National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Tamil Selvan Giri Moorthy
Student ID: x23189975

School of Computing
National College of Ireland

Supervisor: Mr. Bharat Agarwal

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: Tamil Selvan Giri Moorthy

Student ID: X23189975

Programme: MSc Data Analytics **Year:** 2024

Module: MSc Research Project

Lecturer: Mr. Bharat Agarwal

Submission Due Date: 12 December 2024

Project Title: Enhancing SMS Spam Detection Using Deep learning

Word Count: 1377 **Page Count: 11**

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Tamil Selvan Giri Moorthy

Date: 12 December 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	☑
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☑
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☑

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Tamil Selvan Giri Moorthy
X23189975

1 Introduction

This configuration manual will give the information what are the settings and necessary steps that we are followed for this project. The SMS spam Detection project involves building a machine learning, deep learning specially using BERT to classify SMS messages as either ham (non-spam) or spam. The process includes preprocessing text data, training a BERT model, fine tuning it for the classification task, evaluating its performance on test data. The configuration manual guides you through the setup and execution of the project, from data preprocessing to model training, evaluation, and deployment.

2 Environmental Setup

We used Jupter notebook and google colab to run our project. Google colab provides access to GPU and TPU to run a powerful deep learning models like BERT. So, for machine learning and deep learning models we used Jupter notebook and Google colab for BERT model.

2.1 Hardware Requirements

Since Google colab pro runs on cloud-based infrastructure, no specific hardware setup is required from our local machine. But for optimal performance we need to ensure the following.

- **Google Account:** A Google account is required to access Colab
- **Internet Connection:** A stable and fast internet connection is required to handle large datasets for our ham and spam dataset and compute intensive operations like training deep learning models like BERT
- **Google Colab Pro Subscription:** Access to Colab pro is recommended for higher memory and longer runtimes, which are essential for large scale model training and experimentation.

For Jupter notebook we used,

- **Operating System:** Windows 11
- **Processor:** Multi-core processor with Intel i5 or higher and AMD equivalent.
- **Ram:** Minimum of 16GB

2.2 Software Requirements

- **Python:** Python 3.7 or later (typically pre-installed in colab)
- **Transformers:** A library from hugging face for loading pre- trained BERT models and fine-tuning them
- **Keras:** For building and training neural network models (can be used alongside Tensorflow)
- **Scikit-learn:** For data processing, model evaluation and utilities like train-test splitting.
- **NumPy:** For numerical operations and handling arrays.
- **Pandas:** For data manipulation and handling tabular data such as your interaction matrix and the spam dataset.
- **Matplotlib / Seaborn:** For visualizing results, metrics and training curves.
- **Google Drive:** Since we are using Google Colab, we need to mount our google drive to access datasets like the spam dataset and save model outputs.

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
import transformers
from transformers import AutoModel, BertTokenizerFast
import torch
import torch.nn as nn
from transformers import BertModelfrom torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler# import BERT-base
from transformers import AutoModel, BertTokenizerFastfrom sklearn.model_selection import train_test_split
```

Figure 1: Libraries which are used in BERT

This research was implemented using python which is a popular programming language for machine learning models, along with Jupter Notebook as the development environment. We need to check that python 3.12 is installed and properly set up. If it's not installed, you can download it directly from the official website and follow the installation guide. To use Jupyter Notebook install Anaconda Navigator from its official website. Once installed, open Anaconda Navigator, find Jupter Notebook in the application list, and install it. After installation, launch Jupyter Notebook, which will open in your default web browser. From there, you can access and run the research code files. The necessary libraries for both stages of the research are included in the setup.

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from wordcloud import WordCloud, STOPWORDS, ImageColorGenerator
%matplotlib inline
from sklearn.model_selection import train_test_split
# deep learning libraries for text pre-processing
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, GlobalAveragePooling1D, Dense, Dropout, LSTM, Bidirectional,SimpleRNN
from sklearn.metrics import classification_report
```

Figure 2: Libraries which are used in deep learning

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
# data preprocessing
import nltk
nltk.download('stopwords')
nltk.download('punkt')
import string
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from nltk.tokenize import sent_tokenize, word_tokenize
import re
from collections import Counter
from wordcloud import WordCloud
from sklearn.preprocessing import LabelEncoder
# Model Building
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score,confusion_matrix,precision_score
from sklearn.ensemble import RandomForestClassifier
```

Figure 3: libraries which are used in machine learning

3 Tools and Required Setup

3.1 Accessing Google Colab Pro and Jupter Notebook:

Sign in to Google account. Go to Google Colab, we have Google Colab Pro subscription, we can access the enhanced GPU/TPU features. Start a new Colab notebook by clicking on "New Notebook" from the Colab dashboard. In Jupter notebook we can directly do your code in your local system by creating a new file.

3.2 Accessing Datasets from Google Drive:

To run the code with our datasets in Google Colab, we need to access the datasets stored on Google Drive. We can mount Google Drive in Colab as follows:

```
from google.colab import drive
drive.mount('/content/drive')
```

This about prompt will authenticate your google account. Once authentication is verified you can able to access the datasets stored in your google drive

Ham and Spam Dataset:	https://drive.google.com/file/d/1Si98Lgd2QnfHYqM-lv6e3O5aBFnxLA4D/view?usp=sharing

3.3 Running the Code:

Once the dataset and libraries are loaded in both the platforms, we run the code in both colab and jupter notebook. We need to check the code is following the structure required for your research.

- Import necessary libraries for all 3 learning models.
- Then load dataset into pandas dataframes.
- Then understand the shape and structure of the dataset.
- Preprocess the data.
- Train models and evaluated results.

4 Configuration and set up for model Training:

4.1 Mounting Google Drive and Loading Data:

```
from google.colab import drive
drive.mount('/content/drive')
import os
import pandas as pd
import numpy as np
# Step 1: Load the News Data with Embeddings
df = pd.read_csv('/content/drive/MyDrive/spam-ham v2.csv', encoding="latin-1", usecols=["v1", "v2"])
df = df.copy()
df.columns = ["label", "text"]
df = df[["text", "label"]]
df["label"] = df["label"].map({"ham": 0, "spam": 1})
df.head()
```

This code will load the datasets from Google Drive and print the first few rows to verify the data.

4.2 Loading Data in Jupter Notebook:

```
# Specify the path to the SMSSpamCollection file
file_path = "SMSSpamCollection"  # Update the path if the file is in another directory

# Load the file into a pandas DataFrame
messages = pd.read_csv(file_path, sep='\t', header=None, names=['label', 'message'], encoding='latin1')

# Display the first few rows of the DataFrame
print("First 5 rows of the dataset:")
print(messages.head())
```

In Jupter notebook, we directly uploaded the file and printed the first few rows of the dataset.

4.3 Preprocessing steps:

First, we identify the shape of the dataset to understand the how many data are present in the dataset.

```
df.shape
```

```
(5572, 2)
```

```python
# Import necessary libraries
import re
import string

# Define the clean_text function
def clean_text(text):
    '''Make text lowercase, remove text in square brackets, remove links, remove punctuation,
    and remove words containing numbers.'''
    text = str(text).lower()  # Convert text to lowercase
    text = re.sub(r'\[.*?\]', '', text)  # Remove text in square brackets
    text = re.sub(r'https?://\S+|www\.\S+', '', text)  # Remove links
    text = re.sub(r'<.*?>+', '', text)  # Remove HTML tags
    text = re.sub(f"[{re.escape(string.punctuation)}]", '', text)  # Remove punctuation
    text = re.sub(r'\n', '', text)  # Remove newline characters
    text = re.sub(r'\w*\d\w*', '', text)  # Remove words containing numbers
    return text

# Apply the clean_text function to the 'text' column and create a new column 'text_clean'
data['text_clean'] = data['text'].apply(clean_text)

# Display the first 5 rows of the dataframe
data.head()
# removing stop words from the dataset
stop words = set(stopwords.words("english"))
```

Figure 4: preprocessing steps

```python
# Initialize the PorterStemmer
stemmer = PorterStemmer()

# Load your dataset (replace 'your_file.csv' with your actual file path)
# Ensure the file path is correct, and check the column name

# Check if 'text' column exists; print column names if not
if 'text' not in data.columns:
    print("Column names:", data.columns)
    raise ValueError("The dataset does not have a 'text' column.")

# Define the function to apply stemming
def apply_stemming(text):
    if pd.isna(text):  # Handle any potential NaN values
        return ""
    text = re.sub(r'\W', ' ', str(text))  # Remove special characters
    words = nltk.word_tokenize(text)  # Tokenize the text into words
    stemmed_text = ' '.join([stemmer.stem(word) for word in words])  # Apply stemming
    return stemmed_text

# Apply the function to the 'text' column
data['text'] = data['text'].apply(apply_stemming)
```

Figure 5: preprocessing steps

The above figure 4 and 5 shows the code what are the preprocessing steps that we conducted for project. By doing that we removed the unnecessary columns and irrelevant words from the dataset.

4.4 Implementing Machine learning model:

Once we pre-processed the data, we encode labels, applies TF-IDF vectorization with 50 features on spam data and splits the dataset into training 20% and test 80% sets using a random seed for reproducibility.

```
# Label Encoding
encoder = LabelEncoder()
data['result'] = encoder.fit_transform(data['result'])

# TF-IDF Vectorization with Reduced Features
tfidf = TfidfVectorizer(max_features=50)
X = tfidf.fit_transform(data['emails']).toarray()
y = data['result']

# Reduce Training Data Size
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.80, random_state=42  #  test size
)
```

Then we applied random forest classifier and make predications on the test set, and evaluates performance using accuracy, confusion matrix, and precision.

```
# Simplified Random Forest
rf_classifier = RandomForestClassifier(n_estimators=1, random_state=42)
rf_classifier.fit(X_train, y_train)

# Predictions and Evaluation
y_pred_rf = rf_classifier.predict(X_test)
accuracy_rf = accuracy_score(y_test, y_pred_rf)
print(f"Random Forest Accuracy: {accuracy_rf:.2f}")
print("Confusion Matrix:", confusion_matrix(y_test, y_pred_rf))
print("Precision Score:", precision_score(y_test, y_pred_rf))
```

4.5 Implementing Deep learning model:

Mapping ham and spam

```
# Map ham label as 0 and spam as 1
msg_df['msg_type']= msg_df['label'].map({'ham': 0, 'spam': 1})
msg_label = msg_df['msg_type'].values
# Split data into train and test
train_msg, test_msg, train_labels, test_labels = train_test_split(msg_df['message'], msg_label, test_size=0.2,
```

For deep learning model. This code maps ham to 0 and spam to 1, then splits dataset into training 80% and testing 20% sets for message and the corresponding labels using a fixed random seed.

Sequence and Padding:

The below code converts text messages into sequence, then applies padding and truncation to ensure consistent input length for training and testing data.

```
# Sequencing and Padding

training_sequences = tokenizer.texts_to_sequences(train_msg)
training_padded = pad_sequences (training_sequences, maxlen = max_len, padding = padding_type, truncating = trunc_type )
testing_sequences = tokenizer.texts_to_sequences(test_msg)
testing_padded = pad_sequences(testing_sequences, maxlen = max_len,
padding = padding_type, truncating = trunc_type)
```

Checking the shape before and after padding:

```
# Shape of train tensor
print('Shape of training tensor: ', training_padded.shape)
print('Shape of testing tensor: ', testing_padded.shape)
```

```
Shape of training tensor:  (1195, 50)
Shape of testing tensor:  (299, 50)
```

```
# Before padding
len(training_sequences[0]), len(training_sequences[1])
```

```
(27, 24)
```

```
# After padding
len(training_padded[0]), len(training_padded[1])
```

```
(50, 50)
```

This code prints the shapes of padded training and testing tensors, then compares the sequence lengths before and after padding.

Applied the deep learning models

Then applied all the deep learning models and concluded the results for all the models. The below displays only one learning model.

```
# Define the LSTM model with Bidirectional layer for spam detection
model1 = Sequential()

# Embedding layer to convert words into vectors of fixed size
model1.add(Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=max_len))

# Bidirectional LSTM layer for sequence processing
model1.add(Bidirectional(LSTM(units=n_lstm, dropout=drop_lstm, return_sequences=False)))

# Output layer with sigmoid activation for binary classification (spam vs. not spam)
model1.add(Dense(1, activation='sigmoid'))
```

```
model1.compile(loss = 'binary_crossentropy', optimizer = 'adam', metrics=['accuracy'])
```

```
from sklearn.metrics import classification_report
import numpy as np

# Training and evaluation LSTM model
num_epochs = 20
early_stop = EarlyStopping(monitor='val_loss', patience=2)
history = model1.fit(training_padded, train_labels, epochs=num_epochs, validation_data=(testing_padded, test_labels)

# Evaluate the model on the test set to get loss and accuracy
loss, accuracy = model1.evaluate(testing_padded, test_labels)

# Generate predictions on the test set
predictions = model1.predict(testing_padded)

# Convert predictions to class labels (0 or 1) for binary classification
predicted_labels = (predictions > 0.5).astype(int)

# Print loss and accuracy
print(f"Bi-LSTM architecture loss: {loss}")
print(f"Bi-LSTM architecture accuracy: {accuracy * 100:.2f}%")

# Print precision, recall, and F1 score
print("\nClassification Report:")
print(classification_report(test_labels, predicted_labels))
```

4.6 Implementing Bert model:

Split train dataset into train, validation and test sets:

```
from sklearn.model_selection import train_test_split

train_text, temp_text, train_labels, temp_labels = train_test_split(df['text'], df['label'],
                                                                    random_state=2018,
                                                                    test_size=0.3,
                                                                    stratify=df['label'])

# we will use temp_text and temp_labels to create validation and test set
val_text, test_text, val_labels, test_labels = train_test_split(temp_text, temp_labels,
                                                                random_state=2018,
                                                                test_size=0.5,
                                                                stratify=temp_labels)
```

This code first splits the dataset into a training set 70% and temporary set 30% using a train_test_split. The stratify parameter ensures that the label distribution is preserved across both sets. Then, the temporary set (temp_text and temp_labels) is further split into validation set 15% and test run 15%.

Import BERT Model and BERT Tokenizer

```
# import BERT-base pretrained model
from transformers import AutoModel, BertTokenizerFast

bert = AutoModel.from_pretrained('bert-base-uncased')

# Load the BERT tokenizer
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
```

This code imports the BERT- base pretrained model and its tokenizer from the hugging face transformer library for natural language processing tasks.

Tokenization

```
[ ]  max_seq_len = 25
```

```
# tokenize and encode sequences in the training set
tokens_train = tokenizer.batch_encode_plus(
    train_text.tolist(),
    max_length = max_seq_len,
    pad_to_max_length=True,
    truncation=True,
    return_token_type_ids=False
)

# tokenize and encode sequences in the validation set
tokens_val = tokenizer.batch_encode_plus(
    val_text.tolist(),
    max_length = max_seq_len,
    pad_to_max_length=True,
    truncation=True,
    return_token_type_ids=False
)
```

This code tokenizes and encodes text data from the training, validation, and test sets using the BERT tokenizer, applying padding, truncation, and setting a maximum sequence length for all sets.

Convert Interger into Tokens

```
# for train set
train_seq = torch.tensor(tokens_train['input_ids'])
train_mask = torch.tensor(tokens_train['attention_mask'])
train_y = torch.tensor(train_labels.tolist())

# for validation set
val_seq = torch.tensor(tokens_val['input_ids'])
val_mask = torch.tensor(tokens_val['attention_mask'])
val_y = torch.tensor(val_labels.tolist())

# for test set
test_seq = torch.tensor(tokens_test['input_ids'])
test_mask = torch.tensor(tokens_test['attention_mask'])
test_y = torch.tensor(test_labels.tolist())
```

This code converts the tokenized sequences, attention masks, and labels for the training validation, and test sets into PyTorch tensors for use in deep learning models.

Define Model Architecture

```
class BERT_Arch(nn.Module):
    def __init__(self, bert):
        super(BERT_Arch, self).__init__()

        # Initialize the BERT model
        self.bert = bert

        # Dropout layer to prevent overfitting
        self.dropout = nn.Dropout(0.1)

        # ReLU activation function
        self.relu = nn.ReLU()

        # Fully connected layers
        self.fc1 = nn.Linear(768, 512)  # 768: BERT output dimension
        self.fc2 = nn.Linear(512, 2)   # 2: Number of output classes

        # LogSoftmax activation function
        self.softmax = nn.LogSoftmax(dim=1)
```

This code defines BERT- based neural network model using PyTorch. It uses the pre-trained BERT model for feature extraction, followed by a dropout layer, fully connected layers, and a softmax activation for binary classification. The model returns classification logits.

Fine-Tune Bert

```
# function to train the model
def train():

  model.train()

  total_loss, total_accuracy = 0, 0

  # empty list to save model predictions
  total_preds=[]

  # iterate over batches
  for step,batch in enumerate(train_dataloader):

    # progress update after every 50 batches.
    if step % 50 == 0 and not step == 0:
      print('  Batch {:>5,}  of  {:>5,}.'.format(step, len(train_dataloader)))

    # push the batch to gpu
    batch = [r.to(device) for r in batch]

    sent_id, mask, labels = batch

    # ensure sent_id and mask are tensors
    #sent_id = torch.tensor(sent_id, dtype=torch.long, device=device)  # This line is causing the issue
    #mask = torch.tensor(mask, dtype=torch.long, device=device) # This line is causing the issue

    # clear previously calculated gradients
    model.zero_grad()

    # get model predictions for the current batch
    preds = model(sent_id, mask)

    # compute the loss between actual and predicted values
    loss = cross_entropy(preds, labels)

    # add on to the total loss
    total_loss = total_loss + loss.item()
```

The train () function trains a BERT model by iterating over batches from the training data loaders. It computes the loss using cross-entropy, performs backpropagation, updates model parameters, and clip gradients to prevent exploding gradients. The function returns the average loss and model predications for the epoch.

Start Model Training

This code trains and evaluates the model for each epoch, tracking training and validation losses. It saves the model with the best validation loss and prints the loss value after each epoch.

```
# set initial loss to infinite
best_valid_loss = float('inf')

# empty lists to store training and validation loss of each epoch
train_losses=[]
valid_losses=[]

#for each epoch
for epoch in range(epochs):

    print('\n Epoch {:} / {:}'.format(epoch + 1, epochs))

    #train model
    train_loss, _ = train()

    #evaluate model
    valid_loss, _ = evaluate()

    #save the best model
    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'saved_weights.pt')

    # append training and validation loss
    train_losses.append(train_loss)
    valid_losses.append(valid_loss)

    print(f'\nTraining Loss: {train_loss:.3f}')
    print(f'Validation Loss: {valid_loss:.3f}')
```

After we this steps, we evaluated the results for all the models.

Conclusion

This Configuration manual helps in settings up environment, verifying GPU availability, loading and processing datasets, extracting Bert and other machine learning and deep learning models by effectively addressing issues like class imbalance and using advanced models. This setup is crucial for building the project aims to accurately detect the spam messages with high performance.

.