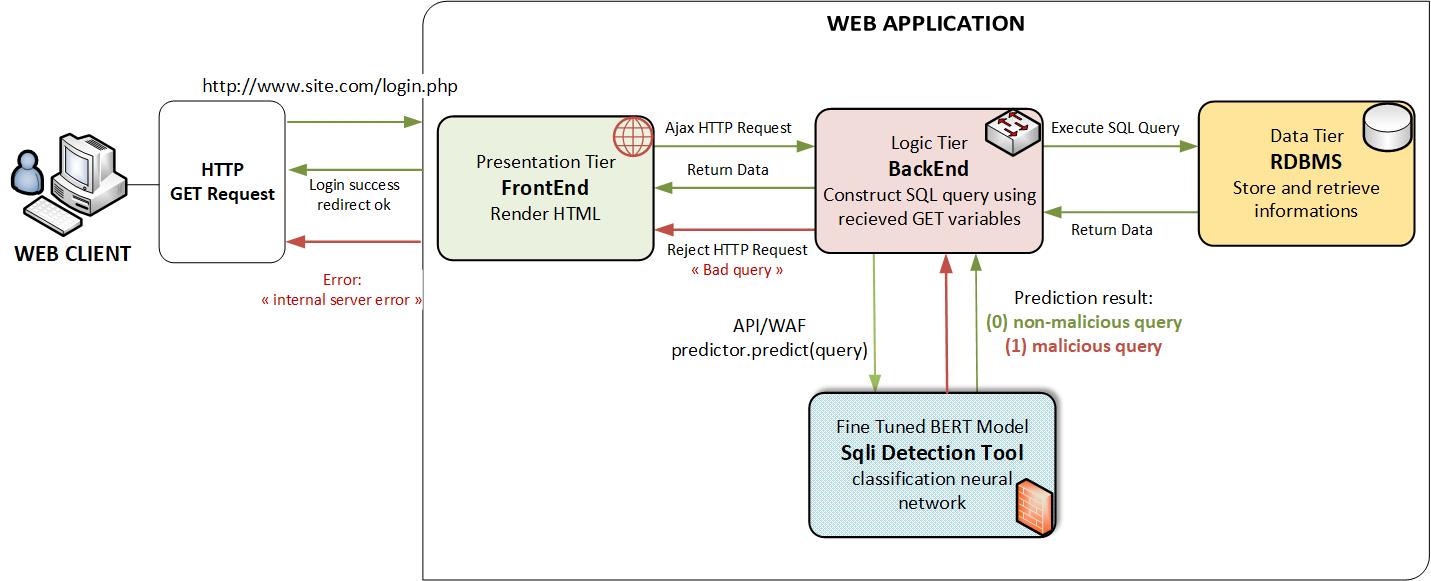
# Chapter 03 : conception and implementation :

1. **Introduction**

In this chapter, we will discuss the design and implementation of our system for detecting SQL injections using deep learning. We will describe conception and the materials used, including the dataset and the type of deep learning architecture employed. Furthermore, we will cover the preprocessing steps taken to ensure the accuracy and efficiency of our system. By detailing our approach to design and implementation, we aim to provide a comprehensive understanding of our methodology for detecting SQL injections through the use of deep learning.

1. **General conception of the solution**

 The deep learning model that we have developed will be used for detecting SQL injection attacks in web applications. The model is based on the Bidirectional Encoder Representations from Transformers (BERT) architecture, which has been fine-tuned on a dataset of SQL injection attacks. The model is designed to function as a middleware layer (API or a WAF) between the web application and the database server as shown in the following Figure.

**Figure:** Sql injection Detection Tool conception and architecture.

Our **Sqli Detection Model** analyzes incoming queries and detects any suspicious patterns that may indicate an SQL injection attack, it can be used on Web Application Firewalls (WAF) or as an API. Once the Sql injection is detected, the tool can either block the request or alert the system administrator, depending on the configuration. The model is highly scalable and can handle a large number of requests in real-time, making it suitable for deployment in high-traffic web applications.

* 1. **BERT for SQL Injection Detection**

BERT (Bidirectional Encoder Representations from Transformers) is a state-of-the-art language model that was introduced by Google in 2018 [6]. It uses a transformer-based neural network architecture that is capable of learning complex language representations from large amounts of unlabeled text data. BERT has achieved remarkable performance on a wide range of natural language processing (NLP) tasks, including question answering, sentiment analysis, and text classification.

* + 1. **Why BERT was chosen**

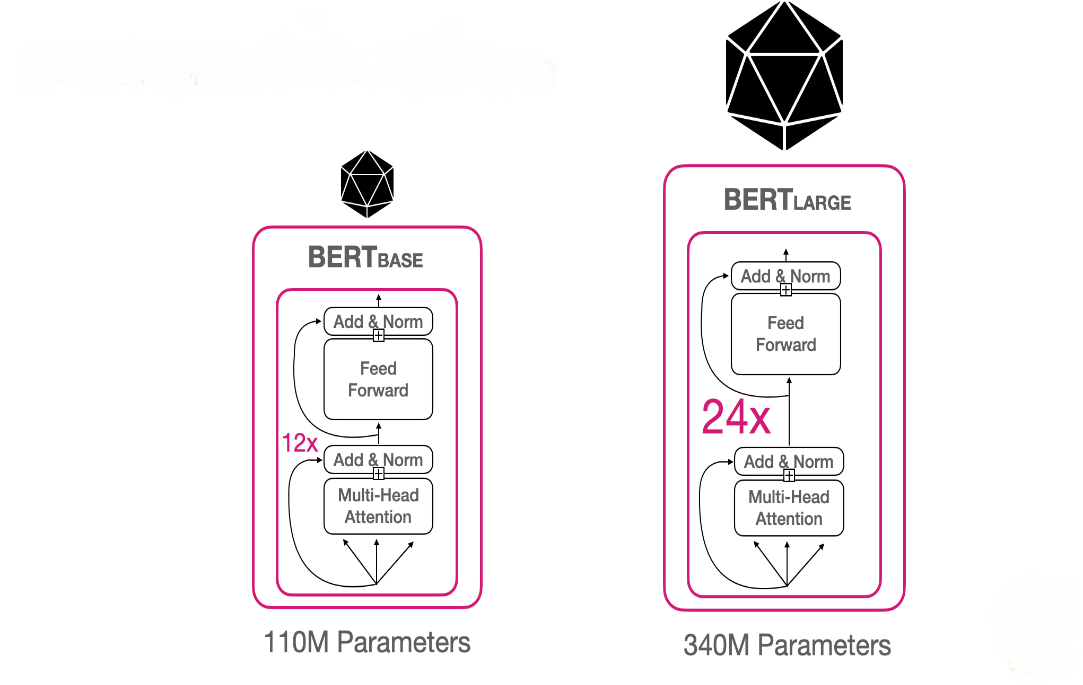
In the context of SQL injection detection, BERT was chosen for its ability to learn rich representations of text data, including queries, comments, and other textual elements that are typically associated with SQL injection attacks. BERT has been shown to outperform other state-of-the-art models on various NLP tasks [7], making it a promising candidate for SQL injection detection. Its transformer-based architecture allows it to process input sequences in a bidirectional manner and generate contextualized word representations. Fine-tuning BERT on a labeled dataset of SQL queries allowed us to develop a model that can detect SQL injection attacks with high accuracy.

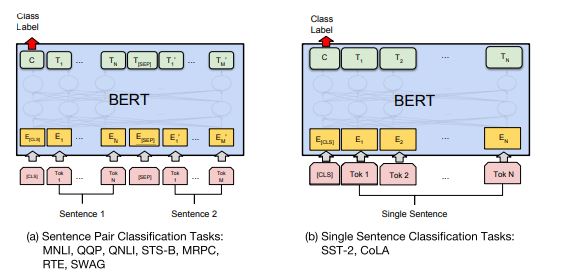
The architecture of BERT is based on the transformer network, which uses self-attention mechanisms to process input sequences. In BERT, a bidirectional transformer encoder is used to generate contextualized word representations. These representations are then fed into a classification layer that predicts whether a given query is vulnerable to SQL injection.

BERT has two variants: BERT-base and BERT-large, which differ in the number of layers and parameters. BERT-base has 12 transformer layers and 110 million parameters, while BERT-large has 24 transformer layers and 340 million parameters [6].

Both variants can be fine-tuned on specific NLP tasks using transfer learning.

* + 1. **Fine-tuning BERT for SQL Injection Detection:**

 To use BERT for SQL injection detection, we fine-tuned the pre-trained BERT model on a labeled dataset of SQL queries.



During fine-tuning, the model was trained to predict whether a given query was vulnerable to SQL injection or not. The fine-tuning process involved adjusting the weights of the classification layer while keeping the weights of the pre-trained BERT layers fixed.

1. **Presentation of development tools**
   1. **Programming language**
      1. **Python**

Python is a popular programming language in the field of machine learning and artificial intelligence due to its simple syntax, extensive libraries, and ease of use. Python provides a variety of libraries for machine learning such as TensorFlow, PyTorch, and Keras, making it a go-to language for many data scientists and machine learning practitioners. Its libraries provide an extensive range of functionalities, from data preprocessing to complex neural network architectures.[1]

Furthermore, Python's community is continuously contributing to its open-source libraries, ensuring a broad range of features and capabilities. Python is also known for its versatility as it can be used not only for machine learning but also for web development, scientific computing, and data analysis. However, it is important to note that while Python is a popular choice, it is not the only programming language used in machine learning. Other languages, such as R and Java, are also used for machine learning tasks.[2]

* 1. **Development environment**
     1. **Google Colab Pro**

Since training deep learning models often requires high-performance hardware or can be time-consuming, we selected Google Colab Pro as the platform for our project. Google Colab Pro is a cloud-based service that provides access to a powerful GPU and RAM, which allows us to train our model in a reasonable amount of time.

Specifically, Google Colab Pro provides access to a Tesla K80 GPU, which has 12 GB of GDDR5 VRAM and 4992 CUDA cores. This GPU is suitable for training deep learning models with moderate to high computational requirements.

In addition, **Google Colab** **Pro** also provides access to TPUs, which are specifically designed for accelerating deep learning computations. TPUs are available for users on a case-by-case basis and require a separate application process.

The TPU offered by Google Colab Pro is the TPU v3-8, which has 8 TPU cores and 64 GB of High Bandwidth Memory (HBM). This TPU is designed for high-throughput deep learning workloads and can accelerate training times by orders of magnitude compared to a traditional GPU.

Overall, the availability of both GPU and TPU on Google Colab Pro made it a suitable platform for our deep learning project, enabling us to train and test our model efficiently and effectively."

Additionally, Google Colab Pro provides a user-friendly interface that allows us to write and execute our code using Jupyter notebooks. This platform also offers other useful features such as version control, collaboration tools, and cloud storage for our data and code.

* + 1. **Jupyter notebook :**

Jupyter notebook is an open-source web application that allows users to create and share documents that contain live code, equations, visualizations, and narrative text. It supports over 40 programming languages, including Python, R, and Julia, making it a versatile tool for data analysis and machine learning.[3]

Jupyter notebooks are interactive and allow users to execute code in a step-by-step manner, making it easy to debug and analyze results. They also support the use of Markdown, a markup language for text formatting, making it easy to create readable and well-structured documents. Jupyter notebooks are widely used in the data science community due to their flexibility, interactivity, and ease of use.[4]

1. **Dataset :**

In order to train a successful and effective deep learning model, the dataset must be carefully processed, to achieve this, we needed to find a Dataset consisting of samples divided into two classes: "malicious queries" and "non-malicious queries". By using this Dataset, the model can learn to distinguish between the two classes and accurately identify SQL injection attacks.

During our search for a suitable Datasets, we came across the "SQL Injection Dataset" list on the Kaggle platform [5]. We found a list of datasets that was created by “Syed Hussain” that contain Three (03) versions:

* **SQLi.csv** (723.15 kB) contains **3951** samples with **78%** classified as normal queries and **28%** as malicious queries.
* **SQLiV2.csv** (3.61 MB) contains **33726** samples with **66%** classified as normal queries and **34%** as malicious queries.
* **SQLiV3.csv** (2.32 MB) contains **30873** samples with **62%** classified as normal queries and **37%** as malicious queries and **1%** as other.

At first vision we choose the **SQLiV2** Dataset because of the size of samples in it in comparison to others, we trained our model with it in the first time, but unfortunately the results were not satisfied while predicting normal queries. After analyzing the situation, we found that there was no normal queries in the dataset, only free text in place flagged as normal queries.

Because of the big issue in **SQLiV2.csv**, we tried the **SQLiV3.csv** Dataset**.** After reviewing it, we identified certain deficiencies that need to be cleared using some preprocessing steps like:

* Remove any unnecessary Strings or characters in the sql statements. We found two comas (,,) in the end of the queries that was removed using a Register-Expression technique using a python script.
* Remove not valid empty columns. We need only two valid columns, the **Statement** and the **Label**, in that Dataset we found two empty and not valid columns that was removed using the Office Excel Software
* Remove empty and free text Rows. We found many empty and free text rows that was removed using the Office Excel Software
* Remove wrong SQL queries. We found not valid SQL statements that was identified and cleared manually

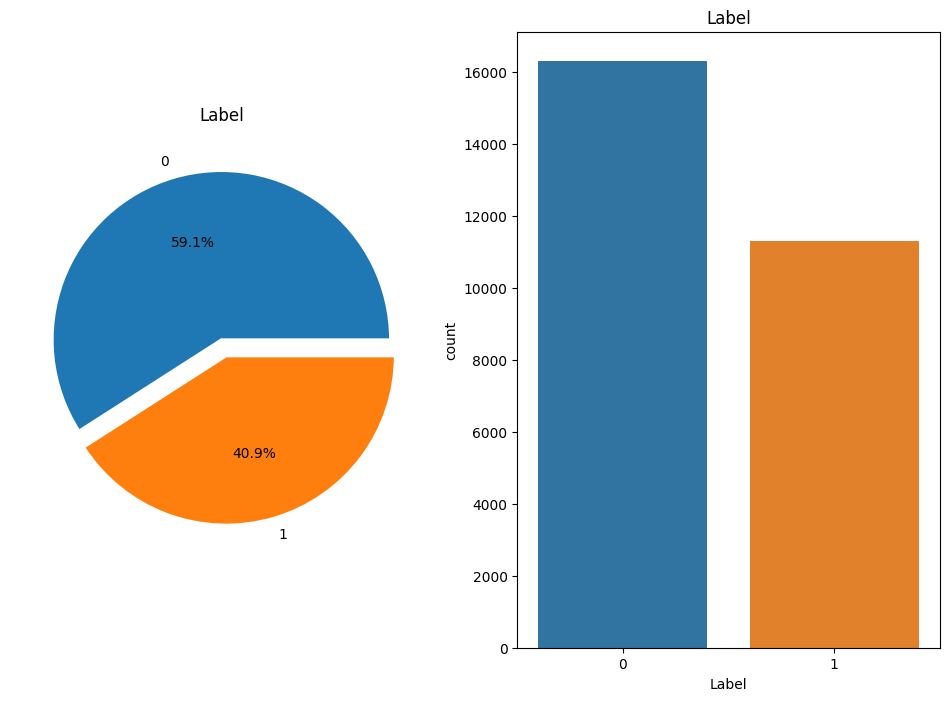
By applying the above-preprocessed steps, we generated a new Dataset with **50%** classified as normal queries and **50%** as malicious queries.

In real life, web applications receives a number of normal queries more than the malicious queries. Basing on this rule, we decided to increase the number of normal queries by generating random SQL statements using an SQL generator that was created with Python.

Our SQL generator use Four (04) separated python files:

* **sql\_SELECT\_generator.py**, generates 2000 random SELECT statements with random data samples.
* **sql\_INSERT\_generator.py**, generates 1000 random INSERT statements with random data samples.
* **sql\_UPDATE\_generator.py,** generates 1000 random UPDATE statements with random data samples.
* **sql\_DELETE\_generator.py,** generates 1000 random DELETE statements with random data samples.

After we added the generated SQL statements to the Dataset, we finally came up with a partitioned version that has **16307** **(59.1%)** "non-malicious queries" against **11298 (40.9%)** "malicious queries".



After clearing the different deficiencies and increased the number of normal queries, we came up with a new valid preprocessed Dataset that has sufficient diversity in both categories and ready to be used in the next step. We ensured that our deep learning model would only train on preprocessed samples that belongs to one of the two classes and could, therefore, learn to differentiate between malicious and non-malicious queries effectively.

1. **Code And Implementation**

The implementation of a deep learning model can be a challenging task, especially when it involves complex architectures and large datasets. In this section, we will present the code and implementation details of our SQL injection detection model based on the BERT architecture. We will describe the steps involved in data preprocessing, model training, and testing, along with code snippets and visualizations of our results.

* 1. **Import necessary libraries and tools :**

Importing necessary libraries and tools is an essential step when working on any data science project. These libraries provide functionality for common data manipulation, exploration, and deep learning tasks. In this project, we used a number of libraries to preprocess and classify text data as the following:

**numpy**: NumPy is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices. We used this library to work with numerical data in our project.

**ktrain**: ktrain is a lightweight wrapper for the Keras deep learning library to help simplify the training of neural networks. We used this library to build and train our text classification model.

**pandas**: Pandas is a library used for data manipulation and analysis. It provides data structures for efficiently storing and manipulating large datasets. We used this library to read in and preprocess our text data.

**chardet**: Chardet is a Python library used for character encoding detection. We used this library to ensure that our text data is properly encoded before processing.

**matplotlib**: Matplotlib is a data visualization library used for creating static, animated, and interactive visualizations in Python. We used this library to create visualizations of our data and model performance.

**sklearn**: Scikit-learn is a library used for machine learning tasks such as classification, regression, and clustering. We used this library to split our data into training and testing sets.

**seaborn**: Seaborn is a data visualization library based on matplotlib. It provides a high-level interface for creating informative and attractive statistical graphics. We used this library to visualize the distribution of our data.

**pickle**: Pickle is a Python module used for serializing and de-serializing Python objects. We used this library to save our trained model for future use.

* 1. **Reading the Dataset**

# Detect the encoding of the CSV file

with open('SQLiV3\_4.csv', 'rb') as f:

    result = chardet.detect(f.read())

# Read the CSV file into a pandas dataframe, using the detected encoding

df = pd.read\_csv('SQLiV3\_4.csv', sep=',',  encoding=result['encoding'])

# Display the first rows of the dataframe

print(df.head(10))

Sentence Label

0 " or pg\_sleep ( \_\_TIME\_\_ ) -- 1

1 AND 1 = utl\_inaddr.get\_host\_address ( ... 1

2 select \* from users where id = '1' or @ @1 ... 1

3 select \* from users where id = 1 or 1#" ( ... 1

4 select name from syscolumns where id = ... 1

5 select \* from users where id = 1 +$+ or 1 =... 1

6 1; ( load\_file ( char ( 47,101,116,99,47... 1

7 select \* from users where id = '1' or ||/1 ... 1

8 select \* from users where id = '1' or \.<\ ... 1

9 ? or 1 = 1 -- 1

The code section above is responsible for reading a CSV file named "SQLiV3\_4.csv" into a Pandas DataFrame object. The first step is to detect the encoding of the CSV file using the chardet library.

The with open ('SQLiV3\_4.csv', 'rb') as f: statement opens the CSV file in read mode and reads its content using the *rb mode* (read binary). The content is then passed to the chardet.detect() method to detect the encoding of the file. The result variable holds the detected encoding.

The next step is to read the CSV file into a Pandas DataFrame using the pd.read\_csv() method. The sep=',' parameter specifies that the CSV file is comma-separated. The encoding=result['encoding'] parameter uses the encoding detected earlier to read the file content. The resulting DataFrame is stored in the df variable.

Finally, the print(df.head(10)) statement is used to display the first ten (10) rows of the DataFrame on the console. This is to ensure that the file has been correctly loaded into the DataFrame.

* 1. **Data exploration and visualization :**

df.shape

(27605, 2)

print("----------------------------------------------------")

df.info()

print("----------------------------------------------------")

----------------------------------------------------

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 27605 entries, 0 to 27604

Data columns (total 2 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 Sentence 27605 non-null object

1 Label 27605 non-null int64

dtypes: int64(1), object(1)

memory usage: 431.5+ KB

----------------------------------------------------

df['Label'].describe()

count 27605.000000

mean 0.409274

std 0.491709

min 0.000000

25% 0.000000

50% 0.000000

75% 1.000000

max 1.000000

Name: Label, dtype: float64

# Count NaN label values.

df['Label'].isna().sum()

0

# Get unique values existing in the label column.

unique\_label = pd.unique(df['Label'])

print(unique\_label)

[1 0]

# Count how much label are 1.

df[df['Label'] == 1].count()

Sentence 11298

Label 11298

dtype: int64

# Count how much label are 0.

df[df['Label'] == 0].count()

Sentence 16307

Label 16307

dtype: int64

# Distribution of the label

import matplotlib.colors as mcolors

f, ax = plt.subplots(1, 2, figsize=(12, 8))

colors = list(mcolors.TABLEAU\_COLORS.values())

df['Label'].value\_counts().plot.pie(explode=[0, 0.1], autopct='%1.1f%%', ax=ax[0], wedgeprops={'edgecolor': 'white', 'linewidth': 2, 'antialiased': True}, colors=colors)

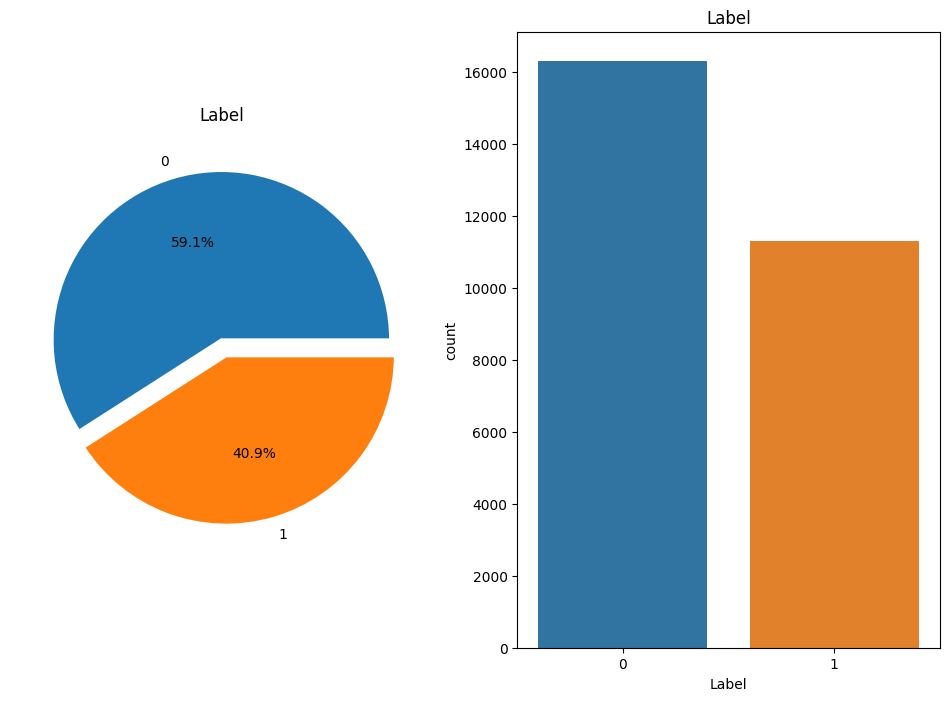
ax[0].set\_title('Label')

ax[0].set\_ylabel('')

sns.countplot(x='Label', data=df, ax=ax[1], palette=colors)

ax[1].set\_title('Label')

plt.show()



The Data exploration and visualization section aims to understand the dataset we are working with. First, we get the shape of the data using df.shape, which returns the number of rows and columns in the dataset. We then use df.info() to get a concise summary of the dataframe. We also check if there are any missing values in the Label column by using the isna() method and summing up the values with sum(). To get a better understanding of the Label column, we use the describe() method, which provides a summary of the distribution of the labels.

Next, we count the number of occurrences of each unique value in the Label column using the value\_counts() method. We then create a pie chart and a bar plot to visualize the distribution of the labels. The pie chart shows the proportion of each label value, while the bar plot shows the count of each label value. We use the plt.subplots() function to create a 1x2 figure, set the size of the figure with figsize, and assign the two subplots to ax[0] and ax[1]. We also set the colors of the plots using mcolors.TABLEAU\_COLORS.values(). Finally, we use plt.show() to display the plots.

* 1. **Split data into training and testing sets and Preprocess data for BERT model:**

sentences = df['Sentence'].tolist()

labels = df['Label'].tolist()

sentences\_train, sentences\_test, labels\_train, labels\_test = train\_test\_split(sentences, labels, test\_size=0.2, random\_state=1500)

(x\_train, y\_train), (x\_test, y\_test), preproc = text.texts\_from\_array(sentences\_train, labels\_train, preprocess\_mode='bert', maxlen=500,class\_names=list(set(labels\_train)))

preprocessing train...

language: en

done.

Is Multi-Label? False

preprocessing test...

language: en

done.

task: text classification

In this section, the code is splitting the data into training and testing sets. It starts by extracting the 'Sentence' and 'Label' columns from the pandas dataframe and creating two lists out of them. Then, it uses the 'train\_test\_split' method from the scikit-learn library to split the sentences and labels into training and testing sets. The test\_size parameter is set to 0.2, which means that 20% of the data is used for testing and the remaining 80% is used for training. The random\_state parameter is set to 1500 to ensure that the data is split in the same way every time the code is run.

Next, the 'texts\_from\_array' method from the ktrain library is used to convert the training and testing sets into arrays that can be used for training a BERT model. The 'preprocess\_mode' parameter is set to 'bert', which means that the data will be preprocessed according to the requirements of the BERT model. The 'maxlen' parameter is set to 500, which means that the maximum length of a sentence is set to 500 tokens. The 'class\_names' parameter is set to the unique labels in the training set, which will be used to create a mapping between the label values and their corresponding names. The method returns four variables: x\_train and x\_test are the preprocessed arrays of sentences, y\_train and y\_test are the label arrays, and preproc is a preprocessor object that was used to preprocess the data.

* 1. **Build BERT model :**

model = text.text\_classifier('bert', (x\_train, y\_train), preproc=preproc)

In this code line, a BERT (Bidirectional Encoder Representations from Transformers) model is being built using the text\_classifier function from the ktrain library. This function takes the name of the model as the first argument (in this case, 'bert'), the training data (x\_train, y\_train), and the preprocessing object preproc as input.

The BERT model is a neural network architecture designed to process text data and learn representations of the meaning of words and the relationships between them. This model uses a deep architecture of transformer blocks that can handle long sequences of text and capture both local and global dependencies between words in a sentence.

The ktrain library provides a simplified interface for training and fine-tuning BERT models for text classification tasks, without requiring extensive knowledge of deep learning and natural language processing.

* 1. **Train BERT model**

learner = ktrain.get\_learner(model=model,

                             train\_data=(x\_train, y\_train),

                             val\_data=(x\_test, y\_test),

                             batch\_size=6)

learner.lr\_find()

simulating training for different learning rates... this may take a few moments...

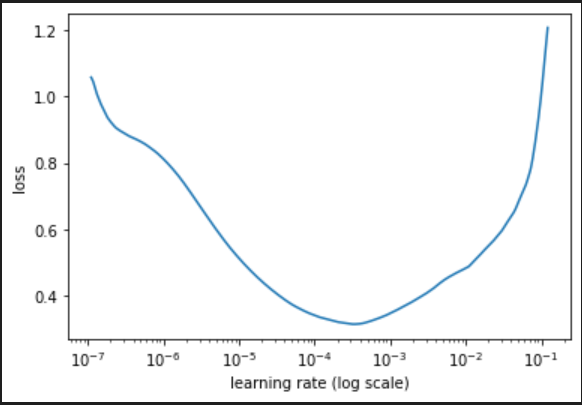
Epoch 1/1024

2712/2712 [==============================] - 184s 58ms/step - loss: 1.7540 - accuracy: 0.6912

done.

Please invoke the Learner.lr\_plot() method to visually inspect the loss plot to help identify the maximal learning rate associated with falling loss.

learner.lr\_plot()



learner.fit\_onecycle(lr=2e-5, epochs=4)

begin training using onecycle policy with max lr of 2e-05...

Epoch 1/4

2712/2712 [==============================] - 342s 119ms/step - loss: 0.0301 - accuracy: 0.9897 - val\_loss: 0.0061 - val\_accuracy: 0.9972

Epoch 2/4

2712/2712 [==============================] - 317s 117ms/step - loss: 0.0065 - accuracy: 0.9985 - val\_loss: 0.0021 - val\_accuracy: 0.9994

Epoch 3/4

2712/2712 [==============================] - 317s 117ms/step - loss: 2.8025e-04 - accuracy: 0.9999 - val\_loss: 0.0020 - val\_accuracy: 0.9994

Epoch 4/4

2712/2712 [==============================] - 317s 117ms/step - loss: 6.5042e-06 - accuracy: 1.0000 - val\_loss: 0.0022 - val\_accuracy: 0.9994

In the "Train BERT model" section, the model is trained using ktrain's get\_learner method and the one-cycle policy, which involves training the model with a learning rate that linearly increases for the first half of the epochs and then linearly decreases for the second half of the epochs.

First, get\_learner is called, which creates a Learner object for the specified model and training data (x\_train, y\_train). It also includes validation data (x\_test, y\_test) to monitor the model's performance during training, and the batch\_size is set to 6.

Then, the fit\_onecycle method is called on the learner object. This method trains the model using the one-cycle policy for a specified lr (learning rate) and number of epochs. In this case, the learning rate is set to 2e-5 and the number of epochs is 4.

During training, the model's loss and accuracy are displayed for each epoch. The goal is to minimize the loss and maximize the accuracy on the validation set to create a well-performing model.

* 1. **Choice of hyperparameters**

The choice of hyperparameters plays a critical role in the design and performance of our system for detecting SQL injections using deep learning. In this section, we discuss the key hyperparameters we selected and the reasoning behind these choices.

**Preprocessing Hyperparameters:**

**Max Length:** We set the maximum length of input sequences to 500. This value was determined based on the analysis of the dataset, ensuring that most SQL injection statements can be adequately captured within this limit.

**Preprocess Mode:** We utilized the "bert" preprocess mode, which applies BERT-specific tokenization and formatting to the input text data. This mode is specifically designed for BERT models and helps optimize the preprocessing step, enhancing the model's ability to understand the context and semantics of the text.

**Model Training Hyperparameters:**

**Batch Size:** We chose a batch size of 6, which determines the number of training samples processed in each iteration. This value strikes a balance between training speed and memory consumption, considering the available computational resources and dataset size.

**Learning Rate:** We set the learning rate to 2e-5, a value recommended by Google for fine-tuning BERT models. This learning rate choice enables effective convergence during training while minimizing the risk of overshooting the optimal solution.

**Number of Epochs:** The model was trained for 4 epochs, meaning the entire training dataset was processed four times. This number of epochs allows the model to learn patterns and generalize well to the dataset without over-fitting.

**Data Split Hyperparameters:**

**Test Size:** We partitioned the dataset into training and testing sets using a test size of 0.2 (20%). This split allocates 80% of the data for training and 20% for testing, ensuring a sufficient amount of data for evaluation while preserving a sizable training set.

By carefully selecting these Hyperparameters, including the test size, max length, preprocess mode, batch size, learning rate, and epochs, we aimed to optimize the performance of our system for detecting SQL injections. These choices were based on prior knowledge, best practices, and recommendations from Google's BERT documentation. The hyperparameters collectively contribute to the effectiveness and accuracy of our model in identifying SQL injection attacks.

* 1. **Evaluate BERT model:**

learner.validate(class\_names=preproc.get\_classes())

57/57 [==============================] - 13s 151ms/step

precision recall f1-score support

0 1.00 1.00 1.00 928

1 1.00 1.00 1.00 880

accuracy 1.00 1808

macro avg 1.00 1.00 1.00 1808

weighted avg 1.00 1.00 1.00 1808

array([[927, 1],

[ 0, 880]])

The code above is used to evaluate the performance of the trained BERT model using the validate() method from the ktrain library.

First, we create a learner object using the trained model, train\_data, val\_data, and batch\_size parameters.

Then, we call the validate() method on the learner object. This method returns a dictionary containing several metrics for each class, such as accuracy, precision, recall, and F1-score.

By passing the class\_names parameter, we can display the metrics for each class in a readable format.

Overall, the validate() method helps us to assess the performance of our BERT model on the validation data and to identify any potential issues, such as overfitting or underfitting.

* 1. **View top losses**

learner.view\_top\_losses(n=30, preproc=preproc)

57/57 [==============================] - 9s 151ms/step

----------

id:263 | loss:3.97 | true:0 | pred:1)

[CLS] create table if not exists ` z ##c ##f \_ submit ##log ##s ` ( ` id ` int ( 6 ) unsigned not null auto \_ inc ##rem ##ent , ` cr ##ms ##ub ##mit ##log ##sta ##tus ` var ##cha ##r ( 20 ) default null , ` cr ##ms ##ub ##mit ##log ##des ##cr ##ib ##tion ` long ##text not null , ` cr ##ms ##ub ##mit ##form ##type ` var ##cha ##r ( 30 ) not null , primary key ( id ) ) engine = inn ##od ##b default char ##set = ut ##f ##8 [SEP]

This code snippet uses the view\_top\_losses() method of the ktrain library's learner object to display the top losses of the trained BERT model.

When a model makes predictions, it assigns a probability score to each class label. The view\_top\_losses() method shows the examples where the model had the most significant difference between the predicted probability and the true label.

The n parameter specifies the number of examples to display. In this case, the top 30 examples will be shown. The preproc parameter is the preprocessor used to process the data before being fed into the model.

By examining the examples with the highest losses, we can identify pattern errors made by the model and possibly adjust the model or improve the training data to address these errors.

* 1. **Make predictions with BERT model**

predictor = ktrain.get\_predictor(learner.model, preproc)

# make predictions

samples = [

     "1'; DROP TABLE users;--;",

     "INSERT INTO users (username, password) VALUES ('testuser', 'testpassword'); DROP TABLE users;",

     "SELECT COUNT(\*) FROM users WHERE username = 'admin' OR 1 = 1",

     "UPDATE users SET password = 'newpassword' WHERE username = 'admin';" ,

     "select \* from generate\_series  (  5980,5980,case when   (  5980  =  5063  )   then 1 else 0 end  )   limit 1--",

     "SELECT TOP 3 \* FROM growth SELECT \* FROM catch 3SELECT \* FROM mainly",

     "SELECT \* FROM users WHERE username = '' OR 1:1",

     "INSERT INTO column ( white, does, certain, curious, first, our )  VALUES  ( 'rose', 'anyone'. close', 'remove', 'force', 'feet', 'fell' )",

     "SELECT \* FROM users WHERE username = '' OR 1=2 --' AND password = 'input\_password'"]

prediction = predictor.predict(samples)

print(prediction)

1/1 [==============================] - 0s 74ms/step [1, 0, 1, 0, 1, 0, 1, 0, 1]

This code block shows how to use the BERT model trained to detect SQL injection attacks to make predictions on new data. The get\_predictor() function loads the trained BERT model and pre-processing pipeline, which are necessary to make predictions on new data. The predict() function takes a single input (in this case, a string containing a SQL injection attack) and returns the predicted label. The code then prints the predicted label for the input samples array that is likely to be flagged as a SQL injection attacks or not. This code can be extended to make predictions on a large number of SQL queries and provide insights on the overall security of the system being tested.

**6. Conclusion:**

In this chapter, we have described the design and implementation of a system for detecting SQL injections using deep learning. We utilized a dataset of SQL injection attacks and non-attacks to train a BERT model for classification. The preprocessing steps involved converting the text data into a format suitable for BERT model input and splitting the data into training and testing sets. Our results showed that the trained model was effective in detecting SQL injection attacks with a high degree of accuracy. By sharing the details of our approach, we have provided a comprehensive understanding of how deep learning can be utilized for SQL injection detection.

**References:**

Géron, A. (2019). Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media, Inc.[1]

Raschka, S., & Mirjalili, V. (2019). Python Machine Learning: Machine Learning and Deep Learning with Python, Scikit-Learn, and TensorFlow 2. Packt Publishing.[2]

Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B. E., Bussonnier, M., Frederic, J., ... & Ivanov, P. (2016). Jupyter Notebooks—a publishing format for reproducible computational workflows. In ELPUB (pp. 87-90).[3]

Rule, A., Tabard, A., & Hollan, J. D. (2018). Rapid prototyping interactive data visualizations with Jupyter notebooks. IEEE transactions on visualization and computer graphics, 24(1), 421-431.[4]

https://www.kaggle.com/datasets/syedsaqlainhussain/sql-injection-dataset [5]

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pages 4171–4186.[6]

Yan Zhang, Huan Zhang, Xuanjing Huang, and Jian Zhao. 2020. Benchmarking Neural Network Models for SQL Injection Detection. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 1772–1783.[7]