Spam Classification using Logistic Regression

by Anshuman Singh

Task

In this assignment, you will build a spam classifier from scratch. No training data will be provided. You are free to use whatever training data that is publicly available/does not have any copyright restrictions (You can build your own training data as well if you think that is useful). You are free to extract features as you think will be appropriate for this problem. The final code you submit should have a function/procedure which when invoked will be able to automatically read a set a emails from a folder titled test in the current directory. Each file in this folder will be a test email and will be named 'email.txt' ('email1.txt', 'email2.txt', etc). For each of these emails, the classifier should predict +1 (spam) or 0 (non Spam). You are free to use whichever algorithm learnt in the course to build a classifier (or even use more than one). The algorithms (except SVM) need to be coded from scratch. Your report should clearly detail information relating to the data-set chosen, the features extracted and the exact algorithm/procedure used for training including hyperparameter tuning/kernel selection if any. The performance of the algorithm will be based on the accuracy on the test set.

Datasets used

For training the classifier model, the following datasets were used:

1. **Enron email dataset**: A subset of the original dataset containing emails from employees of the Enron corporation, which have been categorized into spam and ham (legitimate emails). The dataset used had around 30,000 emails with 16,000 ham and 14,000 spam emails. Dataset link: https://www.kaggle.com/datasets/purusinghvi/email-spam-classification-dataset
2. **SpamAssassin Public Corpus**: A popular dataset compiled by the SpamAssassin project. It contains around 14,000 labeled email messages, with almost equal number of spam and ham emails. Dataset link: https://spamassassin.apache.org/old/publiccorpus/
3. **Ling-Spam Dataset**: It contains a mix of emails from the Linguistic Institute mailing list and spam emails. The dataset contains 2400 ham and 500 spam emails. Dataset link: https://www.kaggle.com/datasets/mandygu/lingspam-dataset
4. **CEAS:** It contains the dataset taken from the CEAS 2008 Spam Filtering Challenge. The dataset contains approx. 17300 ham and 21800 spam emails. Dataset link: https://www.kaggle.com/datasets/naserabdullahalam/phishing-email-dataset
5. **Nigerian fraud**: Contains Nigerian Prince phishing email dataset having 3300 spam emails. Dataset link: https://www.kaggle.com/datasets/naserabdullahalam/phishing-email-dataset
6. **Nazario Spam Email Corpus:** It's well-known spam email dataset that was released by security researcher Jose Nazario. It contains a collection of spam emails gathered by a honeypot (systems designed to lure spammers), used for various research tasks such as spam filtering, email

classification, and security studies. The dataset used for training had 1400 spam and 140 ham emails. Dataset link: https://www.kaggle.com/datasets/naserabdullahalam/phishing-email-dataset

7. **SMS Spam dataset**: This dataset contains SMS messages that are labeled as either spam or ham. It is widely used for spam detection tasks in natural language processing. This dataset had total 5572 messages, with 4825 ham and 747 spam messages. Dataset link: https://archive.ics.uci.edu/dataset/228/sms+spam+collection

In total, the combined dataset used for building the model had around 46k spam and 47k ham emails. The datasets are saved in their original format in the `raw_datasets` folder.

Pre-processing

The datasets used for training the algorithm had different forms. All of the datasets were present in tabular form (.csv files) except SpamAssassin dataset, which was present in separate .txt files. So, in order to keep the pre-processing process consistent, the tabular datasets were converted into .txt files, with each email having a separate .txt file. This was done by appropriately combining the data present in each column of the tabular dataset into a single column and then saving each email data present in the combined column into a separate .txt file. The spam and ham emails are stored in the `dataset/spam` and `dataset/ham` folders inside the working directory.

	sender	receiver	date	subject	body	urls	label
0	MR. JAMES NGOLA. <james_ngola2002@maktoob.com>	webmaster@aclweb.org	Thu, 31 Oct 2002 02:38:20 +0000	URGENT BUSINESS ASSISTANCE AND PARTNERSHIP	FROM:MR. JAMES NGOLA.\nCONFIDENTIAL TEL: 233-2...	0	1
1	Mr. Ben Suleman <bensul2004nng@spinfinder.com>	R@M	Thu, 31 Oct 2002 05:10:00 -0000	URGENT ASSISTANCE /RELATIONSHIP (P)	Dear Friend,\n\nI am Mr. Ben Suleman a custom ...	0	1
2	PRINCE OBONG ELEME <obong_715@epatra.com>	webmaster@aclweb.org	Thu, 31 Oct 2002 22:17:55 +0100	GOOD DAY TO YOU	FROM HIS ROYAL MAJESTY (HRM) CROWN RULER OF EL...	0	1
3	PRINCE OBONG ELEME <obong_715@epatra.com>	webmaster@aclweb.org	Thu, 31 Oct 2002 22:44:20 -0000	GOOD DAY TO YOU	FROM HIS ROYAL MAJESTY (HRM) CROWN RULER OF EL...	0	1
4	Maryam Abacha <m_abacha03@www.com>	R@M	Fri, 01 Nov 2002 01:45:04 +0100	I Need Your Assistance.	Dear sir, \n \nIt is with a heart full of hope...	0	1
5	Kuta David <davidkuta@postmark.net>	davidkuta@yahoo.com	Sat, 02 Nov 2002 06:23:11 +0000	Partnership	ATTENTION: ...	0	1
6	Barrister tunde dosumu <tunde_dosumu@lycos.com>	NaN	NaN	Urgent Attention	Dear Sir,\n\nI am Barrister Tunde Dosumu (SAN)...	1	1

Fig. 1: The Nigerian dataset preview. Nigerian fraud emails contained sender data, receiver data, date, subject, body content, count of urls present in the dataset, alongwith the email label (0: ham, 1: spam). It only had spam emails.

This entire process of converting .csv email datasets to separate email .txt files was carried out for all the following datasets:

Nigerian Fraud, Ling, Nazario, CEAS_08, Enron and SMS spam.

Code for the conversion is given in `Csv2Text.py` file for Nigerian Fraud dataset. For other datasets, the code is almost the same with minor modifications based on the features columns given in the original datasets.

After this step, we proceed to loading the email .txt files and then processing the loaded data to make it suitable for training our machine learning model. First, we go through each email text file, read the text file, and store the text as well as the associated label (spam/ham) in a **pandas** dataframe. Then, we proceed to carrying out pre-processing of each of these emails. The pre-processing involves the following steps:

1. **Cleaning html content:**

   In case the email contains html content, we need to remove it. We do it using the `html_parser()` function. The function uses regular expression to clean up HTML content by removing tags, handling hyperlinks, condensing newlines, and decoding common HTML entities, producing plain text suitable for further analysis. This function uses the `remove_escape_characters_and_punctuations()` as a sub-routine, the working of which is explained next.

2. **Removing escape characters and punctuation characters from the text:**

   Using the `remove_escape_characters_and_punctuations()` function, we replace the following escape characters `'\n' '\t' and '\r'` and the following punctuation characters `'!"#$%&\'()*+,-./:;⇔?@[\\]^_`{|}~'` with `" "`. This cleans up the email text for further processing. This is implemented using the `remove_escape_characters_and_punctuations()` function.

3. **Removing urls:** The `url_swapper()` function is a utility to identify and replace URLs in a text with the `"URL"` placeholder. It is useful for simplifying text data by removing URLs, which are not relevant in their original form for the classification task. It also makes use of regular expressions.

4. **Stemming:** We have used **nltk** package to perform word stemming implemented through the `stemmer()` function.

   Word stemming is the process of reducing a word to its base or root form by removing prefixes, suffixes, and inflections. The goal is to group different forms of a word (like "running," "runner," and "ran") into a common root (like "run"), so that they are treated as the same term in text analysis. Stemming reduces vocabulary Size by converting words to their base form. It also reduces the total number of unique words, simplifying the model's vocabulary and improving memory and computational efficiency. In machine learning tasks, stemming helps models generalize better by treating different word forms as the same feature, improving accuracy. We will be implementing Porter Stemming. Porter Stemming applies a series of predefined rules by removing common word suffixes like "ing," "ed," "ly," and others in a structured way.

All the above pre-processing functions have been combined into a single function named `email_to_text` , saved in the `Email2Text.py` file.

NOTE: Applying the `email_to_text` pre-processing function to each email is a time taking process and may take upto 15 minutes. To save time, we will use the output from a previous run, saved as `processed_emails.csv` in the work directory. To carry out the pre-processing again, run the `ProcessingTrainingData.py` file. The code in the file will use the email .txt files stored inside `dataset/ham` and `dataset/spam` folders to create the `processed_email.csv` file again.

Splitting training and test datasets

Now, after implementing the pre-processing steps, we have our dataset in the form of a pandas dataframe. The “content” column of the pandas dataframe has the email content and the “label” column has the labels (0 for ham and 1 for spam). Before proceeding further, we split our dataset into training and test datasets so that we can test the performance of our algorithms later using the test dataset. We do this using the random module of numpy library. 30% of the total emails are randomly chosen for the test dataset and rest of the emails are put in the training dataset.

Tokenization

After carrying out the pre-processing, we are ready to convert the text into tokens. We will create a custom class called `CustomCountVectorizer` (inspired from sklearn's class `CountVectorizer`) for tokenization.

- **Fitting:** (implemented using `.fit` method) During the fitting process, the `CustomCountVectorizer` class will first go through the pre-processed email texts and identify unique words, creating a vocabulary set (using dictionary class of python) and storing each of these unique words along with their counts. If the frequency of a word in the vocabulary set is less than `min_df` parameter or more than the `max_df` parameter, then that word is dropped from the vocabulary set.
- **Transforming:** (implemented using `.transform` or `.fit_transform` method) During the transform process, the final vocabulary obtained from the fitting step is used to convert all the texts into a feature matrix, where each feature is a unique word present in the vocabulary and the value is the no. of times that word appears in a particular email. Any word which is not present in the vocabulary is ignored during this process.

For our dataset, we used the value of `min_df = 2` . This means that any word which appeared only once in the training texts is dropped. This is done to remove occurrences like unique numerals, combinations of numerals and texts, as well as mis-spelled words, as these are very unlikely to appear again in any email.

Also, we have used the value of `max_df = 0.85` . This means that any word which has a `0.85` or higher frequency (meaning that it appears in more than 84 out of 100 texts) is removed. This is done to drop very common words like `a` , `the` , `and` etc. as these words don't carry any useful information which might help us in our spam classification task.

The above values of `min_df` and `max_df` give us a vocabulary of `119539` words.

The `CustomCountVectorizer` is present in the `CustomCountVectorizer.py` file in the working directory.

Sparse matrix

Tokenization creates a very high dimensional feature matrix. This matrix is sparse in nature as each email contains a fraction of the total words present in the vocabulary. Therefore, `CustomCountVectorizer` converts this feature matrix, originally stored as a dense matrix, into a sparse matrix. This is a compulsory step because if we try to run our algorithms on a dense matrix representation of the feature matrix, then we will run out of memory and our system will crash. Hence, all the operations from the tokenization step onwards till the prediction step (in the machine learning algorithm) are carried out in sparse matrix representation.

To convert dense matrix to sparse matrix, we have used `scipy` library. `scipy` is a standard python library used for scientific and technical computing. We use the `csr_matrix` function from the `scipy.sparse` module for conversion to sparse matrix. This conversion is done by default in the `CustomCountVectorizer` class for the feature matrix.

Also, the label array `y_train` needs to be converted to a sparse array in order to be compatible with the sparse feature matrix. The function used for the same is as follows:

```
## Creating function to convert y array to sparse
def y_to_csr(y):
    """
    Function to convert y array to csr format
    """
    row = []
    col = [0]*y.shape[0]
```

```
    data = []
    for ind, val in enumerate(y):
        row.append(ind)
        data.append(val)
    y_csr = csr_matrix((data, (row, col)), shape = (y.shape[0], 1))
    return y_csr
```

Metrics for evaluation

For evaluating the performance of our algorithm, we will be using the following metrics:

1. **Precision:** It measures the accuracy of the positive predictions made by the model. It indicates the proportion of true positive predictions among all positive predictions:

   ```
   precision = True Positive/ (True Positive + False Positive)
   ```

2. **Recall:** It measures the model's ability to identify all relevant positive instances. It indicates the proportion of true positive predictions among all actual positive instances:

   ```
   recall = TruePositive/(True Positive + False Negative)
   ```

3. **f1_score:** The F1 Score is the harmonic mean of precision and recall. It provides a balance between the two metrics, especially useful in situations where there is an uneven class distribution. A higher F1 score indicates better performance.

   ```
   f1 = (2 * precision * recall)/(precision + recall)
   ```

Precision-recall tradeoff: Since we are dealing with binary classification tasks, we need to monitor both precision and recall. There is a trade-off between precision and recall, and to increase one, the other has to be decreased. This can be done by varying the threshold value. By default, the threshold value is 0.5, meaning prediction is given as positive if the probability is greater than or equal to 0.5. If we increase the threshold, say to 0.6, then a higher probability would be required to get predicted as positive class, resulting in higher precision but lower recall. For the task of spam classification, high precision is important because we don't want important mails to end up in the spam box.

These metrics are calculated using the function `classification_metrics` defined in the `ClassificationMetrics.py` file.

Building the Logistic Regression classifier

We will be using logistic regression algorithm. **Logistic Regression** algorithm has been used to carry out the binary classification task in this assignment. Gradient descent is used to find the optimum weights for the logistic regression model.

The log of maximum likelihood function for logistic regression is given by:

$$log(L) = \Sigma_i^n [y_i * log(sig(w^T * x_i)) + (1 - y_i) * log(1 - sig(w^T * x_i))]$$

The above equation doesn't have a closed form solution. Therefore, to maximize the likelihood function written above, we will apply gradient ascent. The vectorized form of gradient of the above function is give by:

$$\nabla(log(L)) = X * (Y - sig(X^T * w))$$

where,

X is the d x n data matrix with d features and n samples

Y is the label vector with dimensions n x 1

$sig(X^T * w)$ means that sigmoid function is applied to each entry of the vector $X^T * w$, resulting in a vector of n x 1

The gradient ascent step is given by the following equation:

$$w^{t+1} = w^t + \eta * \nabla(log(L(w^t)))$$

where,

η is the step-size

The sigmoid function is given as:

$$sigmoid(x) = \frac{1}{1+e^{-x}}$$

If we apply the above sigmoid function directly, we will run into overflow problems while fitting the logistic regression algorithm. To avoid the overflow problem, we have used the following implementation of the sigmoid function using numpy:

```
import numpy as np

def sigmoid(x):
    """
    Implementation of the sigmoid function
    x is a numpy array
    """
    return np.exp(-np.logaddexp(0, -x))   ## using this to avoid overflow
```

The above implementation prevents overflow for small values of x by using log. It is present in the `Sigmoid.py` file inside the working directory.

We fit the logistic regression model on the training dataset and then evaluate the results on the training and test dataset using the `classification_metrics` function. The parameters used while training the logistic regression model are: `threshold=0.5`, `step_size_multiplier=80`, and `max_iter=5*10**3`. The results are as follows:

Training dataset (for threshold = 0.5):

```
recall: 0.994951069002057
precision: 0.9951371571072319
f1_score: 0.995044104354331
```

Test dataset (for threshold = 0.5):

```
recall: 0.9802655711246424
precision: 0.9812734082397003
f1_score: 0.9807692307692307
```

The above results show decent performance on both the training and the test dataset. This shows that the model is able to generalize well on unseen data. Also, both precision and recall for the model are having high values, which is the ideal case.

The entire training process is carried out by executing the `Training.py` file. It contains the `training_logreg()` function, which gets executed when the file is run. The `training_logreg()` function does everything from loading the data to training the model. The training process needs to output the final fitted logistic regression model and fitted count vectorizer so that predictions can be made on new email .txt files. For this, we use pickling, which is implemented using the `pickle` library. We export the trained logistic regression model and fitted count vectorizer as pickle files which are then read during the testing process for making predictions. All this is done by executing the `Training.py` itself.

Testing process

The testing process has only two steps:

1. Place the emails to be tested in the `test` folder inside the working directory.
2. Run the `Testing.py` file.

The `Testing.py` file contains the code for testing new email .txt files present inside the `test` folder. The `Testing.py` file has the `testing()` function, which is executed to carry out the testing. The function loads the saved logistic regression model and count vectorizer pickle files (saved during training). The fitted count vectorizer is required in testing for transforming the test emails into a feature matrix. Also, we can't use a new instance of count vectorizer in testing because the test feature matrix should be created using the same set of word vocabulary which was used to create the training feature matrix.

After loading the trained logistic regression model and fitted count vectorizer, the `testing()` function loads the emails from .txt files present in the test folder, pre-processes them, vectorizes them using count vectorizer, and then makes predictions on them using the trained logistic regression model. The final predictions are then saved as `preds.csv` file inside the `test` folder. The `preds.csv` file has two columns, one

indicating the email filename and the other indicating whether it is a spam (1) or ham (0). The entire testing process takes around 10 to 15 minutes.

Instructions for running and testing the model

1. The testing process has only two steps:
   a. Place the emails to be tested in the `test` folder inside the working directory.
   b. Run the `Testing.py` file.
2. All the source code is written in Python and give in .py file.
3. Make sure that `nltk` library is installed in the environment in which you are running the code.
4. While running any source code file, the working directory should be the folder in which all source code files are present. So, change your working directory if that is not the case.
5. Do not change the name or location of any sub-file or sub-folder.
6. Due to size restriction of moodle (150 MB), the files present in the `dataset` and `raw_datasets` folders have not been added to the final zip file being uploaded on moodle. The google drive links for these two files are as follows:

   `dataset` : https://drive.google.com/file/d/1xYYXzL8A5bt2Rj4m7GGQZWzQCm974pP5/view?usp=sharing

   `raw_datasets` : https://drive.google.com/open?id=1xZMeuSzCixqih9IqOPuuNODfqRzVkUXk&usp=drive_fs

   Due to the absence of ham and spam .txt files in `dataset` folder, the `ProcessingTrainingData.py` file (used for the pre-processing of email .txt files from scratch) won't execute. If the `ProcessingTrainingData.py` is to be executed to carry out pre-processing, kindly download the spam and ham email .txt files from the `dataset` link mentioned above and place the emails in the `spam` and `ham` folders respectively before running the `ProcessingTrainingData.py` file.

Source Codes

▼ **Csv2Text**

```
import pandas as pd
import os

# specifying the .csv dataset path for Nigerian Fraud emails
spam_datasets_path = os.path.join(os.cwd(), "raw_datasets")

df_Nigerian_Fraud = pd.read_csv(os.path.join(spam_datasets_path, "Nigerian_Fraud.csv"))  ##loading the dataset
df_Nigerian_Fraud = df_Nigerian_Fraud.fillna("") ## imputing the NaN values with ""

## combining all the columns into a single email column
df_Nigerian_Fraud["sender"] = "Sender: " + df_Nigerian_Fraud["sender"]
df_Nigerian_Fraud["receiver"] = "Receiver: " + df_Nigerian_Fraud["receiver"]
```

```
df_Nigerian_Fraud["date"] = "Date: " + df_Nigerian_Fraud["date"]
df_Nigerian_Fraud["subject"] = "Subject: " + df_Nigerian_Fraud["subject"]
df_Nigerian_Fraud["body"] = "Body: " + "\n" + "\n" +  df_Nigerian_Fraud["body"]
df_Nigerian_Fraud["email"] = df_Nigerian_Fraud["sender"] + "\n" + df_Nigerian_Fraud["receiver"]
+ "\n" + df_Nigerian_Fraud["date"] + "\n" + df_Nigerian_Fraud["subject"]+ "\n" + df_Nigerian_Fra
ud["body"]
df_Nigerian_Fraud["email"] = df_Nigerian_Fraud.apply(lambda row: row["email"] + " URL "*int(row
["urls"]), axis = 1)

save_path = os.path.join(os.cwd(), "dataset")
ham_path = os.path.join(save_path, "ham")
spam_path = os.path.join(save_path, "spam")

## function to save the emails from pandas dataframe to .txt file
def write_email_to_file(df, ham_path, spam_path):
    """
    Email should be in column named "email"
    label should be in column named "label"
    label: 1 for spam and 0 for ham
    """
    for i, text in enumerate(df.loc[df["label"] == 0, "email"]):
        with open(os.path.join(ham_path, f"nigerian_fraud_ham_{i}.txt"), 'w') as f:
            f.write(text)

    for i, text in enumerate(df.loc[df["label"] == 1, "email"]):
        with open(os.path.join(spam_path, f"nigerian_fraud_spam_{i}.txt"), 'w') as f:
            f.write(text)

write_email_to_file(df_Nigerian_Fraud, ham_path, spam_path)
```

▼ **ClassificationMetrics**

```
import numpy as np

def classification_metrics(y_true, y_preds):
    """
    y_true: True values
    y_preds: predicted values
    """
    y_true = np.array(y_true).ravel()
    y_preds = np.array(y_preds).ravel()

    true_pos = np.sum(np.where((y_preds == 1) & (y_true == 1), 1, 0))
    false_pos = np.sum(np.where((y_preds == 1) & (y_true == 0), 1, 0))
    true_neg = np.sum(np.where((y_preds == 0) & (y_true == 0), 1, 0))
    false_neg = np.sum(np.where((y_preds == 0) & (y_true == 1), 1, 0))

    recall = true_pos/(true_pos + false_neg)
    precision = true_pos/(true_pos + false_pos)
    f1_score = 2*precision*recall/(precision + recall)
```

```
print(f"recall: {recall}")
print(f"precision: {precision}")
print(f"f1_score: {f1_score}")
```

▼ **CustomCountVectorizer**

```
## Defining the CustomCountVectorizer class
from collections import Counter
from scipy.sparse import csr_matrix  ## for converting dense matrix to sparse matrix

class CustomCountVectorizer:
    def __init__(self, min_df=1, max_df=1.0):
        """
        Initializes the CustomCountVectorizer with min_df and max_df options.

        Parameters:
        - min_df: Minimum document frequency (ignores words in fewer documents than this).
        - max_df: Maximum document frequency (ignores words in more documents than this).
        """
        self.min_df = min_df
        self.max_df = max_df
        self.vocabulary_ = {}

    def fit(self, texts):
        """
        Fits the vocabulary on the provided list of texts.

        Parameters:
        - texts: List of documents (texts).
        """
        doc_count = len(texts)
        term_doc_freq = Counter()

        for text in texts:
            terms = set(text.lower().split())
            term_doc_freq.update(terms)

        # Apply min_df and max_df thresholds
        min_docs = self.min_df if isinstance(self.min_df, int) else int(self.min_df * doc_count)
        max_docs = self.max_df if isinstance(self.max_df, int) else int(self.max_df * doc_count)

        self.vocabulary_ = {
            term: idx for idx, (term, freq) in enumerate(term_doc_freq.items())
            if min_docs <= freq <= max_docs
        }

        # Re-index vocabulary to keep it contiguous
        self.vocabulary_ = {term: i for i, (term, idx) in enumerate(self.vocabulary_.items())}

    def transform(self, texts):
```

```
        """
        Transforms the texts into a sparse document-term matrix based on the vocabulary.

        Parameters:
        - texts: List of documents (texts).

        Returns:
        - Sparse document-term matrix with each row representing a document and columns represe
nting terms.
        """
        rows, cols, data = [], [], []

        for i, text in enumerate(texts):
            term_counts = Counter(text.lower().split())
            for term, count in term_counts.items():
                if term in self.vocabulary_:
                    idx = self.vocabulary_[term]
                    rows.append(i)
                    cols.append(idx)
                    data.append(count)

        # Create sparse matrix in CSR format
        sparse_matrix = csr_matrix((data, (rows, cols)), shape=(len(texts), len(self.vocabulary_)), dty
pe=int)
        return sparse_matrix

    def fit_transform(self, texts):
        """
        Combines fit and transform into one step for convenience.

        Parameters:
        - texts: List of documents (texts).

        Returns:
        - Sparse document-term matrix.
        """
        self.fit(texts)
        return self.transform(texts)
```

▼ **Email2Text**

```
################################ Importing dependencies ##########################
##

## The cwd path should be the path to the folder in which the .py file containing code is present
## All other supporting functions are also present in the same directory

import nltk
import re
import string
```

```
#*****************************************************************************


# function to remove escape characters and punctuations
def remove_escape_characters_and_punctuations(text):
    # Replace escape characters with a space
    cleaned_text = text.replace('\n', ' ').replace('\t', ' ').replace('\r', ' ')

    # Replace punctuation with spaces
    cleaned_text = cleaned_text.translate(str.maketrans(string.punctuation, ' ' * len(string.punctuati
on)))

    # Remove any extra spaces that may have been added
    cleaned_text = ' '.join(cleaned_text.split())

    return cleaned_text


# html parser
def html_parser(html):
    # Step 1: Remove content within <head> tags
    text = re.sub('<head.*?>.*?</head>', '', html, flags=re.M | re.S | re.I)

    # Step 2: Replace <a> tags with "HYPERLINK"
    text = re.sub('<a\s.*?>', ' HYPERLINK ', text, flags=re.M | re.S | re.I)

    # Step 3: Remove all remaining HTML tags
    text = re.sub('<.*?>', '', text, flags=re.M | re.S)

    # Step 4: Remove extra newlines
    text = re.sub(r'(\s*\n)+', '\n', text, flags=re.M | re.S)

    # Step 5: Replace common HTML entities manually
    html_entities = {
        '&amp;': '&',
        '&lt;': '<',
        '&gt;': '>',
        '&quot;': '"',
        '&#39;': "'",
        ' ': ' '
    }
    for entity, char in html_entities.items():
        text = text.replace(entity, char)

    return remove_escape_characters_and_punctuations(text)

## Url detector and swapper
def url_swapper(text):
    if not isinstance(text, str):
        return ""
```

```
    # Define a regex pattern to identify URLs
    url_pattern = re.compile(
        r'((http|https|ftp):\/\/)?'         # Protocol (optional)
        r'(\w+(\-\w+)*\.)+'                 # Subdomain(s)
        r'([a-z]{2,6})'                     # Domain
        r'(:\d+)?'                          # Port (optional)
        r'(\/[\w\-\.\~\?=&%\+;]*)*',        # Path (optional)
        flags=re.IGNORECASE
    )

    # Substitute all matched URLs with "URL"
    text = url_pattern.sub(" URL ", text)

    return text

# stemmer function
def stemmer(text):
    ps = nltk.PorterStemmer()
    text_list = text.split(" ")
    out_list = [ps.stem(w) for w in text_list]
    return " ".join(out_list)

## combining all the pre-processing functions
def email_to_text(email):
    email = html_parser(email)   ## parsing html content
    email = url_swapper(email)   ## replacing urls with URL
    email = stemmer(email)       ## carrying out stemming
    return email
```

▼ **LogisticRegression**

```
import numpy as np

from scipy.sparse import csr_matrix, hstack   ## for converting dense matrix to sparse matrix

from Sigmoid import sigmoid

class logistic_regression():
    """
    X should be a sparse matrix of dimensions nxd
    y should be an numpy array of dimensions nx1
    """
    np.random.seed(23)   #setting random seed for reproducibility

    def del_fx(self, X, y, w):
        """
        Returns the value of the derivative of the loss function with respect to w
        """
        return -1 / y.shape[0] * X.T @ (y - sigmoid(X @ w))

    def __init__(self, threshold = 0.5, max_iter = 10**3, step_size_multiplier = 1, intermediate_w = Fal
```

```
se):
        self.iter_ = max_iter
        self.ssm_ = step_size_multiplier
        self.intermediate_w_ = intermediate_w
        self.fit_ = False
        self.threshold = threshold

    def add_bias_(self):   ## adds bias column
        X_ = hstack([csr_matrix(np.ones((self.n, 1))), self.X])
        return X_

    def fit(self, X, y):
        self.n = X.shape[0]
        self.X = X
        self.X = self.add_bias_()
        self.y = y
        self.w = np.random.randn(self.X.shape[1], 1)
        self.grads = [np.linalg.norm(self.del_fx(self.X, y, self.w))]

        ## implementing gradient descent
        iters = 0
        while (iters < self.iter_) and self.grads[-1] > 10**-6:
            self.w = self.w - self.ssm_ * 1/(iters*(10**-3) + 1) * self.del_fx(self.X, y, self.w)
            iters += 1
            self.grads.append(np.linalg.norm(self.del_fx(self.X, y, self.w)))
        self.fit_ = True
        return self.grads

    def predict(self, X_test, weights = None):
        if self.fit_:
            X_test = hstack([csr_matrix(np.ones((X_test.shape[0], 1))), X_test])
            return np.where(sigmoid(X_test @ self.w) < self.threshold, 0, 1)
        else:
            print("Model not fitted yet")
```

▼ **ProcessingTrainingData**

```
################################ Importing dependencies ##########################
##

## The cwd path should be the path to the folder in which the .py file containing code is present
## All other supporting functions are also present in the same directory

import os
from Email2Text import email_to_text
import pandas as pd

#*********************************************************************************

base_path = os.getcwd()
dataset_path = os.path.join(base_path, "dataset")
```

```
ham_path = os.path.join(dataset_path, "ham")
spam_path = os.path.join(dataset_path, "spam")
ham_files = [os.path.join(ham_path,i) for i in os.listdir(ham_path)]
spam_files = [os.path.join(spam_path,i) for i in os.listdir(spam_path)]
df = pd.DataFrame(columns = ["email", "label"])

data = [] # Initialize an empty list to store data temporarily

# Process ham emails
for file in ham_files:
    with open(file, mode="rb") as f:
        content = f.read().decode("utf-8", errors="ignore")
        data.append({"email": content, "label": 0})

# Process spam emails
for file in spam_files:
    with open(file, mode="rb") as f:
        content = f.read().decode("utf-8", errors="ignore")
        data.append({"email": content, "label": 1})

# Create a DataFrame from the accumulated data
df = pd.DataFrame(data)

print(f"total number of emails loaded: {len(df)}")
print(f"spam emails loaded: {len(spam_files)}")
print(f"ham emails loaded: {len(ham_files)}")


df["content"] = df["email"].apply(lambda x: email_to_text(x)) ## applying all the pre-processing fu
nctions to process raw emails
df.to_csv(os.path.join(os.getcwd(),"processed_emails.csv"), columns = ["content", "label"])  ## ex
porting pre-processed emails to load later
```

▼ **Training**

```
################################ Importing dependencies ##########################
##

## The cwd path should be the path to the folder in which the .py file containing code is present
## All other supporting functions are also present in the same directory


from LogisticRegression import logistic_regression
from ClassificationMetrics import classification_metrics
from CustomCountVectorizer import CustomCountVectorizer
import os
import pickle
import numpy as np
import pandas as pd

from scipy.sparse import csr_matrix  ## for converting dense matrix to sparse matrix
```

```
#*****************************************************************************


############################## Defining helper functions ##########################
#####

## Function to split training and test datasets
def train_test_splitter(n, test_size = 0.3):
    """
    Function to randomly split dataset into train and test datasets. Return indices for splitting.
    n: no. of samples
    test_size : fraction of samples to be put in the test set
    """

    np.random.seed(23)  ## setting random seed for re-producability
    test_n = int(test_size * n)
    rand_ind = np.random.permutation(n)
    test_ind = rand_ind[:test_n]
    train_ind = rand_ind[test_n:]
    return train_ind, test_ind

## Creating function to convert y array to sparse
def y_to_csr(y):
    """
    Function to convert y array to csr format
    """
    row = []
    col = [0]*y.shape[0]
    data = []
    for ind, val in enumerate(y):
        row.append(ind)
        data.append(val)
    y_csr = csr_matrix((data, (row, col)), shape = (y.shape[0], 1))
    return y_csr

#*****************************************************************************

############################## Defining main training function ####################
#########

def training_logreg():
    """
    Main function for training logistic regression classifier. Returns fitted model and count-vectorize
r objects.
    Takes pre-processed emails from .csv files saved as "processed_emails.csv"
    """

    ## Deleting preivously created logreg model pickle file
    if os.path.exists(os.path.join(os.getcwd(), "logreg_c.pkl")):
        os.remove(os.path.join(os.getcwd(), "logreg_c.pkl"))
```

```
    ## Deleting preivously created model count vectorizer file
    if os.path.exists(os.path.join(os.getcwd(), "cv.pkl")):
        os.remove(os.path.join(os.getcwd(), "cv.pkl"))

    df = pd.read_csv(os.path.join(os.getcwd(), "processed_emails.csv"))  ## loading pre-processed
emails from csv file
    df = df.dropna()

    X = np.array(df["content"])   ## Feature matrix
    y = np.array(list(df["label"]))  ## labels

    ## checking if all the entries in the training set are of string type (and not nan)
    for i, text in enumerate(X):
        if not isinstance(text, str):
            print(f"The following index has non-string entry: {i}")

    ## splitting the dataset into train and test datasets randomly
    ## test dataset has 30% of the total emails
    train_ind, test_ind = train_test_splitter(len(df), 0.3)
    X_train, y_train = X[train_ind], y[train_ind]
    X_test, y_test = X[test_ind], y[test_ind]

    cv = CustomCountVectorizer(min_df = 2)  ## fitting custom count vectorizer on training data
    cv.fit(X_train)

    with open(os.path.join(os.getcwd(), "cv.pkl"), "wb") as file: ## saving the fitted cv for testing usi
ng pickle
        pickle.dump(cv, file = file)

    print(f"Words in vocabulary: {len(cv.vocabulary_)}")

    X_train_cv = cv.transform(X_train)  ## Fitting CountVectorizer on the training dataset
    X_test_cv = cv.transform(X_test)  ## Transforming test dataset

    y_train_csr = y_to_csr(y_train)   ## converting y_train to a sparse array

    ## Fitting  logistic regression
    logreg_c = logistic_regression(threshold=0.5, step_size_multiplier=80, max_iter=5*10**3)  ## fit
ting the logistic regression model
    grads = logreg_c.fit(X_train_cv, y_train_csr)

    with open(os.path.join(os.getcwd(), "logreg_c.pkl"), "wb") as file: ## saving the fitted cv for test
ing using pickle
        pickle.dump(logreg_c, file = file)


    ## calculating the metrics on the training dataset
    print("Logistic Regression performance on training dataset:")
    classification_metrics(y_train, logreg_c.predict(X_train_cv))
```

```
    ## calculating the metrics on the test dataset
    print("Logistic Regression performance on test dataset:")
    classification_metrics(y_test, logreg_c.predict(X_test_cv))

    return logreg_c, cv

print("Training.py file execution started")
training_logreg()
```

▼ **Testing**

```
############################## Importing dependencies ########################
##

## The cwd path should be the path to the folder in which the .py file containing code is present
## All other supporting functions are also present in the same directory

import os
from Email2Text import email_to_text
import pickle

import pandas as pd

#*****************************************************************************

def testing():

    ## loading the logreg model from saved pickle file
    if not os.path.exists(os.path.join(os.getcwd(), "logreg_c.pkl")):
        raise ValueError("Logistic regression pickle file not found")
    else:
        with open(os.path.join(os.getcwd(), "logreg_c.pkl"), "rb") as file:
            logreg_c = pickle.load(file=file)

    ## loading the fitted cv from saved pickle file
    if not os.path.exists(os.path.join(os.getcwd(), "cv.pkl")):
        raise ValueError("CV pickle file not found")
    else:
        with open(os.path.join(os.getcwd(), "cv.pkl"), "rb") as file:
            cv = pickle.load(file=file)

    ## Testing by loading data files in the test folder
    test_path = os.path.join(os.getcwd(), "test")
    test_files = [os.path.join(test_path, i) for i in os.listdir(test_path) if i.endswith(".txt")]

    ## Remove preds.csv file from created any previous test run
    try:
        os.remove(os.path.join(test_path, "preds.csv"))
    except Exception as e:
        pass
```

```
    data = []
    # Process the emails
    for file in test_files:
        with open(file, mode="rb") as f:
            content = f.read().decode("utf-8", errors="ignore")
            data.append({"email": content})

    df_test = pd.DataFrame(data)
    print(f"No. of emails in test folder: {len(df_test)}")

    ## Testing Transformation steps
    ## extracting info from the emails
    df_test["content"] = df_test["email"].apply(lambda x: email_to_text(x))
    X_test = df_test["content"]
    X_test = cv.transform(X_test)

    preds = logreg_c.predict(X_test)  # making predictions

    # saving predictions in preds.csv in the test folder
    pred_list = [{"filename": tf.split(os.sep)[-1], "pred": i} for i,tf in zip(preds.ravel(), test_files)]
    df_preds = pd.DataFrame(pred_list)
    df_preds.to_csv(os.path.join(test_path, "preds.csv"), index = False)

print("Testing.py file execution started")
testing()
```