Diploma Thesis

*Predict if the SQL injection query can get the access to the database*

Mayur Kagathara

(mayur.kagathara2508@gmail.com)

Diploma, University of Hyderabad

February 25, 2022

**Supervised by : Applied AI Team**

**Content**

[**1. Problem definition**](#_3rbx0bz1n8zz) **3**

[1.1 Problem Background](#_psyrwv49r07c) 3

[1.2 Literature Survey](#_mj93cwc8ikkm) 4

[1.3 SQL injection (SQLi)](#_pvtp8e6ui9vz) 4

[**2. Data Acquisition**](#_h5h753c7vx2u) **6**

[2.1 Dataset](#_xsn3d5a4uw0h) 6

[2.2 Dataset review](#_ww8h6svjkc67) 6

[2.3 Solution Approach](#_aygolocheuud) 6

[2.3.1 KPI Selection](#_bwetjo8vsvgb) 7

[2.3.2 Possible solutions](#_yegxgkt081g6) 7

[**3. Exploratory Data Analysis (EDA)**](#_lfu8a05oeo1g) **8**

[3.1 Data Cleaning](#_qsoppyc991ji) 8

[3.1.1 Duplicate removal](#_c3co2vh8utr2) 8

[3.1.2 Special characters and extra spaces](#_l3r3v4mb4ehq) 8

[3.1.3 Outlier removal](#_t44jql24b5ib) 8

[3.2 Data Analysis](#_z8b7l8p0i6yf) 9

[3.2.1 Special Characters Count](#_pdva1nraq5n) 9

[3.2.2 Query Length Distribution](#_955l29yv700p) 9

[3.2.3 Feature Engineering](#_lvra6eg5fsqy) 10

[3.3 Conclusion of EDA phase](#_ervn2985fds2) 12

[4.1 Modeling](#_f97ljovedae0) 13

[4.1.1 K-Nearest Neighbors Classifier](#_koyfi78zjadc) 13

[4.1.2 Multinomial Naive Bayes Classifier](#_iwny47xphoi1) 14

[4.1.3 Logistic Regression Classifier](#_137s97upqnc4) 15

[4.1.4 Support Vector Classifier](#_1oz36ddez9bt) 16

[4.1.5 Random Forest Classifier](#_puu21xwiupd7) 17

[4.1.6 Gradient Boosted Decision Trees Classifier](#_j4x6covh5nh) 18

[4.1.7 Multi Layer Perceptron Classifier](#_l0kd2b7s5e7d) 19

[4.2 Error Analysis](#_g7zyjeuqm8w0) 20

[4.2.1 Feature Engineering](#_b2dwi741g19h) 20

[4.2.2 Retraining](#_1rr3092v39jz) 21

[4.2.3 Summary and Comparison](#_tws8ff4i21v) 22

[**5. Advanced Modeling And Interpretability**](#_ys7irvkntu2f) **23**

[5.1 RFC Interpretability](#_s040u98u9v8g) 23

[5.2 LIME](#_s1dw2f7i6shj) 24

[5.3 Advance Modeling](#_p268ibtfkv1l) 26

[5.3.1 Multi Layer Perceptrons (MLP)](#_datn1l6qg1n8) 26

[5.3.2 Recurrent Neural Network (RNN)](#_ef5b0z8gkn1f) 27

[5.3.3 Multi Layer Recurrent Neural Network (Multi RNN)](#_6xqupq7yy57q) 28

[**6. Model Deployment**](#_57qjo7bhhqa8) **31**

[6.1 Streamlit](#_mxl4q3lk1ckk) 31

[6.2 Flask](#_c5vcyec52dpb) 32

[**Appendix:**](#_ojqxdkfqxfsy) **33**

[Streamlit app Screenshot](#_8ev5ta8cro1) 33

[Code for streamlit:](#_vsd3euz2lqgk) 33

[Flask API screenshot:](#_9qtozlgjfdp9) 34

[Code for Flask:](#_tf1qxw31pi6z) 34

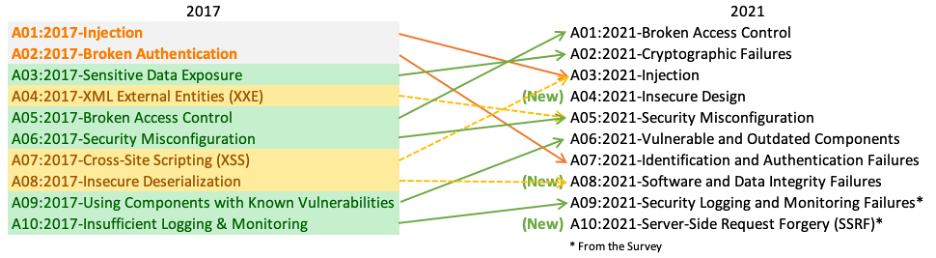
[API reference:](#_wr21eonwf8ba) 35

[**References:**](#_a079s1nyusrl) **37**

# **1. Problem definition**

## **1.1 Problem Background**

In today’s world, everything is going online and becoming digital. With new features and convenience it comes with risk of data and other security concerns. Cyber-crime is also increasing day by day with novel attacks. The threats include attacks such as Cross Site Scripting (XSS), Denial of Service Attack (DoS0, and Structured Query Language (SQL) Injection attacks.



[Credits: <https://owasp.org/www-project-top-ten/> ]

Below are some important points about SQL injection attacks:

1. SQL injection attack was the number one application security risk in 2017 and now it is the third most common and dangerous security risk in 2021. (Ref: <https://owasp.org/www-project-top-ten/>)
2. 94% of the applications were tested for some form of injection.
3. With a max incidence rate of 19%, an average incidence rate of 3%, and 274k attacks occurred.

It is very important to prevent this kind of attack to secure the sensitive data and failing to do so can result in loss of data, trust and potential customers. SQL injection attacks are also commonly used to get unauthorized access, confidential data and break the server/system as well. An application is vulnerable to attack when:

* User-supplied data is not validated, filtered, or sanitized by the application.
* Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
* Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
* Hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures.

## 

## **1.2 Literature Survey**

To understand the problem deeply and get some insights on approaches of the solution of the problem below theses were reviewed,

1. [SQL Injection Detection Using Machine Learning](https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1727&context=etd_projects)
2. [SQL Injection Detection Using Machine Learning Techniques and Multiple Data Sources](https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1649&context=etd_projects)

Some observations from above literature survey,

1. Different datasets and approaches are used in both theses.
2. Good results can be obtained using simple machine learning algorithms like naive bayes as well. We can get better accuracy Gradient boosting compared to Naive Bayes.
3. Artificial Neural Network(ANN) performs nearly the same as Boosting algorithm.
4. ANN can be improved to achieve higher precision and recall.

## **1.3 SQL injection (SQLi)**

SQL injection is broadly classified into three categories:

1. In-band SQLi (Classic)
2. Inferential SQLi (Blind)
3. Out-of-band SQLi



1. **In-band SQLi**:  
   The attacker uses the same channel of communication to launch SQL injection attacks and gather the results. This is the most common type of SQLi attack.
   1. **Error based SQLi**:  
      The attacker injects the query that causes the database to produce the error messages. These error messages can be potentially used to get the information about the database and data structure.
   2. **Union Based SQLi**:  
      The attacker takes advantage of the UNION SQL operator which can be used to run multiple select statements. This way they can get the multiple dataset response in the one HTTP response. This response may contain confidential data as well.
2. **Inferential SQLi (Blind)**  
   The attacker sends the payloads to the server and observes the response and behavior of the server to learn more about the database structure.
   1. **Boolean**:  
      The attacker sends the SQL query to the database prompting the application to return a result. The result depends on whether the query is true or false. Based on the result, the information within the HTTP response will modify or stay unchanged.
   2. **Time-based**:  
      The attacker sends a SQL query to the database which makes the database wait for some periods in seconds before it can react. Based on the response time the attacker can understand if the query is true or false. This way they can work out if the message they used returned true or false.
3. **Out-of-band SQLi**  
   This type of attacks are only possible when certain features on the database server are enabled. When an attacker can’t use the same channel to launch the attack and gather information, Out-of-band SQLi is performed. These techniques count on the capacity of the server to create DNS or HTTP requests to transfer the data to an attacker.

# **2. Data Acquisition**

## **2.1 Dataset**

There are a couple of ways to get the data for this problem.

* Capture the SQL-HTTP requests coming to the server and label them manually. This data gathering can be very slow and expensive as well. This can be very helpful when dealing with the specific kind of application/data.
* Use publicly available dataset consisting of a variety of SQL injection attacks. There are a couple of datasets available for public use and that can be used to train the model as well. This is very easy and inexpensive. But this can result in very bad performance in real time if the application is facing a very different kind of SQL injection missing in the training dataset. This also does not cover the new kind of attack patterns.

For this scope of project we are going to use the below dataset available of kaggle for public use.  
Name : SQL Injection Dataset (by [Sajid576](https://www.kaggle.com/sajid576/sql-injection-dataset))

Link : <https://www.kaggle.com/sajid576/sql-injection-dataset>

## **2.2 Dataset review**

This dataset was originally created by [syedsaqlainhussain](https://www.kaggle.com/syedsaqlainhussain). Then it was cleaned and improved by [Sajid576](https://www.kaggle.com/sajid576/sql-injection-dataset).

**Filename** : Modified\_SQL\_Dataset.csv

**Size** : 2.28 MB

**Columns** : 2

* *Query* (String) : This column contains the actual query from the web traffic.
* *Label* (Integer) : This column represents if the query is malicious or not.
  + 0 : The query is normal
  + 1 : The query is malicious or contains the injection attack.

**Row count** : 30905

**Problem** : Binary Classification

Dataset is small enough in size to process on a single machine. There is no special need with respect to acquisition and processing.

## **2.3 Solution Approach**

As we are dealing with the text data we should be able to apply all the NLP techniques to tokenize and vectorize the data. But in our case we need to consider that we are not having English or any other language sentences. Also the literal meaning of most of the English-words used in the SQL query have different meanings.

Keywords and other letters used in the SQL query have special meaning and function.

### 

### **2.3.1 KPI Selection**

Selection of KPI (Key Performance Indicator) is very critical. We will optimize the KPI to decide the model performance as well as selecting the best model. KPI or business metric defines what matters the most to the business and what has to be given importance over others. Business metric is decided based on the strategic goals business wants to achieve by solving this problem.

SQL injection attacks can highly impact the business and we need to identify the most of the attacks. Also it needs to be done in realtime to filter out the queries from the incoming web traffic. If it takes a long time to classify the request, it hampers the end user experience. Also doing the misclassification of a normal query as malicious can lead to bad end-user experience.

For this problem identifying the most of the malicious attacks with high accuracy in minimum or real time is very important.

Given this constraint, we need to get high precision and recall. So that we make sure that model is able to classify the most of the attacks (recall) at the same time not making type-1 error (precision). Here precision and recall have the same importance. Thus we can use F1-score as a business-matric.

F1-Score

Alternative matric we can use here is accuracy. Accuracy also makes sure that we are having maximum True Positives and True Negatives. But accuracy will fail in case of an imbalanced dataset. Here we have a slightly imbalanced dataset, which makes F1-Score a better option over accuracy.  
 Our dataset has 63% values from class 1 and 37% from class 0.

Accuracy will fail in case the model is getting most of the negative class correct and missing positive class value. For example, making the same percentage of error in both the classes will result in more errors for positive class, because the proportion of positive class is considerably more than negative class. So we need to consider this fact in model selection and optimization.

### **2.3.2 Possible solutions**

**Rule Based filtering:**

This is a naive and first cut solution which works well for well-known attacks which occur in the same fashion. If some kind of attack query has specific structure we can filter those queries from the traffic. This solution only works for very specific types of queries and does not generalize well.

**Dictionary Based Filtering:**

This is very similar to the first solution. But here we keep track of historical attacks data and keep on updating the dictionary. This type of solution works very well to identify the similar kind of attacks which are already there in the dictionary. But it fails if the query is little tweaked and does not match with the historical query. This also does not generalize well on future data and is not adaptable to small changes in the attack query.

**Machine Learning Based Filtering:**

In this approach we train a machine learning model (or Neural network model) to learn the pattern of the attack from the train data. For this approach to work, we need a lot of labeled data. But once we have enough data for the training, we can pose a problem as binary classification to train machine learning model. This model can be used to identify the SQL injection attack from the future traffic. Tweaks in the SQLi can be handled by the model easily.

# **3. Exploratory Data Analysis (EDA)**

EDA is very crucial and the most important phase in the whole machine learning lifecycle. It helps to identify patterns in the data and understand the data better. Understanding of the data helps to build better machine learning models and most importantly to debug the error and model.

We will divide the EDA process into two parts:

1. Data cleaning
2. Data Analysis
3. Feature Engineering

## **3.1 Data Cleaning**

We only have one feature column named Query in the dataset. We will check for duplicates, special characters, extra spaces, outliers in the data cleaning process.

### **3.1.1 Duplicate removal**

We checked for duplicates in the raw data. And we found below duplicate queries in the raw data.

1. #NAME?
2. 1.86E+15
3. 1.94E+15
4. 26%
5. 28%
6. 29%
7. 7.75E+15

These duplicate values look like numbers and not like SQL queries. There are only 20 duplicate values out of **30919** Queries. So we can safely remove these queries from the dataset.

**Code:**

raw\_dataset = pd.read\_csv("Modified\_SQL\_Dataset.csv")

raw\_dataset[raw\_dataset.duplicated(keep=False)].sort\_values(by=['Query'])

raw\_dataset.drop\_duplicates(keep=False, inplace=True)

### **3.1.2 Special characters and extra spaces**

We can observe multiple special characters and spaces in the queries. We can try to train a machine learning model by removing the special characters. If this does not work, we can include special characters afterwards.

Special characters might have some meaning in the SQL query, so it is good to create two datasets one with and another without special characters.

Though we have text data but all SQL queries, so there is no need to remove stop words. We cleaned the data and took the dataset with all special characters and extra spaces removed. Then we again dropped the duplicate observations. At last we were left with **28140** data points.

### **3.1.3 Outlier removal**

As a part of data analysis and finding outliers, we broke the query into words and took the number of words as query length. From the percentile values and histogram analysis, we were able to find two outliers from the data.

10th percentile for query length = 1.0

50th percentile for query length = 7.0

90th percentile for query length = 19.0

95th percentile for query length = 26.0

99.9th percentile for query length = 50.0

100.0th percentile for query length = 542.0

We removed the queries with length equal to zero and more than 500.

## **3.2 Data Analysis**

### **3.2.1 Special Characters Count**

As discussed in the data cleaning phase, we will remove all the special characters and spaces from the query and will include them in case of failure. So as a part of analysis, we will do analysis of the number of special characters present in the query and its distribution for positive and negative classes.

For the raw dataset without outlier removal we get distribution of count of special characters as below,

No\_of\_special\_chars = raw\_dataset['Query'].map(lambda x: \

len( re.findall( '[^a-zA-Z0-9\s]' ,x)))

fig3, ax\_sc = plt.subplots(1, 1, figsize=(15,8))

sns.histplot(no\_of\_special\_chars[raw\_dataset['Label']==1],\

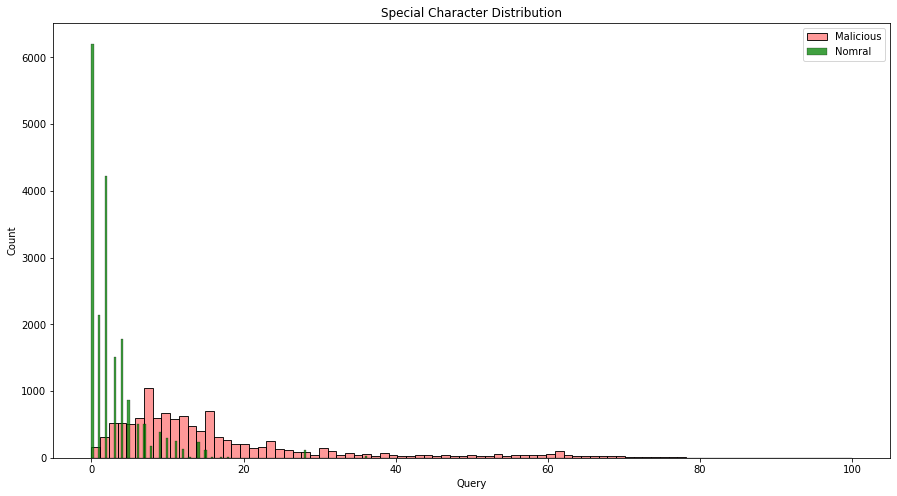
ax=ax\_sc, color='red', label='Malicious', binrange=(0,100) ,alpha=0.4)

sns.histplot(no\_of\_special\_chars[raw\_dataset['Label']==0], \

ax=ax\_sc, color='green', label='Nomral', binrange=(0,100))

ax\_sc.set\_title("Special Character Distribution")

ax\_sc.legend()



Here we can see, Number of special characters in a query has an almost same distribution for normal and malicious queries. Hence, this is not a good feature to make classification. But in the modeling phase, we can try this feature to improve the score if other features are not performing well.

### **3.2.2 Query Length Distribution**

We define query length as the number of words in the query after cleaning. As discussed in the topic 3.1.3 Outlier Removal we were able to identify outliers using query length. Here we will try to check if there is any major difference in query length of positive and negative classes. We plot the query length distribution for normal and malicious queries in the cleaned dataset,

normal\_queries = dataset\_all\_cleaned[dataset\_all\_cleaned['Label']==0]

malicious\_queries = dataset\_all\_cleaned[dataset\_all\_cleaned['Label']==1]

normal\_queries\_length = normal\_queries.Query.map(lambda x:len(x.split()))

malicious\_queries\_length = malicious\_queries.Query.map(lambda x:len(x.split()))

fig2, ((ax21, ax22)) = plt.subplots(1, 2, figsize=(15,6))

sns.histplot(normal\_queries\_length,binrange=(0,50),\

ax=ax21, color='green', label='Nomral')

sns.histplot(normal\_queries\_length,binrange=(50,150),\

ax=ax22, color='green', label='Nomral')

sns.histplot(malicious\_queries\_length,binrange=(0,50),\

ax=ax21, color='red', label='Malicious', alpha=0.4)

sns.histplot(malicious\_queries\_length,binrange=(50,150),\

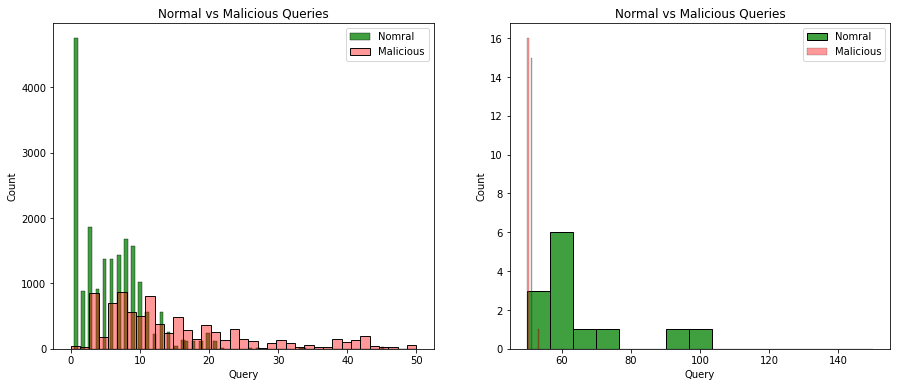
ax=ax22, color='red', label='Malicious', alpha=0.4)

ax21.set\_title("Normal vs Malicious Queries")

ax22.set\_title("Normal vs Malicious Queries")

ax21.legend()

ax22.legend();



Query length for both classes have almost the same distribution. So it won't be a very useful feature in classifying Labels. This feature can be used in the modeling phase if tokenization features are not working well.

### **3.2.3 Feature Engineering**

We have a query in text format. We can convert the query into vectors using simple count vectorizer, TFIDF vectorizer and using deep learning methods like word2vec as well. For the scope of this phase we will try count and TFIDF vectorizer to convert the query into a high dimensional array.

Once we have vectorized the train data, we will train the Random Forest Classifier model to get the top most features using feature importance score. And this will also work as a baseline model for the next modeling phase, where we will explore other machine learning and deep learning techniques.

* **Vectorization**:
  + Count vectorizer:   
    We construct the bag of words and then the count of the word in the query and encode the vector using the count as the appearance of the word in the query.  
    from sklearn.feature\_extraction.text import CountVectorizer

vectorizer = CountVectorizer(ngram\_range=(1,3), max\_features=2500)

X\_train\_vec = vectorizer.fit\_transform(X\_train)

X\_test\_vec = vectorizer.transform(X\_test)

* + TFIDF vectorizer:  
    We calculate Term Frequency(TF) and Inverse Document Frequency(IDF) of each word in the query and calculate TFIDF value of the word. We use this value to vectorize queries.  
    For both vectorizers we have taken the top 2500 features (words) from unigrams, bigrams and trigrams. We can increase the ngram range if it fails in a later phase.  
    from sklearn.feature\_extraction.text import TfidfVectorizer

tfidf\_vectorizer = TfidfVectorizer(ngram\_range=(1,3), max\_features=2500)

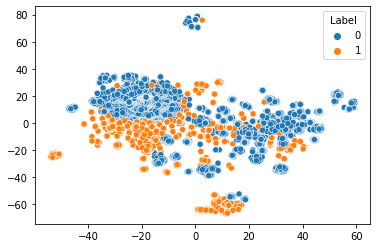
X\_train\_tfidfvec = tfidf\_vectorizer.fit\_transform(X\_train)

X\_test\_tfidfvec = tfidf\_vectorizer.transform(X\_test)

* **T-SNE analysis**:  
  To visualize the data from 2500 dimensions space we can use T-Distributed Stochastic Neighbor Embedding (T-SNE). It converts the higher dimension data into 2D space which can be visualized using scatter plots very easily.  
  After trying various values of perplexity and iterations we were able to achieve below visualization,

from sklearn.manifold import TSNE

embedded\_data=TSNE(perplexity=400, n\_iter=3000).fit\_transform(X\_train\_vec, y\_train)

sns.scatterplot(x=embedded\_data[:,0],y=embedded\_data[:,1],hue=y\_train, legend='full')  


From above visualization and other T-SNE output of different perplexity, we can say that both classes are making good clusters and should be separable using non-linear machine learning models with high accuracy.

* **Random Forest Features**:  
  Decision tree based method Random Forest Classifier can be used to learn the feature importance automatically based on information gain. For the counter vectorizer and TFIDF vectorizer we got almost the same f1-score, accuracy and confusion matrix. Below are the top 50 features for TFIDF based random forest classifier model,

RFC\_model\_tfidf = RandomForestClassifier()

RFC\_model\_tfidf.fit(X\_train\_tfidfvec, y\_train)

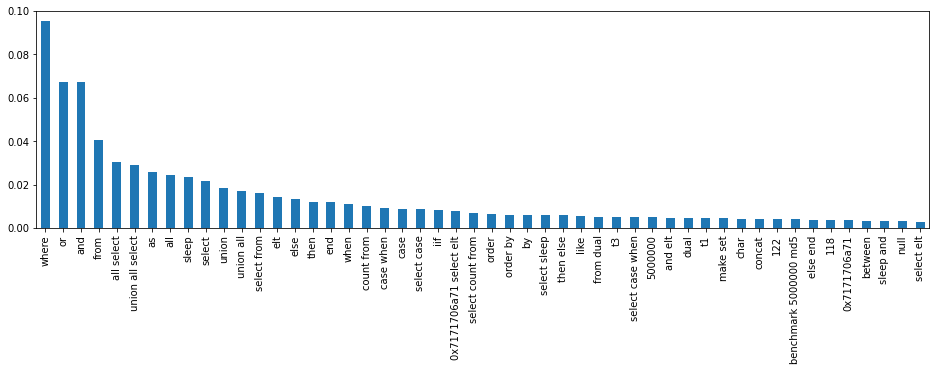
importances = RFC\_model\_tfidf.feature\_importances\_

forest\_importances = pd.Series(importances, \

index=tfidf\_vectorizer.get\_feature\_names())

forest\_importances.sort\_values(inplace=True, ascending=False)

forest\_importances[:50].plot(kind='bar', figsize=(16,4))



* **Special Characters Analysis**:  
  Before moving to the next phase, we want to make sure of the importance of special characters in the SQL queries with respect to classifying the labels. We trained the same random forest classifier on the dataset containing special characters. We also tried the Grid-Search cross validation to tune the hyper parameters of the Random forest model. But in the result we got a little less f1-score, accuracy than the model trained without special characters.

## **3.3 Conclusion of EDA phase**

1. Data is clean and requires minimal data cleaning.
2. Count vectorizer and TFIDF are performing exactly the same.
3. Only BoW and TFIDF based features are adequate to achieve a good score in the baseline model.
4. Default Random forest classifier is able to achieve a good score using only count based features.
5. Special characters in the query are not very important features to predict dependent variables.
6. Dataset is very prone to overfitting and requires attention in the next phase.
7. There is no need to use complex models like Word2Vec or BERT to encode the queries. Simple BoW and TFIDF works really well in the baseline model.
8. Statistics of baseline model (Random Forest Classifier):

| **Data** | **Accuracy** | **f1-score** | **TP** | **TN** | **FP** | **FN** |
| --- | --- | --- | --- | --- | --- | --- |
| BoW without special characters | 99.62 | 99.38 | 2570 | 5840 | 2 | 30 |
| TF-IDF without special characters | 99.62 | 99.38 | 2571 | 5839 | 3 | 29 |
| TF-IDF with special characters | 99.49 | 99.31 | 3363 | 5834 | 3 | 44 |

**Note:** Deep learning based features and methods will be experimented in the next phase which is modeling and error analysis phase. We are getting superb scores in the baseline model. So it doesn’t make sense to use complex models as we do not have much room for improvements.

Please refer to [EDA\_Notebook](https://github.com/mayurkagathara/sqli_detection/blob/master/Notebook/Phase_2_EDA.ipynb) for code snippets mentioned in topic 2 and 3. (<https://github.com/mayurkagathara/sqli_detection/blob/master/Notebook/Phase_2_EDA.ipynb>)

**4. Modeling and Error Analysis**

In the modeling phase we will train different machine learning models starting from less complex models. We will also tune the model using Grid Search Cross Validation or Random Search Cross Validation. We will gradually increase the complexity of the model to achieve higher performance. Once we are done with the modeling part, we will choose the best model with respect to performance and time complexity.

After the best model is selected we will do error analysis. In the error analysis, we will try to find the particular pattern if present which is causing the model to make a mistake in classification. Based on the error analysis we will add some more features into the best fitted model and retrain it.

## **4.1 Modeling**

We will use a dataset without special characters as it was working best with Random Forest Classifier in the EDA phase. We will start with simple models like KNN and will train complex Multi Layer Perceptron Classifiers at last. We have 2500 TF-IDF based features. We will train and tune below models,

1. K-Nearest Neighbors Classifier
2. Multinomial Naive Bayes Classifier
3. Logistic Regression Classifier
4. Support Vector Classifier
5. Random Forest Classifier
6. Gradient Boosted Decision Trees Classifier
7. Multi Layer Perceptron Classifier

### **4.1.1 K-Nearest Neighbors Classifier**

K-Nearest Neighbors Classifier (KNN Classifier) works by classifying points based on its K Nearest Neighbors. Here we have only 1 hyperparameter - the value of K in KNN. To decide on K nearest neighbors it uses distance metrics like euclidean, minkowski etc.

As observed in the TSNE result, there are clear groups of points for both the classes and are not jumbled. KNN-Classifier should be able to perform well in this case. Due to high dimensionality, it is taking more time in the training and testing phase.

We are able to get more than 99% accuracy in the train and test dataset with default value of k=5 and minkowski distance metric. By doing a grid search with cross validation for the value of K we can see that there is very little difference in f1-score for k=5 and k=7. Higher value of K is better to avoid over-fitting.

knn\_model\_gcv = KNeighborsClassifier()

knn\_params = {'n\_neighbors':(1,3,5,7,9,11,13,19), 'metric':('euclidean','minkowski')}

knn\_clf\_gcv = GridSearchCV(estimator = knn\_model\_gcv, param\_grid = knn\_params, n\_jobs=-1, scoring='f1')

knn\_clf\_gcv.fit(X\_train\_tfidfvec, y\_train)

cv\_results = pd.DataFrame(knn\_clf\_gcv.cv\_results\_)

sns.set\_style("dark")

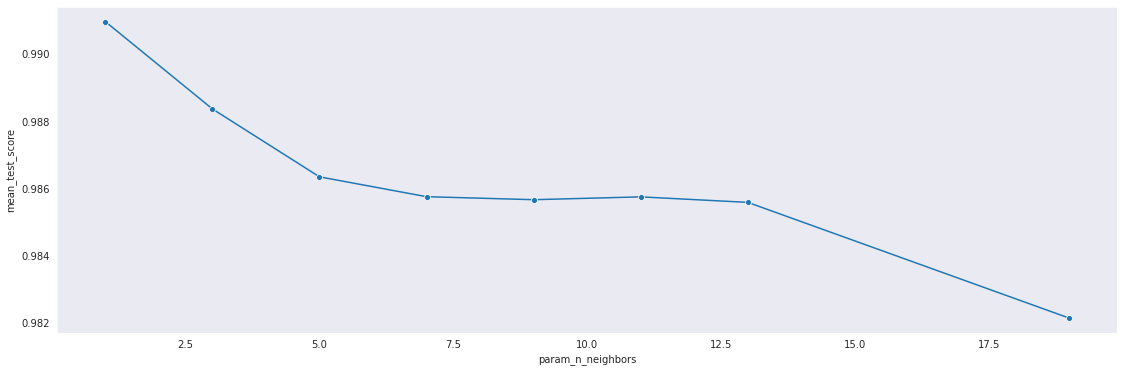
fig = plt.figure(figsize=(19,6))

sns.lineplot(data = cv\_results[cv\_results[\

'param\_metric']=='minkowski'], \

x='param\_n\_neighbors', \

y='mean\_test\_score', marker="o")



We are getting the same performance by using euclidean and minkowski distance metric, Because default value of power parameter for minkowski distance (p) is 2 which works the same as euclidean distance.

We trained KNN-Classifier again with K=7 and got good performance with below results,

| Training Accuracy | 99.2283 % |
| --- | --- |
| Testing Accuracy | 99.4788 % |
| F1-Score | 99.1479 % |
| False Positive Rate | 0.0685 % (FP = 40) |
| False Negative Rate | 1.5385 % (FN = 4) |
| ROC-AUC | 99.1965 % |

### **4.1.2 Multinomial Naive Bayes Classifier**

Naive Bayes works by applying Bayes’ theorem with the “naive” assumption of conditional independence between every pair of features given the value of class variable. Naive bayes does not consider the interaction of the features and pattern in the group of words. It takes the counting or probability of features with respect to dependent variables. Due to high dimensionality it takes a long time during the training and testing phase, as it has to calculate conditional probability for each feature.

We have only one hyperparameter α (alpha) which is the smoothing parameter in Multinomial Naive Bayes (MNB) likelihood calculation. If α ≥ 0, it prevents zero probabilities for features not present in the learning samples. If α = 1, it is called Laplace Smoothing, while α < 1 is called Lidstone smoothing.

mnb\_model\_gcv = MultinomialNB()

mnb\_params = {'alpha':(0, 0.1, 0.5, 1, 10, 100, 500)}

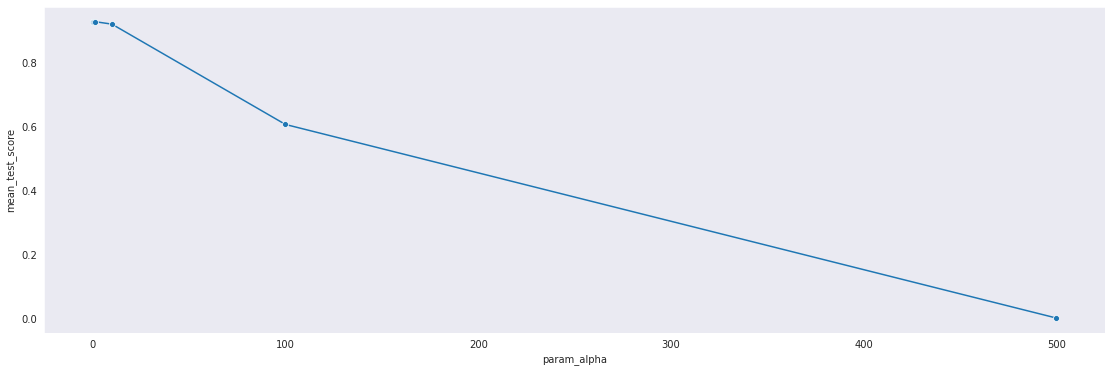
mnb\_clf\_gcv = GridSearchCV(estimator = mnb\_model\_gcv, param\_grid = mnb\_params, n\_jobs=-1, scoring='f1')

mnb\_clf\_gcv.fit(X\_train\_tfidfvec, y\_train)

cv\_results = pd.DataFrame(mnb\_clf\_gcv.cv\_results\_)

fig = plt.figure(figsize=(19,6))

sns.lineplot(data = cv\_results, x='param\_alpha', y='mean\_test\_score', marker="o")



By default the value of α is 1 in sklearn Multinomial Naive Bayes(MNB) classifier. Even After doing Grid Search for the best value of α, we found α = 1 works best for the MNB classifier. We trained MNB classifier with α = 1 and got average performance compare to KNN-model with below results,

| Training Accuracy | 95.8824 % |
| --- | --- |
| Testing Accuracy | 95.5105 % |
| F1-Score | 92.3512 % |
| False Positive Rate | 1.1469 % (FP = 67) |
| False Negative Rate | 12.0000 % (FN = 312) |
| ROC-AUC | 93.4266 % |

### **4.1.3 Logistic Regression Classifier**

Logistic regression is a linear model for classification. It works best only when data is linearly separable. It is also noticed that Logistic Regression (LR) works better in higher dimension space, because it is easy to find the hyperplane in the higher dimension which can separate the positive and negative class. It tries to minimize the logistic loss and can be trained using Stochastic Gradient Descent by taking log-loss.

There are mainly two hyper parameters penalty and regularization parameter C.

For the LR classifier, we can set the penalty to three values - ‘l1’, ‘l2’ and ‘elastic net’. The value of C is the inverse of regularization strength. Value of C is inversely proportional to regularization.

In the sklearn library we have different options for solvers as well.

lr\_model\_gcv = LogisticRegression()

lr\_params = {'C':(0.1, 0.5, 1.0, 5.0, 10.0), 'penalty':('l1', 'l2', 'elasticnet'),

'solver':('lbfgs', 'saga'), 'max\_iter':[1000]}

lr\_clf\_gcv = GridSearchCV(estimator = lr\_model\_gcv, param\_grid = lr\_params,

n\_jobs=-1, scoring='f1')

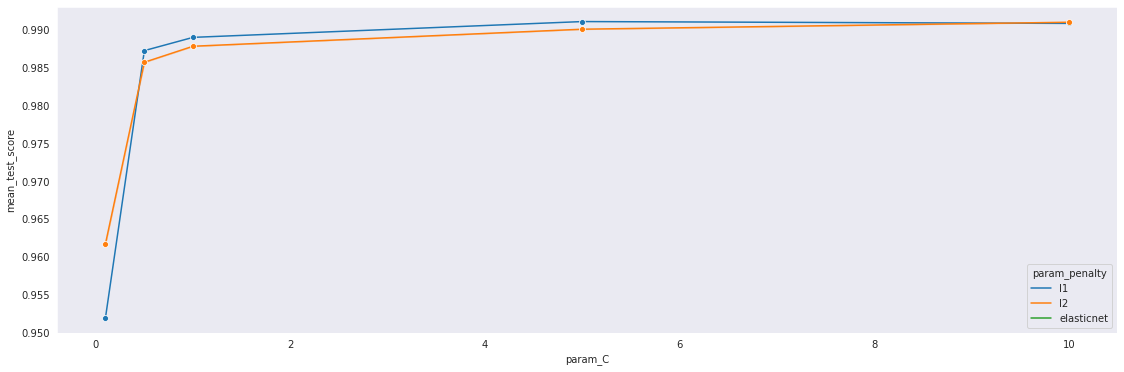
lr\_clf\_gcv.fit(X\_train\_tfidfvec, y\_train)

cv\_results = pd.DataFrame(lr\_clf\_gcv.cv\_results\_)

figure = plt.figure(figsize=(19,6))

sns.lineplot(data = cv\_results, x='param\_C', y='mean\_test\_score',

hue='param\_penalty' ,marker="o")



By doing hyperparameter tuning, we found that for solver ‘saga’, LR model works best with C = 5.0 with L1 penalty. By doing hyperparameter tuning we were able to improve the performance and the best tuned model gave below results,

| Training Accuracy | 99.7360 % |
| --- | --- |
| Testing Accuracy | 99.5854 % |
| F1-Score | 99.3237 % |
| False Positive Rate | 0.0856 % (FP = 5) |
| False Negative Rate | 1.1538 % (FN = 30) |
| ROC-AUC | 99.3803 % |

### 

### **4.1.4 Support Vector Classifier**

Support Vector Machine (SVM) works by minimizing hinge loss and is a binary classifier. There are two versions of Support Vector Classifier(SVC) -Primal and Dual. Primal form of SVC can be used to classify the binary data which is linearly separable. Dual form of SVC uses the kernel like Radial Basis Function(RBF), Polynomial to achieve the non-linear surface to separate the data which is not linearly separable and works well at higher dimensions. Only problem with SVC is its higher training and test time complexity. We also observed that SVC is giving best results but also takes longer time to execute. There are two hyper parameters in SVC, which kernel to use and value of regularization parameter C. The value of C is the inverse of regularization strength. Value of C is inversely proportional to regularization.

svc\_model\_gcv = SVC()

svc\_params = {'kernel':('linear', 'rbf'), 'C':(0.1, 0.5, 1.0, 5.0, 10.0)}

svc\_clf\_gcv = GridSearchCV(estimator = svc\_model\_gcv,

param\_grid = svc\_params, n\_jobs=-1, scoring='f1')

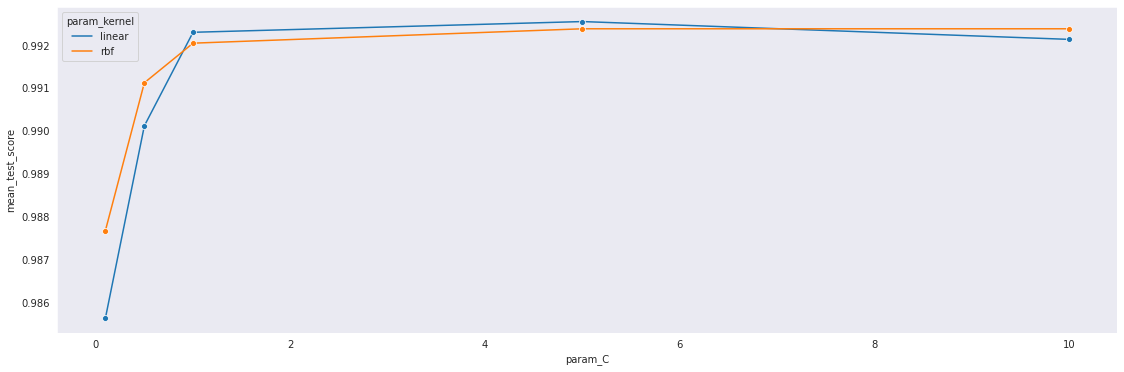
svc\_clf\_gcv.fit(X\_train\_tfidfvec, y\_train)

cv\_results = pd.DataFrame(svc\_clf\_gcv.cv\_results\_)

figure = plt.figure(figsize=(19,6))

sns.lineplot(data = cv\_results, x='param\_C',

y='mean\_test\_score', hue='param\_kernel' ,marker="o")



We found that SVC with C=5.0 for linear kernel is working best followed by RBF kernel. Below is the performance results for the best model,

| Training Accuracy | 99.7106 % |
| --- | --- |
| Testing Accuracy | 99.6209 % |
| F1-Score | 99.3815 % |
| False Positive Rate | 0.0514 % (FP = 3) |
| False Negative Rate | 1.1154 % (FN = 29) |
| ROC-AUC | 99.4166 % |

### **4.1.5 Random Forest Classifier**

Tree based classifiers are best fit when we have so many features since we are working with subsets of data. It is faster to train the decision trees because we are working only on a subset of features in the data at a time, so we can easily work with hundreds and thousands of features. There are so many hyperparameters for Random Forest Classifiers (RFC) but we only care about the number of estimators and class\_weight. Number of estimators directly impacts the performance and fitting of RFC.

By doing hyperparameter tuning, We can see that n\_estimators=50 and class\_weight = 'balanced' works best. We should also consider n\_estimators=200 and class\_weight='balanced\_subsample', because it is having a smaller standard deviation in score. RFC takes less training and testing time than SVC and achieves the same F1-score and performance.

RFC\_model\_gcv = RandomForestClassifier()

RFC\_params = {'n\_estimators':(10,50,100,200,300),\

'class\_weight':(None, 'balanced', 'balanced\_subsample')}

RFC\_clf\_gcv = GridSearchCV(estimator = RFC\_model\_gcv,\

param\_grid = RFC\_params, n\_jobs=-1, scoring='f1')

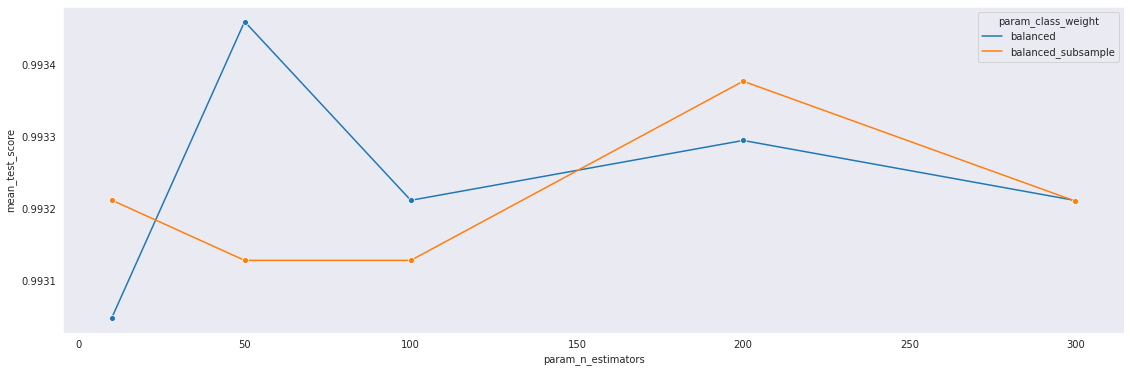
RFC\_clf\_gcv.fit(X\_train\_tfidfvec, y\_train)

cv\_results = pd.DataFrame(RFC\_clf\_gcv.cv\_results\_)

fig = plt.figure(figsize=(19,6))

sns.lineplot(data = cv\_results, x='param\_n\_estimators', y='mean\_test\_score',\

hue='param\_class\_weight' ,marker="o")



Best RFC model with n\_estimators = 50 and ‘balanced’ class\_weight gives below result,

| Training Accuracy | 99.7715 % |
| --- | --- |
| Testing Accuracy | 99.6209 % |
| F1-Score | 99.3815 % |
| False Positive Rate | 0.0514 % (FP = 3) |
| False Negative Rate | 1.1154 % (FN = 29) |
| ROC-AUC | 99.4166 % |

### **4.1.6 Gradient Boosted Decision Trees Classifier**

Gradient Boosted Decision Trees classifier works by building an additive model in a forward stage-wise fashion. It trains weak learners(estimators) to reduce the error at every stage. GBDT learners can not be trained in parallel so it takes much time compared to Random Forest Classifier.

There are mainly two hyperparameters- learning rate and n\_estimators. By doing hyperparameter tuning we can see that there is a very small difference in mean test score for learning rate 0.5 and 0.1. It is better to choose a smaller learning rate since it is reliable and also the standard deviation score is also less than the higher learning rate.

GBDT\_model\_gcv = GradientBoostingClassifier()

GBDT\_params = {'n\_estimators':(50,100,200,300), 'learning\_rate':(0.01, 0.1, 0.5, 1.0)}

GBDT\_clf\_gcv = GridSearchCV(estimator = GBDT\_model\_gcv,\

param\_grid = GBDT\_params, n\_jobs=-1, scoring='f1')

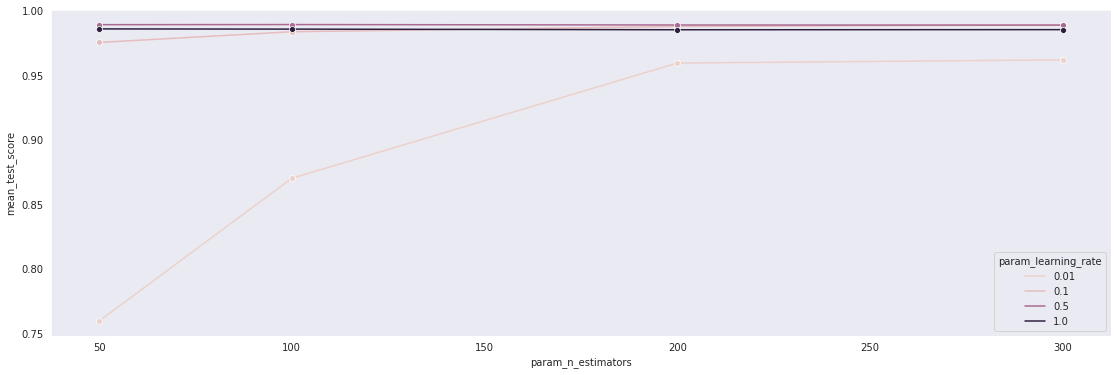
GBDT\_clf\_gcv.fit(X\_train\_tfidfvec, y\_train)

cv\_results = pd.DataFrame(GBDT\_clf\_gcv.cv\_results\_)

fig = plt.figure(figsize=(19,6))

sns.lineplot(data = cv\_results, x='param\_n\_estimators',\

y='mean\_test\_score', hue='param\_learning\_rate' ,marker="o")



For learning rate 0.5 we are able to get almost the same test\_score with respect to learning rate 0.1 and n\_estimators=300. As we increase the number of estimators(learners) from 300 to 500, performance is getting better but the training time is also getting longer.

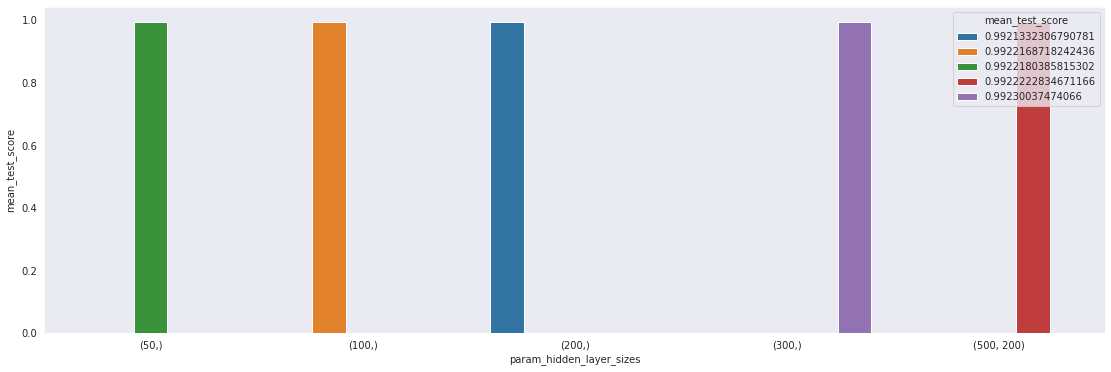
We got below result for n\_estimators = 500 and learning rate = 0.1,

| Training Accuracy | 99.7665 % |
| --- | --- |
| Testing Accuracy | 99.5617 % |
| F1-Score | 99.2845 % |
| False Positive Rate | 0.0685 % (FP = 4) |
| False Negative Rate | 1.2692 % (FN = 33) |
| ROC-AUC | 99.3311 % |

### **4.1.7 Multi Layer Perceptron Classifier**

As part of the experiment, we also tried a Multi Layer Perceptron(MLP) classifier. MLP classifier uses backpropagation to minimize the cross entropy loss function. It is really useful to fit a non-linear model to the training data. There are so many hyper parameters for MLP-Classifier like learning rate, solver, activation function, hidden layer size, alpha (regularization term), etc. It takes a very long time for training, so we have kept all parameters as best constant value and tuned only hidden layer size.

We used Relu as activation with Adam solver, learning rate=0.001 and alpha=0.0001.



It took a very long time in training and grid search cross validation.

From hyperparameter tuning, we found one hidden layer with 300 nodes works best and gives below results,

| Training Accuracy | 99.7665 % |
| --- | --- |
| Testing Accuracy | 99.6091 % |
| F1-Score | 99.3623 % |
| False Positive Rate | 0.0685 % (FP = 4) |
| False Negative Rate | 1.1154 % (FN = 29) |
| ROC-AUC | 99.4081 % |

Taking time complexity and score in consideration, we can take Random Forest Classifier as the best final model and proceed with error analysis. Support vector classifier and MLP classifier are able to get the same score as the RFC model, but it takes a considerable amount of time compared to RFC. Also RFC is a simpler and more interpretable model than SVC and MLP.

Taking all above points into consideration, we will do error analysis on the RFC model with n\_estimators = 50 and class\_weight = ‘balanced’.

## **4.2 Error Analysis**

Out of **8442** test data points, our best RFC model is making a total of **32 mistakes** which is less than **0.4%**. In our case it is possible to examine the queries where the model is making mistakes.

Our primary goal is to reduce the false negatives to improve the recall. Because it is more important to get the high recall than precision. It is okay to make type-1 errors rather than type-2 in our case.

By looking at the queries, we were able to find that some trivial pattern like 1=1 was not identified by model to classify that query as malicious. Most of the queries have only one word. Some queries look normal but used to access database level information. It is very hard to identify such a query as it does not contain any malicious pattern and keywords like sleep, union, some number = some number.

So we decided to introduce some extra features along with TF-IDF features to improve the recall value and overall F1-score.

### **4.2.1 Feature Engineering**

Next, we will try to use some extra features with our best model to see if it helps to improve the error further.

1. **Special Characters Count** : Number of special characters in the raw query.  
   Number of special characters has slightly different distribution in both the classes and might help to improve the model performance.
2. **Query Length** : Number of words in the query after removing special characters.  
   Query length has almost the same distribution for malicious and normal queries. But in some ranges like more than 20, distribution is very different for both the classes. This can be a very good feature to improve the model performance.
3. **Count of *‘number=number*’ pattern** : Number of times some digit = digit pattern occurs in the query. eg 15 = 5, 1=1 etc.  
   It is very common for attackers to use this kind of pattern along with other conditions in the where clause. Also there are some errors in the best model where this pattern is missed. We can either use boolean value for existence of this pattern or take count as well for this pattern occurrence. This can also help to mitigate the trivial errors and improve the best model performance further.

### **4.2.2 Retraining**

We retrained the best RFC model with above three features and TF IDF features.

final\_RFC\_model\_FE = RandomForestClassifier(n\_estimators=50, class\_weight='balanced',

random\_state=42)

# X\_train\_tfidf\_FE contains 2500 TFIDF features and 3 engineered features

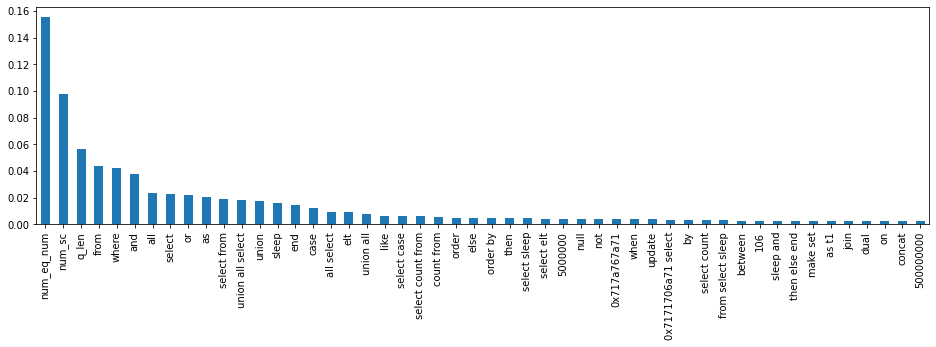
final\_RFC\_model\_FE.fit(X\_train\_tfidf\_FE, y\_train)

importances = final\_RFC\_model\_FE.feature\_importances\_

forest\_importances = pd.Series(importances, index=X\_train\_tfidf\_FE.columns)

forest\_importances.sort\_values(inplace=True, ascending=False)

forest\_importances[:50].plot(kind='bar', figsize=(16,4))



We used the feature importance score available in Random Forest Classifier based on the average information gain in all the base learners (estimators) to check the importance score of the new three engineered features.

From the above feature importance graph we can see that newly added features have high importance and are top 3 features. This helped to achieve the improvisation in model performance in F1-score and recall value. Below is the performance of RFC model with engineered feature,

| Training Accuracy | 99.9035 % |
| --- | --- |
| Testing Accuracy | 99.7631 % |
| F1-Score | 99.6142 % |
| False Positive Rate | 0.0342 % (FP = 2) |
| False Negative Rate | 0.6923 % (FN = 18) |
| ROC-AUC | 99.6367 % |

### **4.2.3 Summary and Comparison**

| **Model** | **Accuracy** | **F1-Score** | **TP** | **TN** | **FP** | **FN** | **FPR** | **FNR** | **Comments** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **KNN** | 99.4788% | 99.1479% | 2560 | 5838 | 4 | 40 | 0.0685% | 1.5385% | Takes longer time |
| **MNB** | 95.5105% | 92.3512% | 2288 | 5775 | 67 | 312 | 1.1469% | 12.000% | Worst than KNN |
| **LR** | 99.5854% | 99.3237% | 2570 | 5837 | 5 | 30 | 0.0856% | 1.1538% | Better and faster than KNN |
| **SVC** | 99.6209% | 99.3815% | 2571 | 5839 | 3 | 29 | 0.0514% | 1.1154% | Takes very long time |
| **RFC** | 99.6209% | 99.3815% | 2571 | 5839 | 3 | 29 | 0.0514% | 1.1154% | Takes less time, best results. |
| **GBDT** | 99.5617% | 99.2845% | 2567 | 5838 | 4 | 33 | 0.0685% | 1.2692% | Takes very long time |
| **MLP** | 99.6091% | 99.3623% | 2571 | 5838 | 4 | 29 | 0.0685% | 1.1154% | Takes the longest time. |
| **RFC(FE)** | 99.7631% | 99.6142% | 2582 | 5840 | 2 | 18 | 0.0342% | 0.6923% | Better with engineered features |
| **SVC(FE)** | 99.6328% | 99.4010% | 2572 | 5839 | 3 | 28 | 0.0514% | 1.0769% | Better than SVC but very slow |

Note: We have taken the test accuracy and f1-score in the above tables.

Please refer to [Modeling\_Error\_Analysis.ipynb](https://github.com/mayurkagathara/sqli_detection/blob/master/Notebook/Phase_3_Modeling_Error_Analysis.ipynb) for code snippets mentioned in topic 4. (<https://github.com/mayurkagathara/sqli_detection/blob/master/Notebook/Phase_3_Modeling_Error_Analysis.ipynb>)

# 

# **5. Advanced Modeling And Interpretability**

In the advanced modeling phase we will train different advanced machine learning models like multi level perceptron, Recurrent neural network and deep recurrent neural network. In the first part of this phase, we will try to interpret the best RFC model with engineered features using LIME analysis.

## **5.1 RFC Interpretability**

Our best model is Random Forest Classifier with engineered features. Implicitly Random forest is interpretable using gini score and information gain score. At each node we can calculate probability using a number of datapoints.

Random forest classifier has multiple estimators which are fitted on the various parts of the data. We can take any one of the estimators and try to print how it is making the prediction using the tree.

At each level of the tree we have an if else like condition on one of the features. This feature is chosen based on information gain which is the difference of entropy or gini impurity.

We will use the plot\_tree function from sklearn to print the tree.

fig = plt.figure(figsize=(15, 10))

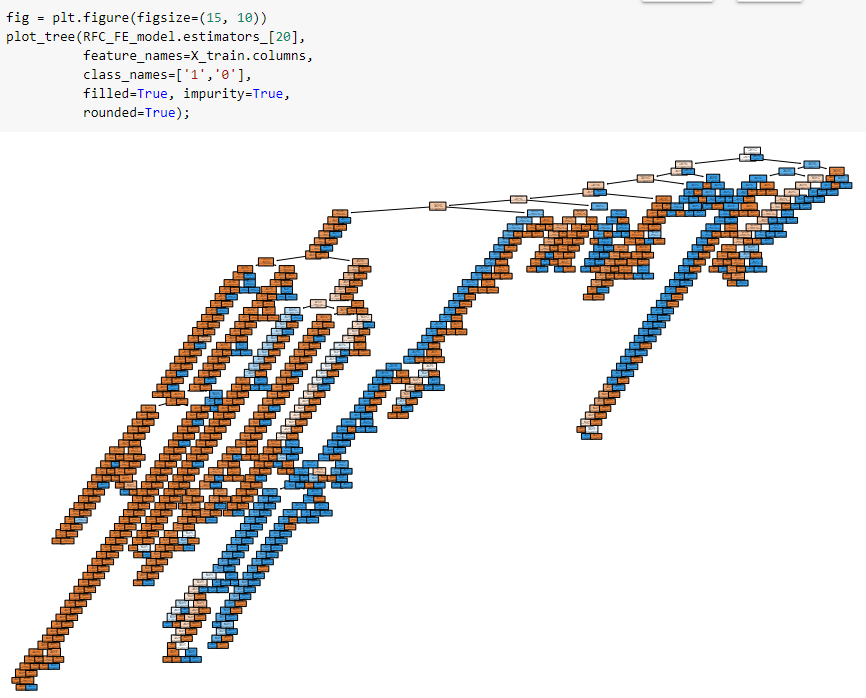
plot\_tree(RFC\_FE\_model.estimators\_[20],

feature\_names=X\_train.columns,

class\_names=['1','0'],

filled=True, impurity=True,

rounded=True);



Here we can see that the tree is very deep and there are so many features (total 2503 features with 2500 TFIDF features and 3 engineered features). Tree is not easily interpretable due to high dimensionality and is not very useful for output interpretability.

## **5.2 LIME**

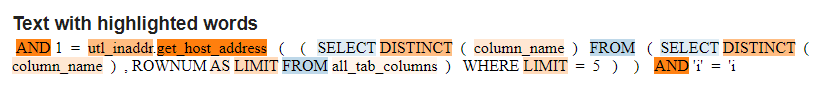
LIME stands for **L**ocal **I**nterpretable **M**odel-agnostic **E**xplanations. Lime is a technique used in machine learning that approximates any black box machine learning model with a local, interpretable model to explain each individual prediction. It works on following idea,

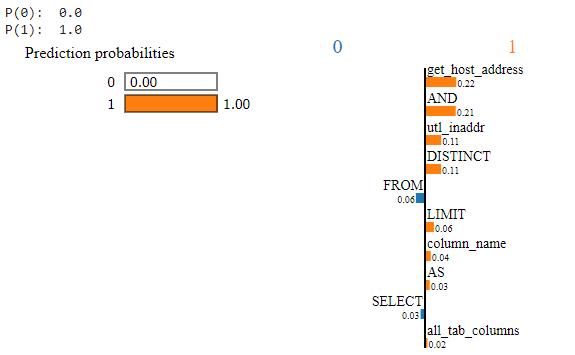
1. Select the query datapoint for which we need explanation.
2. Find the neighbors using similarity scores like cosine similarities, euclidean distance, manhattan distance or any other similarity metrics.
3. Train an interpretable model on this new dataset with the variations.
4. Surrogate model is explainable and less complex (which is made sure using optimization formula), so we can create an explanation for the query point.

**Note** : Please refer to i-python notebook for lime code.

Here are some examples of true classes.

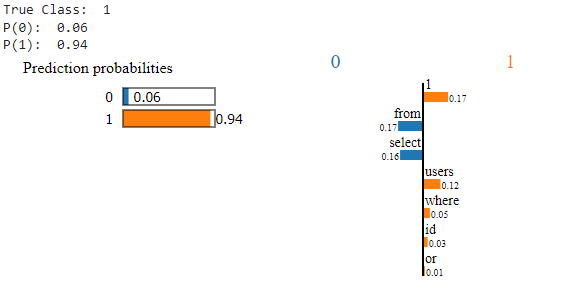
1. Example 1



  
In the above graph value in orange represents features score for class 1 and blue for class 0.

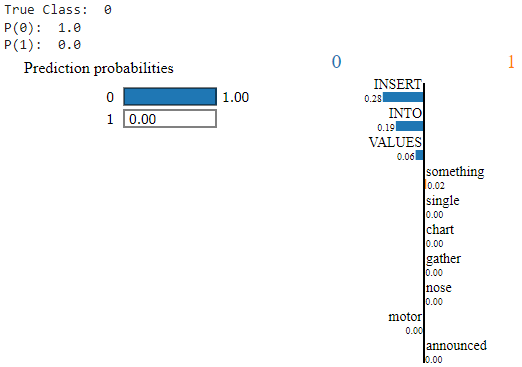
Here in this example we can see that get\_host\_address has the highest contribution towards the model predicting this query as a malicious query. Other features like utl\_inaddr, all\_tab\_columns make sense as it stores Database internal information.  
We have some features like Select and Form which are mostly not harmful and common in most of the queries so it does not add any value to predict this query as malicious.

1. Example 2  
   



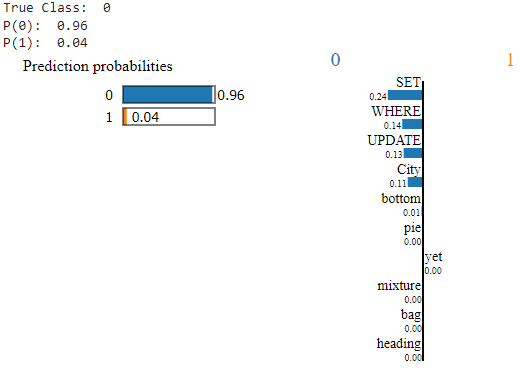
Here in this example select and form are common words but users are very common in malicious queries, probably attackers trying to get users' information. Here 1=1 will return all the information of the users table from the database. LIME is able to explain the importance of word users and 1 in the query to make it malicious by giving them big scores.

1. Example 3  
   



This query is simple Insert into operation to add some values into the table and LIME is able to get the most important features INSERT, INTO, VALUES to mark this query as non-malicious. It has not given importance to any of the inserted values.

1. Example 4  
   



This query is used to update the value in the table and is not malicious. LIME is able to get the sensible features responsible for non-malicious queries. One thing to notice here is, word ‘WHERE’ has a positive score for class 0 for this query and a positive score for class 1 for malicious query in example 2.

## **5.3 Advance Modeling**

We do not have much room for improvement as discussed in the previous phase. All erroneous points are mostly one word and don’t have any pattern to classify the query. But it is worth trying a deep learning model and comparing the results with a classical machine learning algorithm.

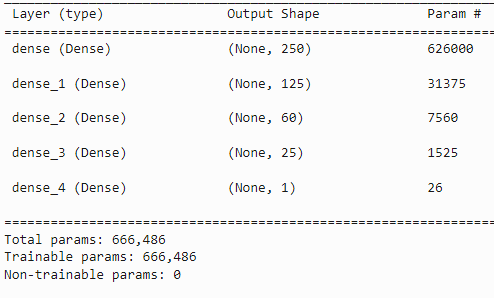
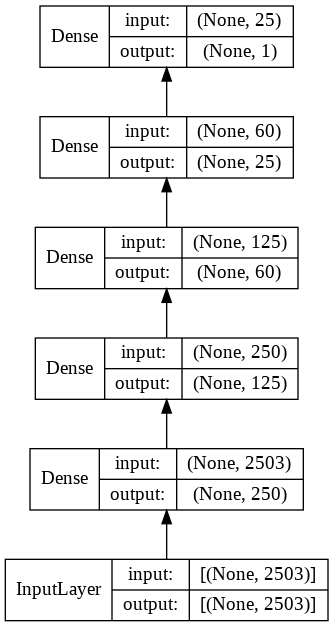
We will use the same approach as machine learning techniques starting from the simple multi layer perceptrons neural network and then trying single layer bidirectional RNN (recurrent neural network). At last we will also try multi level RNN as well.

### **5.3.1 Multi Layer Perceptrons (MLP)**

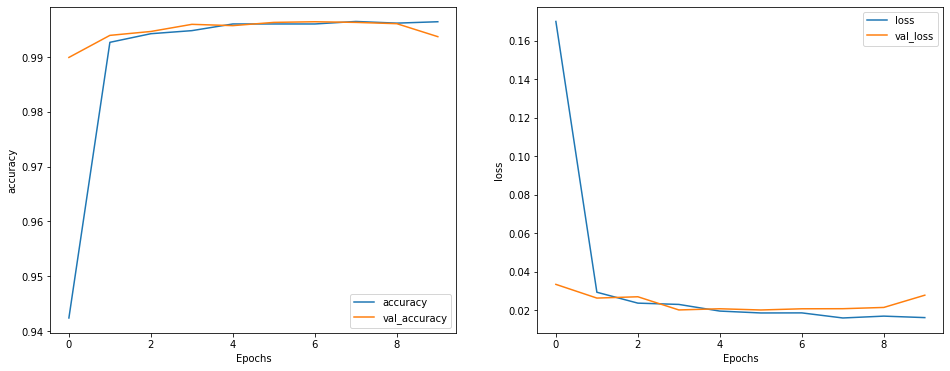
MLP’s are the classical type of neural network. They are composed of one or more layers of neurons. Data is fed to the input layer, there may be one or more hidden layers providing levels of abstraction, and predictions are made on the output layer, also called the visible layer.

We have tried very simple MLP models with a total 4 hidden layers. We have used Adam optimizer with learning rate 0.0001 and optimized the model for binary cross entropy.

After 10 epochs we got really good accuracy for the train and validation dataset and from the accuracy vs epochs graph we can see that the model is not overfitting or underfitting. Loss for both train and validation data is decreasing and accuracy is increasing.

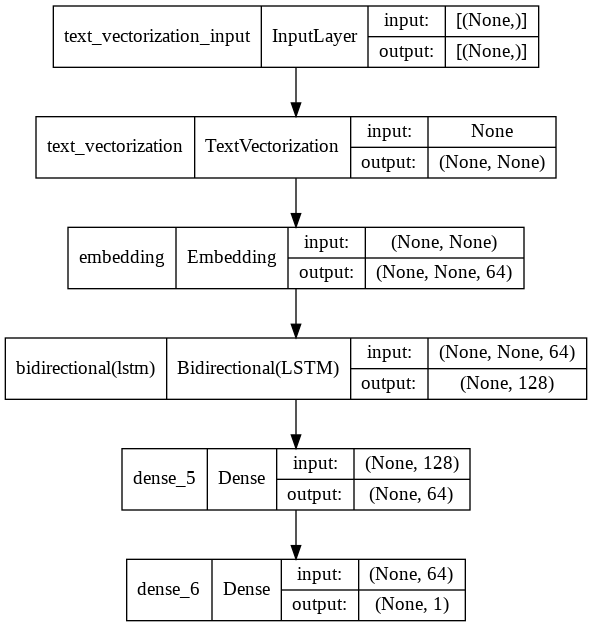
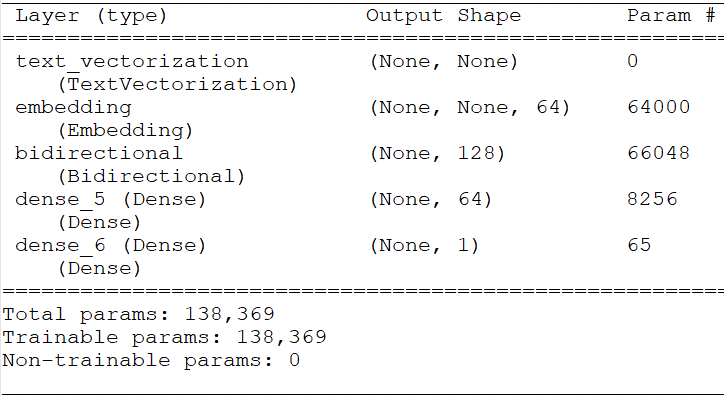


Accuracy and loss vs epochs graph for MLP,

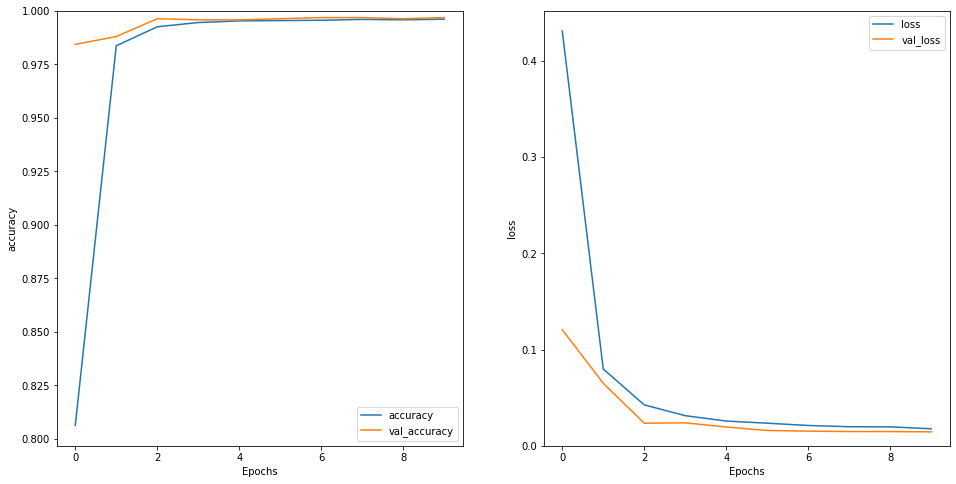


### **5.3.2 Recurrent Neural Network (RNN)**

Recurrent Neural networks work really well with time series and text data. Text data can be imagined as a series of words and RNN can be used for classification tasks. We will use bidirectional RNN to provide the context vector which is passed to the dense layer to perform the classification task.

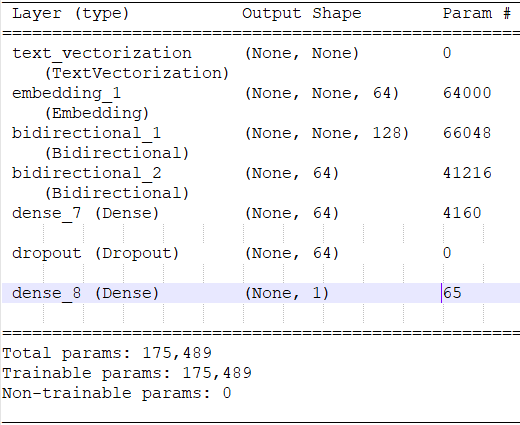
 

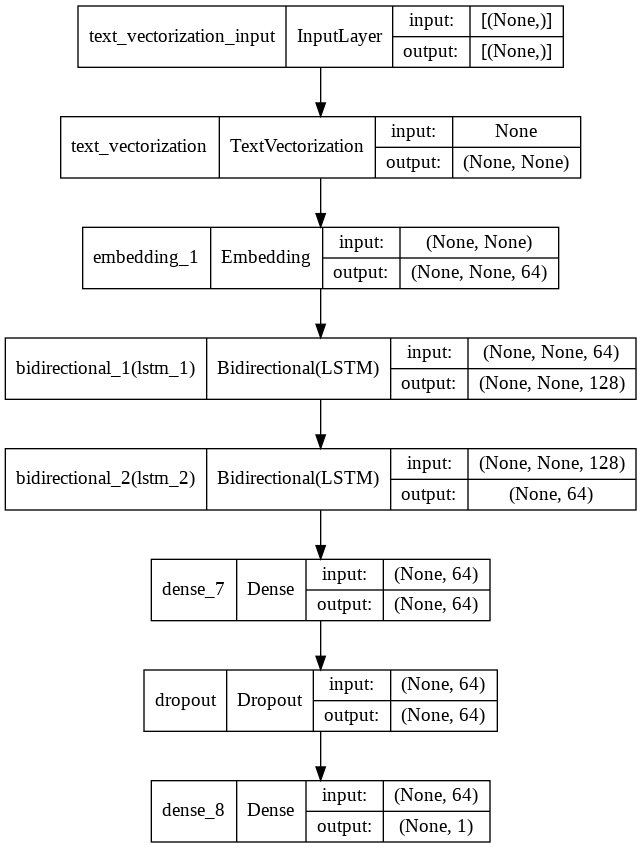
Accuracy and loss vs epochs graph for RNN,

****

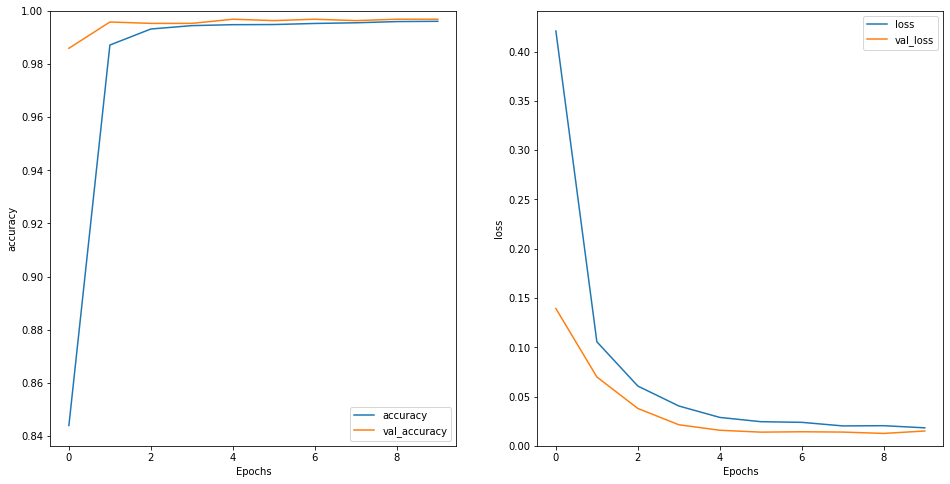
### **5.3.3 Multi Layer Recurrent Neural Network (Multi RNN)**

We can also create a stack of RNN layers same as deep learning models to build more complex models and learn more deep features from the sequence data. Output of one layer of bidirectional RNN is given to another layer of bidirectional RNN. On top of this we have a dense layer which takes input from the RNN layer and performs prediction tasks. To avoid overfitting we have also used a dropout layer.





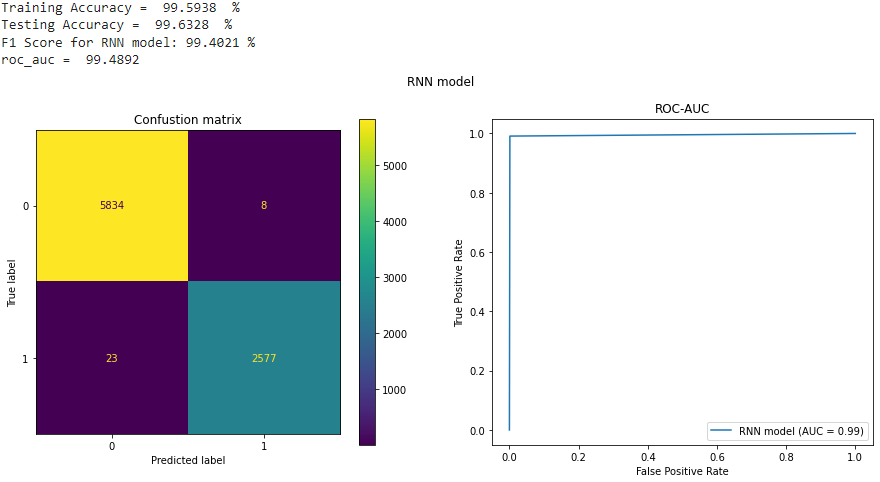
Let’s see the accuracy and loss vs epochs graph for multilayer RNN model.



As we can see from the above graphs, accuracy is increasing with the epochs and stays constant after 4-5 epochs for both training and validation dataset. Also the loss is decreasing with epochs and stays constant after 6 epochs for training and validation dataset. This makes sure that model is neither over fitting nor underfitting.

We got almost the same results for all three deep learning models. (Please refer to the jupyter notebook.)

Below is the evaluation for the multilayer RNN model.



Please refer to [Advanced\_Modeling.ipynb](https://github.com/mayurkagathara/sqli_detection/blob/master/Notebook/Phase_4_Advanced_Modeling.ipynb) for code snippets mentioned in topic 4. (<https://github.com/mayurkagathara/sqli_detection/blob/master/Notebook/Phase_4_Advanced_Modeling.ipynb>)

# **6. Model Deployment**

We chose the best Random forest model with engineered features for deployment. We have followed the below deployment process,

1. Save the best model and TFIDF vectorizer as a pickle file.
2. Create a module with all utility functions which can clean the raw input query, transform it using TFIDF vectorizer, and predict the class of the query.
3. Create a main function for deployment which leverages the functions from the module.

We have deployed the model using stramlit and flask REST API.

## **6.1 Streamlit**

Streamlit is a python library which is very easy to use and helps to create beautiful data science apps. Streamlit also provides free hosting service to host the web-app. We have created a github repository to host the streamlit code and finally host the streamlit app on the streamlit cloud servers which takes the code from the github repository.

Link of the Github Repo: <https://github.com/mayurkagathara/sqli_detection>

Directory structure:

**sqli\_detection**

├── streamlit\_app.py Main file to run the streamlit app\_

├── prediction\_module.py Prediction module with all the utility functions

├── requirements.txt Dependencies

├── README.md

├── **Model**

├──final\_RFC\_FE\_model.model Pickled best RF Model

├──tfidf\_vec.sav Pickled TFIDF vectorizer

├── **Documentation** Folder for documentation

├── **Notebook**  Folder for jupyter notebooks

├── **data**  Folder for data

**How to run in local**

1. Clone the repo using git clone:  
   git clone https://github.com/mayurkagathara/sqli\_detection
2. Install dependencies using pip:  
   pip install -r requirements.txt
3. Go to CMD and run the following command:  
   streamlit run streamlit\_app.py  
   (Run it from the root directory of the project).

This will start a streamlit server in the local,

> streamlit run streamlit\_app.py

You can now view your Streamlit app in your browser.

Local URL: <http://localhost:8501>

Network URL: <http://192.168.xx.xx:8501>

In any internet browser go to the above URL to access the application. To access the hosted streamlit application go to <https://share.streamlit.io/mayurkagathara/sqli_detection> (The app will go to sleep if it is not used for 10 days as per streamlit free use community policy). Please find an app screenshot in the appendix section.

## **6.2 Flask**

We can use flask to create the web app as well. But it requires good html and css scripting knowledge. So we decided to use the flask\_restful library to create the API endpoint which can be utilized to send a simple POST request to find out if the query is a SQLi attack or not.

For use in actual scenarios we must have an API endpoint which can be utilized by different teams to use the machine learning model to predict the query class in real time. For example website backend teams can use the API to check the query for SQL injection attack before passing to the database.

Most of the famous scripting languages support the REST API and which makes it easy to use and integrate different systems.

Directory structure:

**mysite**

├── final\_RFC\_FE\_model.model Pickled best model

├── flask\_app.py Main app to run flask app

├── prediction\_module.py Prediction module with all the utility functions

├── requirements.txt Dependencies

├── tfidf\_vec.sav Pickled TFIDF vectorizer

**How to run in local**

1. Get the code from <https://github.com/mayurkagathara/sqli_detection/tree/master/Flask>
2. Install dependencies using pip:  
   pip install -r requirements.txt
3. Go to CMD and run the following command:  
   python flask\_app.py  
   (Run it from the root directory of the project).  
   This will start the local Flask webserver in the localhost.

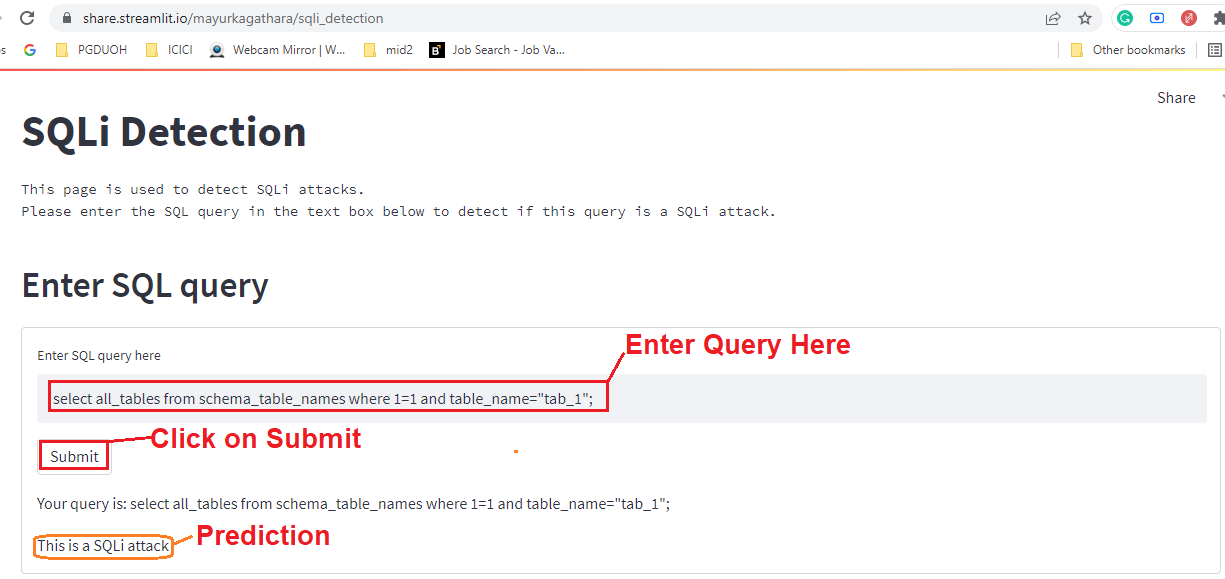
We have also deployed this flask app on the internet using pythonanywhere free tier service. Flask REST API endpoint can be accessed from anywhere (use endpoint /predict to make the post request) using the link <https://mayurkagathara.pythonanywhere.com>.

More information regarding API can be found in the Appendix section.

# 

# **Appendix:**

## **Streamlit app Screenshot**

****

## **Code for streamlit:**

**import streamlit as st**

**from prediction\_module import predict\_class**

**st.set\_page\_config(page\_title="SQLi Detection",page\_icon="👾",layout="wide") # Set the page title and icon**

**st.title("SQLi Detection") # Set the title of the page**

**st.text("""This page is used to detect SQLi attacks.**

**Please enter the SQL query in the text box below to detect if this query is a SQLi attack.""") # Set the text of the page**

**st.header("Enter SQL query") # Set the header of the page**

**with st.form("SQLi Detection"): # Create a form with name "SQLi Detection"**

**query = st.text\_input("Enter SQL query here") # Create a text input with name "Enter SQL query here"**

**if st.form\_submit\_button("Submit"): # if the user clicks the submit button**

**isSQLi = predict\_class(query) # Call the predict\_class function and store the result in isSQLi**

**st.write("Your query is:", query) # Write the query in the text box**

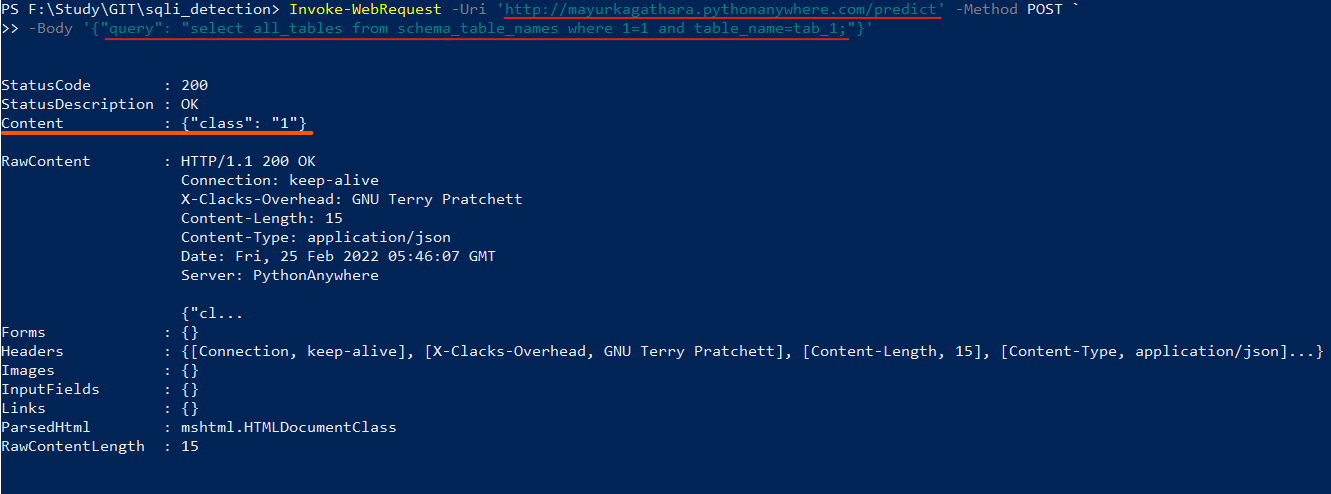
**if isSQLi: # If the query is a SQLi attack**

**st.write("This is a SQLi attack") # Write "This is a SQLi attack"**

**else:**

**st.write("This is not a SQLi attack") # else write "This is not a SQLi attack"**

## **Flask API screenshot:**



This API call can be made from most of the programming languages and simply from curl and powershell.

## **Code for Flask:**

**import flask as fl**

**from flask\_restful import Resource, Api**

**from prediction\_module import predict\_class**

**app = fl.Flask(\_\_name\_\_)**

**api = Api(app)**

**#create post method to return the class of the query**

**class Predict(Resource):**

**def post(self):**

**data = fl.request.get\_json(force=True, silent=True, cache=False)**

**if data:**

**query = data['query']**

**return { "class" : str(predict\_class(query))}**

**else:**

**return {'message': 'No data received'}, 400**

**def get(self):**

**return fl.redirect("/")**

**api.add\_resource(Predict, '/predict')**

**#create get method to return documentation**

**class Docs(Resource):**

**def get(self):**

**msg = """<html><p>There is an endpoint named predict. <br>**

**It takes a query as a parameter and returns the class of the query. <br>**

**It accepts a json object with a query as a parameter. <br>**

**The query is a string. <br>**

**The response is a json string. class = 1 if query is sql injection, 0 otherwise. <br><br>**

**Example: <br>**

**curl -X POST http://127.0.0.1:5000/predict -H "Content-Type: application/json" -d '{"query": "Select \* from Employee"}' <br>**

**The response is: {"class": "0"}</p></html>**

**"""**

**return fl.Response(msg, mimetype='text/html')**

**api.add\_resource(Docs, '/')**

**@app.errorhandler(404)**

**def own\_404\_page(error):**

**return fl.Response("<html><p>404: Page not found. Goto <a href='/'>home</a></p></html>", status=404, mimetype='text/html')**

**if \_\_name\_\_ == '\_\_main\_\_':**

**app.run()**

## **API reference:**

**URL:** <http://mayurkagathara.pythonanywhere.com/predict> (or <http://localhost:5000/predict> for localhost)

**Method:** POST

**Parameters:**

* **query:** SQL query in string format.
* **type:** json

**Example:**

1. **Curl  
   curl -X POST http://mayurkagathara.pythonanywhere.com/predict -H "Content-Type: application/json" -d "{\"query\": \"union select id from table where 102=102\"}"**
2. **PHP  
   <?php**

**$url = "http://mayurkagathara.pythonanywhere.com/predict";**

**$curl = curl\_init($url);**

**curl\_setopt($curl, CURLOPT\_URL, $url);**

**curl\_setopt($curl, CURLOPT\_POST, true);**

**curl\_setopt($curl, CURLOPT\_RETURNTRANSFER, true);**

**$headers = array(**

**"Content-Type: application/json",**

**);**

**curl\_setopt($curl, CURLOPT\_HTTPHEADER, $headers);**

**$data = '{"query": "union select id from table where 102=102"}';**

**curl\_setopt($curl, CURLOPT\_POSTFIELDS, $data);**

**$resp = curl\_exec($curl);**

**curl\_close($curl);**

**var\_dump($resp);**

**?>**

1. **Java  
   URL url = new URL("**[**http://mayurkagathara.pythonanywhere.com/predict**](http://mayurkagathara.pythonanywhere.com/predict)**");**

**HttpURLConnection http = (HttpURLConnection)url.openConnection();**

**http.setRequestMethod("POST");**

**http.setDoOutput(true);**

**http.setRequestProperty("Content-Type", "application/json");**

**String data = "{\"query\": \"union select id from table where 102=102\"}";**

**byte[] out = data.getBytes(StandardCharsets.UTF\_8);**

**OutputStream stream = http.getOutputStream();**

**stream.write(out);**

**System.out.println(http.getResponseCode() + " " + http.getResponseMessage());**

**http.disconnect();**

1. **javascript**

**var url = "http://mayurkagathara.pythonanywhere.com/predict";**

**var xhr = new XMLHttpRequest();**

**xhr.open("POST", url);**

**xhr.setRequestHeader("Content-Type", "application/json");**

**xhr.onreadystatechange = function () {**

**if (xhr.readyState === 4) {**

**console.log(xhr.status);**

**console.log(xhr.responseText);**

**}};**

**var data = '{"query": "union select id from table where 102=102"}';**

**xhr.send(data);**

# **References:**

1. <https://portswigger.net/web-security/sql-injection>
2. <https://www.imperva.com/learn/application-security/sql-injection-sqli/#:~:text=SQL%20injections%20typically%20fall%20under,data%20and%20their%20damage%20potential>.
3. <https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1727&context=etd_projects>
4. <https://blog.pythonanywhere.com/121/>
5. <https://help.pythonanywhere.com/pages/#ive-got-an-existing-web-app-that-im-trying-to-deploy>
6. <https://docs.streamlit.io/>
7. <https://docs.streamlit.io/streamlit-cloud>
8. <https://flask.palletsprojects.com/en/2.0.x/>
9. <https://stackoverflow.com/>
10. <https://reqbin.com/>