is changed. Ensemble learning could also help with reducing these errors.

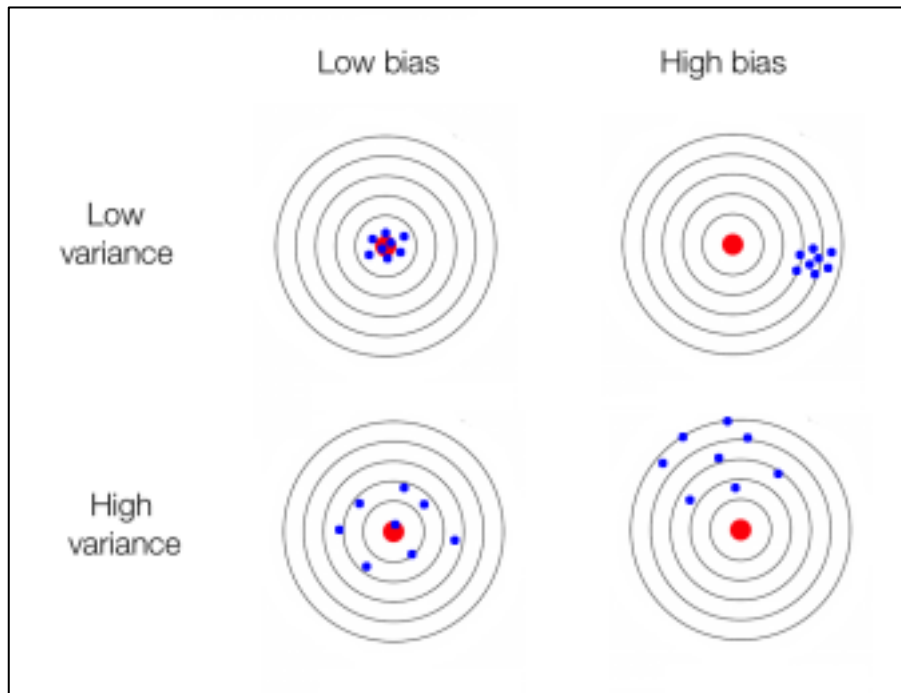


Fig. 5 Bias Variance Trade-off [11]

Figure 5 shows the Bias Variance errors. The red dot in the middle denotes the actual values and the blue dots denote the predicted values by the model. It can be inferred from the figure 5 that in case of higher variance, the predicted values are farther apart from each other and in case of high bias the predicted values are farther apart from the actual values. Ensemble learning can be achieved by the following two ways that are discussed briefly in the next section.

1. Bagging
2. Boosting

iii. Bagging and Boosting

Bagging is an ensemble learning approach that predicts a value of data by using multiple supervised learning models and then combining the results of all these individual learning models by a chosen technique. The technique used for combining these results could be any including by weighting the results, taking their average, voting for the maximum result, etc. Bagging is also called as Bootstrap Aggregation. Bagging can help with reducing variance errors by using multiple supervised learning models and then combining their result. An example of bagging technique is Random Forest Algorithm. In this approach, multiple decision trees are created on random subsets of training data and results are collected from each decision tree. A final result is then selected from these results by taking an average of all the results from individual supervised learning models. Figure 6 shows the architectural depiction of bagging and boosting techniques. As can be observed from the figure, all the individual models are used in parallel in Bagging approach.

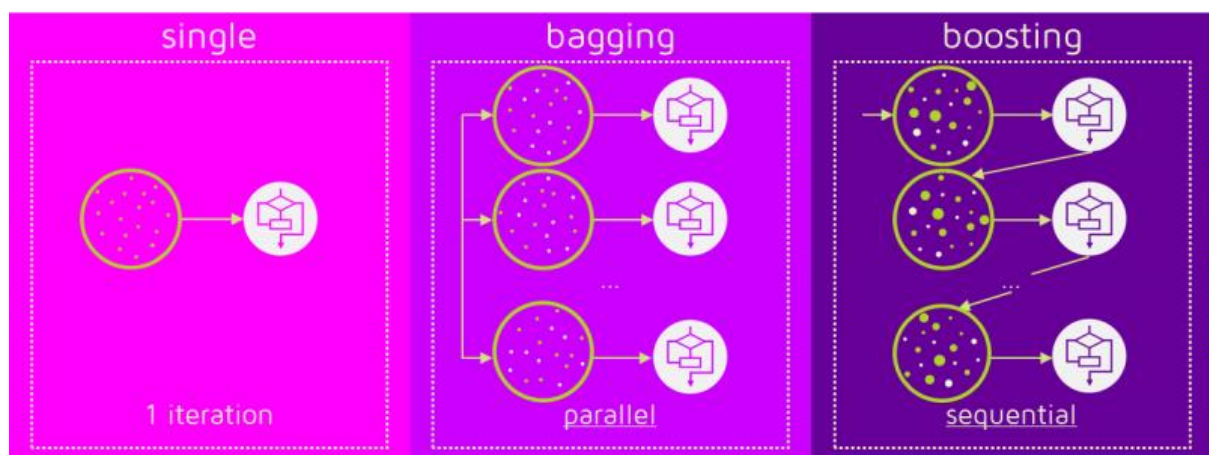


Fig. 6 Ensemble Learning – Bagging vs Boosting [12]

Boosting on the other hand is an Ensemble learning approach that also uses multiple supervised learning models in combination to provide better predictive results. The difference is the way in which boosting uses these multiple models. Instead of using them in parallel, in boosting the multiple models are used sequentially. In this technique each predictor model learns and tries to minimize the errors from the previous predictor model. Boosting algorithms can reduce the bias errors introduced due to small size of datasets. An example of boosting algorithm is Gradient Boosting.

In this paper, Gradient Boosting approach is used to classify and detect SQL Injection attacks. One of the important reasons of choosing this approach is because not enough data is available to train the machine learning models. Naïve Bayes technique has been implemented to detect SQL Injection attacks because it can be trained even on small datasets. However, that can lead to high bias errors as it is possible that data will not be classified correctly all the time. The hope is that using Gradient Boosting approach results in better accuracy while classifying the SQL Injection queries and overall provides better results and higher ratio of detecting an SQL Injection attack.

V. Methodology

Gradient boosting is an Ensemble learning method to reduce errors and provide predictions with better accuracy. Gradient Boosting algorithm uses simple classifiers, mostly decision trees, in a sequential manner, to provide results.

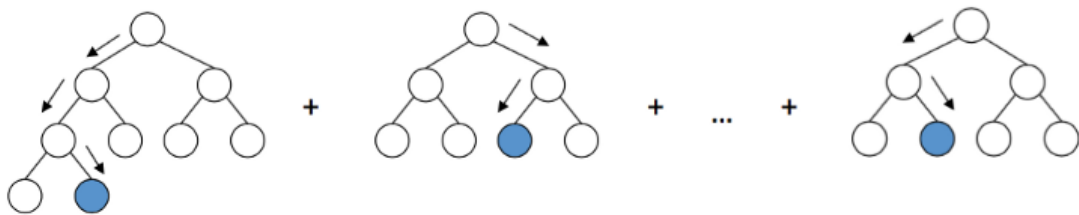


Fig. 7 Multiple Decision Tree Classifiers used for Gradient Boosting

The algorithm first uses the simple classifier to classify the data. Then the results are considered to calculate the errors or the data points that were not easily fit by the simple classifier. The algorithm then focuses on those data points in the next round and tries to fit them as well. In this way the errors are reduced, and outlier data points are also taken into consideration. But overdoing this remodeling can also cause overfitting. Hence learning to stop remodeling at an acceptable accuracy and error rate is also an important point to consider with this approach. In this paper, we use Gradient Boosting machine learning approach to detect and prevent SQL Injections.

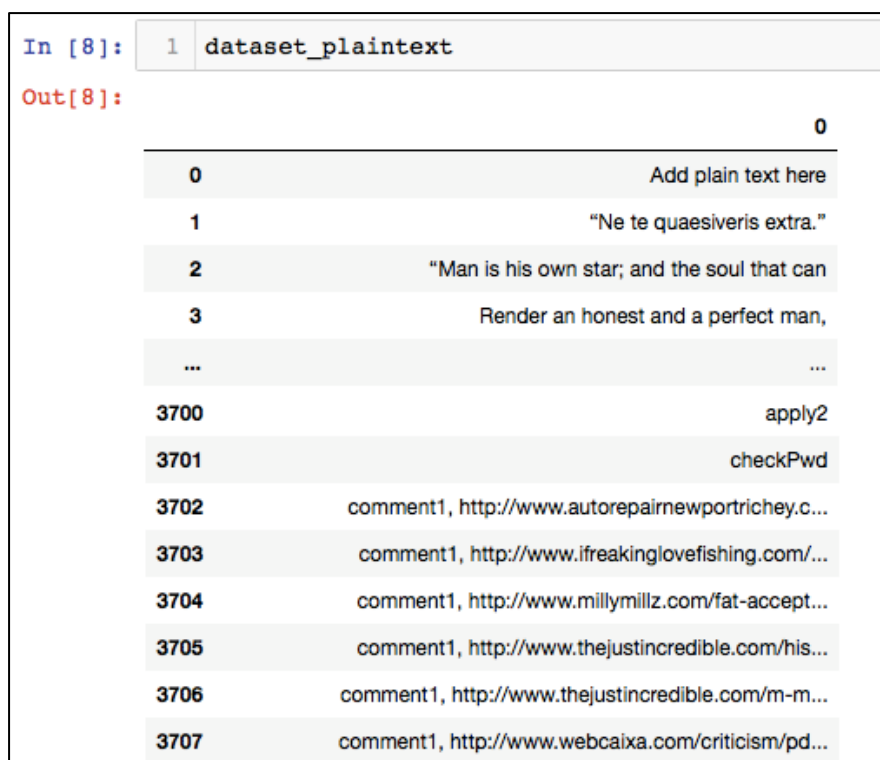
VI. Dataset

The dataset consists of the following two parts: Their descriptions are given under respective sections.

i. Plain-Text Dataset

This dataset consists of plain-text sentences and has around four thousand rows. The plain-text dataset has been created with payloads received from html forms. The dataset consists of a combination of URL's, special characters, textual data and numerical data.

Figure 8 shows a small sample of plain-text dataset.



```
In [8]: 1 dataset_plaintext
```

```
Out[8]:
```

	0
0	Add plain text here
1	"Ne te quaesiveris extra."
2	"Man is his own star; and the soul that can
3	Render an honest and a perfect man,
...	...
3700	apply2
3701	checkPwd
3702	comment1, http://www.autorepairnewportrichey.c...
3703	comment1, http://www.ifreakinglovefishing.com/...
3704	comment1, http://www.millymillz.com/fat-accept...
3705	comment1, http://www.thejustincredible.com/his...
3706	comment1, http://www.thejustincredible.com/m-m...
3707	comment1, http://www.webcaixa.com/criticism/pd...

Fig. 8 Sample Plain-text dataset

Following features of this dataset make it a good choice for this problem.

A. Diversity

The dataset not only contains just the textual data, but it is comprised of special characters and numbers. This is helpful while training the model to identify SQL Injections with better accuracy and avoid false positives.

B. Size

The dataset is large enough in size for our model to be trained properly.

C. Source

The dataset is created by collecting user inputs from a form in a web application.

Because of the source of the dataset, there is more probability of wide range of scenarios being covered for training the model efficiently.

ii. SQL Injection Dataset

Gathering a dataset for this problem was challenging as no datasets with public access to actual SQL Injection attacks that were launched are available. The dataset for SQL Injections has been created from a tool named Libinjection [13]. Libinjection is an open source tool that is used for penetration testing of web applications. It passes SQL Injections as payload to web applications and analyses if the application is vulnerable to SQL Injection attack. By the use of this tool, all the payloads generated by libinjection were captured for a particular instance and a dataset consisting of all these payloads is used as the SQL Injection dataset [14]. This dataset contains around six thousand SQL Injections of all the

VII. Tokenization:

In machine learning analysis that consists of text-based datasets, tokenization usually is the first and most important step in data preprocessing. In tokenization, sequence of characters are broken down into small pieces called ‘tokens’. Tokenization also includes removing certain characters sometimes. This practice is usually performed in word-based learning. A common example of tokenization in NLP (Natural Language Processing) is shown in figure 8 below. It can be seen how each part of the sentence is tokenized at every step.

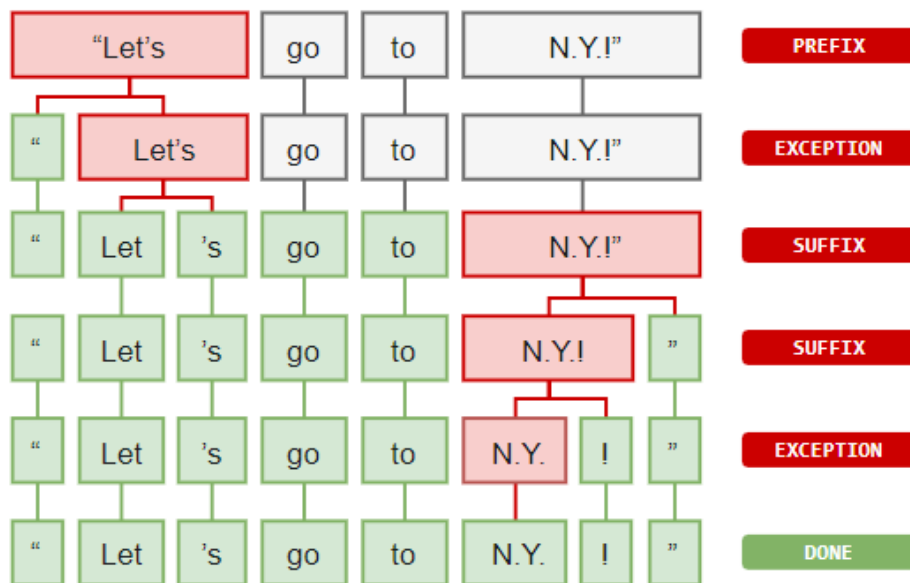


Fig. 10 Tokenization in NLP [15]

However, in our case, since we are trying to identify SQL Injections, every character in both the datasets is retained and the tokens are generated using regular expressions instead of tokenizing words. Sequence of characters are grouped together for tokenization in this approach.

i. Regular Expressions

Regex in python is used for tokenizing each entry in both the SQL Injection and plain-text datasets. They define a sequence of characters in a string format. Regular expressions are popularly used in pattern matching.

In this paper, *re.compile()* method is used to compile a regular expression sequence into a regular expression object.

```
In [ ]: import re
# this regex is being used to make tokens
sql_regex = re.compile("(?P<UNION>UNION\s+(ALL\s+)?SELECT) | (?P<PREFIX>({'\''\"})|((\''|\"|\s)|\d+|\w+)\s))(\s|\&\&|and|or
```

Fig. 11 re.compile() for Tokenization

The regular expression object is created using multiple SQL queries and SQL reserved words. Tokenization is implemented by lexical analysis using regular expression in python. Groupby() method is used to split the objects into tokens. After creating tokens from the dataset, feature extraction is performed on the dataset using the tokens. The token object has three parameters.

A. Token_Count:

The token_count parameter stores the number of times a particular token is present in the entire dataset.

B. Token_Value:

The token_value parameter stores the actual values of tokens that are created.

C. Token_Type:

The `token_type` parameter categorises each token as 'plain' or 'sqli'. Type 'sqli' tokens are generated from the SQL Injection dataset and type 'plain' tokens are generated from the plain-text dataset.

Besides the three parameters listed above, tokens are also grouped together using `groupby()` function, based on the sequence in which they most commonly occur together.

VIII. Feature Extraction

After tokenization of the dataset, feature extraction is performed on the data and first step in feature extraction is calculating the G-test Scores for all token values in the dataset. Prior to calculating the G-test scores, a dataframe is created using Pandas library in python. This dataframe acts as the new dataset and has the following columns.

- Token_Count
- Token_Value

G-test score calculation is then performed on this new dataset.

i. Step 1 : Calculate G-test Score

G-test Score is also called the likelihood ratio. It is considered to be an alternative to Chi-Square Test. G-test score is usually used when there is one nominal variable, that means there are two classes to classify. For example, if 'Sex' is the nominal variable, then 'Male' and 'Female' will be the two classes. This feature makes G-test score perfect to be used in our approach, since the classification is to classify data in two classes that are plain-text and SQL injection. G-test scores help to determine the variation of a prediction from ideal prediction and it is applied to categorical data.

To calculate the G-test scores, some pre-processing needs to be done on the data. The numerical values for counts in the data is converted to float type. Two types of G-test scores are calculated in this case.

- Observed G-test Score
- Expected G-Test Score

Expected G-test score is calculated based on the total tokens, number of tokens in a particular row and the types of tokens. Expected G-test score is the ideal score that should have been if the data was normally distributed. Observed G-test score is the actual score of the occurrence of data.

ii. **Step 2 : Calculate Entropy**

Next step in feature extraction is calculating Entropy of each row in the dataset. Entropy helps to measure the randomness of the data. If the data is very similar to each other, entropy of such dataset will be low, and if the data is diverse, entropy of such data will always be high. Decision trees use Entropy to split data. The goal of a decision tree is to split the data in such a way that similar data is grouped together. Hence the decision trees validate their split based on entropy. If the entropy decreases, they go ahead with the split, and if the entropy increases, they try to split at some other point.

The formula for Entropy is shown in figure 10 below.

$$Entropy = - \sum p(X) \log p(X)$$

Fig. 12 Entropy Formula

Where $p(x)$ = number of x / total number of features.

iii. Step 3 : Calculate G-test score mean

In this step, average value of G-test scores is calculated for each token in the dataset. A new dataset is generated and stored in Dataframe using pandas library in python. This dataset is actually used for training the model using Gradient Boosting Classifier. The dataset contains the following columns as shown in figure 11 below.

	raw_sql	type	sql_tokens	token_seq	token_length	entropy	sql_g_means	plain_g_means
0	Add plain text here	plain	[PLAIN]	[['PLAIN']]	1	3.536887	-5314.077226	11318.951712

Fig. 13 Dataset with Extracted Features

IX. Experiments

i. Experiment 1 : Naïve Bayes

Similar approach as above is used to implement Naïve Bayes Algorithm. Tokens are created and grouped together based on their occurrence.

Step 1 : Prior probabilities are calculated in the following manner.

- Calculate total number of rows in the dataset = `total_cnt`
- Calculate the number of SQL Injection rows in the dataset = `sqli_cnt`
- Calculate the number of plain-text rows in the dataset = `plain_cnt`

Prior probabilities are calculated as:

$$P(sqli) = sqli_cnt / total_cnt$$

$$P(plain) = plain_cnt / total_cnt$$

Where:

P (sqli) is the Probability of SQL Injection

P (plain) is the Probability of plain text

Step 2 : Next we calculate the likelihood of a new input being a SQL Injection or plain-text.

Likelihood is similar to the G-test score in this case and is calculated based on the number of tokens matching with the new input. Likelihood is calculated in the following manner.

- Calculate total number of tokens that match with the user input =
`Match-Token-Cnt`

Finally, likelihood is calculated by using the formula:

$$\text{Likelihood (sql)} = \text{Match_Token_Cnt} / \text{sql_cnt}$$

$$\text{Likelihood (plain)} = \text{Match_Token_Cnt} / \text{plaint_cnt}$$

Where:

Likelihood (sql) = The possibility that the new input is a SQL Injection

Likelihood (plain) = The possibility that the new input is a plain-text

Figure 14 shows the prediction accuracy of Naïve Bayes classifier.

	Accuracy
Naïve Bayes Classifier	92.8

Fig. 14 Naïve Bayes Classifier Results

ii. Experiment 2 : Gradient Boosting Classifier

Research suggests that Ensemble approaches in machine learning could provide better predictive accuracy than other classifiers.

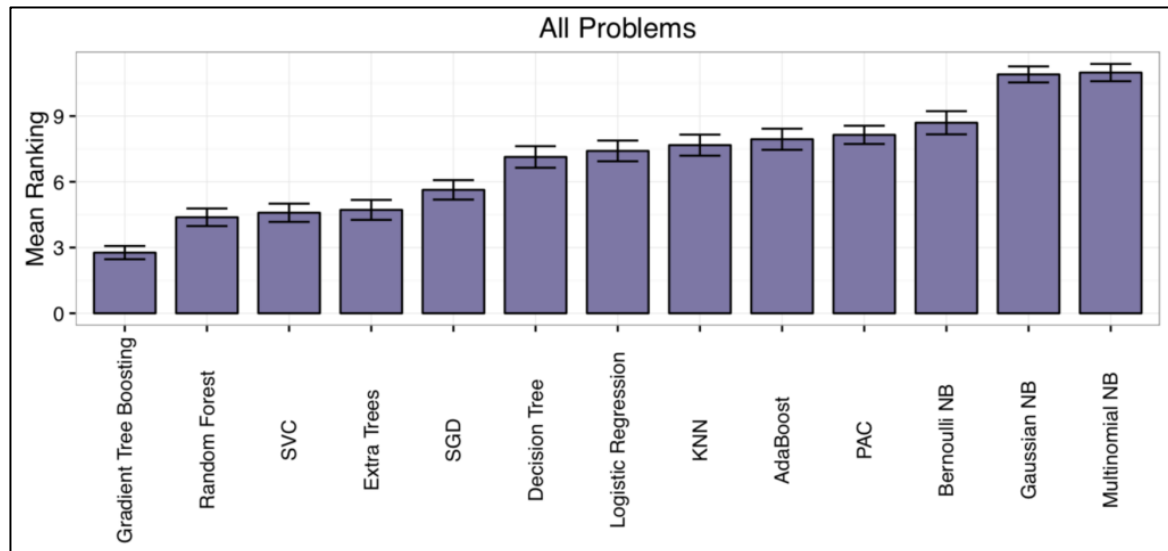


Fig. 15 Algorithm Mean Ranking [16]

Figure 13 shows a box-graph plot of comparisons of performance of various machine learning algorithms after applying them to various datasets. Gradient Boosting and Random Forest algorithms have lowest mean ranking, i.e. highest performance than other machine learning algorithms. Both the algorithms fall under Ensemble learning category. Hence Gradient Boosting Classifier was chosen to implement SQL Injection detection on our dataset. Gradient Boosting Classifier has been implemented from the ensemble part of Scikit-Learn library in Python. Parameter tuning is of high significance in ensemble learning algorithms. The parameter tuning for our problem is done as explained below.

A. Parameter Tuning

- **n_estimators**

This parameter decides the number of boosting stages that will be performed. Its default value is 100. The default value is used in this case. Large number of estimators gives better performance.

- **learning_rate**

Learning rate decides the contribution of each tree. The default value of this parameter is 0.1, which is also retained in this implementation.

- **max_depth**

max_depth decides the maximum depth of individual trees. The default value of this parameter is 3 however it was observed that the model performed better with a depth of 2. Hence the depth is set to 2.

- **random_state**

Random state can be set as an integer, 'RandomState' or 'None'. If it is set as an integer value, that particular node will always be used as a seed to generate trees. If it is set as 'RandomState', random_state will be used as a random number generator for trees. If it is set as 'None', np.random is used as a random number generator for trees.

It was observed that the model gave different results each time when it is run when the seed is randomly generated. To avoid inconsistency in results, this parameter is set to 0, so that the 0th node is always used as a seed for tree generation. Figure 16 shows the accuracy of predictions done by Gradient Boosting model.

	Accuracy
Gradient Boosting Classifier	97.4

Fig. 16 Gradient Boosting Classifier Results

X. Results and Analysis

Research shows that boosting approaches in machine learning can help reduce bias in models. The results from both experiments show that Gradient Boosting approach does perform better in terms of prediction accuracy. Figure 17 shows a comparison of the results of two experiments.

Parameters	Naïve Bayes	Gradient Boosting
Accuracy	92.8	97.4

Fig. 17 Accuracy Naïve Bayes vs Gradient Boosting

Gradient Boosting Classifier model is implemented for this problem as ensemble learning methods are said to perform better than simple classifiers, it seemed to be the right fit considering the criticality of failing to identify even a single SQL Injection. The model identifies all types of SQL Injections correctly as shown in below figures.

Identifying Union Based SQL Injection:

```
In [26]: 1 check_data = 'Hello UNION ALL SELECT NULL,version()--'
2 res = Check_is_sql(check_data)
3 if res == 1:
4     print ("This is a SQL Injection - %s" % check_data)
5 else:
6     print ("This is Plain Text - %s" % check_data)

This is a SQL Injection - Hello UNION ALL SELECT NULL,version()--
```

Fig. 18 Identifying Union Based SQL Injection

Identifying Error Based SQL Injection:

```
In [25]: 1 check_data = 'Hello 1 AND extractvalue(rand(),concat(0x3a,version()))--'
2 res = Check_is_sql(check_data)
3 if res == 1:
4     print ("This is a SQL Injection - %s" % check_data)
5 else:
6     print ("This is Plain Text - %s" % check_data)

This is a SQL Injection - Hello 1 AND extractvalue(rand(),concat(0x3a,version()))--
```

Fig. 19 Identifying Error Based SQL Injection

Identifying Boolean SQL Injection:

```
In [24]: 1 check_data = 'Hello AND 1=1'
2 res = Check_is_sql(check_data)
3 if res == 1:
4     print ("This is a SQL Injection - %s" % check_data)
5 else:
6     print ("This is Plain Text - %s" % check_data)

This is a SQL Injection - Hello AND 1=1
```

Fig. 20 Identifying Boolean SQL Injection

Identifying Time Based SQL Injection:

```
In [23]: 1 check_data = 'Hello 1 1 AND sleep(10)--'
2 res = Check_is_sql(check_data)
3 if res == 1:
4     print ("This is a SQL Injection - %s" % check_data)
5 else:
6     print ("This is Plain Text - %s" % check_data)

This is a SQL Injection - Hello 1 1 AND sleep(10)--
```

Fig. 21 Identifying Time Based SQL Injection

Identifying Plain-text:

```
In [27]: 1 check_data = 'Hello UNION How have you been'
2 res = Check_is_sql(check_data)
3 if res == 1:
4     print ("This is a SQL Injection - %s" % check_data)
5 else:
6     print ("This is Plain Text - %s" % check_data)

This is Plain Text - Hello UNION How have you been
```

Fig. 22 Identifying Plain-Text

```
In [28]: 1 check_data = 'Hello AND Quiet Crisis, 6476, http://www.webcaixa.com/surreal/'
2 res = Check_is_sql(check_data)
3 if res == 1:
4     print ("This is a SQL Injection - %s" % check_data)
5 else:
6     print ("This is Plain Text - %s" % check_data)

This is Plain Text - Hello AND Quiet Crisis, 6476, http://www.webcaixa.com/surreal/
```

Fig. 23 Identifying Plain-Text

As seen from above figures, the implemented model is able to classify between SQL Injection and Plain-text data. The machine learning model is able to identify almost all types of SQL Injections.

Ensemble approaches - of which Gradient Boosting is an example, also have certain trade-offs.

Trade-offs:

- Ensemble learning approaches tend to take longer time in the learning phase but are better learners.
- This approach can be easily susceptible to overfitting. In this project, overfitting is avoided by experimenting with the number of boosting stages to be performed. Selecting a number in higher range helps with the overfitting issue.

- Gradient Boosting approach is computationally expensive than simple classifiers in terms of memory and computation. More memory is needed to store multiple trees.

XI. Conclusion and Future Work

SQL Injection attacks remain to be one of top concerns for cyber security researchers. Signature based SQL Injection detection methods are no longer reliable as attackers are using new types of SQL Injections each time. There is a need for SQL Injection detection mechanisms that are capable of identifying new, never before seen attacks. Applying machine learning to the field of cyber-security is being considered by many researchers. Since machine learning in cyber-security is still a developing research area, there are not many libraries and open source tools that are machine learning specific and apply to problems related to threats and attacks.

In this thesis, the SQL Injection detection problem is approached by applying machine learning algorithms. Classification method is used to classify the incoming traffic as a SQL Injection or plain text. Two machine learning classification algorithms are implemented on the problem, which are, Naïve Bayes Classifier and Gradient Boosting Classifier. Naïve Bayes classifier machine learning model provides results with an accuracy of 92.8%. Ensemble learning methods are said to provide results with better accuracy as they implement multiple simple classifiers to improve error and accuracy. Hence Gradient Boosting Classifier from ensemble learning is selected to be implemented on the SQL Injection classification problem. Various combinations of parameters are tuned and tried together and an accuracy of 97.4% is achieved by the Gradient Boosting classifier. From this project it can be concluded that machine learning approaches can be used for SQL Injection detection, and Gradient Boosting classifier algorithm provides better accuracy than Naïve Bayes approach.

For future work this project could further be modified in terms of usability and efficiency. The machine learning approach for detecting SQL Injections could be used in combination with other SQL Injection detection mechanisms such as static code analysis and web application firewalls. The machine learning model can also be advanced further with better feature extraction. In this project tokenization approach is used to create features for the machine learning model. Other approaches can be used for feature extraction and training the model in more effective manner.

REFERENCES

1. J. Abirami, R. Devakunchari and C. Valliyammai, "A top web security vulnerability SQL injection attack — Survey," *2015 Seventh International Conference on Advanced Computing (ICoAC)*, Chennai, 2015, pp. 1-9.
2. Diallo, A. K., Al-sakib, K. P.: A survey on SQL injection:vulnerabilities, attacks, and prevention techniques.2011. Retrieved from http://irep.iium.edu.my/769/1/ISCE2011_paper323.pdf and accessed on 10th June, 2017.
3. C. Gould, Zhendong Su and P. Devanbu, "JDBC checker: a static analysis tool for SQL/JDBC applications," *Proceedings. 26th International Conference on Software Engineering*, Edinburgh, UK, 2004, pp. 697-698.
4. Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan, "CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks", *ACM Transactions on Information and System Security (TISSEC)*, v.13 n.2, p.1-39, February 2010.
5. William G. J. Halfond , Alessandro Orso, "AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks", *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, November 07-11, 2005.
6. Gregory Buehrer , Bruce W. Weide , Paolo A. G. Sivilotti, "Using parse tree validation to prevent SQL injection attacks", *Proceedings of the 5th international workshop on Software engineering and middleware*, September 05-06, 2005
7. A. Joshi and V. Geetha, "SQL Injection detection using machine learning," *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, Kanyakumari, 2014, pp. 1111-1115.
8. William GJ. Halfond and Alessandro Orso," Preventing SQL Injection Attacks Using AMNESIA" *ICSE'06*, May 20-28, 2006, Shanghai, China ACM 06/0005.
9. Takeshi Matsuda, Daiki Koizumi, Michio Sonoda, Shigeichi Hirasai, "On predictive errors of SQL injection attack detection by the feature of the single character"

- Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on 9-12 Oct 2011, On Page 1722-1727.
10. Angelo Ciampa, Corrado Aaron Visaggio, Massimiliano Di Penta : "A heuristic-based approach for detecting SQL-injection vulnerabilities in Web applications".
 11. P. Joshi, Dissecting Bias vs. Variance Tradeoff In Machine Learning, 2015 [Online] Available: <https://prateekvjoshi.com/2015/10/20/dissecting-bias-vs-variance-tradeoff-in-machine-learning/>
 12. Xristica, What is the difference between Bagging and Boosting?, 2016 [Online] Available: <https://quantdare.com/what-is-the-difference-between-bagging-and-boosting/>
 13. Nick Galbreath, 'libinjection', 2012. [Online] Available: <https://github.com/client9/libinjection.git>
 14. foospidy/payloads, libinjection_bypasses.txt, 29 June 2007. [Online] Available: <https://github.com/foospidy/payloads.git>
 15. Linguistic Features, 'Spacy'. [Online] Available: <https://spacy.io/usage/linguistic-features>
 16. J. Brownlee, 'Start With Gradient Boosting, Results from Comparing 13 Algorithms on 165 Datasets', March 30, 2018. [Online] Available: <https://machinelearningmastery.com/start-with-gradient-boosting/>
 17. G Buehrer, B.W. Weide, P.A.G Sivilotti, Using Parse Tree Validation to Prevent SQL Injection Attacks, in: 5th International Workshop on Software Engineering and Middleware, Lisbon, Portugal, 2005, pp. 106-113.
 18. Z. Su and G Wassermann "The essence of command injection attacks in b applications". In ACM Symposium on Principles of Programming Languages (POPL'2006), January 2006.
 19. S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In Proceedings of the 2nd Applied Cryptography and Network Security Conference, pages 292-302, June 2004.
 20. RA. McClure, and J.H. Kruger, "SQL DOM: compile time checking of dynamic SQL statements," Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, pp. 88-96, 15-21 May 2005.

21. Ke Wei, M. Muthuprasanna, Suraj Kothari, "Preventing SQL Injection Attacks in Stored Procedures" Proceedings of the 2006 Australian Software Engineering Conference (ASWEC'06 IEEE).
22. P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks. ACM Trans. Inf. Syst. Secur., 13(2): 1-39, 2010.
23. Mei Junjin, "An Approach for SQL Injection Vulnerability Detection," Proc. of TTNG '09, pp.1411-1414, 27-29 April 2009.
24. YongJoon Park, JaeChul Park, "Web Application Intrusion Detection System for Input Validation Attack "Third 2008 International Conference on Convergence And Hybrid Information Technology.
25. Needleman, S.B., Wunsch, C.D. "A general method applicable to the search for similarities in the amino acid sequence of two proteins", .T.Mol.Biol.48:443-453, 1970.
26. Sangita, R., Avinash, K. S., Ashok S. S.: Detecting and Defeating SQL Injection Attacks International Journal of Information and Electronics Engineering, 2011.
27. Nausheen, K.: Detection and Prevention of SQL Injection Attacks by Request Receiver, Analyzer and Test Model. 2011.
28. Cristian, N., et al.: CBRid4SQL: A CBR Intrusion Detector for SQL Injection Attacks. 2010.
29. Shikhar Jain & Alwyn R. Pais," Model Based Approach to Prevent SQL Injection Attacks on.NET Applications" International Journal of Computer Science & Informatics, Volume-1, Issue-11, 2011.