2021

Quality of SQL Code Security on StackOverflow and Methods of Prevention

Robert Klock
Oberlin College

Quality of SQL Code Security on StackOverflow and Methods of Prevention

Robert Klock
Oberlin College

Dr. Cynthia Taylor
Oberlin College

Abstract

This paper explores the frequency at which SQL/PHP posts on the website Stackoverflow.com contain code susceptible to SQL Injection, a common database vulnerability. Specifically, we analyze whether other users give notice of the vulnerability or provide an answer that is secure. The majority of questions analyzed were vulnerable to SQL Injection and were not corrected in their answers or brought to the attention of the original poster. To mitigate this, we present a machine learning bot which analyzes the poster's code and alerts them of potential injection vulnerabilities, if necessary.

Keywords: StackOverflow, SQL Injection, security, text mining, machine learning

1 Introduction

StackOverflow is a Q&A website for programmers to ask and answer questions pertaining to computer science, oftentimes for programming and software development. It is used by experts and beginners alike and provides an environment for the exchange of ideas and advice. However, the ease of use associated with StackOverflow can be harmful when provided code is insecure or security vulnerabilities go ignored or undetected by viewers. As presented by Fischer et al. in [1], code snippets from StackOverflow are continuously used as is in real world applications and often contain security vulnerabilities. Posts related to databases on StackOverflow frequently do not include discussion of potential Injection vulnerabilities in posted code when appropriate.

This paper explores the practices around insecure SQL and PHP code on StackOverflow, specifically the amount of code snippets that are injectable and the frequency at which answers and comments point out or correct those issues. We outline the behaviors associated with insecure SQL and PHP posts on StackOverflow and give an outline of various classification techniques to detect and prevent database injection.

2 Motivation

Injection, including SQL Injection, is the number one web application security vulnerability as of 2020 according to the Open Web Application Security Project® (OWASP) nonprofit [6].

A SQL Injection occurs when a user with malicious intentions gains access to a website's database by exploiting poor handling of user input. Successful injections can result in access to confidential database information and deletion or modification of data, among other turnouts. SQL Injection sets itself apart from other security vulnerabilities because of its relatively straightforward prevention methods (Real Escape, Bounded User Input, etc.) and subpar discussion in university-level database textbooks [7]. The combination of lack of education surrounding SQL Injection prevention, ease and frequency of use of code provided on StackOverflow, and relatively uncomplicated solution of injection vulnerabilities led to the research project outlined in this paper.

We show that PHP/SQL code on StackOverflow is frequently insecure and not mentioned or prevented in answers or comments. To mitigate this, we present a bot that automatically detects, warns, and educates StackOverflow users about the possibility of SQL Injection in their code. While the specific rate at which software developers copy provided code into production is not known, it is clear that it is a somewhat common practice despite its potential danger.

3 Related Work

In [1], Fischer et al. discuss how StackOverflow users oftentimes copy the code provided to them online directly into their production software. In particular, they show the frequency at which code from StackOverflow was used as is in Android applications available on the Google Play app store. Alarmingly, 15.4% of the 1.3 million applications analyzed contained a security-related code snippet. More so, 97.9% of those code snippets were insecure, defined as:

> Snippets that contained obviously insecure code, e.g. using outdated algorithms or static initialization vectors and keys for symmetric cryptography, weak RSA keys for asymmetric cryptography, insecure random number generation, or insecure SSL/TLS implementations.

Despite their inherent insecurity, these code snippets still made it into production applications, which could be potentially devastating.

In [7], Taylor and Sakharkar discuss the coverage of SQL Injection and prevention in undergraduate database courses. In the top 50 computer science programs in the United States (at the time of writing) seven textbooks were used in total. Of those, only two explicitly mentioned SQL Injection. Others even provided examples that are susceptible to Injection.

4 Methods

4.1 Obtaining Data

The entirety of StackOverflow was downloaded over the course of many days using curl. This was done for ease of querying as we were specifically looking for pages related to SQL or PHP. Since the database was only downloaded once, all posts that were analyzed during this project were at least a year old (most fell into the pre-2016 years). Although some tools and paradigms undergo drastic changes over the course of a year, SQL Injection-vulnerable practices do not change as fast, so the gap in analyses poses little disadvantage to our study. Additionally, pages on StackOverflow are likely to be stagnant after their first year, which provided an opportunity for us to observe them consistently across different trials.

Later in the paper we will outline the machine learning methods used to classify vulnerable code snippets. The model presented in this paper utilized batch learning, which means a set of data was trained and tested on and no new data was passed in. Another training possibility is referred to as *online* learning, where a model learns on some data and is continuously retrained on new data. This would be the case if we periodically pulled data from StackOverflow for retraining, but we did not implement that.

The full data obtained from StackOverflow had multiple issues of clarity. First, it was human-produced in an uncontrolled environment. Questions may be unclear, comments may be incorrect, etc. With that, the code we obtained was still difficult to analyze. It is rare for someone to post the entirety of their application or codebase in a single StackOverflow post, so we only had a small glimpse into the practices going on (sometimes, though, these were the most critical parts of their application). Additionally, programmers may include irrelevant comments in their code that contain terms that could otherwise be interpreted as SQL code. With this, code was extensively cleaned to remove irrelevant characters (queries like SELECT were kept, but the $ before $SELECT would be taken away).

4.1.1 Qualitative Coding

In order to analyze the corpus of StackOverflow posts effectively, Eliana Grossof developed a qualitative coding criteria for quick and consistent analysis and categorization. This method involves creating a series of binary variables to describe a post (i.e. *relevant, contains a code snippet, is SQL-injectable*). Throughout the duration of the project, we analyzed 1117 StackOverflow posts (questions, answers, and comments) and qualitatively coded them. Then, each post was re-coded independently by another team member for verification. If a mismatch occurred in the code representation of a post between two people they met and discussed a final code representation of the post. StackOverflow allows for multiple answers for a single post. When we refer to the answer of a post, we are referring to the answer with the most upvotes. All other answers get categorized with “everything else”.

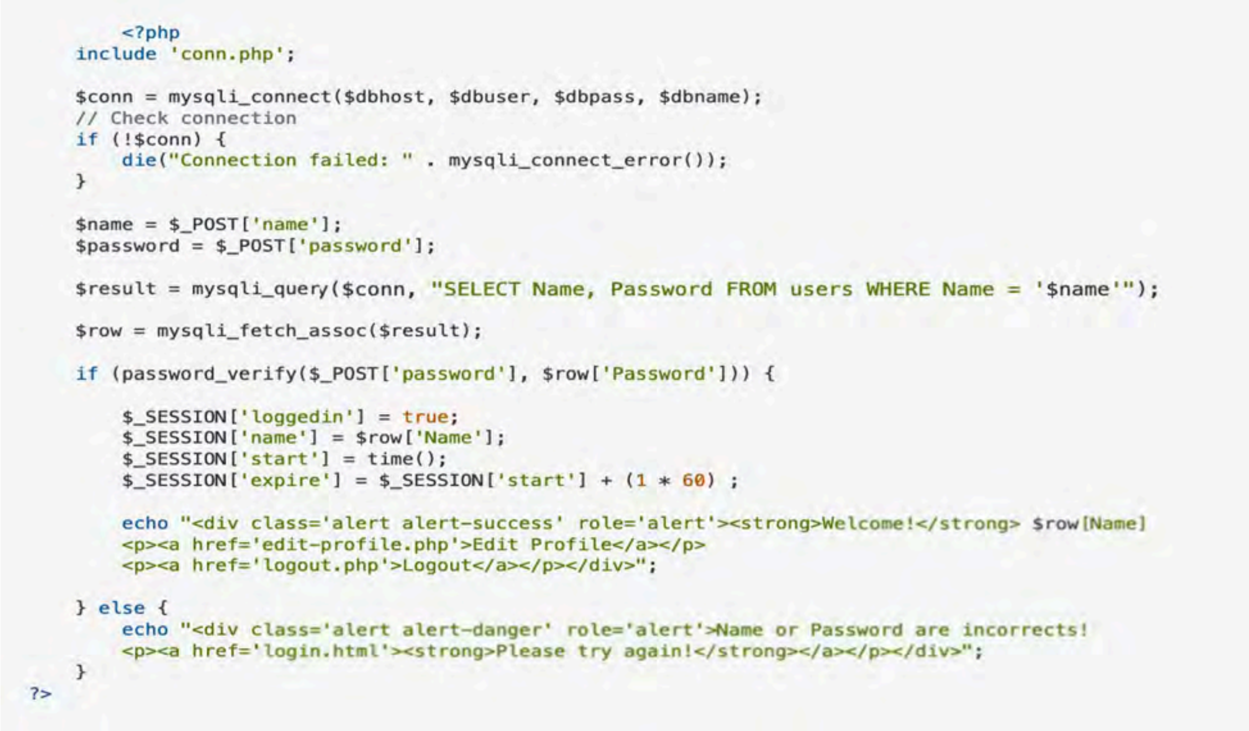
R	SQLI	RE	L	CO	P	MSQLI	BUI
1	1	0	0	1	0	0	0

Table 1: Qualitative codes for the code snippet in Figure 1 indicating the values of relevancy (R), vulnerability to SQL Injection (SQLI), use of Real Escapes (RE), external links (L), presence of a code snippet (CO), use of at least one prepared statement (P), mention of SQL Injections (MSQLI), and use of bounded user input (BUI).

Table 1 shows the corresponding qualitative codes for the code snippet of a StackOverflow post shown in Figure (1).

5 Detect-and-Warn Bot Motivation and Approaches

One behavioral pattern emerged upon analysis of hundreds of comments: oftentimes when people had glaring vulnerabilities in their code, commenters would simply



```
    <?php
include 'conn.php';

$conn = mysqli_connect($dbhost, $dbuser, $dbpass, $dbname);
// Check connection
if (!$conn) {
    die("Connection failed: " . mysqli_connect_error());
}

$name = $_POST['name'];
$password = $_POST['password'];

$result = mysqli_query($conn, "SELECT Name, Password FROM users WHERE Name = '$name'");

$row = mysqli_fetch_assoc($result);

if (password_verify($_POST['password'], $row['Password'])) {

    $_SESSION['loggedin'] = true;
    $_SESSION['name'] = $row['Name'];
    $_SESSION['start'] = time();
    $_SESSION['expire'] = $_SESSION['start'] + (1 * 60) ;

    echo "<div class='alert alert-success' role='alert'><strong>Welcome!</strong> $row[Name]
    <p><a href='edit-profile.php'>Edit Profile</a></p>
    <p><a href='logout.php'>Logout</a></p></div>";

} else {
    echo "<div class='alert alert-danger' role='alert'>Name or Password are incorrects!
    <p><a href='login.html'><strong>Please try again!</strong></a></p></div>";
}
?>
```

Figure 1: An example code snippet from a StackOverflow post with its corresponding qualitative code representation shown in Table 1.

tell the poster to research SQL Injection on their own instead of providing background information, links to resources, or a secure solution. To alleviate some of the burden of trying to find helpful information, our bot will not only detect instances of vulnerable code, but also provide some educational resources for the poster to educate themselves.

A variety of approaches were explored to implement the bot to detect and warn StackOverflow users of Injection vulnerabilities in their code. These include Linear Support Vector Machines (SVMs), Naive Bayes, Decision Trees, and deep neural networks. Extensive data pre-processing was used to transform the code from StackOverflow posts into a format suitable for machine learning algorithms. The pipeline to obtain data for the models consisted of downloading StackOverflow pages, gathering the code from each page by looking for code blocks in HTML posts which were tagged with ⟨code⟩ tags, and cleaning and encoding that text into vectors using TF-IDF vectorization.

6 Encoding with TF-IDF

Machine learning algorithms cannot easily deal with raw natural language. Instead, it must be transformed into numerical format. TF-IDF Vectorization is one of the ways we can put words into a format that classification algorithms can work with. TF stands for term frequency, which is the amount of a time a term occurs in a document. The more times a term appears in a document, the larger its term frequency. IDF stands for inverse document frequency, which is the frequency of that term appearing in other documents. The advantage of using TF-IDF vectorization over other "bag of words" representations is that TF-IDF places less emphasis on terms that appear frequently over all documents in a corpus. For example, the term "a" occurs frequently in many texts and is relatively uninteresting (it does not tell us much about how a given sequence of terms is different from others). Terms with a high term frequency and document frequency (i.e. they occur many times in each document of a corpus) are called "stop words" and get encoded with a small weight with TF-IDF vectorization. Importantly, TF-IDF embeddings do not preserve the ordering of words in a document.[1]

So, "I love CS, CS hates me" yields the same encoding as "I hate CS, CS loves me". This poses issues in situations where ordering is critical, but we found reasonable

[1] Note: In our implementation, we make use of a TF-IDF parameter that takes in a corpus of stop words and ignores any instances of them. This allows the model to ignore unhelpful terms, even if those terms may not be considered "stop words" in the data. We chose to ignore all typical stop words in the English language.

success in classification of code regardless. Implementing a bot that uses an order-preserving method of text embedding has merit for a future project.

$$TF(t,d) = \frac{f_{t,d}}{\sum t' \in d \; f_{t',d}} \tag{1}$$

$$IDF(t,d) = \; log \frac{N}{|d \in D : t \in d|} \tag{2}$$

$$TFIDF(t,d,D) = \; TF(t,d) * IDF(t,D) \tag{3}$$

Equation (1) shows the computation of term frequency given a term t and document d. $f_{t,d}$ is the frequency of a particular term in the given document, it is divided by the total amount of unique terms appearing in the given document. Equation (2) is the calculation of the IDF, which is the logarithmically scaled inverse ratio of documents containing the given term over all documents in the corpus N. TF-IDF is the product of these two calculations.

In natural language, the significance or meaning of a term is not always explicitly unique to the term itself. Sometimes the terms around a given term affect its meaning. This is especially the case in Bag of Words embeddings like TF-IDF. Since order is not preserved, “The sky is blue, the sky is not green” is encoded the same as “The sky is not blue, the sky is green”. These sentences have drastically different meanings because of the location of the “not”. We can increase the number of “grams” to improve on this and introduce some order into our embedding. In the original sentence (with one gram) the respective grams are [The, sky, is, blue, not, green]. With two grams, we add the grams [The sky, sky is, is blue, blue the, the sky, is not, not green]. This is a bit easier to decipher, and it becomes clear that the “not” is directly related to the word “green”. We achieved best performance with one to three grams for the Linear SVM.

The utilization of various amounts of grams greatly increases the amount of terms in the resulting vector. Fortunately, the implementation of TF-IDF vectorization we use takes two other parameters: maximum document frequency and minimum document frequency. Maximum document frequency disqualifies grams that appear in more than a certain percentage of documents (80%, for example), limiting the amount of stop words that make their way into the eventual vectorization. Minimum document frequency disqualifies grams that do not appear in at least a certain amount of documents (for example, a term must appear in at least 50 documents). Both of these parameters are tuned by hand and reasonable values can be found via grid search.

7 Classification Techniques

A few different techniques for classification were explored over the course of this project, including Linear SVMs, Naive Bayes, Decision Trees, and deep neural networks. Each implementation was built out by myself, with the exception of Naive Bayes which was implemented by Sam Fertig. Evans Muzulu assisted in the creation of the Decision Trees model. All models were trained on the same batch of training data (using 495 questions) and evaluated during training using k-fold cross-validation. After analyzing the training performance of each model (its cross-validation accuracy), Linear SVM and deep neural networks were selected for further fine-tuning to improve performance.

Cross-validation, specifically k-fold cross-validation, is a technique used to better evaluate the performance of a model on its training data. During training, the training set is split up into smaller chunks called folds. Then, the model is trained on n-1 folds and validated on the final fold. The validation accuracy of all classifiers was averaged across folds to get a mean validation accuracy.

7.1 Linear SVM

We chose to work with Linear SVMs because of their ability to perform well on complex and small datasets and their capability to generalize well with soft margin classification [2]. Unlike traditional logistic regression models, Linear SVMs work by finding a decision boundary between two classes that maximizes the distance from each class. This forms what is referred to as a “wide road”. This, paired with soft-margin classification, led to a model that was fairly accurate at classifying vulnerable code blocks from StackOverflow.

As opposed to having a hard-set boundary dividing classes from each other, soft margin classification allows for some outliers of one class to belong to another class. This makes a model robust to outliers and increases its likelihood to generalize well to new data. The nature of our problem and data is not so clear cut (there are no hard-set rules for when code is or is not SQL Injectable, and not all information is accessible) so soft margin classification comes in handy when analyzing data. We made use of a relatively small hyperparameter value of C, which modulates the width of the “road” (larger numbers, like 100, shrinks the width). This led to more margin violations but leaves room for the model to generalize to more data.

Analyzing the accuracy of a model is not always the best way to understand its full performance. Two better metrics include precision and recall, which comprise the

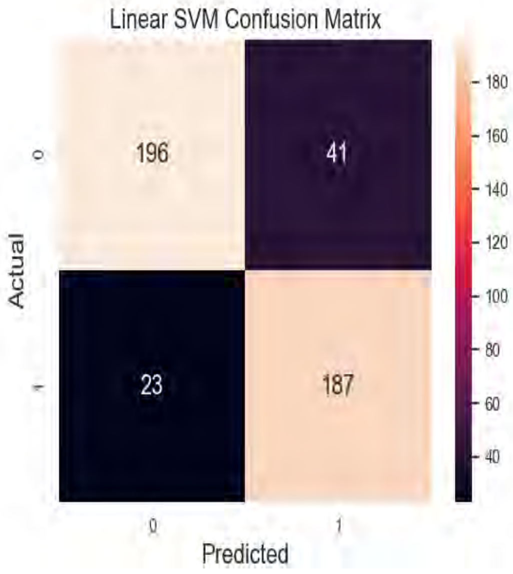


Figure 2: Confusion matrix of the Linear SVM model. From top left clockwise are the total occurrences of true negatives, false positives, true negatives, and false negatives.

axes of a confusion matrix plot for a model.

$$precision = \frac{\text{True Positives}}{\text{True Positives + False Positives}} \quad (4)$$

$$recall = \frac{\text{True Positives}}{\text{True Positives + False Negatives}} \quad (5)$$

The equations for precision and recall are shown in (4) and (5) respectively. The precision keeps track of how often the model made a correct positive classification. Recall measures the proportion of correctly identified positive instances. Together, they make up the vertical and horizontal axes in Figure (2). During training, we made use of k-fold cross validation with a fold value of 8. Overall, the SVM's validation accuracy was 84% and its test accuracy dropped to 74.2 %. The confusion matrix outlining the ways in which the Linear SVM classified code is shown in Figure (2).The Linear SVM's precision score was 82% and its recall score was 89%. We did not mind having many false negatives as this would prevent the bot from over alerting StackOverflow users and eventually being taken down on account of spam. With that said, the confusion matrix top left and bottom right corners are important, as they indicate the true positives and negatives. The bottom left quadrant indicates the frequency of false positives. Those are the instances where our model was not confident and misclassified an injectable snippet as safe. Our team feels this is better than potentially overwarning people whose code is actually secure (which would add more instances to the top right corner).

7.2 Naive Bayes

Naive Bayes classifiers make use of Bayes' Theorem, which is shown in (6)[2]. Similar to Decision Trees, Naive Bayes models learn which attributes are associated with high probabilities of belonging to a certain class. In our case, certain grams may be more frequently used in SQL Injectable code than others. When that gram appears in a new instance, it may push the classifier closer to believing that instance is SQL Injectable. This process of combining the different probabilities of different grams belonging to each class is continued for grams in new data, by which a final classification is made [5]. Similar to our method of embedding text into vectors, Naive Bayes does not make use of order of probabilities (this is why it is called *Naive*). This may seem like an oversimplification to work with, but in practice it has been shown to perform well.

In our case, however, Naive Bayes did not perform well. It predicted that all instances in the validation set were SQL Injectable (see Figure (3)). In some cases, this was a correct prediction. However, we do not want to scare every StackOverflow user that posts about SQL into thinking that their code is insecure, so we did not continue with the Naive Bayes model.

$$P(h|D) = \frac{P(D|h) * P(h)}{P(D)} \quad (6)$$

7.3 Decision Trees

As presented in [5], Decision Trees work by systematically sorting an instance into different possible categorizations based on the presence of certain attributes in said instance. The various combinations of attributes that comprise a given instance form a path from the root of the tree to a leaf, by which a final classification is made.

A Decision Tree is formed from training data by analyzing the entropy and information gain of a certain attribute which adjusts where the attribute is analyzed in the tree (higher information gains correspond to decisions closer to the root). The entropy and information gain of an attribute A in a set S are shown in equations

[2] P(h|D) is the posterior probability: that hypothesis *h* is true given *D*. P(D|h) is the probability of *D* given *h* is true. P(h) is the probability of *h* being true (also called prior probability), and P(D) is the probability of *D* regardless of *h*.

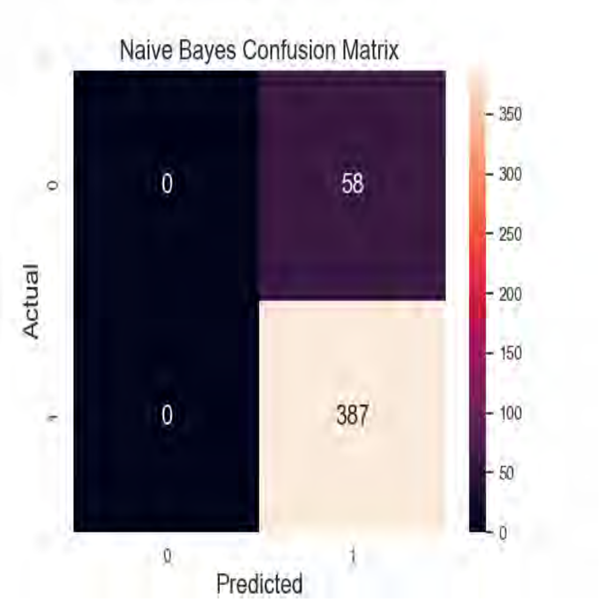


Figure 3: Confusion matrix of the Naive Bayes classifier. It did not perform well because it believed everything it came across was SQL Injectable.

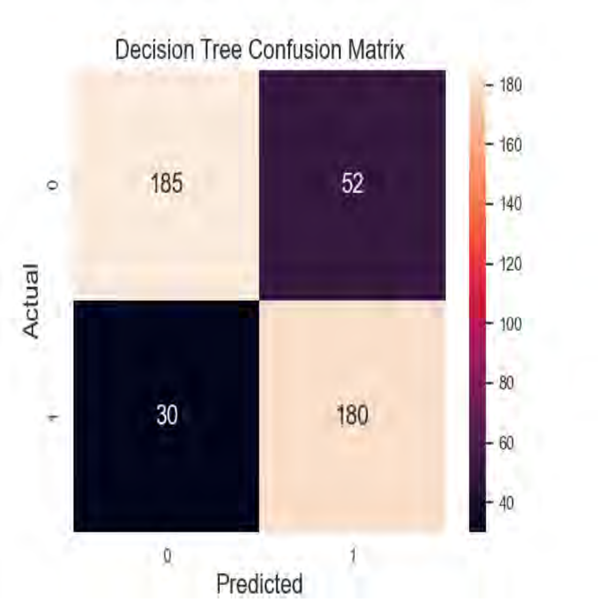


Figure 4: Confusion matrix of the Decision Tree classifier. It performed slightly worse than the Linear SVM model during training and validation.

(7) and (8)[3][4] The validation accuracy over 8 folds for Decision Trees was 79%, which dropped to 66% after testing. Since Linear SVM performed better, we chose to not continue with Decision Trees, though Random Forests may be explored in the future.

$$E(S) = \sum_{i=1}^{c} -p_i log_2 p_i \tag{7}$$

$$Gain(S,A) = E(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} E(S_v) \tag{8}$$

7.4 Deep Neural Net

Many attempts were made at crafting a deep neural network for classification. However, all models overfit the data and did not perform well on validation sets. Multiple different configurations and regularization techniques were used to improve performance of the neural network. Once overtraining was detected, dropout was implemented. Dropout is a regularization technique that was proposed by Hinton et al. in [3]. It works by randomly ignoring certain neurons during training, which keep a neural network from overfitting the data. This was a promising approach, as it is especially effective for networks that operate on small amounts of data, but no improvement was found. We also attempted to optimize the model with Adam optimization, but saw little improvement [4]. This led us to believe that the problem is not appropriate for a deep neural network (its use is overkill). Or, there was simply not enough data to give it at the time which would allow it to learn the necessary patterns to detect SQL Injection-vulnerable code. However, this does not mean that one cannot be used down the line once more data is gathered or the model is implemented in a way that allows for online learning.

8 The Bot

As of the completion of this thesis, the bot has not been fully implemented. However, it is expected to be finished by the end of this Spring semester. We will make use of the StackOverflow API to periodically pull and analyze

[3] In (7), p_i is the proportion of an instance S belonging to class i. c is the total number of different possible classes. In our case, c = 2 since a binary classification is made: a post's code either SQL Injectable or not.

[4] In (8) A is a given attribute and $Values(A)$ is all possible values for A. S_v is the subset of all training examples S where an attribute has value v. So, the second half of equation (8) calculates the total loss of entropy for a given attribute, which tells us more information when making a decision about an example.

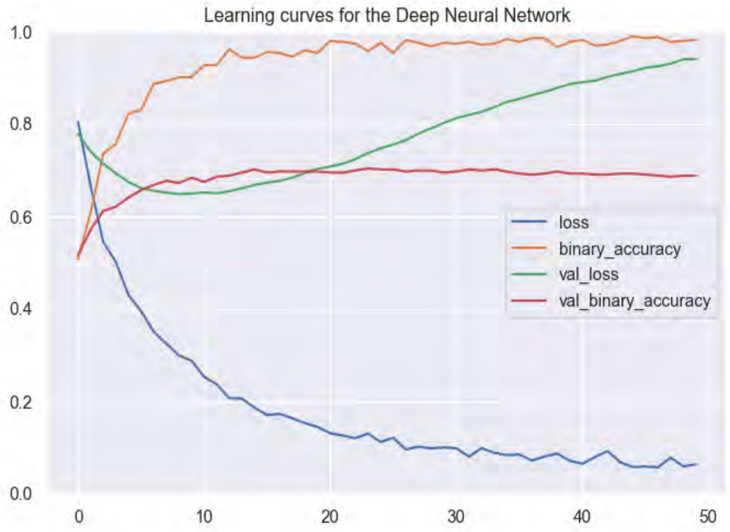


Figure 5: Learning curves for the deep neural network during training: the mean training loss and accuracy during each epoch and the mean validation loss and accuracy at the end of each epoch. Training accuracy and validation accuracy are ideally supposed to rise closer to 1 during training. Training and validation loss should have decreased during training. However, validation loss takes a dip (not even into an appropriate range), then begins to increase. Validation accuracy increases and then plateaus, showing no improvement in learning. This can be caused by a number of factors including a small validation set, which we have.

posts from StackOverflow with PHP or SQL labels. This is done via a cron job running on OCCS. From there, posts will be passed through another machine learning model in development which will determine if the post in question is relevant to us or not (likely using a Linear SVM again). If so, the post will be analyzed by the Linear SVM classifier described above. Posts with a positive classification will be added to a queue for commenting. The queue will be emptied by posting a comment on each post which will warn the user of possible vulnerabilities in their code. This comment also will also include a link to a web page we set up on OCCS that discusses the project, the bot, and the necessary information for developers to protect themselves against SQL Injections.

9 Results

Overall, we found StackOverflow to not be SQL Injection warning savvy. A very small percentage (four out of all 1117) of answers to vulnerable questions contained code that answered the poster's question and fixed their security flaws. This could be attributed to the relatively sub par coverage of Injection prevention techniques in the literature, vulnerabilities that went undetected, or unwillingless to alert the poster, among other possibilities.

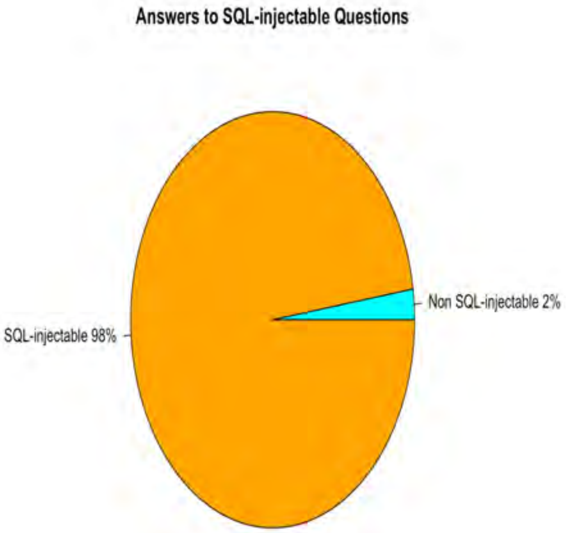


Figure 6: Percentage of analyzed answers which do not offer a solution for the original post's vulnerability. Very few users provided secure solutions to insecure posts.

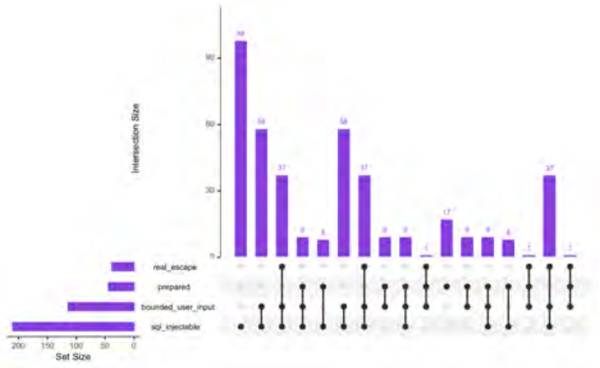


Figure 7: Upset plot showing the total amount of questions recorded that used real escapes, prepared statements, bounded user input, or were SQL Injectable (lower left). It also shows the frequency of various qualitative coding combinations across all questions. Clearly, the majority of questions were coded as injection-vulnerable. Few made use of all three prepared statements, bounded user input, and real escapes.

Many pages ended up with identical combinations of qualitative codes. This may be an indication that coding practices which lead to insecure code are learned (though, it could just be a pattern that emerged for no reason). The frequency of each combination of codes for all posts, questions, and everything else on a given StackOverflow page that we analyzed are expressed in Figures (7), (8), and (9), respectively, as upset plots, which were created by Eliana Grossof. The bottom left corners of these figures show the amount of total times a post element was coded as SQL Injectable, made use of bounded user input, included a prepared statement, or used a real escape. The main plot shows the distributions of various combinations of these codes across each page's subsections. It is clear that the widest range of code combinations occurs in the questions on StackOverflow. Comments and answers most frequently just contain SQL Injectable advice or code. Additionally, behaviors that make use of more than one qualitative coding trait are more frequent than those that demonstrate a single behavior. Of course, these are only hypotheses that were extrapolated from the data and not verified in an experiment to find causality.

10 Future Work

Future work (including time spent on the project during the rest of this semester) consists of finalizing the setup of the bot to post comments to StackOverflow on our behalf, honing the messages and resources (or potentially writing our own) to communicate effectively the

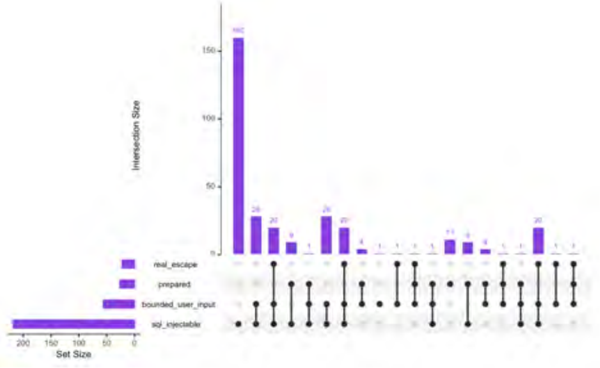


Figure 8: Upset plot which follows the same format as in Figure (7), but for the qualitative codes representing all answers analyzed from StackOverflow. Again, the majority of answers are SQL Injectable. However, there is a large drop off in the amount of answers that fall into most other combinations of codes.

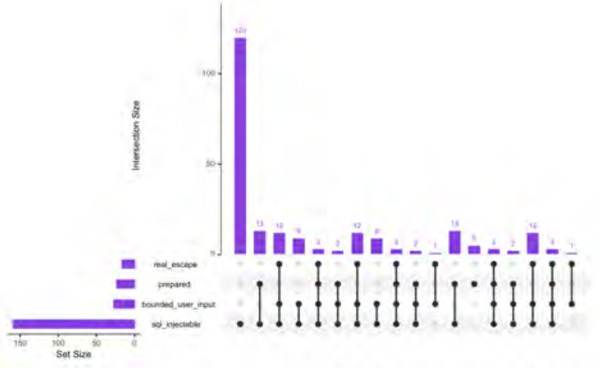


Figure 9: Upset plot for all instances of "everything else", which is all non top-upvoted answers and comments on a given StackOverflow post. As with the answers presented in Figure (8), the majority of everything else on a post was SQL Injectable and very few instances provided code that made use of real escapes, bounded user input, or prepared statements.

importance of preventing SQL Injection to StackOverflow users, deploying the classifier to effortlessly run on OCCS, and setting up potential pipelines to periodically retrain the model on new data and instances from StackOverflow. Further on, work could be done to set up an effective deep neural network once more data is gathered. We have also briefly discussed reestablishing Decision Trees or Random Forests as a viable classifier for their ability to provide “justification” of why a post was tagged. As mentioned earlier, various future works could be done to explore the effectiveness of different vectorization techniques to transform the raw code into vectors for the various models.

11 Acknowledgements

I would like to express my gratitude to Dr. Cynthia Taylor for advising this project. Eliana Grossof was essential to the early development and success of this research. I would like to thank Evans Muzulu and Sam Fertig for their incredible work and great time as collaborators. Additionally, help and support from other department members, like Dr. Adam Eck, was much appreciated. Ella Rumsey for reviewing, editing, and supporting me along the way. Finally, I would like to thank my family for their support and encouragement.

References

[1] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack Overflow Considered Harmful? The Impact of Copy Paste on Android Application Security. pages 121–136, 2017.

[2] Aurelien Geron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. O’Reilly Media, 2017.

[3] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.

[4] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[5] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[6] OWASP. Owasp top ten web application security risks. 2020.

[7] Cynthia Taylor and Sahell Sakharkar. ’);drop table textbooks;–: An argument for sql injection coverage in database textbooks. *CIGSCE*, 50:191–197, 2019.