National College of Ireland

Novel Text and Image Based Approach to Android Malware Detection

MSc Research Project
Data Analytics

Agnideep Pal
Student ID: x21141631

School of Computing
National College of Ireland

Supervisor: Prof. Anderson Simiscuka

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Agnideep Pal
Student ID:	x21141631
Programme:	Data Analytics
Year:	2022
Module:	MSc Research Project
Supervisor:	Prof. Anderson Simiscuka
Submission Due Date:	15/12/2022
Project Title:	Novel Text and Image Based Approach to Android Malware Detection
Word Count:	1475
Page Count:	12

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:	Agnideep Pal
Date:	1st February 2023

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):	

Novel Text and Image Based Approach to Android Malware Detection

Agnideep Pal
x21141631

1 Introduction

This manual will act as a guide to setup all the experiments exactly as it was performed during the research. It will detail the information related to source of dataset, ways to fetch it, necessary software tools, system requirements to run them, code segments explaining the experiments, model implementation and evaluation.

2 System Configuration

This section lists all the system wide configuration during the research.

2.1 Hardware Requirements

- **OS :** Windows 11 Home Single Language
- **Processor :** Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz 2.50 GHz
- **Installed Ram :** 8.00 GB (7.84 GB usable)

2.2 Software Requirements

- **Jupyter Notebook :** This is an interactive web-based python development environment suitable for carrying out experiments proposed in this research project. This tool is used in the project for writing and validating most of the code as discussed in the Section 3.
- **AndroGuard :** A popular interactive python tool which provides valuables APIs to analyze and decompile Android executable files dynamically.
- **Open CV :** This ia a popular image processing library , widely used in computer vision applications.
- **Google Colab :** This is a cloud based google platform that provides both free and paid computing resources like CPU, GPU and TPU for carrying out resource intensive operations like training a large neural network. This platform is extensively used in this research project to carry out multiple experiments and training different models.

- **Google Drive :** This is a popular cloud-based storage platform provided by Google. This is used in the project to store important data files (like Numpy and Text) as an input for the different Image and Text based neural network models employed in the experiments
- **TensorFlow :** It is a versatile platform that provides many APIs for building, validating , tuning and evaluating complex neural network models. These APIs are extensively used in the project to build proposed models.

3 Project Structure

The folder structure of the source code is provided below in Figure 1 to give a clarity on how the research code is organized.

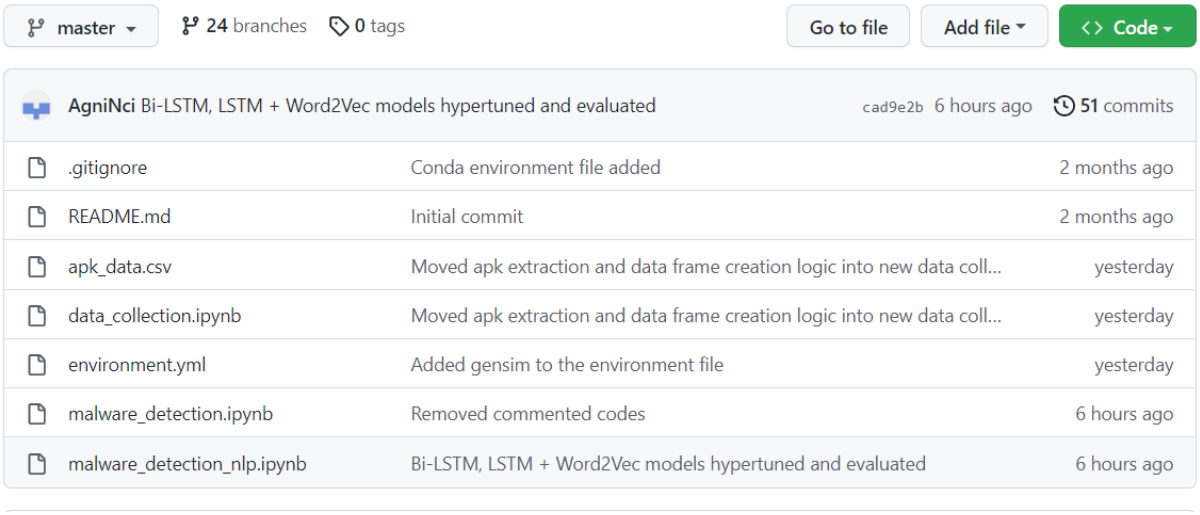


Figure 1: Github Master Branch

Among the files visible in the above figure, **apk_data.csv** holds all the meta information of the APKs used in the project. The notebooks have been divided into three separate files based on their roles in the project. **Data_collection.ipynb** handles the data collection strategies. Whereas other two notebooks are used for image and text based analysis. The environment.yml lists all the required libraries.

4 Project Setup Steps

Two distinct strategies are used in this research project to perform accurate classifications of the android malwares. These are respectively **Image** and **Text** based. The experimental setup for each of them can be collectively categorized under several broad steps, as discussed in detail in the following sub-sections.

4.1 Creating a GitHub Repo

A repository is created in the college Github Account and is cloned into local file system at the beginning of the project. The link to the GitHub repository is given below.

https://github.com/AgniNci/Android-Malware-Detection.git

4.2 Setting up Conda Environment:

4.2.1 Step 1 : Creating config file

An **"environment.yml"** file is created in the project directory. The purpose of this file is to maintain a dedicated **"conda"** environment named **"Research"** for the research project. It holds the list of all the important 3 rd party libraries that needs to be installed. The file details are provided below in the Figure 2

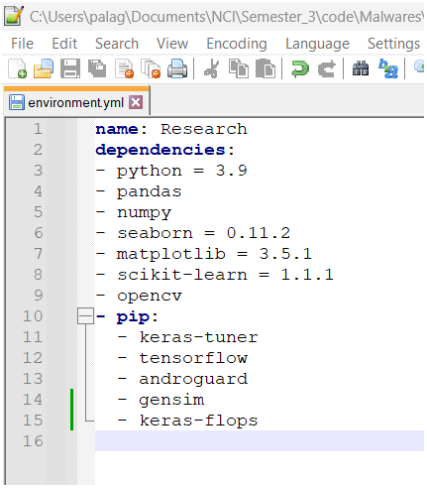
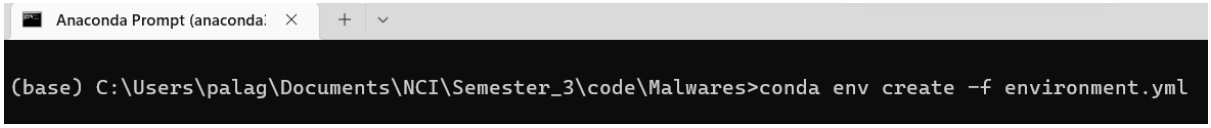


Figure 2: Environment File

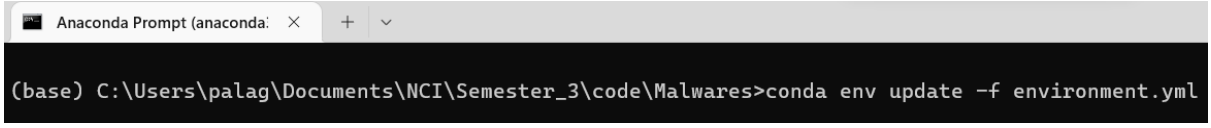
4.2.2 Step 2 : Installing libraries

Next navigating to the project directory and running the below commands, the environment is created and updated conveniently from a **"Anaconda Prompt"** terminal.

- Create Environment



- Update Environment



4.2.3 Step 3 : Create Jupyter Kernel

In this step a Jupyter Kernel with the same name as conda env i.e.. **"Research"** is created by running the commands below within the conda terminal. The kernel created here helps in switching to the appropriate runtime environment from the notebook.

- Installing Ipykernel

```
(base) C:\Users\palag\Documents\NCI\Semester_3\code\Malwares>conda install ipykernel
Collecting package metadata (current_repodata.json): done
Solving environment: done
```

- Creating Kernel

```
(base) C:\Users\palag\Documents\NCI\Semester_3\code\Malwares>ipython kernel install --user --name=Research
Installed kernelspec Research in C:\Users\palag\AppData\Roaming\jupyter\kernels\research
```

5 Data Pre-processing

This section details the implementation steps as follows:

5.1 Extracting Data :

- This step begins with the downloading of zip files containing all APKs from the official website of University of New Brunswick, Canada. The url is provided below. `https://www.unb.ca/cic/datasets/invesandmal2019.html` (Taheri et al. (2019))
- Using the code as displayed in the Figure 3 below, the APK files in the downloaded dataset is extracted from the zip files into their respective folders.

```
In [ ]:
for file in benign:
    with zipfile.ZipFile(file, 'r') as zip_ref:
            zip_ref.extractall('../../Research/apks/')

In [ ]:
for file in malwares:
    with zipfile.ZipFile(file, 'r') as zip_ref:
            zip_ref.extractall('../../Research/apks/')
```

Figure 3: Extract APK from zip files

- The file names for the benign applications in the dataset are named after their packages and needs to be renamed using their **MD5** digest. This must be done to uniquely identify each APK file during the experiments. As far as the malwares are concerned, they are already named accordingly and do not need further correction. The code for generating md5 and renaming the files is provided in the Figure 4 below.

In [ ]:
```python
#Generating md5 hash
def md5(fname):
    hash_md5 = hashlib.md5()
    with open(fname, "rb") as f:
        for chunk in iter(lambda: f.read(4096), b""):
            hash_md5.update(chunk)
    return hash_md5.hexdigest()
```

In [ ]:
```python
#Rename the benign apk files with its md5 digest
for apk in benign_apks:
    if "2015" in apk:
        continue
    md5_digest = md5(apk)
    os.rename(apk, apk.split("\\")[0] + "/" + md5_digest + ".apk")
```

Figure 4: Generating MD5 Hash & Renaming APKs

- Using the available information of the APK files, a panda dataframe is created for organizing in training, testing and validation data along with performing basic exploratory analysis. The data frame created is shown in the Figure 5.

Out[145]:

	id	is_malicious	type	label	root_path
0	1c4e357a8ec5f13de4ffd57cc2711afe	1	dowgin	Adware	../../Research/apks/Adware\dowgin
1	1d15765ffee294f27da4865356a994bd	1	dowgin	Adware	../../Research/apks/Adware\dowgin
2	31c657bf77ea817893d9ba13dbcdf749	1	dowgin	Adware	../../Research/apks/Adware\dowgin
3	37b993b5f59bbed7538265885429e4c4	1	dowgin	Adware	../../Research/apks/Adware\dowgin
4	3e30f2644a2e9f1b81f7f5a810e5f6ce	1	dowgin	Adware	../../Research/apks/Adware\dowgin
...	...	...	...	...	...
2099	17c290bc50f76292ebb10586babb8ac0	1	AvForAndroid	Scareware	../../Research/apks/Scareware\AvForAndroid

Figure 5: Generated Data frame

5.2 Preparing Data for Image Based Malware Detection

- The Data frame created in the previous step is then traversed to fetch the individual APK files from the file system and decompiled them using **"AndroGuard"** using the code in the Figure 6.

```python
#Apk Decompilation using AnalyzeAPK api from AndroGuard
a,d,dx = AnalyzeAPK(os.path.join(row['root_path'] + '/' + row['id'] + '.apk'))
img = apk2image(a, d, dx, img_dimension)
```

Figure 6: APK de-compilation using AndroGuard

- Consequently, they are converted to RGB images and saved in jpeg format files. This happens in three distinct phases. In each phase a distinct characteristic feature of the application is encoded into a colour channel. The **Manifest file** is encoded into **Green Channel**, **opcodes** into **blue channel** and **external api calls** into **red channel**. The process overview is depicted below in the Figure 7

- Finally, the images are converted to **NumPy array** and stored in local filesystem as **.npy** file for training the model using the code provided in the Figure 8.

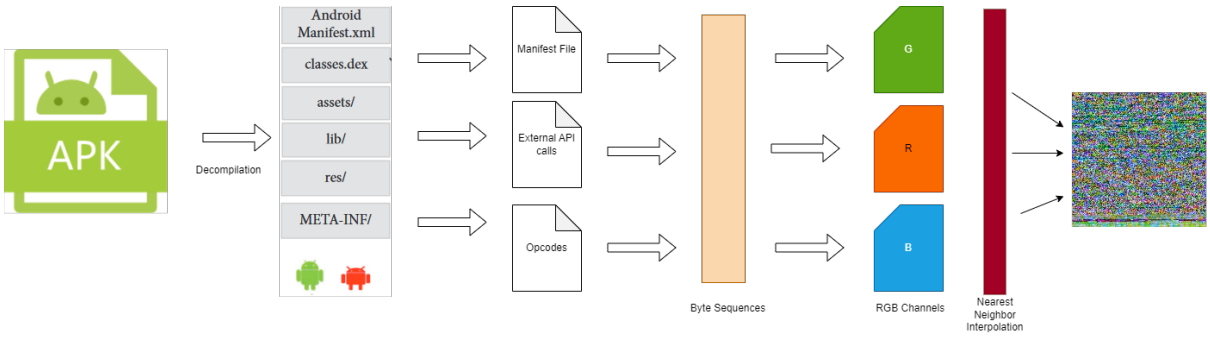


Figure 7: APK to RGB Image conversion

```
#converting images to numpy array (.npy file) for training deep learning models
families_path = f'C:/Users/palag/Documents/NCI/Semester_3/Research/images/dim_{img_dimension}p/*'
families_npy_path = 'C:/Users/palag/Documents/NCI/Semester_3/Research/npys/'
from numpy import load
count = 0
array = []
for family in glob.glob(families_path):
    image_name = family.split("\\")[1]
    array.append(plt.imread(os.path.join(families_path[:-1] + image_name)))
    count += 1
np.save(os.path.join(families_npy_path + f'pixel_data_{img_dimension}p.npy'), array)
```

Figure 8: Numpy file creation from Image Pixels

5.3 Preparing Data for NLP based Malware Detection

- The DataFrame is again traversed to fetch the APK files and dynamically decompile using **"AndroGuard"**
- The decompiled outputs were then further processed to extract the opcodes using the code illustrated in the Figure 9

```
#Opcode Sequence Extraction from dex files
def apk_to_api_sequences(a, d, dx):
    methods = []
    for m_a in dx.get_methods():
        m = m_a.get_method()
        if m_a.is_external():
        # External methods do not have opcodes
            continue
        idx = 0
        method_opcodes = []
        for ins in m.get_instructions():
            # You can access the instruction here and do stuff with it...
            method_opcodes.append(ins.get_name( ))
            idx += ins.get_length()
        methods.append("".join(method_opcodes).strip())
    print(f'All Methods Count: {len(methods)}')
    unique_methods = list(set(methods))
    print(f'All Unique Methods Count: {len(unique_methods)}')
    return unique_methods
```

Figure 9: APK to Opcode sequence translation

- Next the opcodes sequences are subjected to unique sampling, introduced in this research. The goal of this sampling is to extract largest sub sequences appropriate for the optimum performance of the language models while retaining much of the

information and context. The overview of the whole process is illustrated in the Figure 11. This is achieved programmatically by the code in the Figure 10

```python
#This function takes large sequences of opcodes as an input and returns smaller representation
def sample_sequences_from_text(text_list, num_sequences, seq_size = 420):
    sub_sample_sizes = [int(seq_size * 0.4), int(seq_size * 0.3), int(seq_size * 0.2), int(seq_size * 0.1)]
    #print(f'Sub Sample Sizes : {sub_sample_sizes}')
    chunks = np.array_split(text_list, len(sub_sample_sizes))
    sequences = []
    for num in range(num_sequences):
        np.random.shuffle(chunks)
        seq = []
        for i in range(len(sub_sample_sizes)):
            seq += np.ndarray.tolist(chunks[i][:sub_sample_sizes[i]])
        sequences.append(" ".join(seq))
    return sequences
```

Figure 10: Opcode Sequence Sampling

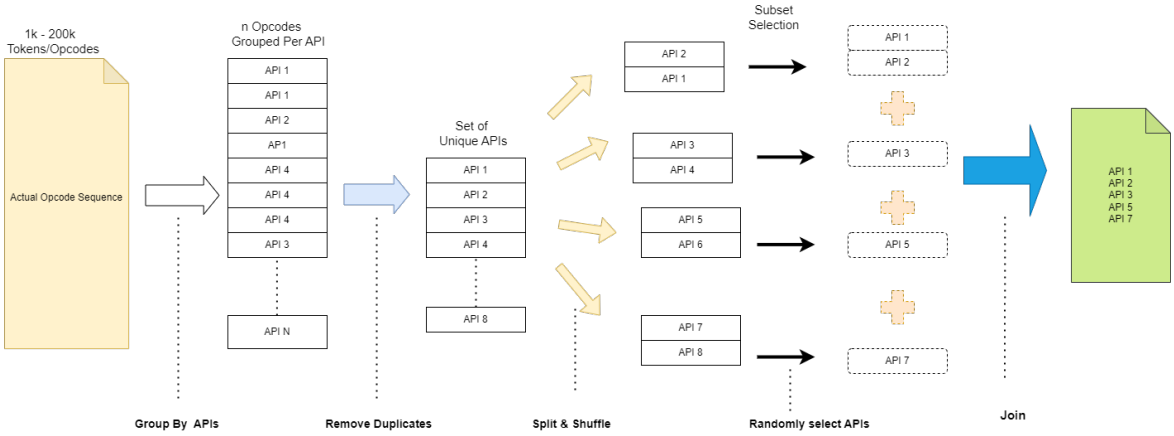


Figure 11: Opcode Sequence Sampling Overview

5.4 Uploading Input Files to Colab

To make use of built-in computing resources in Colab, the appropriate input files (.npy file for Image Based Models and .csv file for NLP based models) are uploaded to the google drive .

6 Model Implementation & Hyperparameter Tuning

Since the research dwell arounds detecting android malwares, it is obvious that the resultant model should be able to run on any handheld android device. As a result, one pre-trained models for Image Processing, suitable for mobile devices was chosen for the experiments. This model is **EfficientNetB4**. This pre-trained model uses weights derived from **"ImageNet"** where it has been trained with millions of data points. Also, another CNN model was proposed by me as well. Below are the implementation of the individual models with their hyper-parameters tuning. For NLP based Malware detection **LSTM** and **Bi-LSTM** models along with word2vec embedding was used.

6.1 Image Based Models

6.1.1 EfficientNetB4 Implementation & Evaluation

The code in the Figure 12 below shows the initialization of the EfficientNetB4 model with the pre-trained **"imagenet"** weights. The model is then compiled with the appropriate hyperparameters like optimizer, loss function and evaluation criteria as shown below.

```
#EfficientNetB4 excluding topmost fully connected layer
#Pooling : Global Average pooling
#Optimizer : Adam
base_model = tf.keras.applications.EfficientNetB4(include_top=False, weights='imagenet', input_shape = input_shp, pool
base_model.trainable = False
model = models.Sequential()
model.add(base_model)
model.add(layers.Dense(1, activation='sigmoid'))
print(model.summary())
```

Figure 12: EfficientNetB4 Initialization

6.1.2 EfficientNetB4 Hyperparameter Tuning

The following code in the Figure 13 illustrates the hyper-parameter tuning of the top layers of the model using novel bandit-based tuning (Li et al. (2017)) algorithm. The algorithm is available as an API in Keras.

```
def build_efficientb4_model(hp):
    # hp_conv_layers = hp.Int("conv_layers", 2, 4, default=3)
    hp_dense_layers = hp.Int("dense_layers", 1, 4, default=3)
    hp_dense_units = hp.Int("dense_units", 4, 64, step = 4, default = 2)
    hp_dropout_rate=hp.Choice("dropout_rate", values=[0.0, 0.1, 0.2, 0.3, 0.4, 0.5])
    hp_activation=hp.Choice('activation', values=['relu', 'tanh'])
    hp_optimizer = hp.Choice('optimizer', values=['adam', 'SGD', 'rmsprop'])
    optimizer = tf.keras.optimizers.get(hp_optimizer)
    optimizer.learning_rate = hp.Choice("learning_rate", [0.1, 0.01, 0.001], default=0.01)
    inputs = tf.keras.Input(shape=input_shp)
    x=inputs
    base_model = tf.keras.applications.EfficientNetB4(include_top=False, weights='imagenet', input_shape = input_shp, poolin
    base_model.trainable = False
    x = base_model(x)
    for i in range(hp_dense_layers):
        x = tf.keras.layers.Dense(hp_dense_units, activation=hp_activation)(x)
    x = tf.keras.layers.Dropout(hp_dropout_rate)(x)

    # The last layer contains 10 unitsfor the number of classes.
    outputs = tf.keras.layers.Dense(units=1, activation="sigmoid")(x)
    model = tf.keras.Model(inputs=inputs, outputs=outputs)
    model.compile(
        loss="binary_crossentropy", metrics=["accuracy"], optimizer=optimizer,
    )
    return model
```

```
effb4_tuner = kt.Hyperband(
    build_efficientb4_model,
    objective='val_accuracy',
    max_epochs=10,
    hyperband_iterations=2,
    overwrite=True)
```

```
effb4_tuner.search(X_train, y_train, epochs=10, validation_data=(X_val, y_val))
```

Figure 13: HyperBand Tuning of EfficientNetB4

6.1.3 CNN Base Model

The Figure 14 shows the model summary of the basic CNN derived by tuning with stateof-art hyperband algorithm.

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 156, 156, 3)]     0

 conv2d (Conv2D)             (None, 156, 156, 36)      1008

 max_pooling2d (MaxPooling2D  (None, 78, 78, 36)       0
 )

 conv2d_1 (Conv2D)           (None, 78, 78, 8)         7208

 max_pooling2d_1 (MaxPooling  (None, 39, 39, 8)        0
 2D)

 conv2d_2 (Conv2D)           (None, 39, 39, 36)        7236

 max_pooling2d_2 (MaxPooling  (None, 20, 20, 36)       0
 2D)

 conv2d_3 (Conv2D)           (None, 20, 20, 36)        11700

 max_pooling2d_3 (MaxPooling  (None, 10, 10, 36)       0
 2D)

 flatten (Flatten)           (None, 3600)              0

 dropout (Dropout)           (None, 3600)              0

 dense (Dense)               (None, 1)                 3601

=================================================================
Total params: 30,753
Trainable params: 30,753
Non-trainable params: 0
```

Figure 14: Base CNN Model Summary

6.2 NLP based Models

6.2.1 Word2Vec Embedding

- Word2Vec model from Gensim is used for training the embedding space and embedding the opcode sequences intelligently. This is illustrated in the Figure 15

```
#Word2Vec Vectorization
w2v = Word2Vec(embedding_input, vector_size=100,
                                window=5,
                                min_count=2,
                                workers = 3)
```

Figure 15: Word2Vec Training

- After the training is over the generated word to vector embeddings are stored in the local file system using the code provided in the Figure 16.
- This vector file is then used to generate an embedding matrix in the in order to load the embeddings into the embedding layer of the sentiment analysis models.

```
#Save the generated word vectors in the local file system
word_vectors = w2v.wv
word_vectors.save("word2vec.wordvectors")
```

Figure 16: Saving Word2Vec Embeddings

The code for generating the embedding matrix is provided in the Figure 17.Simultaneously, Keras Tokenizer is also used to tokenize the text in order to prepare the data to pass on to the neural network. The token generation using Tokenizer is explained in the Figure 18.

```
#Initializing embedding matrix
nb_words = min(vocab_size, len(word_index))+1
embedding_matrix = np.zeros((nb_words, embedding_dim))
for word, i in word_index.items():
    if word in wv.index_to_key:
        embedding_matrix[i] = wv.get_vector(word)
print('Null word embeddings: %d' % np.sum(np.sum(embedding_matrix, axis=1) == 0))
```

Figure 17: Generating Embedding Matrix

```
#Setting Hyperparameters for Keras Tokenizer
vocab_size = len(vocab)
oov_tok = ''
embedding_dim = 100
max_length = 200 # choose based on statistics, for example 150 to 200
padding_type='post'
trunc_type='post'
# tokenize sentences
tokenizer = Tokenizer(num_words = vocab_size, oov_token=oov_tok, filters="")
tokenizer.fit_on_texts(X_train)
word_index = tokenizer.word_index
# convert train dataset to sequence and pad sequences
train_sequences = tokenizer.texts_to_sequences(X_train)
train_padded = pad_sequences(train_sequences, padding='post', maxlen=max_length)
# convert Test dataset to sequence and pad sequences
test_sequences = tokenizer.texts_to_sequences(X_test)
test_padded = pad_sequences(test_sequences, padding='post', maxlen=max_length)
print(f'Word Index Length : {len(word_index)}')
```

```
Word Index Length : 85480
```

Figure 18: Tokenizer Initialization

- A LSTM model with max input length of 200 is designed along with one Dense layer as shown in the Figure 19. Also, visible in the figure is the number of LSTM units used and which is 128.
- A Bi-LSTM model with same input length is designed as described in the Figure 20. Similar to the previous model 128 Bi-LSTM units were used in the Bi-LSTM model.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 200, 100)          9876900

 lstm (LSTM)                 (None, 128)               117248

 dense (Dense)               (None, 64)                8256

 dense_1 (Dense)             (None, 32)                2080

 dense_2 (Dense)             (None, 1)                 33

=================================================================
Total params: 10,004,517
Trainable params: 10,004,517
Non-trainable params: 0
_________________________________________________________________
```

Figure 19: Summary of one of the LSTM model used

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, 200, 100)          9876900

 bidirectional (Bidirectiona  (None, 128)              84480
 l)

 dense_3 (Dense)             (None, 64)                8256

 dense_4 (Dense)             (None, 32)                2080

 dense_5 (Dense)             (None, 1)                 33

=================================================================
Total params: 9,971,749
Trainable params: 9,971,749
Non-trainable params: 0
```

Figure 20: Summary of one of the Bi-LSTM model used

References

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A. and Talwalkar, A. (2017). Hyperband: a novel bandit-based approach to hyperparameter optimization, *The Journal of Machine Learning Research* **18**(1): 6765–6816.

Taheri, L., Kadir, A. F. A. and Lashkari, A. H. (2019). Extensible Android Malware Detection and Family Classification Using Network-Flows and API-Calls, *2019 International Carnahan Conference on Security Technology (ICCST)*, pp. 1–8. ISSN: 2153-0742.