BERT-Based Spam Detection

Written by Ivan Radonjic

June 2024

Introduction

In this document, we present a comprehensive explanation of a neural network model leveraging the power of BERT for text classification tasks. BERT, a state-of-the-art language representation model developed by Google, excels in understanding the context of words in a sentence, making it highly effective for various natural language processing (NLP) applications.

This project focuses on classifying spam emails, aiming to accurately distinguish between spam and legitimate messages. By using BERT, the model can effectively understand the nuances in email content, enhancing its ability to identify spam with high precision and recall.

Code Breakdown

```
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_text as text

import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report

from matplotlib import pyplot as plt
import seaborn as sn
```

Figure 1: Import Libraries

First, we need to import the essential libraries required to complete this task. We will use TensorFlow to construct, train, and test our model. Pandas will facilitate dataset manipulation, allowing us to handle and preprocess the data efficiently. NumPy will be utilized for array operations, providing support for various mathematical functions. SciKit-learn offers valuable tools for splitting our dataset into training and testing sets and for evaluating our model's performance. Additionally, Matplotlib and Seaborn will be employed to create visualizations, enabling us to analyze and interpret the performance metrics of our model effectively.

```
df = pd.read_csv("spam.csv")
df.head(5)
```

	Category	Message
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...

Figure 2: Import Dataset

In Figure 2, we utilize Pandas to import data from a CSV file into a DataFrame, which allows us to efficiently slice, manipulate, and analyze our dataset. This process enables us to handle the data in a structured format, making it easier to perform various preprocessing tasks and prepare the data for model training and evaluation.

When addressing a binary classification problem, it is crucial to have a balanced number of training examples for each class. This ensures that the model does not become biased toward one class over the other. Next, we will verify the balance of our dataset to ensure it contains an equal number of examples for both output cases.

```
df.groupby('Category').describe()
```

	count	unique	top	freq
Category				
ham	4825	4516	Sorry, I'll call later	30
spam	747	641	Please call our customer service representativ...	4

Figure 3: Imbalanced Dataset

In Figure 3 above, we observe a significant imbalance in the dataset, with 747 spam email examples and 4825 non-spam email examples. To address this issue, we will create a new dataset that includes all spam email examples and an equal number of non-spam email examples, selected randomly. This approach will help to balance the classes and ensure that our model trains effectively without bias towards the majority class.

```
#Create new dataframe
df_spam = df[df['Category'] == 'spam']
print('Shape Spam')
print(df_spam.shape)

#Create new dataframe
df_ham = df[df['Category'] == 'ham']
print('\nShape non-Spam')
print(df_ham.shape)

#sample a random 747 from the ham dataframe to have an even amount of training data
df_ham_downsampled = df_ham.sample(df_spam.shape[0])
print('\nShape non-Spam Downsized')
print(df_ham_downsampled.shape)

#Concantenate
df_balanced = pd.concat([df_spam, df_ham_downsampled])
print('\nConcatenated Data; Spam + non-Spam Downsized')
print(df_balanced.shape)
```

```
Shape Spam
(747, 2)

Shape non-Spam
(4825, 2)

Shape non-Spam Downsized
(747, 2)

Concatenated Data; Spam + non-Spam Downsized
(1494, 2)
```

Figure 4: Downsize non-spam and concatenate even dataset

To address the issue of an imbalanced dataset, we utilized the Pandas library to create two separate DataFrames: one for spam emails and one for non-spam emails. The first two outputs in Figure 4 show the sizes of each class, with 747 examples of spam emails and 4825 examples of non-spam emails. To balance the dataset, we limit the number of non-spam examples to match the number of spam examples. We achieve this by randomly sampling 747 instances from the non-spam DataFrame using the Pandas '.sample()' method. The third output in Figure 4 confirms that the downsampled non-spam dataset now contains 747 examples. Finally, we concatenate the two DataFrames to create a balanced dataset, resulting in a combined DataFrame with an equal number of examples for each class.

```
df_balanced['Category'].value_counts()
```

```
Category
spam    747
ham     747
Name: count, dtype: int64
```

```
df_balanced['spam'] = df_balanced['Category'].apply(lambda x: 1 if x == 'spam' else 0)
df_balanced.sample(5)
```

	Category	Message	spam
1205	spam	WIN a year supply of CDs 4 a store of ur choic...	1
892	ham	I am great princess! What are you thinking abo...	0
880	spam	U have a Secret Admirer who is looking 2 make ...	1
1769	ham	How. Its a little difficult but its a simple w...	0
123	spam	Todays Voda numbers ending 7548 are selected t...	1

Figure 5: Double Check Downsizing

In Figure 5, we verify the results of our previous operations. First, we use the Pandas .value_counts() function on the 'Category' column to confirm the number of instances for each class, ensuring that our dataset is balanced with 747 spam and 747 non-spam emails. Next, we create a new column called 'spam' to label each example as either 1 (spam) or 0 (non-spam). This is accomplished using the .apply() function, which assigns a 1 to rows where the 'Category' is

'spam' and a 0 to rows where the 'Category' is 'ham'. Finally, we inspect five random examples from the dataset to ensure that the labeling function was correctly applied. This step is crucial to confirm that our dataset is accurately labeled and ready for training our model.

```
X_train, X_test, y_train, y_test = train_test_split(df_balanced['Message'], df_balanced['spam'], stratify = df_balanced['spam'])
```

Figure 6: Split the data

This line of code int Figure 6 above splits the balanced dataset into training and testing sets using the train_test_split function from Scikit-learn. The stratify parameter ensures that the split maintains the same proportion of spam and non-spam examples in both the training and testing sets, preserving the class distribution.

In the following figure, we leverage pre-trained BERT models available through TensorFlow Hub to preprocess and encode text data. We define two Keras layers, bert_preprocess and bert_encoder, using URLs from TensorFlow Hub.

```
bert_preprocess = hub.KerasLayer("https://tfhub.dev/tensorflow/bert_en_uncased_preprocess/2")
bert_encoder = hub.KerasLayer("https://tfhub.dev/tensorflow/bert_en_uncased_L-12_H-768_A-12/3")

def sentence_embedding(sentences):
    preprocessed_text = bert_preprocess(sentences)
    return bert_encoder(preprocessed_text)['pooled_output']
```

Figure 7: BERT Layers

The bert_preprocess layer handles the preprocessing steps required for BERT, including tokenization, adding special tokens (such as [CLS] and [SEP]), and converting tokens to IDs. The bert_encoder layer is a BERT model that transforms the preprocessed text into dense vectors representing the input sentences. The sentence_embedding function accepts a list of sentences and passes them through the preprocessing layer, which prepares the text for encoding. The preprocessed text is then fed into the BERT encoder. From the encoder, we extract the 'pooled_output', which is a fixed-size tensor representing the entire input sequence. This output is essential for our downstream classification task, as it captures the semantic meaning of the input sentences in a high-dimensional space.

```
#Bert Layers
text_input = tf.keras.layers.Input(shape = (), dtype = tf.string, name = "text")
preprocessed_text = bert_preprocess(text_input)
outputs = bert_encoder(preprocessed_text)

#Neural Network Layers
Layer_1 = tf.keras.layers.Dropout(0.1, name = 'dropout')(outputs['pooled_output'])
Layer_2 = tf.keras.layers.Dense(1, activation = 'sigmoid', name = 'output')(Layer_1)

#Final Model
my_model = tf.keras.Model(inputs = [text_input], outputs = [Layer_2])
```

Figure 8: Model

Figure 8 illustrates the Python code used to construct our text classification model. The model begins with a text input layer that handles raw text data. When text passes through this layer, two key processes occur. First, the text is preprocessed—tokenized, input masks created, etc. Second, the bert_encoder transforms this preprocessed text into numerical vectors that represent the semantic meaning of the input.

Following the text preprocessing and encoding, the neural network component of the model begins with Layer_1. This layer applies a 10% dropout regularization to the pooled_output from BERT. Dropout regularization helps prevent overfitting by randomly deactivating a portion of the input units during the training phase, thus enhancing the model's generalization capabilities.

The subsequent layer, Layer_2, is a dense layer with a single neuron and a sigmoid activation function. This layer outputs a probability value between 0 and 1, indicating the likelihood that the input text is spam. The sigmoid activation function is ideal for binary classification tasks as it maps the input to a probability score.

The final model is defined by specifying the input and output layers, encapsulated within a TensorFlow Keras Model (my_model). This comprehensive architecture integrates raw text input, BERT-based preprocessing and encoding, dropout regularization, and binary classification. Together, these components enable the model to effectively distinguish between spam and legitimate emails, leveraging BERT's robust contextual understanding for high precision and recall.

```
Layer (type)                 Output Shape              Param #   Connected to
==================================================================================================
text (InputLayer)            [(None,)]                 0         []

keras_layer (KerasLayer)     {'input_mask': (None,     0         ['text[0][0]']
                             128),
                              'input_type_ids':
                             (None, 128),
                              'input_word_ids':
                             (None, 128)}

keras_layer_1 (KerasLayer)   {'pooled_output': (       109482241 ['keras_layer[0][0]',
                             None, 768),                          'keras_layer[0][1]',
                              'encoder_outputs':                  'keras_layer[0][2]']
                              [(None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768)],
                              'default': (None,
                             768),
                              'sequence_output':
                              (None, 128, 768)}

dropout (Dropout)            (None, 768)               0         ['keras_layer_1[0][13]']

output (Dense)               (None, 1)                 769       ['dropout[0][0]']

==================================================================================================
Total params: 109,483,010
Trainable params: 769
Non-trainable params: 109,482,241
__________________________________________________________________________________________________
```

Figure 9: Model Summary

Figure 9 above is the summary of our model, it provides gives an overview of a neural network designed for text classification, specifically utilizing BERT for encoding. The model comprises a total of 109,483,010 parameters, with only 769 being trainable. The vast majority of parameters belong to the pre-trained BERT model, which remains frozen during training to leverage its pre-learned knowledge. This architecture efficiently combines BERT's powerful text encoding capabilities with a straightforward neural network to achieve high accuracy in classifying emails as spam or non-spam.

```
my_model_metrics = [
    tf.keras.metrics.BinaryAccuracy(name = 'Binary Accuracy'),
    tf.keras.metrics.Precision(name = 'Precision'),
    tf.keras.metrics.Recall(name = 'Recall')
]

my_model.compile(
    optimizer = tf.keras.optimizers.Adam(),
    loss = tf.keras.losses.BinaryCrossentropy(),
    metrics = my_model_metrics
)
```

```
my_model.fit(X_train, y_train, epochs = 10)
```

```
Epoch 1/10
35/35 [==============================] - 113s 3s/step - loss: 0.6079 - Binary Accuracy: 0.6732 - Precision: 0.6720 - Recall: 0.6768
Epoch 2/10
35/35 [==============================] - 107s 3s/step - loss: 0.4949 - Binary Accuracy: 0.8018 - Precision: 0.7683 - Recall: 0.8643
Epoch 3/10
35/35 [==============================] - 107s 3s/step - loss: 0.4248 - Binary Accuracy: 0.8545 - Precision: 0.8428 - Recall: 0.8714
Epoch 4/10
35/35 [==============================] - 107s 3s/step - loss: 0.3813 - Binary Accuracy: 0.8786 - Precision: 0.8643 - Recall: 0.8982
Epoch 5/10
35/35 [==============================] - 107s 3s/step - loss: 0.3504 - Binary Accuracy: 0.8804 - Precision: 0.8660 - Recall: 0.9000
Epoch 6/10
35/35 [==============================] - 107s 3s/step - loss: 0.3299 - Binary Accuracy: 0.8991 - Precision: 0.8901 - Recall: 0.9107
Epoch 7/10
35/35 [==============================] - 107s 3s/step - loss: 0.3096 - Binary Accuracy: 0.9080 - Precision: 0.8946 - Recall: 0.9250
Epoch 8/10
35/35 [==============================] - 107s 3s/step - loss: 0.2996 - Binary Accuracy: 0.9000 - Precision: 0.8810 - Recall: 0.9250
Epoch 9/10
35/35 [==============================] - 107s 3s/step - loss: 0.2831 - Binary Accuracy: 0.9036 - Precision: 0.8993 - Recall: 0.9089
Epoch 10/10
35/35 [==============================] - 107s 3s/step - loss: 0.2729 - Binary Accuracy: 0.9143 - Precision: 0.9000 - Recall: 0.9321
<keras.callbacks.History at 0x2ab199150f0>
```

Figure 10: Model Metrics, Compiler and training

The training process for our text classification model, which leverages BERT for text encoding, involves compiling the model with specific evaluation metrics and an optimizer before fitting it to the training data. We selected BinaryAccuracy, Precision, and Recall as our metrics to provide a comprehensive understanding of the model's performance. The model is compiled using the Adam optimizer, known for its efficiency in handling sparse gradients and large datasets, along with the BinaryCrossentropy loss function, which is suitable for binary classification tasks.

During the training phase, the model is trained over 10 epochs. In the first epoch, the model starts with a loss of 0.6079, achieving a binary accuracy of 67.32%, precision of 67.20%, and recall of 67.68%, indicating moderate initial performance. By the second epoch, there is a notable improvement, with the loss decreasing to 0.4949, binary accuracy rising to 80.18%, precision to 76.83%, and recall to 86.43%. This indicates that the model is becoming more effective at correctly identifying spam emails.

As training progresses, the third epoch shows further enhancement, with the loss reducing to 0.4248, binary accuracy increasing to 85.45%, precision to 84.28%, and recall to 87.14%. The trend continues with the fifth epoch showing a loss of 0.3504, binary accuracy of 88.04%, precision of 86.60%, and recall of 90.00%.

The ninth epoch continues the positive trend, with a loss of 0.2831, binary accuracy of 90.36%, precision of 89.93%, and recall of 90.89%. Finally, in the tenth epoch, the model achieves a loss of 0.2729, binary accuracy of 91.43%, precision of 90.00%, and recall of 93.21%, demonstrating robust performance in distinguishing between spam and legitimate emails. This training process

effectively showcases the model's ability to improve and adapt, leading to high accuracy and reliability in email classification.

```
my_model.evaluate(X_test, y_test)
```

```
12/12 [==============================] - 37s 3s/step - loss: 0.2836 - Binary Accuracy: 0.9037 - Precision: 0.8756 - Recall: 0.9412
[0.28355422616004944,
 0.903743326663971,
 0.8756219148635864,
 0.9411764740943909]
```

Figure 11: Evaluating Model on Test Data

After training, the model's performance was evaluated on the test dataset using the evaluate method. The results demonstrate the model's effectiveness, achieving a loss of 0.2836, a binary accuracy of 90.37%, a precision of 87.56%, and a recall of 94.12%. These metrics indicate that the model maintains high accuracy and reliability in identifying spam emails even on unseen data, showcasing its robustness and generalization capabilities.

```
y_predicted = my_model.predict(X_test)
y_predicted = y_predicted.flatten()

y_predicted = np.where(y_predicted > 0.5, 1, 0)
cm = confusion_matrix(y_test, y_predicted)
```

Figure 12: Generate Confusion Matrix

After obtaining the predicted probabilities from the model using my_model.predict(X_test), the predictions were flattened and converted into binary classifications using a threshold of 0.5. This produced an array of binary values, with 1 representing spam and 0 representing non-spam. The confusion matrix, generated using confusion_matrix(y_test, y_predicted), provides a detailed breakdown of the model's performance by showing the counts of true positives, true negatives, false positives, and false negatives. This matrix helps in assessing the model's accuracy and identifying areas for potential improvement.

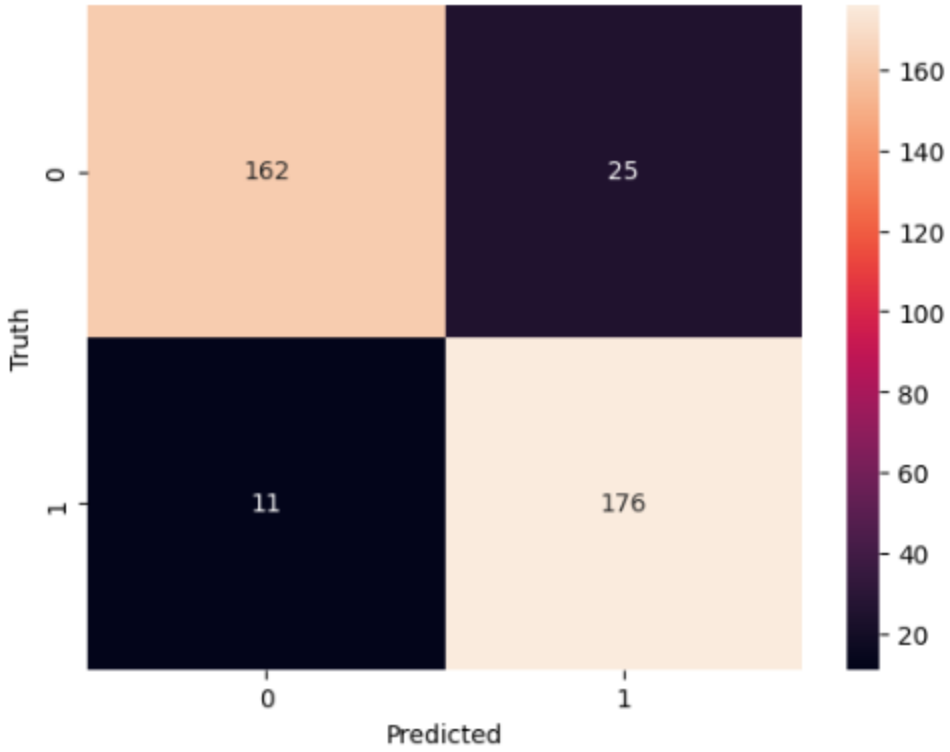

Figure 13: Confusion Matrix

The confusion matrix for our model's predictions shows the following results:

- **True Negatives (162):** The model correctly identified 162 non-spam emails.
- **False Positives (25):** The model incorrectly classified 25 non-spam emails as spam.
- **False Negatives (11):** The model incorrectly classified 11 spam emails as non-spam.
- **True Positives (176):** The model correctly identified 176 spam emails.

These results indicate that the model has a good balance between identifying spam and non-spam emails, with high true positive and true negative counts, suggesting effective performance in classifying emails correctly.

Finally, we can analyze the classification report generated by the classification_report method from the Sci-kit Learn Library.

	precision	recall	f1-score	support
0	0.94	0.87	0.90	187
1	0.88	0.94	0.91	187
accuracy			0.90	374
macro avg	0.91	0.90	0.90	374
weighted avg	0.91	0.90	0.90	374

Figure 14: Classification Report

The classification report provides detailed metrics on the model's performance for classifying emails as spam (1) and non-spam (0):

- **Precision:** The precision for non-spam emails is 0.94, meaning that 94% of emails predicted as non-spam were actually non-spam. For spam emails, the precision is 0.88, indicating that 88% of emails predicted as spam were truly spam.
- **Recall:** The recall for non-spam emails is 0.87, meaning that 87% of actual non-spam emails were correctly identified. For spam emails, the recall is 0.94, indicating that 94% of actual spam emails were correctly identified.
- **F1-Score:** The F1-score, which is the harmonic mean of precision and recall, is 0.90 for non-spam and 0.91 for spam, indicating a balanced performance in identifying both classes accurately.
- **Support:** Both classes have 187 instances in the test set, showing balanced class distribution.

The overall accuracy of the model is 0.90, and both the macro average and weighted average F1-scores are also 0.90, reflecting consistent performance across classes.

Conclusion

In conclusion, our neural network model leveraging BERT for text encoding demonstrates robust performance in classifying emails as spam or non-spam. The model achieved a high overall accuracy of 90%, with balanced precision and recall metrics for both classes. The detailed analysis of the confusion matrix and classification report confirms the model's ability to generalize well on unseen data, effectively identifying spam emails while maintaining a low rate of false positives and false negatives. These results highlight the model's potential for practical deployment in email filtering systems, providing a reliable tool for enhancing email security and user experience.