Baseline Implementation

Adam Woods - W17014929

KF5012 Software Engineering Practice - AI Stream
Word Count: 420

February 2020

1 Baseline Implementation

The purpose of the baseline implementation was to start on the basic functionality of the program in a way that easily allows for further development during the iterative development stage. This meant that other details of the program can be added without changing the entire program and if a part of the program did need changes they could be done easily. This meant that a module design had to be adopted. This section will cover the structure of the baseline program, the functionality of it at its baseline state, the baseline's performance and what the next steps for the program is.

1.1 Structure and Functionality

The functionality of the baseline implementation is very limited and only includes the absolute necessities for the program to do its primary job. This primary job of course is to identify if a given message is spam or not. Therefore, in its baseline state the program only does a few things as seen in figure X. As you can see the program will be broken into three parts, main.py, model.py and utils.py. This is our modular design and will allow one part to be changed without needing to alter the whole program.

1.2 Performance

The performance of the baseline implementation is quite good as seen in figure X and Y. These results were achieved using a test size of 20% meaning that we train on 4457 samples and validate on 1115 samples. By the 20th epoch we have a training accuracy of 99.98% and a loss value of 0.0013 and we have a test accuracy of 99.19% and a loss value of 0.0562. It is worth noting that no dropout is used and the data being used is heavily skewed in favour of real messages as seen in figure X.

1.3 Further Development

In the iterative development stage functionality will be added to the program so that the model can be trained on the data and then messages can be input for the model to predict. Work will be done on the dataset to try and negate its imbalanced nature and more work will be done on the pre-processing of data, perhaps using PCA or LDA to perform dimensionality reduction. More useful functions that come with keras can be used to improve performance such as earlyStopping which will ensure that the model never over-fits the data. Finally, functionality will be added to identify which words are more indicative of a spam message and other similar analysis on the data.

Loads the data set	
Converts the data into and integer format using CountVectorizer	main.py
Splits it into testing and training using 20% as test data	
Trains a model using the training data and tests it against the test data	model.py
Produces a graph of the loss over epoch and accuracy over epoch	utils.py

Figure 1: General steps for the baseline implementation.

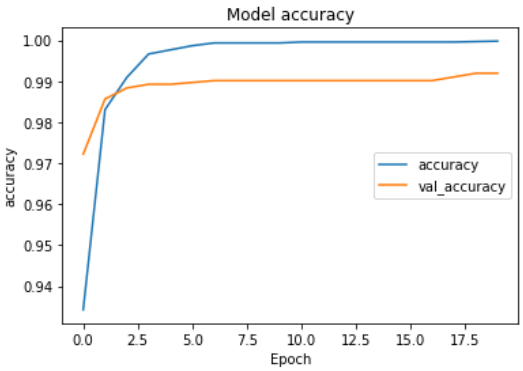

Figure 2: Accuracy over epochs graph.

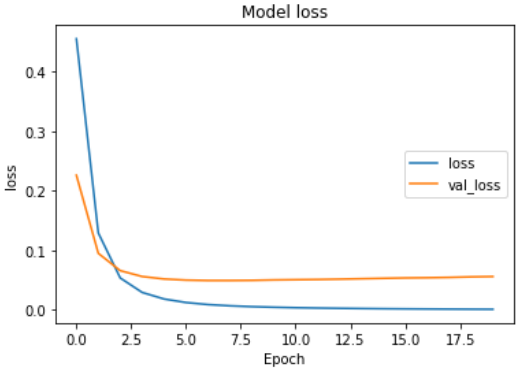

Figure 3: Loss value over epochs graph.

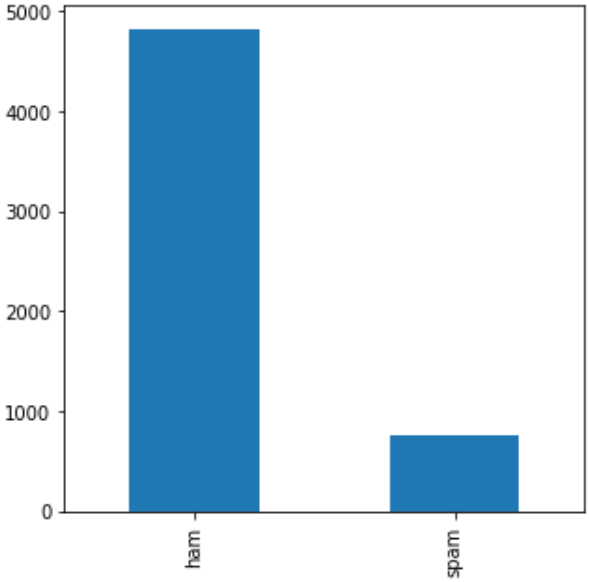

Figure 4: Chart showing the balance of data.

2 Code

2.1 Main

```
# -*- coding: utf-8 -*-
"""
Created on Sat Mar 21 14:52:48 2020
@author: adamw
"""
```

```
#Imports
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from model import train_model

pd.set_option('display.max_colwidth', 1000)

#Loads the csv file
dataframe = pd.read_csv("Data/sms_messages.csv")

#splits the labels and features using the tabs at the top of the file
df_label = dataframe['Category']
df_feature = dataframe['Message']

#converts the strings into numeric vector using BoW
def create_doc_feature_df(sparse_mat, feature_names):
    return(pd.DataFrame.sparse.from_spmatrix(sparse_mat, columns=feature_names))
count_vect = CountVectorizer(stop_words='english')
#converrts messages to BoW
count_vect.fit(df_feature)
feature = create_doc_feature_df(count_vect.transform(df_feature),
count_vect.get_feature_names())

#converts labels into numeric/binary values
count_vect.fit(df_label)
label = create_doc_feature_df(count_vect.transform(df_label),
count_vect.get_feature_names())

#split the new data into test and train sets
x_train, x_test, y_train, y_test = train_test_split(feature,
label, test_size=0.2, random_state=42)
train_model(x_train, x_test, y_train, y_test)
```

2.2 Model

```
# -*- coding: utf-8 -*-
"""
Created on Sat Mar 21 14:42:20 2020
@author: adamw
"""
#imports
from keras import Sequential
from keras.layers import Dense
from utils import test_train_epoch_graph
```

```
#model function called by main.py
def train_model(x_train, x_test, y_train, y_test):
    model = Sequential()
    model.add(Dense(50, input_dim=8440, activation='relu'))
    model.add(Dense(2, activation='sigmoid'))
    model.compile(loss='binary_crossentropy',
    optimizer='adam', metrics=['accuracy'])

    history = model.fit(x_train,y_train, epochs=20,
    batch_size=50, validation_data=(x_test, y_test))
    test_train_epoch_graph(history, 'loss', 'val_loss')
    test_train_epoch_graph(history, 'accuracy', 'val_accuracy')
    return history
```

2.2.1 Utils

```
# -*- coding: utf-8 -*-
"""
Created on Sat Mar 21 14:44:42 2020
@author: adamw
"""
#imports
import matplotlib.pyplot as plt

#graph plot function ccalled by model.py
def test_train_epoch_graph(history, train, test):
    plt.plot(history.history[train])
    plt.plot(history.history[test])
    plt.title('Model '+ train)
    plt.ylabel(train)
    plt.xlabel('Epoch')
    plt.legend([str(train), str(test)], loc='center right')
    plt.show()
```