International Journal of Information Security (2024) 23:1541–1556
https://doi.org/10.1007/s10207-023-00791-y

REGULAR CONTRIBUTION

A deep learning approach based on multi-view consensus for SQL injection detection

Arzu Gorgulu Kakisim[1]

Published online: 9 January 2024

Abstract
SQL injection (SQLi) attacks are one of the oldest and most serious security threats, consistently ranking among the top ten critical web security risks. Traditional defense mechanisms against SQL injection predominantly use blacklists to disallow common injection characters or terms. However, the major challenge for these systems is to create a comprehensive list of potential SQLi characters, terms, and multi-terms that encompass various types of SQLi attacks (time-based, error-based, etc.), taking into account various SQL datasets (such as MySQL, Oracle, and NoSQL). Recently, some research studies have concentrated on feature learning from SQL queries by applying some well-known deep architectures to detect SQLi attacks. Motivated by a similar objective, this research introduces a novel deep learning-based SQLi detection system named "Bidirectional LSTM-CNN based on Multi-View Consensus" (MVC-BiCNN). The proposed method implements a pre-processing step that generates multiple views from SQL data by semantically encoding SQL statements into their corresponding SQL tags. By utilizing two different main layers, which are bidirectional long short-term memory (LSTM) and convolutional neural network (CNN), the proposed method learns a joint latent space from multi-view representations. In the detection phase, the proposed method yields separate predictions for each representation and assesses whether the query constitutes an SQLi attack based on a consensus function's output. Moreover, Interpretable Model-Agnostic Annotations (LIME), one of the methods of Explainable Artificial Intelligence (XAI), is employed for the purpose of interpreting the model's results and analyzing the SQL injection (SQLi) inputs. The experimental results demonstrate that MVC-BiCNN outperforms the baseline methods, yielding 99.96% detection rate.

Keywords SQL injection · Deep learning · Code injection · Information security · XAI

1 Introduction

SQL injection (SQLi) is a type of code injection attack that enables the execution of malicious SQL statements on the database server behind web applications by inserting SQL commands into various entry points, such as Web forms, domain names, or page requests [1, 2]. SQLi attacks are versatile and can be employed to gain unauthorized access to a wide range of SQL databases, including but not limited to MySQL, Oracle, and NoSQL. In this way, the attackers gain the capability to carry out extensive manipulations within the database with the authority of a database administrator, endowed with comprehensive database privileges. Consequently, sensitive information, such as customer data, personal records, and confidential trade data, becomes susceptible to compromise. The attackers may modify data in a database, or add new data [3]. A notorious example of SQLi in action occurred during the 2011 PlayStation Network breach. Attackers organized a SQLi attack that resulted in the theft of personal information belonging to 77 million players, as well as the pilfering of credit card details of approximately ten thousand individuals. [1] Following that event, another significant SQL injection attack occurred in 2014, wherein Russian hackers managed to exfiltrate over 1.2 billion identifiers and passwords from in excess of 420,000 websites. [2] Since 2011, SQL injection attacks remain a notable and recurring con-

✉ Arzu Gorgulu Kakisim
arzu.kakisim@medeniyet.edu.tr

[1] Istanbul Medeniyet University, Computer Engineering, Istanbul, Turkey

[1] Playstation network hackers access data of 77 million users (2011) [online]. Website https://www.theguardian.com/technology/2011/apr/26/playstation-network-hackers-data [accessed 23/06/2022].

[2] Russian hackers amass over a billion internet passwords (2014) [online]. Website https://www.nytimes.com/2014/08/06/technology/

cern. As a matter of fact, the Open Web Application Security Project (OWASP) identifies injections as the third most dangerous threat to web application security in its OWASP Top 10 2021 report.[3]

SQL is a structured language that has a lexical and syntactical system consisting of standard grammar rules. SQL query could be treated as a token sequence that includes keywords, commands, identifiers, literal values and other symbols. Traditional SQL injection defense systems mostly use blacklist or whitelist to filter out errors or illegal commands, keywords and characters, or evaluate the correctness of source code [4]. The obstacle to these systems is the difficulty in creating an effective list of all the characters or keywords which could enable an attack. Considering the flexible and changing nature of languages, the main disadvantage of these systems is their vulnerability to zero-day attacks consisting of non-blacklisted keywords. Therefore, machine learning algorithms are used mostly as a detector to discover the abnormality in the use of characters or keywords observed in user entries and to prevent attacks or suspicious logins [5]. Existing machine learning-based detection methods mostly utilize individual frequency information of input terms, ignoring correlation among sequential multi-term patterns belonging to the input that is generally observed as a token sequence. To extract sequential multi-term patterns, some methods use N-gram-approaches that are computationally expensive due to their tendency to generate a substantial volume of features.

To address sequential dependencies within the input data, and thus increase the detection performance, recently, a few researchers have focused on detecting SQLi attacks using deep learning architectures [6, 7]. Deep learning architectures, which perform feature engineering by establishing relationships between features through defined hidden layers and also have the capacity to learn from errors, are used in various fields, including but not limited to malware detection, intrusion and anomaly detection. These architectures aim to learn long-term dependencies between time steps of sequence data as well as local distinctive features. To improve the performance of deep learning architectures, some methods utilize a feature extraction process to represent SQL strings using specific attributes. These attributes may encompass characteristics like the length of SQL statements, the count of detected keywords, or the frequency of special characters. Subsequently, these extracted features are utilized as input for deep learning models [8]. Some SQLi detection methods encode SQL strings using syntactic specifiers that define semantic expressions of SQL commands or keywords. Some of these approaches transform SQL strings into new sequences containing only the syntactic identifiers of SQL terms [9], while some other methods aim to enrich SQL strings with semantic identifiers by preserving the original SQL terms [10]. Furthermore, some methods transform SQL strings into fixed two-dimensional matrices to feed deep learning models with image-like data inputs [11]. The experimental results presented in these studies show that deep learning architectures that take SQL statements obtained by pre-processing and re-representation processes as inputs achieve more effective detection performances than deep learning architectures that are fed directly by the original SQL strings. However, although the deep architectures such as convolutional neural network (CNN) [10] and long short-term memory networks (LSTM) or bidirectional LSTM (Bi-LSTM) [12, 13] are certainly not new, it is still an open problem which architectures are better at detecting SQLi attacks and which combination of input data will produce better results for these architectures. In particular, treating even one of the millions of injection attacks as benign can lead to detrimental outcomes for users. Consequently, the development of detection models characterized by minimal false positive rates remains an important issue in this field.

Deep learning algorithms often operate as black box systems, hiding any insight into decision-making processes. To address this limitation, explainable artificial intelligence (XAI) algorithms have been presented to provide transparency and interpretability for complex artificial intelligence models. These algorithms furnish interpretable insights into the functioning of the model, bridging the gap between human comprehension and artificial intelligence. From this standpoint, comprehending the rationale behind the decisions made by an attack detection model carries significant value in unraveling the inner workings of both legitimate inputs and potential attacks. In this direction, recently, XAI models have begun to be integrated into deep learning-based detection systems, including intrusion detection and malware detection.

In this paper, a deep learning architecture, called bidirectional LSTM-CNN based on multi-view Consensus (MVC-BiCNN), is presented for SQL injection detection. The proposed system generates three different representations of the SQL data and learns a joint space composed of these views by utilizing bidirectional LSTM and CNN layers. The proposed method can discover meaningful hidden information about inputs from different perspectives and reveals the high degree of correlations between these different views. MVC-BiCNN applies BiLSTM to capture the sequential contextual information in the input queries, and CNN to learn the important local features that reveal which terms play an important role in the query. The presented hybrid model combines Bi-LSTM and TextCNN architectures, both of which are well-suited for text classification tasks. Since SQL strings are a kind of sequential text-like data, the principal goal is to

russian-gang-said-to-amass-more-than-a-billion-stolen-internet-credentials.html [accessed 23/06/2022].

[3] Welcome to the owasp top 10-2021 [online]. Website https://owasp.org/Top10/ [accessed 23/06/2022].

design an adapted LSTM-CNN architecture that is capable of capturing diverse n-gram features within the input data while also revealing correlations among diverse n-gram characteristics within long-term dependencies. This strategy is aimed at empowering the model to excel in recognizing patterns and relationships, both of short-term and long-term nature, with remarkable accuracy and effectiveness. In the detection phase, MVC-BiCNN obtains multi-view representations for a new query, and learns output scores of BiCNN architecture for all views. Using a consensus decision, it classifies the query as an attack or legitimate. The main contributions of this work are presented as follows:

- An effective data pre-processing step is presented to generate multiple views of a given SQL query. Each SQL token is encoded with one of twenty-one different SQL tags to which it semantically corresponds. Moreover, each SQL query is enriched with these tags in order to express the information of all queries more comprehensively, which means adding different feature subsets to the feature space. Enriched representations are more effective for extracting both correlation between input terms and functional similarity between commands. Experimental studies show that both machine learning and deep learning architectures, when fed with enriched representations of their inputs, provide higher detection accuracy compared to using the input representations in their raw form.
- A multi-view consensus-based deep architecture is proposed that jointly learns SQL representations from different views of SQL queries. Thus, it enables us to reveal the common malicious patterns in SQL injection attacks using information from three different angles. The proposed method uses a hybrid architecture to combine the advantages of CNN and Bi-LSTM. Thus, it is aimed to extract local spatial patterns using CNN and capture long-range dependencies in sequential input data using LSTM. Experimental studies show that the proposed hybrid architecture achieves lower false-positive and false-negative values than other deep learning architectures and machine learning algorithms.
- The analysis results generated by the LIME model, a component of XAI toolkit, are presented with the primary aim of offering explainability and transparency. These results serve to unveil the decision-making process employed by the proposed model. Building upon the decisions reached by the model, a comprehensive examination is conducted to delve into the intricate patterns, behaviors, and implications that differentiate SQL injection (SQLi) inputs from benign inputs.
- A detailed comparison of the performance of different deep learning architectures used in SQL injection detection is presented using four different datasets.

The remainder of the paper is organized as follows: A literature review is presented in Section 2. In Section 3, the proposed SQLi detection system is presented. The experimental setup and experimental results are given in Section 4. Finally, the conclusions are discussed in Section 5.

2 Related Works

One of the pioneering approaches, JDBC checker, was introduced by Gould et al. [14]. The JDBC checker aims to identify potential malicious content and errors in SQL queries. It is a static analysis tool that obtains a list of all potential SQL strings for an application and then checks all those strings for potential malicious activity. The main problem of this approach is to properly model and store all the potential SQL queries. William et al. [15] presented AMNESIA that applies string matching between valid SQL queries and dynamically created queries to identify injection attacks. First, it generates all the query strings that could be generated by the application through static code analysis. It then monitors the compatibility of queries in the dynamic phase with the statically generated model. The authors in [16] presented an automated fix solution that replaces vulnerable code with generated secure code to prevent SQL injection attacks. They constructed a fixed generation query corpus that consists of SQL statements, SQL queries, and execution calls to validate user input types. The method is not a complete prevention mechanism, nor does it provide injection detection. The authors in [17] presented a technique to dynamically extract the programmer intended structure of SQL queries. They aim to mine programmer intentions, to generate programmer intended SQL queries, and then to transform applications with those additional SQL queries. To identify the malicious queries, their method compares programmer intended SQL query with the actual SQL query. Xiao et al. [18] proposed a detection method that aims to analyze the user behaviors and SQL execution responses. Their method extracts the predefined URLs and the corresponding SQL queries and then applies URL-SQL mapping between the request and the SQL query. However, it tends to increase false alarms because there exist many uncertain factors during the extraction of invariants.

To detect anomalies in the usage of characters or keywords that are observed in user inputs, many machine-learning-based methods have been presented [19]. Choi et al. [20] used the n-gram approach for feature extraction and applied support vector machine (SVM) classifier for the training phase. Joshi et al. [21] used Naive Bayes classifier with Role-based Access Control mechanism to detect SQL injection attacks. They construct a feature space that consists of tokens of the SQL query. Kamtuo and Soomlek [22] applied different classifiers such as support vector machine (SVM),

boosted decision tree and artificial neural network for SQL injection detection. The authors in [23] used a String Subsequence Kernel algorithm to reveal similarities between SQL query strings. They aimed to discover common subsequence properties of malicious queries. Their model utilizes SVM classifier to identify the new coming suspicious payload.

Recently, researchers have focused on implementing deep learning architectures to detect SQL injection attacks. The main aim is to create high-level feature maps for capturing the correlation of SQL expressions in deep hidden layers. Fang et al. [12] converted SQL query strings to the corresponding syntactic functions (semantic tags), transforms those new strings into word vectors as the input of the model and applied long short-term memory (LSTM) architecture for SQL injection detection. Qi Li et al. [24] presented a new model (ADF), which is an improved version of the Deep Forest. Their method utilizes deep forests using sliding windows to scan the raw SQL features. After obtaining all feature vectors generated by these forests, it aggregates the vectors into transformed feature vectors of the original input features. They integrated the AdaBoost algorithm into their deep forest model to estimate the impact of those features on the classification phase and update the weights of features on each layer. The authors in [25] implemented a convolutional neural network (CNN) model that consists of three padding convolution layers, three max-pooling layers, one full connectivity layer and one hidden layer. Abaimov et al. [10] used CNN model that takes enriched SQL queries as the input. They enriched each SQL query using semantic tags corresponding to query tokens. Xin Xie et al. [11] presented a SQL injection detection framework based on CNN with Elastic-Pooling layer that is an improved version of maximum pooling. Differently from other works, they transformed the SQL query string to a matrix that consists of the embedding vectors of the characters in the query. Their CNN model applies three different convolution layers with different sized kernels on those matrices. The authors in [24] used LSTM model to detect SQL injections for intelligent transportation systems. They also generated new SQL injection samples by combining the basic elements of SQL payloads for the training of their architecture. The authors in [8] used multi-layer perceptron network for SQLi detection. They selected a different number of features from URLs and conducted statistical research on these features.

3 Proposed Method

The first step of the proposed method is to generate multiple views from SQL queries to enrich the feature space using the semantic tags of SQL terms. The second step is to merge these views to augment the amount of data. Our aim is to more effectively model the proximity of relations between the SQL terms and their semantic tags. The third step is to learn the representations of SQL queries using a bidirectional CNN (BiCNN) deep learning architecture using this joint space. During the detection stage, multiple views are derived for newly incoming SQL queries, followed by the application of a classification process to each view. A consensus score is then calculated for each entry as to whether the entry is SQLi or not. This section is dedicated to elucidating the process of the multi-view generation and providing a comprehensive overview of the proposed architecture.

3.1 Multi-view representations for SQL inputs

SQL injections can consist of different attributes such as SQL related keywords, characters, parameters, usernames, passwords or table names. In order to accelerate the training of a learning model and increase the detection rate, it is very important to accurately reveal more meaningful attributes among the attributes of the input through the pre-processing process. Therefore, the proposed method performs the following three stages in the pre-processing module:

3.1.1 Tokenization

In the pre-processing phase, each SQL payload in corpus S is tokenized to reveal all attributes that are observed in SQL payload such as SQL commands “select”, “union”, “drop”, etc., table names, identifiers such as passwords or usernames, operators (&,$,#, etc.), parentheses, and punctuation marks. It is aimed to split an entire SQL input into smaller units as terms. Figure 1 illustrates the tokenization processes for different SQL inputs. However, some expressions such as table names, numbers or passwords are generally not meaningful as other SQL tokens. These expressions are omitted from the SQL queries to filter out noisy information. For instance, the cleaned version of the query s = {admin’ or ’1’=’1’–} could be the following: T = {admin ’ or ’ ’ = ’ ’ –}. As it is seen, integer numbers are deleted from SQL queries during the tokenization process. In addition, a threshold value has been used to delete some identifiers and usernames that are not observed frequently to reduce the number of attributes.

3.1.2 Extraction of Semantic SQL Tags:

SQL tags to which SQL statements correspond can be used to reveal similarities or relationships between SQL statements. For instance, Data Definition Language (DDL) is a set of SQL commands that are used to create, modify and delete database structures. Expressions such as “create” to create the database or its objects, “drop” to delete objects from the database, and “alter” to change the structure of the database are examples of DLL commands. Although the “create”, “drop”, or “alter” commands perform different functions, they basically

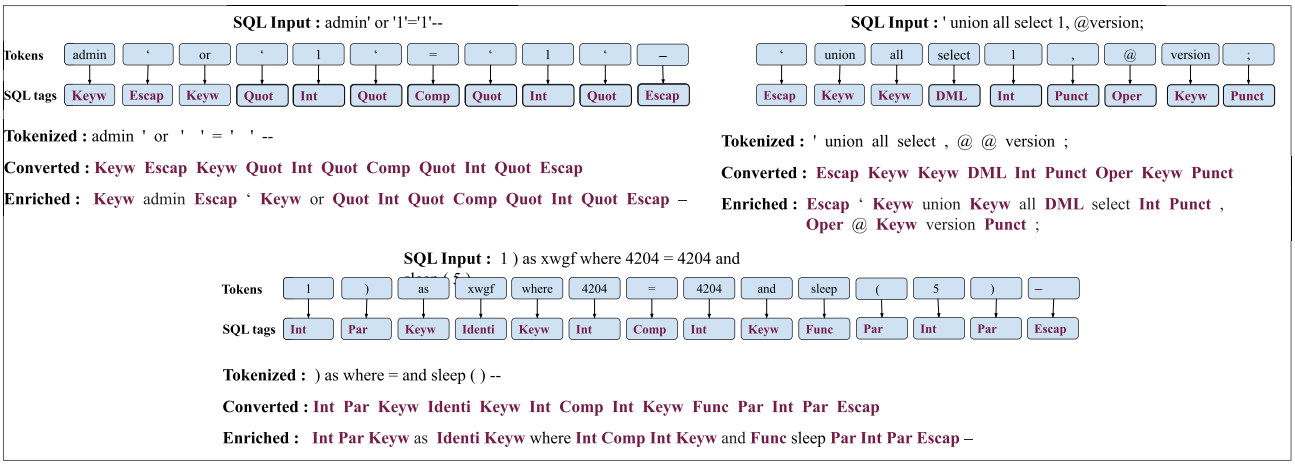

Fig. 1 Illustration of multi-view SQL representations according to the proposed pre-processing step

Table 1 Semantic SQL tags used for sample SQL statements

Semantic Label	SQL Expression	Semantic Label	SQL Expression	Semantic Label	SQL Expression
DLL	create, alter	Punctuation (Punct)	.;	Error	", $
DML	insert, select	Identifier (Identifi)	pass123, jessica	Where	where
DCL	revoke, grant	Identifierlist	('name', 'password')	Paranthesis (Par)	()
Keyword (Keyw)	or, and, exec	Quotes (Quot)	' '	Function (Func)	sleep(), benchmark()
Integer (Int) or Float	59887, 10.1	Comparison	, !, =,	CTE	with
Escape (Escap)	' –	Wildcard	*	Order	asc, desc
Hexadecimal	0x28	Builtin	date, varchar, unsigned	Operator (Oper)	@, #, %, /

refer to manipulations on the database. To reveal the functional similarity between commands, after obtaining tokens of a SQL input, each token is replaced with its semantic label using Python SQLparse library. This library provides support for formatting SQL statements. Moreover, a separate list has been created for other SQL instructions that this library does not contain. As a result, a SQL dictionary D has been created that lists twenty-one SQL tags and their corresponding commands. The semantic equivalents of the expressions observed in the SQL inputs are shown in Table 1 with examples. Each obtained SQL token is tagged with one of 21 different semantic tags to which it semantically corresponds. These tags could be following: Data Definition Language (DDL), Data Manipulation Language (DML), Data Control Language (DCL), or Common Table Expression (CTE). Figure 1 illustrates the extracted semantic tags of the sample of an SQL query. For instance, the semantic SQL tags of "union", "select", "1", "=" are "Keyword, "DML", "Integer" and "Comparison", respectively.

3.1.3 Enriching SQL inputs with SQL tags:

Using SQL queries only by encoding them with semantic tags may cause us to ignore information about the specific tasks that the commands perform. Therefore, CODDLE [10] uses a different encoding approach, combining SQL tokens with SQL tags. In this approach, after each SQL token observed in the SQL string, the semantic SQL tag of that token is appended to the string. Inspired by CODDLE, the proposed method uses the same approach to enrich SQL strings. By integrating semantic information into tokenized SQL strings, the specific and general functional expressions of the SQL command will be observed together. Thus, it is aimed to improve the detection rate of the learning model since proximity, relationships and correlations between SQL statements can be better revealed during the learning phase. The novelty of the proposed method is that it utilizes twenty-one different semantic tags as given in Table 1, differently from CODDLE, which uses only three semantic tags (expression, single, and operator). Thus, the proposed method uses a more detailed coding approach to enrich SQL statements. Considering that the semantic equivalents of some expressions integrated into the enriched version of SQL inputs are sufficient, some expressions such as numbers, punctuation, and parentheses are deleted from the enriched versions. Figure 1 illustrates the enrichment process for different SQL strings.

Using the pre-processing step mentioned above, SQL data is encoded to three different views. These views are called

Algorithm 1 : Extracting multi-view representations.

Input: S, D
Output: T, C, E
procedure MULTIVIEWREP(S)
  for $s \in S$ **do**
    $q \leftarrow$Tokenizing SQL payload s
    for $q_i \in q$ **do**
      $c_{q_i} \leftarrow$Obtain semantic tag of q_i using D
      $t_q \leftarrow t_q \cup q_i$
      $c_q \leftarrow c_q \cup c_{q_i}$
      $e_q \leftarrow e_q \cup c_{q_i} \cup q_i$
    $t_q \leftarrow$ Deleting noisy tokens from t_q
    $e_q \leftarrow$ Deleting noisy tokens from e_q
    $T \leftarrow T \cup t_q$
    $C \leftarrow C \cup c_q$
    $E \leftarrow E \cup e_q$

as tokenized T, converted C and enriched E SQL inputs, respectively. Algorithm 1 presents the pre-processing step to generate these three different views. The initial step involves subjecting each sample within the S dataset to a tokenization process. After this process, a predefined semantic tag in the D list is assigned to each token. Subsequently, three versions of the input sample are derived: the tokenized version t_q, the converted version c_q, and the enriched version e_q. Expressions called noise such as numbers, table names, numbers or passwords that can be observed in the generated tokenized and enriched sequence representations are deleted. Finally, by iterating through these steps for all samples within the dataset, tokenized T, converted C, and enriched E sets of data are obtained.

3.2 MVC-BiCNN architecture

In this section, the proposed deep learning architecture for SQL injection detection is presented. The proposed system is shown in Fig. 2. It firstly combines obtained multiple views from SQL inputs and obtains a corpus Q containing all generated representations of T, C, E. Thereafter, it uses an embedding layer as the first hidden layer of a network in order to generate input vectors. The embedding layer enables us to convert each SQL token into a fixed-length vector of predetermined size d. In natural language processing, embedding approaches are used for capturing the morphological, syntactic and semantic information of words. Similarly, in the proposed model, the embedding layer transforms each unique SQL term $q_i \in Q$ to a real-valued low-dimensional vector ϕ_i by discovering relationships in SQL strings considering sequential distributions of those strings.

Since SQL strings are a type of sequential data, BiLSTM (bidirectional LSTM) layer is used to properly capture both long-term dependencies. BiLSTM is a variant of recurrent neural network (RNN) model that is commonly used for processing time-series data and other sequential data to capture high-order data correlations and patterns. While traditional deep neural networks assume that the inputs and outputs are independent of each other, RNNs have an architecture that can model short-term dependencies using memory units (hidden states) that allow them to persist data. It uses previous outputs as inputs to influence the current input and output. Among many variants of RNNs, long short-term memory (LSTM) is the most popular architecture, which is capable of learning long-term dependencies. LSTM is composed of a set of recurrently connected memory cells and three gates as an input gate, an output gate and a forget gate. The gates are used for providing write, read and reset operations for the cells, which means selectively remembering part of information and passing it on to the next state to control the information flow. The LSTM network can only preserve information from the past. To take the advantage of the future context, another variant of LSTM, called as BiLSTM (bidirectional LSTM), was designed. BiLSTM consists of two LSTMs: one receiving the input in a forward direction, and the other in a backward direction. Thus, it has a capability of capturing bidirectional long-term dependencies between time steps of sequence data.

Let $\phi_i^s \in R^d$ be the d-dimensional embedding vectors for the i-th term in a given SQL string s. Let $\phi^s \in R^{nxd}$ refer to the string s where n denotes the length of s. BiLSTM layer is fed by the embedding matrix ϕ. LSTM treats each token of the SQL string as a separate input occurring at time t. At time-step t, the memory c_t and the hidden state h_t are updated with the following equations:

$$
\begin{aligned}
f_t &= \sigma(\phi_t U^f + h_{t-1} W_f + b_f) \\
i_t &= \sigma(\phi_t U^i + h_{t-1} W_i + b_i) \\
o_t &= \sigma(\phi_t U^o + h_{t-1} W_o + b_o) \\
C_t &= tanh(\phi_t U^i + h_{t-1} W_c) \\
c_t &= \sigma(f_t * c_{t-1} + i_t * C_t) \\
h_t &= tanh(C_t) * o_t
\end{aligned} \tag{1}
$$

where W represents the recurrent connection between the preceding hidden layer to the current hidden layer, U denotes the weight matrix that establishes connections from the inputs to the hidden layer, C is a candidate hidden state derived from the current input and the previous hidden state, and c represents the internal memory of the unit, which is a combination of the previous memory, multiplied by the forget gate, and the newly computed hidden state, multiplied by the input gate [26]. b_i, b_f and b_o are biases, and σ and $tanh$ represent to the sigmoid function and the hyperbolic tangent function, respectively. The initial stage involves determining which information should be preserved and which should be discarded from the cell. To accomplish this task, within the forgetting gate layer, both the current input ϕ_t and the previous hidden state h_{t-1} are processed through the sigmoid

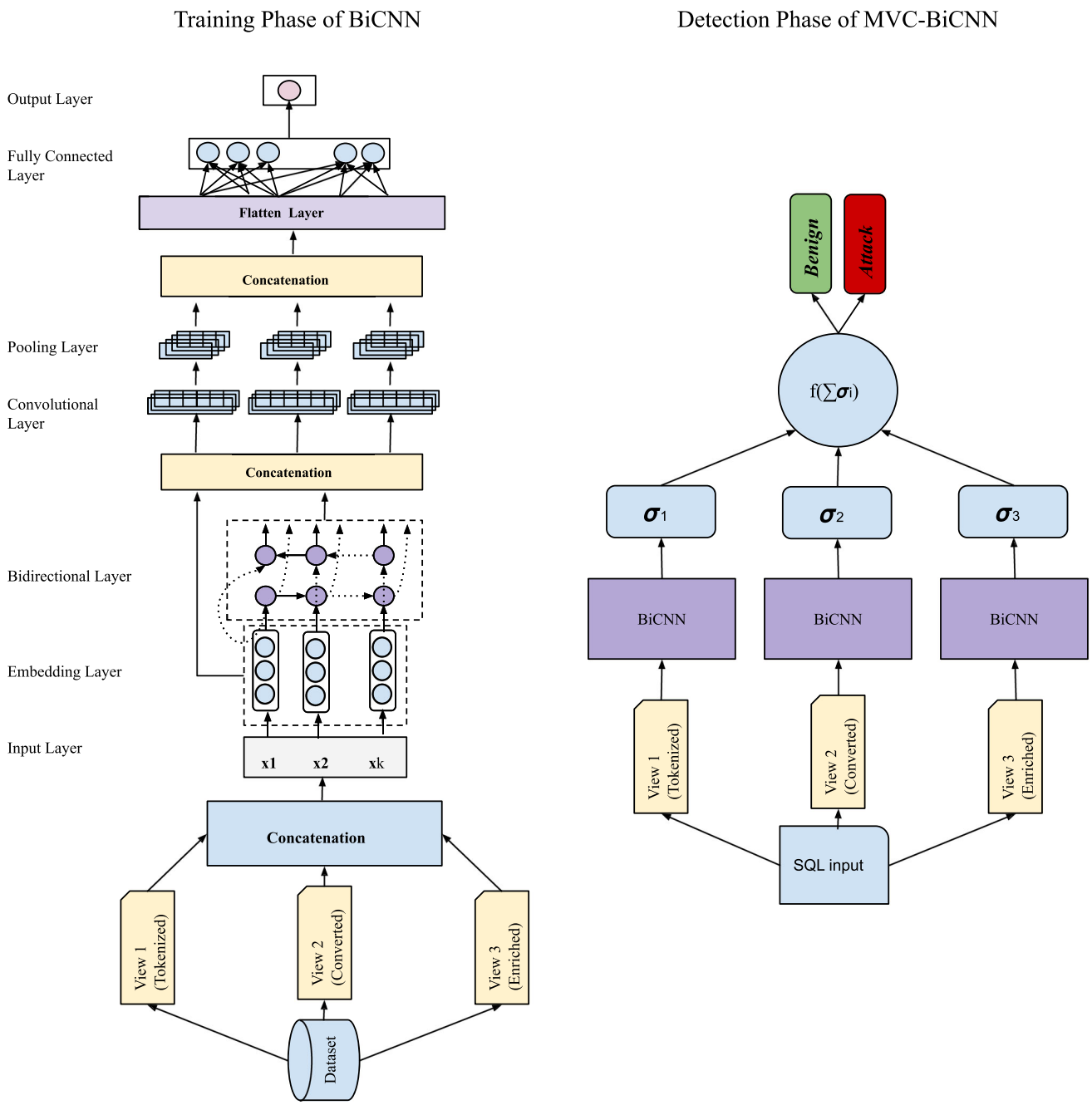

Fig. 2 The illustration of proposed architecture for SQL injection detection

function. The function produces an output that falls within the range of 0 to 1. When this output approaches 1, it signifies the complete retention of information. Conversely, if the value approaches 0, it indicates the complete elimination or forgetting of the information. Input gate processes h_{t-1} and ϕ_t through a sigmoid function to determine which information should be updated. Subsequently, a new candidate cell C_t is computed using a $tanh$ function. The input from the previous cell state c_{t-1} is multiplied by the forget gate output f_t. This output is then combined with the output of the input gate i_t to update the new candidate cell state C_t. Following this, the output gate passes h_{t-1} and current input ϕ_t to the sigmoid function. To generate the current hidden state, the result of this multiplication is combined with the output of the $tanh$ function of C_t. The ultimate outputs of the LSTM network consist of the current state c_t and the current hidden state h_t. The weight matrices of the forget, input, and output of the LSTM are represented by U_f, U_i, and U_o, respectively. Since BiLSTM contains two opposite networks for the forward and backward sequence context, an annotation

is obtained by concatenating forward and backward contexts as follows:

$$h_t = [\overrightarrow{h_t}, \overleftarrow{h_t}] \tag{2}$$

where $\overrightarrow{h_t} = \{\overrightarrow{h_1}, \overrightarrow{h_2}, ..., \overrightarrow{h_n}\}$ and $\overleftarrow{h_t} = \{\overleftarrow{h_1}, \overleftarrow{h_2}, ..., \overleftarrow{h_n}\}$ correspond to two hidden state sequences from the forward and the backward directions, respectively. With this way, the each SQL sequence can be represented as $h = \{h_1, h_2, ..., h_n\}$.

In the next step, the outputs of the embedding layer and bidirectional layer are concatenated to preserve both long-term dependencies and correlations between SQL tokens. In addition, the input data for the next layer are enriched using this feature aggregation phase. These two outputs are concatenated to form a new global feature vector matrix as follows:

$$H = [h \oplus \phi] \tag{3}$$

where $\oplus$ is a matrix concatenating operation.

The vector matrix H is passed to a CNN network to capture discriminative local SQL patterns. CNNs consist of one or more convolution layers and are used in the Natural Language Processing applications for local feature extraction. Similarly, to extract the most influential n-grams of different semantic aspects from the SQL queries, it is aimed to utilize convolution and the max-pooling operations. Within the convolutional layer, a collection of k filters is employed to process sub-matrices of the vector matrix H. Each filter $F \in R$ with dimensions $l \times w$, where l and w correspond to the height and width of convolution filter, is employed to process a window comprising l words to create a new feature v_i from a window of vectors $H_{i:i+l-1}$ as follows:

$$v_i = f(F * H_{i:i+l-1} + b) \tag{4}$$

where $b \in R, f$ and $H_{i:i+l-1}$ are the bias, the nonlinear activation function rectified linear unit $ReLU$ and the concatenation of $H_i, ..., H_{i+l-1}$, respectively. The filter scans each possible window of the matrix H and conducts convolution operations, resulting in the generation of a feature map m where m refers to $[v_0, v_1, ..., v_{n-l+1}]$. Using k convolution filters, k different feature maps are generated. In the final component of the convolutional layers, the rectified linear unit (ReLU) function is employed as an activation function to enhance nonlinearity in the outputs. The ReLU function sets all negative values in the vectors to zero while retaining all positive values. Additionally, zero padding is applied to maintain the original size of all sequences when convolutional filters are applied.

Following the application of a convolutional layer, it is customary to incorporate a pooling layer with the objective of reducing the dimension of the feature maps. In the proposed system, a pooling layer is employed, which utilizes the max-pooling operation over the feature map m and extracts the maximum value, denoted as $\hat{m} = max(m)$, as the ultimate feature. This pooling process facilitates the identification of the most prominent feature for each filter. After revealing k features from the feature map, the pooling results are merged $\hat{m} = \hat{m_1}, \hat{m_2}, ..., \hat{m_k}$ to generate the output of the CNN layer.

To capture the correlations of the given SQL input in different n-gram ranges, the proposed architecture applies multiple parallel convolutional neural networks that process the vector matrix using different kernel sizes. It consists of three filter region sizes: 2, 3 and 4. Since three different convolutional processes were carried out in the convolutional layer, in the pooling layer, pooling operation is performed separately for all three different feature map sets. In the next step, the feature maps obtained from three different sets are concatenated to form a single feature matrix M as follows:

$$M = [\hat{m^1} \oplus \hat{m^2} \oplus \hat{m^3}] \tag{5}$$

This feature vector is then passed to a fully connected layer consisting 250 hidden units with ReLU activation function. Since SQL injection detection is a supervised binary classification problem that assigns the samples into one of two classes: malicious or legitimate SQL payloads, this hidden layer is followed by a sigmoid classifier that returns the probability score of each class. The sigmoid binary cross-entropy is used to optimize a class one-versus-all loss based on max-entropy. Adam stochastic optimizer algorithm is used for training the network using the back-propagation algorithm.

In the detection phase, tokenized, converted and enriched views are generated for a new coming SQL input. For each view, the detection stage is applied using BiCNN. Three different sigmoid results from BiCNNs are used as input values for a function f. The function f sums the three sigmoid values and produces a detection result by applying a threshold value th. A value less than the threshold indicates that this SQL query is legitimate, and a value greater than the threshold indicates that the query is an attack.

4 Experimental evaluation

In this section, the datasets, experimental setup is introduced and the experimental results are discussed. The detection performances are reported by comparing the proposed method with different methods. The analysis results produced by the LIME model are presented to interpret the model's decision-making process.

Table 2 Dataset description

The dataset	SQLi	Legitimate
Dataset 1[4]	158	994
Dataset 2[5]	1594	994
Dataset 3[6]	5433	994
Dataset 4[7]	22559	19009
Dataset 5	29379	20002

4.1 Dataset and experimental setup

The experiments have been conducted on publicly available four different datasets, which are commonly used in the literature [6, 10]. Table 2 summarizes the statistics of the datasets. These datasets were downloaded from Github and Kaggle websites. In addition, we combined these datasets by removing repetitive samples observed across multiple datasets and adding new legitimate samples and then constructed new dataset that consists of 20002 malicious SQLi samples and 29379 benign samples. The final dataset contains different types of SQLi payloads such as time-based, union-based, Boolean-based or error-based. Moreover, the dataset consists of injections from different SQL databases such as MSSQL, MYSQL, Oracle, or NoSQL. After filtering noisy tokens, the number of unique tokens is 476 for the tokenized version of SQL data, 21 for the converted version, and 488 for the enriched version. The truncating and padding strategy has been used for making all sequences the same length. The length of sequences is set to 50. To ensure a balanced distribution of SQL injection (SQLi) and legitimate samples, 80% of the samples in each class were determined as train and 20% as test. The fivefold cross-validation is applied to create training and test samples.[5]

For the LSTM model, the number of hidden units was tested in the range of [16, 32, 64]. For the CNN model, the number of filters was tested in the range of [16, 32, 64]. In the context of experimental tests focusing on accuracy performance, the number of memory units for the LSTM layer is specified as 32, the number of filters for the CNN layer is set to 32, and the embedding dimension is set to 32. The size of hidden units of dense fully connected layer is set to 250 using a search strategy. For output layer, the sigmoid activation function is selected as the binary classifier. The training batch size is set to 100. To train the network using the back-propagation algorithm, Adam stochastic optimizer is selected. Binary cross-entropy is used as the loss function. To mitigate overfitting, early stopping is applied, with validation loss monitoring in the "max" mode, and the patience is set to 3.

All experiments were carried out in PC with an Intel(R) Xeon(R) CPU (2.20GHz) with 13GB RAM. We use Python 3.7.13, sklearn 1.0.2 library for machine learning algorithms, keras 2.8.0 to build neural and Tensorflow 1.4.1 as a backend computation.

4.2 Baseline Methods

Five different learning models are selected as baseline methods to be used for comparison:

TF-IDF+RF: This model uses the term and inverse term frequency (TF-IDF) transform to represent features. It indicates how frequently an SQL token occurs in the SQL input. IDF reduces the weight of terms that occur very frequently in the set of SQL inputs and increases the weight of terms that occur rarely. After obtaining TF-IDF feature space, RandomForest model was used for the learning and testing phases.

FastText: FastText [27] is a simple and efficient model for learning word representations and sentence classification. It allows us to quickly represent attributes at word level or character-n-gram level. It allows representations of n-gram words to be represented as sums of n-gram vectors. It provides a fast and memory-efficient mapping of the n-grams using the hashing trick. The model uses a simple neural network with only one layer. The text representations are first fed into a lookup layer, where the embeddings are fetched for every single word. Then, an average pooling is applied to obtain a single averaged embedding for the whole text. Finally, the output of the pooling phase is fed to a linear classifier.

CNN: This model applies a one-dimensional convolutional layer to the input embedding vectors. Then, a max-pooling is performed to reduce the dimensionality of the local features. The details are described in Section 3.

LSTM: This model performs a BiLSTM layer to the input embedding vectors. The details are described in Section 3.

TextCNN: The model [28] is a variant of the CNN model that uses multiple convolutional networks in parallel. This model mainly uses a one-dimensional convolutional layer that performs multiple sizes of filters with various window sizes to discover different granularities of features. In this study, the filter sizes are selected as 2, 3, and 4, respectively.

[4] owasp payloads [online]. Website: https://github.com/foospidy/payloads/blob/master/owasp/jbrofuzz/sqli.txt [accessed: 23/06/2022].

[5] sql injection payload list [online]. Website: https://github.com/payloadbox/sql-injection-payload-list [accessed: 23/06/2022].

[6] sql injection [online]. Website: https://github.com/SuperCowPowers/data_hacking/tree/master/sql_injection/data [accessed: 23/06/2022]

[7] sql-injection-dataset [online]. Website: https://www.kaggle.com/datasets/syedsaqlainhussain/sql-injection-dataset [accessed: 23/06/2022]

Table 3 SQLi detection performances of different methods based on multiple view representations

	Methods	False Positive (FP)			False Negative (FN)		
		Tokenized	Converted	Enriched	Tokenized	Converted	Enriched
Dataset1	TF-IDF+RF	18	4	4	24	30	20
	FastText	1	1	1	118	114	89
	CNN	3	4	**0**	15	11	**9**
	LSTM	8	4	1	14	24	12
	TextCNN	4	1	**0**	**9**	11	10
	BiCNN	4	1	1	12	10	**9**
Dataset2	TF-IDF+RF	30	56	19	31	46	31
	FastText	48	126	64	18	43	34
	CNN	25	27	5	20	20	**8**
	LSTM	19	26	12	24	29	21
	TextCNN	15	22	**4**	22	16	**8**
	BiCNN	20	13	8	22	10	**8**
Dataset3	TF-IDF+RF	30	22	10	25	23	22
	FastText	59	18	10	21	29	21
	CNN	13	**5**	**5**	14	11	3
	LSTM	15	14	**5**	21	12	3
	TextCNN	13	7	**5**	11	7	2
	BiCNN	13	7	**5**	11	6	**1**
Dataset4	TF-IDF+RF	181	78	47	679	72	52
	FastText	138	49	37	430	130	73
	CNN	98	58	**20**	320	44	26
	LSTM	54	43	23	368	40	33
	TextCNN	70	30	24	345	59	24
	BiCNN	62	34	**20**	346	39	**18**
Dataset5	TF-IDF+RF	179	145	67	934	116	82
	FastText	177	67	52	526	211	114
	CNN	91	43	42	429	134	50
	LSTM	87	51	40	455	81	53
	TextCNN	65	39	36	466	72	47
	BiCNN	95	56	**32**	412	58	**44**

4.3 Evaluation criteria

To evaluate the effectiveness of the proposed method, false-positive (FP), false-negative (FN), accuracy and detection rate metrics are used. FN refers to the number of SQLi samples incorrectly predicted as benign, and FP refers to the number of benign samples incorrectly predicted as SQLi attacks. Accuracy corresponds to the ratio of number of correct predictions to the total number of input samples. This metric is calculated as follows:

$$ACC = \frac{TP + TN}{P + N} \tag{6}$$

where P and N represent the total number of SQLi and legitimate samples, respectively. TP is the number of SQLi attacks that are classified as injection attack. True negative (TN) is the number of legitimate queries that are classified as benign. The TP rate (TPR), alternatively referred to as detection rate (DR), represents the percentage of the total SQLi samples that are correctly recognized as SQLi. This metric is quantified as follows:

$$DR = \frac{TP}{P} \tag{7}$$

4.4 Experimental Results

The detection results of different methods are shown in Table 3 and Table 4. The best performance results are given in bold text. First, the detection performances of the baseline methods and the proposed BiCNN architecture have been evaluated using the views obtained by the proposed method. Each view obtained in the multi-view generation process was

Table 4 SQLi detection performances of BiCNN and MVC-BiCNN for different datasets

	Methods	FP	FN
Dataset 1	BiCNN (Enriched)	1	9
	MVC-BiCNN	**0**	**4**
Dataset 2	BiCNN (Enriched)	8	8
	MVC-BiCNN	**4**	**0**
Dataset 3	BiCNN (Enriched)	5	1
	MVC-BiCNN	**2**	**1**
Dataset 4	BiCNN (Enriched)	20	18
	MVC-BiCNN	**6**	**8**
Dataset 5	BiCNN (Enriched)	32	44
	MVC-BiCNN	**6**	**13**

used as input data separately, and learning–testing processes were applied for each method.

The results in Table 3 demonstrate that the proposed BiCNN outperforms the baseline methods in terms of FP and FN scores for Dataset 3, Dataset 4, and Dataset 5, which are considerably larger than other datasets. It is observed that our method has comparable performance to CNN and TextCNN for Dataset 1 and Dataset 2. For all datasets, the lowest FP and FN scores were obtained by BiCNN, which takes enriched SQL representations as input. Similarly, almost all methods achieve better performance for all datasets when they use the enriched representations as input obtained by the proposed method. This result shows that the enriching process is very effective in revealing malicious patterns in SQL queries.

Considering Dataset 5 containing all datasets, the second highest results were obtained by TextCNN. The performance of this method was followed by CNN. This is because the TextCNN method, unlike CNN, performs multiple parallel convolutional operations, which allows it to discover a much larger number of sequential multi-word expressions in different n-gram ranges. LSTM has comparable performance with CNN in terms of FP and FN results. However, FastText, which takes trigrams generated from SQL expressions as inputs, gives relatively lower detection scores than TF-IDF+RF that is performed without considering the order of the expressions in the query. The results show that all methods achieve better performance when they use enriched versions of SQL inputs as input. The likely reason is that SQL semantic tags, which provide a better understanding of the role of a particular SQL command or symbol, help more effectively discover functional similarities and correlations between original SQL tokens.

The detection performance values of the proposed MVC-BiCNN are given in Table 4. According to the results, MVC-BiCNN provides higher detection performance than BiCNN that uses only enriched representations. This result shows that MVC-BiCNN more successfully uncovers meaningful

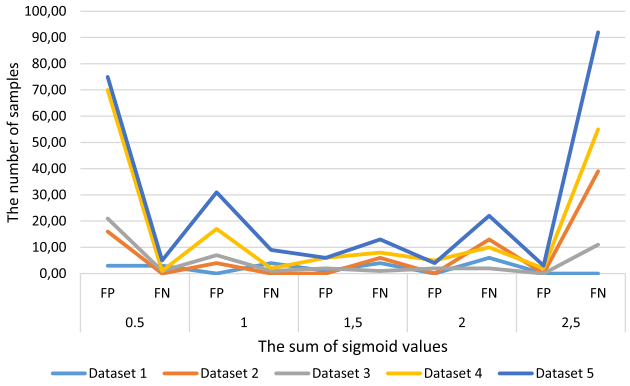

Fig. 3 The influence of the th parameter of consensus function on detection rate

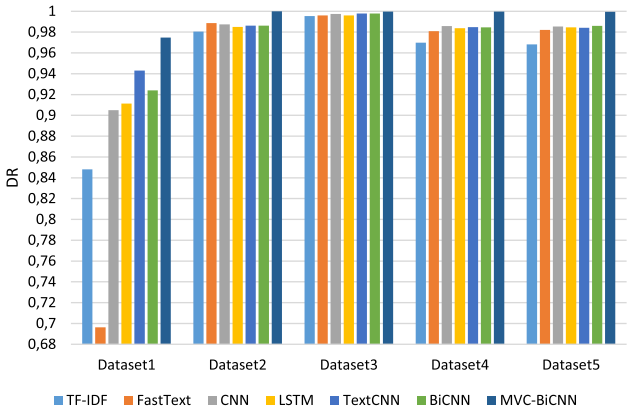

Fig. 4 The detection rates of the different methods

latent patterns in SQL injection queries by learning a joint space consisting of three different view representations.

Figure 3 demonstrates FP and FN results obtained by varying the threshold value parameter used in the function that generates the consensus decision of the method. Different detection results were obtained by varying the total sigmoid value between 0.5 and 2.5. The lowest FP and FN values for all datasets were generally obtained when the total sigmoid value was equal to 1.5. As the value of the total sigmoid value increases, the FN values tend to increase and the FP values tend to decrease. In the opposite case, FN values generally decrease and FP values increase. However, since the value of FN is even more important in detecting SQL injection, it is concluded that the most appropriate values for the proposed method are the values between 0.5 and 1.5.

Figure 4 presents a comparative analysis between MVC-BiCNN and the baseline approaches that directly utilize original SQL queries as their input. MVC-BiCNN improves the detection performance by approximately 15% for Dataset 1, 2% for Dataset 2, 0.44% for Dataset 3, 3% for Dataset 4, and 3.23% for Dataset 5 when contrasted with TF-IDF with RF. In comparison with deep learning-based methods, MVC-BiCNN provides the highest detection rates for all

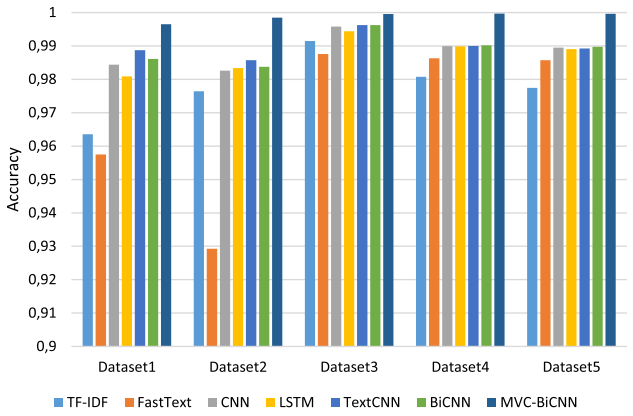

Fig. 5 The accuracy performances of the different methods

datasets. TextCNN gives the second highest detection results for Dataset 1 and Dataset 2, while BiCNN obtains the second highest detection results for Dataset 3, Dataset 4, and Dataset 5, which contains more samples compared to the other datasets. Similar results are observed in the accuracy values of the methods given in Fig. 5.

The proposed detection system is compared to some state-of-the-art methods that provide SQLi detection. However, the accuracy performances of these methods in the literature were obtained using different datasets or the same datasets including different numbers of samples. The results presented in Table 5 show that MVC-BiCNN achieves higher accuracy than other methods. MVC-BiCNN demonstrates an enhancement in detection accuracy of approximately 3% when compared to the CODDLE system [10]. The most likely reason for this might be that the CODDLE method enriches queries using only three different semantic tags. However, our system uses 21 different tags for the enrichment phase, which enables better modeling of functional similarity between queries. The other most important reason is that MVC-BiCNN uses a hybrid architecture consisting of a BiLSTM and CNN that takes into account both previous and subsequent contexts in queries, while CODDLE uses only CNN architecture. The second highest result was achieved in the study referenced as [9]. They created syntax trees of SQL entries to use them as LSTM inputs. However, the number of samples they use in the learning and testing phase is quite low compared to the number of samples used in our proposed method. The extent to which methods validated with a limited dataset may not be reliably extended to new SQLi instances.

Figure 6 is presented to show the loss and accuracy performances of the proposed deep model over 50 epoch for one fold dataset. The figure contains the performances of both the training and validation processes. It is observed that the model loss performance increases after the 10^{th} epoch. It is seen that the accuracy performance of the model reaches its maximum in the first few epochs and converges to between 99.6 and 99.4 after the 10^{th} epoch. In this study, the number of epochs was determined as 10 based on the loss performance of the model, and in addition, the early stopping approach was used to stop the training when no improvement was observed in the loss function for 3 iterations to prevent overfitting.

4.5 Explanation of the model results

In this section, the LIME model is used to explain the decision-making process of the proposed model over some sample inputs. LIME is an Explainable AI tool that provides explanations of individual predictions of the proposed model by taking into account certain local features [30]. To achieve this objective, LIME employs an approach aimed at minimizing the loss function, which quantifies the proximity between

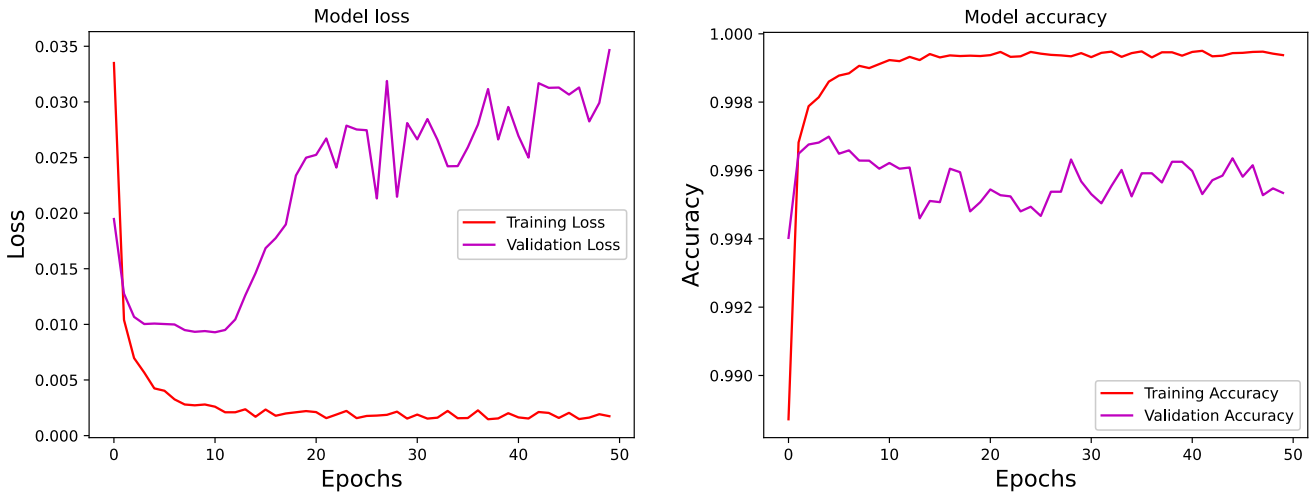

Fig. 6 The loss and accuracy performances of the proposed method

Table 5 SQLi detection performances of the state-of-the-art methods

The methods	Main approaches	#of SQLi/Legitimate	ACC
[9]	Syntax Tree + LSTM	1,312/1,317	99.77
[8]	Feature selection + Multilayer perceptron	7,095/76,960	99.67
[29]	Adaptive Deep Forest	10,000/10,000	98.00
[10]	Enrichment SQLs with 3 semantic tags + CNN	13,000/1,020	95.70
[12]	Converted SQLs with 24 Semantic Tags + LSTM	15,032/6,841	98.60
MVC-BiCNN	Multi-views + Bidirectional LSTM-CNN	29,379/20,002	**99.96**

a locally interpretable model and the original model in relation to the specific observation.

Figure 7 shows SQLi test examples that are correctly classified by the proposed model. The proposed model takes tokenized, converted, and enriched representations as separate inputs, giving a probability estimate for each input. In Fig. 7, the probabilities produced by the model for these three views, the attributes that are effective in the model's decision-making process and the weights of those attributes are demonstrated. The proposed model correctly predicts that the first SQLi example input is SQLi with a 97 % probability. In the realm of our analysis, it becomes evident that a crucial determinant in the model's decision-making process is the 'by' attribute, often referred to as a keyword. It is seen that the 'order' attribute contributes positively to the probability of the sample SQLi entry being in the Legitimate class. In this case, it is observed that this attribute contradicts the decision of the model. However, it is imperative to emphasize that our deep learning model derives its final decision not merely from the discrete evaluation of these two attributes but also from the correlation between them. The LIME results reveal that the converted representation of first SQLi example results in a 2% enhancement in the model's probability, as compared to the tokenized representation. The most distinctive attribute here is the 'operator' attribute, which actually represents '-'. Other effective attributes, respectively, are 'integer' and 'keyword', which actually correspond to '9087', and the 'order' attributes. The enriched representation significantly bolsters the model's decision, yielding a 100-percent probability of identifying the input as SQLi. This outcome surpasses the detection probabilities obtained through the other two representations. It is notable that the attributes governing the model's decision are largely consistent across both representations.

For the second SQLi sample input, the model correctly predicts with a 95-percent probability that the sample is SQLi. The attributes that support the model's decision are 'resource', 'grant', 'name', 'to', and 'connect' attributes. The pivotal attribute influencing this outcome is the 'identifier', which refers to the 'name' attribute. When the enriched version is given as input, the model gives the highest value for the probability of being SQLi. As seen here, the 'identifier' attribute still conflicts with the model's decision, but has a lower weight. The attribute 'keyword' is listed as the most distinctive attribute that supports the model's decision.

Figure 8 provides LIME explanations of the model's results when legal samples are used as inputs. The proposed model correctly predicts that the first example input is legitimate with a 93 % probability. The 'health' attribute aligns with the model's decision, whereas the 'statistics' attribute appears to contradict it. This discrepancy primarily arises from the fact that, despite 'statistics' serving as a clear identifier in this context, the model categorizes it as a 'keyword'. It can be inferred that the primary reason for this keyword conflicting with the model's judgment is its frequent occurrence in SQL injection (SQLi) examples. When we consider the converted representation, the probability of the input being legitimate is computed at 97 percent. In this context, the 'keyword' attribute also lends support to the model's decision. It can be deduced that the utilization of a keyword following an identifier is predominantly observed in legitimate examples. Notably, using the enriched version of the input, the model yields a probability of 100 percent, indicating that all existing features exert a positive contribution on the model's decision. The model misclassifies the second example input with a high probability. Attributes that support this decision are 'select' or 'from' etc. are general SQL tags. For the converted version of this example, on the contrary, the model achieves 100% accuracy. The SQL tags used also have a positive impact on the model's decision. The enriched representation of the input enables the model to make the correct decision in this example, but to obtain a lower prediction probability value compared to the converted representation.

In summation, our analysis underscores the positive contribution of enriched input representations for both examples in enhancing the model's decision-making process. LIME results show that the enriched representation can reduce the probability of misclassification that would be obtained when using only the converted version or only the tokenized version, and therefore can be effective in obtaining lower FP and TN values.

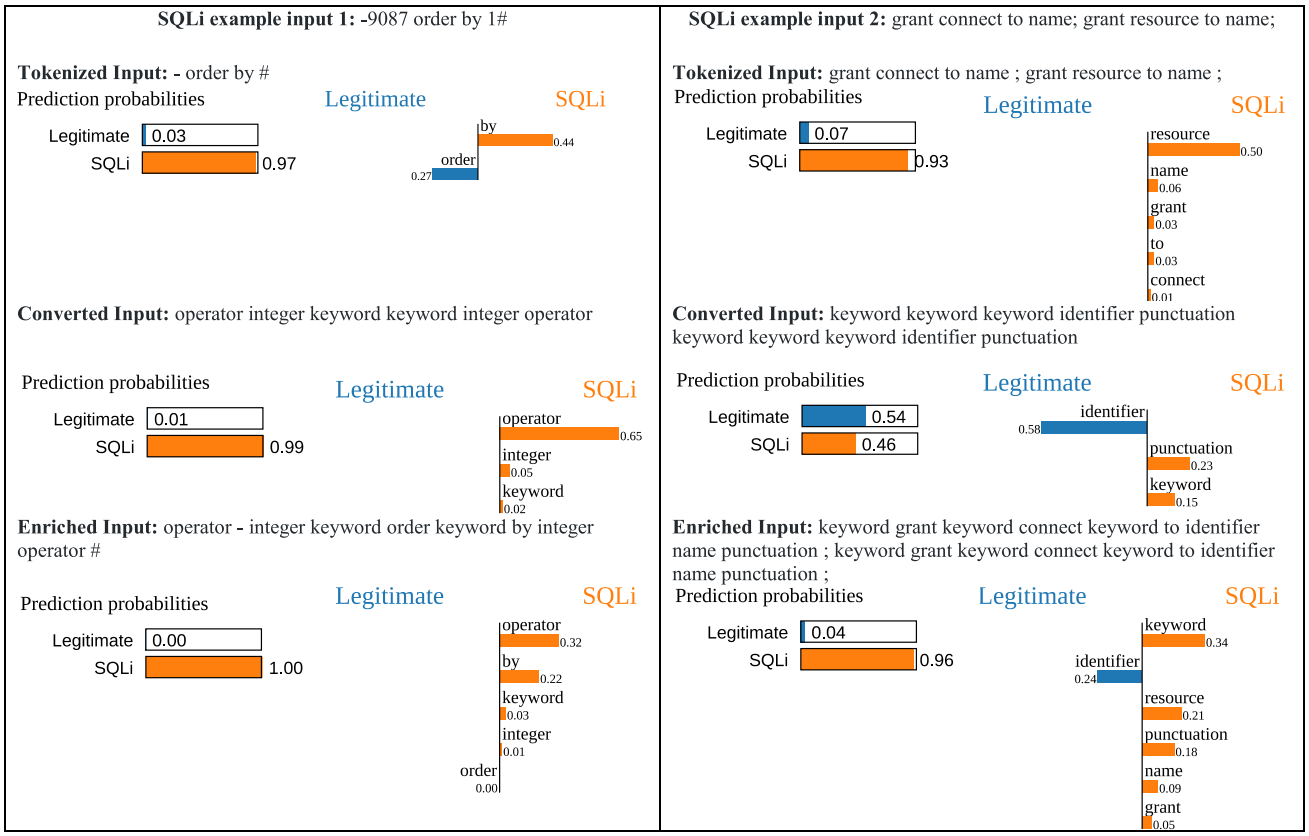

Fig. 7 The explanations generated by LIME for given SQLi samples

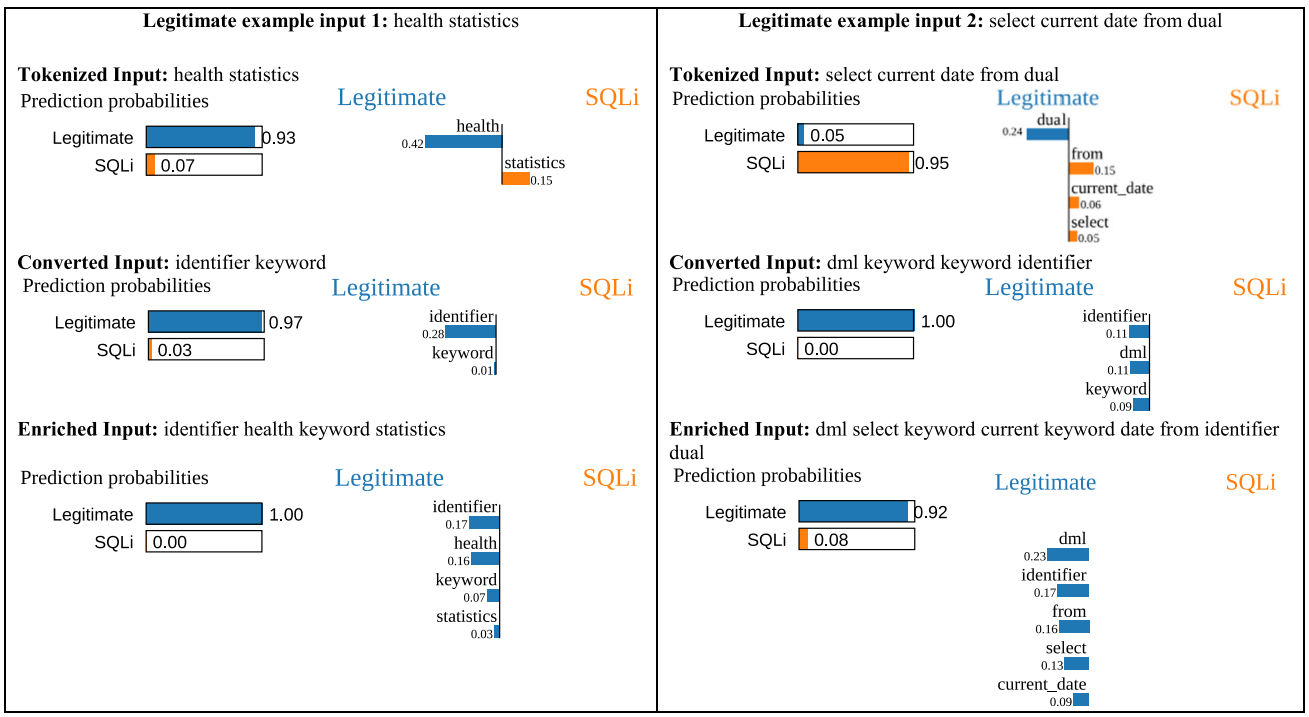

Fig. 8 The explanations generated by LIME for given legitimate samples

5 Conclusion

In this paper, a new deep learning-based SQL injection detection system, called MVC-BiCNN, is presented. The proposed system transforms the original SQL inputs into three different views that enrich the feature space with different feature subsets. To generate multi-views, the system uses twenty-one SQL semantic tags with SQL terms, which helps to more effectively capture the role of a particular SQL command or symbol, considering functional similarities. Experimental results confirm that the utilization of enriched representations for inputs achieves improved detection performance in both machine learning and deep learning architectures. The proposed deep learning system merges generated views to learn a latent joint space, which provides discovering the meaningful correlations between different feature subsets shared by multiple views. The proposed system focuses on leveraging a CNN architecture with multiple filtering processes in a single layer to extract high-level features of multiple terms and LSTM architecture to capture long-term dependencies between SQL token sequences. During the detection phase, a consensus function is employed to facilitate a collective decision-making process, ultimately determining whether a newly arrived input should be classified as an attack or not. This approach ensures that multiple perspectives and sources of information contribute to the decision, enhancing the robustness of the detection system. Empirical studies provide evidence that the multi-view consensus-based hybrid architecture consistently outperforms other deep learning architectures by yielding lower false-positive and false-negative values. Additionally, the XAI toolkit was used to interpret the decision-making process used by the proposed model, and analysis of the patterns that distinguish SQL injection (SQLi) entries from legitimate entries were presented.

Future work may focus on two key aspects. One aspect involves adapting the system to address different types of code injection attacks, such as Cross-Site Scripting (XSS), by configuring new settings and rules to effectively detect and mitigate these threats. Another promising direction is the augmentation of the system through dynamic monitoring approaches. By integrating dynamic monitoring mechanisms, the system can continuously adapt and evolve to counter evolving security challenges.

Funding Open access funding provided by the Scientific and Technological Research Council of Türkiye (TÜBİTAK).

Data Availability The datasets and code generated during the current study are available from the corresponding author on reasonable request.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

1. Lee, I., Jeong, S., Yeo, S., Moon, J.: A novel method for SQL injection attack detection based on removing SQL query attribute values. Mathematical and Computer Modelling, 55 (1-2), (2012) (Jan 1) 58-68. https://doi.org/10.1016/j.mcm.2011.01.050
2. Shar, L. K., Tan, H. B.: Defeating SQL injection. Computer, 46 (3), (2012) (Aug 10) 69-77.https://doi.org/10.1109/MC.2012.283
3. Atoum, J. O., Qaralleh, A. J.: A hybrid technique for SQL injection attacks detection and prevention. International Journal of Database Management Systems, 6 (1), (2014) (Feb 1) 21. https://doi.org/10.5121/ijdms.2014.6102
4. Aliero, M. S., Ghani, I., Qureshi, K. N., Rohani, M. F.: An algorithm for detecting SQL injection vulnerability using black-box testing. Journal of Ambient Intelligence and Humanized Computing, 11 (1), (2020) (Jan) 249-66. https://doi.org/10.1007/s12652-019-01235-z
5. Latchoumi, T. P., Reddy, M. S., Balamurugan, K.: Applied machine learning predictive analytics to SQL injection attack detection and prevention. European Journal of Molecular & Clinical Medicine, 7 (02), (2020)
6. Zhang, W., Li, Y., Li, X., Shao, M., Mi, Y., Zhang, H., Zhi, G.: Deep Neural Network-Based SQL Injection Detection Method. Security and Communication Networks, (2022) (March 24)
7. Alaoui, R. L., Nfaoui, E. H.: Deep Learning for Vulnerability and Attack Detection on Web Applications: A Systematic Literature Review. Future Internet, 14(4), (2022) (Apr 13) 118. https://doi.org/10.3390/fi14040118
8. Tang, P., Qiu, W., Huang, Z., Lian, H., Liu, G.: Detection of SQL injection based on artificial neural network. Knowledge-Based Systems, 190:105528, (2020) (Feb 29). https://doi.org/10.1016/j.knosys.2020.105528
9. Zhuo, Z., Cai, T., Zhang, X., Lv, F.: Long short–term memory on abstract syntax tree for SQL injection detection. IET Software, 15 (2), (2021) (Apr) 188-97. https://doi.org/10.1049/sfw2.12018
10. Abaimov, S., Bianchi, G.: CODDLE: Code-injection detection with deep learning. IEEE Access, 7, (2019) (Sep 13) 128617-27. https://doi.org/10.1109/ACCESS.2019.2939870

11. Xie, X., Ren, C., Fu, Y., Xu, J., Guo, J.: Sql injection detection for web applications based on elastic-pooling cnn. IEEE Access, 7, (2019) (Oct 21) 151475-81. https://doi.org/10.1109/ACCESS.2019.2947527
12. Fang, Y., Peng, J., Liu, L., Huang, C.: WOVSQLI: Detection of SQL injection behaviors using word vector and LSTM. InProceedings of the 2nd international conference on cryptography, security and privacy (2018) (Mar 16) 170-174
13. Fang, Y., Huang, C., Su, Y., Qiu, Y.: Detecting malicious JavaScript code based on semantic analysis. Computers & Security, 93:101764, (2020) (Jun 1). https://doi.org/10.1016/j.cose.2020.101764
14. Gould, C., Su, Z., Devanbu, P.: JDBC checker: A static analysis tool for SQL/JDBC applications. IEEE InProceedings. 26th International Conference on Software Engineering (2004) (May 28) 697-698
15. Halfond, W. G., Orso, A.: AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks. InProceedings of the 20th IEEE/ACM international Conference on Automated software engineering, (2005) (Nov 7) 174-183
16. Bisht, P., Madhusudan, P., Venkatakrishnan, V. N.: CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks. ACM Transactions on Information and System Security (TISSEC), 13(2), (2010) (Mar 5) 1-39
17. Thomas, S., Williams, L.: Using automated fix generation to secure SQL statements. In Third International Workshop on Software Engineering for Secure Systems, IEEE, (SESS'07: ICSE Workshops 2007), (2007)(May 20) 9-9
18. Xiao, Z., Zhou, Z., Yang, W., Deng, C.: An approach for SQL injection detection based on behavior and response analysis. In2017 IEEE 9th International Conference on Communication Software and Networks (ICCSN), IEEE, (2017) (May 6) 1437-1442
19. Hasan, M., Balbahaith, Z., Tarique, M.: Detection of SQL injection attacks: a machine learning approach. In2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA) (2019) Nov 19 (pp. 1-6). IEEE
20. Choi, J., Kim, H., Choi, C., Kim, P.: Efficient malicious code detection using n-gram analysis and SVM. In2011 14th International Conference on Network-Based Information Systems,IEEE, (2011) (Sep 7) 618-621
21. Joshi, A., Geetha, V.: SQL Injection detection using machine learning. In2014 international conference on control, instrumentation, communication and computational technologies (ICCICCT),IEEE, (2014) (Jul 10) 1111-1115
22. Kamtuo, K., Soomlek, C.: Machine Learning for SQL injection prevention on server-side scripting. In2016 International Computer Science and Engineering Conference (ICSEC), IEEE, (2016) (Dec 14) 1-6
23. McWhirter, P. R., Kifayat, K., Shi, Q., Askwith, B.: SQL Injection Attack classification through the feature extraction of SQL query strings using a Gap-Weighted String Subsequence Kernel. Journal of information security and applications, 40, (2018) (Jun 1) 199-216. https://doi.org/10.1016/j.jisa.2018.04.001
24. Li, Q., Wang, F., Wang, J., Li, W.: LSTM-based SQL injection detection method for intelligent transportation system. IEEE Transactions on Vehicular Technology, 68 (5), (2019) (Jan 17) 4182-91. https://doi.org/10.1109/TVT.2019.2893675
25. Luo, A., Huang, W., Fan, W.: A CNN-based Approach to the Detection of SQL Injection Attacks. In 2019 IEEE/ACIS 18th International Conference on Computer and Information Science (ICIS), IEEE, (2019) (Jun 17) 320-324
26. Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., Schmidhuber, J.: LSTM: A search space odyssey. IEEE transactions on neural networks and learning systems, 28 (10), (2016) (Jul 8) 2222-32. https://doi.org/10.1109/TNNLS.2016.2582924
27. Fang, Y., Qiu, Y., Liu, L., Huang, C.: Detecting webshell based on random forest with fasttext. InProceedings of the 2018 International Conference on Computing and Artificial Intelligence, (2018) (Mar 12) 52-56
28. Qin, B., Wang, Y., Ma, C.: API call based ransomware dynamic detection approach using textCNN. In2020 International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE), IEEE, (2020) (Jun 12) 162-166
29. Li, Q., Li, W., Wang, J., Cheng, M.: A SQL injection detection method based on adaptive deep forest. IEEE Access, 7:145385, (2019) (Oct 1); 94
30. Ribeiro, M. T., Singh, S., Guestrin, C. : " Why should i trust you?" Explaining the predictions of any classifier. In Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining, (2016) (August) 1135-1144