Assignment-2

1. Files and It's Contains

1. `main.py` : The models are trained on preprocessed text data, evaluated on a test dataset, and also used to make predictions on additional unseen data(Data from `test` folder).
2. `data_preprocessing.py` : Performs text preprocessing and preparation of a dataset for machine learning. It includes several steps: loading the dataset, cleaning and preprocessing the text data, splitting the data into training and testing sets, and vectorizing the text features
3. `test_data.py` : This file load, preprocess, and vectorize email data from `test` folder, preparing it for input into machine learning models. It relies on functions from a `data_preprocessing` function from `data_preprocessing.py` to handle text cleaning and vectorization.
4. `naive_bayes.py` : In this file I have implemented code of Naive-Bayes from scratch.
5. `logistic_regression.py` : In this file I have implemented code of Logistic Regression from scratch.

2. Loading and Preprocessing Data

2.1 Library Imports

For preprocessing data I have imports several libraries:

```
import pandas as pd
from nltk.stem.porter import PorterStemmer
import re
import os
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
```

- `pandas` : For loading and handling tabular data.
- `PorterStemmer` from `nltk` : For stemming words to their root form.
- `re` : For regular expression operations, used in text cleaning.
- `os` : For file path management.
- `train_test_split` and `CountVectorizer` from `sklearn` : For splitting the data and converting text into numeric format, respectively.

2.2 Dataset Loading

Data to train the models where taken from https://www.kaggle.com/datasets/ozlerhakan/spam-or-not-spam-dataset which originates from https://spamassassin.apache.org/old/publiccorpus/ i.e. Apache SpamAssassin's public datasets which is a **Public Dataset.**

1. In data_processing.py file I loaded the downloaded data from internet.

   ```
   directory_name = os.path.dirname(__file__)
   data_file_path = os.path.join(directory_name, 'spam_or_not_spam.csv')
   data = pd.read_csv(data_file_path)
   ```

   By getting directory path using `directory_name = os.path.dirname(__file__)` then getting path for data file using `os.path.join` then code loads a CSV file named `spam_or_not_spam.csv` containing the dataset. The dataset contains columns `email` (containing emails) and `label` (containing the 1(spam)/0(non-spam)). This data is contained by variable name `data`.

2. In test_data.py I have taken data from `'test'` folder using code:

```
def load_emails(folder_path):
    email_data = []
    # Check if the folder exists
    if not os.path.exists(folder_path):
        print(f"The folder '{folder_path}' does not exist.")
        return

    # Read each .txt file in the folder
    for filename in os.listdir(folder_path):
        if filename.endswith(".txt"):
            file_path = os.path.join(folder_path, filename)
            with open(file_path, 'r', encoding='utf-8') as file:
                email_content = file.read()
                email_content = email_content.split('\n')
                email_content = ' '.join(email_content)
                email_data.append({"email_content": email_content})
    return email_data


directory_name = os.path.dirname(__file__)
folder_path = os.path.join(directory_name, 'test')

# Create DataFrame
emails = load_emails(folder_path)
email_df = pd.DataFrame(emails, columns=["email_content"])
```

By calling function `load_emails` which takes `folder path` as input. If folder doesn't exists it prints `The folder folder_path does not exist`. If folder exists it will read files one by one and add there contents to list `email_data` which will be converted into `DataFrame` using Pandas this is carried by variable by name `email_df`.

2.3 Text Preprocessing Function

1. In data_processing.py file I wrote a function to preprocess data which is also used every where.

```
def preprocess_text(text):
    stop_words = {...}
    stemmer = PorterStemmer()
    text = text.lower()
    text = re.sub(r'[^a-zA-Z\s]', '', text)
    words = [stemmer.stem(word) for word in text.split() if word not in stop_words]
    return ' '.join(words)
```

The `preprocess_text` function performs several preprocessing steps:

- **Stop Words Removal**: A set of commonly used English stop words is defined and removed from the text.
- **Lowercasing**: Converts the text to lowercase to ensure consistency. Using `text = text.lower()`.
- **Special Character Removal**: Uses a regular expression to remove any non-alphabetical characters. Using `text = re.sub(r'[^a-zA-Z\s]', '', text)`.

- **Stemming**: The `PorterStemmer` reduces each word to its root form, helping to normalize different forms to the same word. Example: words like writing, writes will be converted to write.

2.4 Applying Preprocessing to Dataset

1. In data_processing.py file the text preprocessing function is applied using following code.

```
data['clean_text'] = data['email'].apply(lambda x: preprocess_text(x)
                                     if isinstance(x, str) else "")
data = data[data['clean_text'] != ""]
X_train, X_test, y_train, y_test = train_test_split(data['clean_text'],
                            data['label'], test_size=0.2, random_state=42)
```

This code applies the `preprocess_text` function to each entry in the `email` column, creating a new column, `clean_text`, which contains the preprocessed text. Rows with empty `clean_text` are removed from the dataset. Then data was split into train test using function `sklearn.model_selection.train_test_split`. The dataset is split into training and testing sets, with 80% of the data used for training and 20% for testing. The `random_state` parameter is set for reproducibility.

2. In test_data.py the same was done by following code:

```
email_df = pd.DataFrame(emails, columns=["email_content"])
# Preprocessing Data
email_df['clean_data'] = email_df['email_content'].apply(lambda x: preprocess_text(
                                        if isinstance(x, str) else "")
```

By creating new dataset using emails from `test` folder. Then applied preprocessing function on that data.

2.5 Vectorization

As models like **Logistic Regression and SVM** doesn't take words as inputs the text data was converted into into a matrix of token counts. This was achieved by:

```
vectorizer = CountVectorizer()
X_train_vect = vectorizer.fit_transform(X_train).toarray()
X_test_vect = vectorizer.transform(X_test).toarray()
```

This vectorizer was initiated in data_processing.py file and same was used in test_data.py file and was assigned to `final_input_test_vect` variable.

```
final_input_test_vect = vectorizer.transform(final_input_test).toarray()
```

3. Naive Bayes Classifier

Here's a detailed breakdown of the code, including the mathematical concepts involved which were used for building Naive-Bayes model:

3.1 Class Initialization: `__init__`

```
class NaiveBayes:
    def __init__(self):
        self.spam_words = defaultdict(int)
```

```
        self.not_spam_word = defaultdict(int)
        self.spam_count = 0
        self.not_spam_count = 0
```

The classifier begins by initializing dictionaries and counts for words in spam and not spam:

- `spam_words`: A dictionary to store word frequencies in spam emails.
- `not_spam_word`: A dictionary to store word frequencies in non-spam emails.
- `spam_count` and `not_spam_count`: Counts the total number of words in spam and non-spam emails.

These structures help store frequency information, which the classifier uses to calculate probabilities.

3.2 Training Method: `fit`

```
def fit(self, X, y):
    for text, label in zip(X, y):
        for word in text.split():
            if label == 1:
                self.spam_words[word] += 1
                self.spam_count += 1
            else:
                self.not_spam_word[word] += 1
                self.not_spam_count += 1
```

The `fit` method trains the classifier by computing word frequencies in each category based on a labeled dataset, where:

- `X`: The list of email texts.
- `y`: The list of labels (1 for spam and 0 for not spam).

Each word in a given email is split and counted based on its label. If the email is spam (`label == 1`), the word count is added to `spam_words`, and `spam_count` is incremented. If it's not spam (`label == 0`), the word count goes to `not_spam_word`, and `not_spam_count` is incremented.

1. **Priors Calculation**

   ```
   self.p_spam = y.sum() / y.shape[0]
   self.p_not_spam = 1 - self.p_spam
   ```

   The code calculates the prior probabilities, P(spam)P(\text{spam})P(spam) and P(not spam)P(\text{not spam})P(not spam), as follows:

   - $P(\text{spam}) = \frac{\text{number of spam emails}}{\text{total emails}}$
   - $P(\text{not spam}) = 1 - P(\text{spam})$

   The prior probabilities represent the likelihood that an email is spam or not before observing any specific word.

2. **vocabulary size**

   ```
   vocab = set(self.spam_words.keys()).union(set(self.not_spam_word.keys()))
   self.vocab_size = len(vocab)
   ```

The vocabulary size (`vocab_size`) is the total number of unique words in both spam and not spam emails. This is used later in Laplace smoothing.

3.3 Word Probability Calculation: `word_prob`

```
def word_prob(self, word, label):
    if label == 1:
        return (self.spam_words[word] + 1) / (self.spam_count + self.vocab_size)
    else:
        return (self.not_spam_word[word] + 1) / (self.not_spam_count + self.vocab_size
```

The `word_prob` method computes the probability of a word given a class (spam or not spam) using **Laplace smoothing** to handle cases where a word may not appear in one class:

- If `label == 1` (spam), the probability is:

$$P(\text{word} \mid \text{spam}) = \frac{\text{spam\_words[word]} + 1}{\text{spam\_count} + \text{vocab\_size}}$$

- If `label == 0` (not spam), the probability is:

$$P(\text{word} \mid \text{not spam}) = \frac{\text{not\_spam\_word[word]} + 1}{\text{not\_spam\_count} + \text{vocab\_size}}$$

3.4 Prediction Method: `predict`

```
def predict(self, emails):
    predictions = []
    for email in emails:
        emial_words = email.split()
        result = math.log(self.p_spam/self.p_not_spam)
        for word in emial_words:
            p1 = self.word_prob(word, 1)
            p0 = self.word_prob(word, 0)
            result += (math.log((p1*(1 - p0))/((1 - p1)*p0))) + math.log((1-p1)/(1-p0)
        if result >= 0:
            predictions.append(1)
        else:
            predictions.append(0)
    return predictions
```

The `predict` method classifies each email as spam or not spam. Here's a breakdown of its key steps:

Log-Likelihood Calculation

1. **Initialize the Log-Odds with Priors**:
   - The `result` variable starts with the log of the ratio of priors:

$$\log\left(\frac{P(\text{spam})}{P(\text{not spam})}\right)$$

2. **Word Contribution**:
   - For each word in the email, the code calculates the probability of the word given the spam and not spam classes.
   - Using log-probabilities helps handle the product of small numbers (to prevent underflow) and translates the conditional probabilities into an additive form:

$$\text{result}+ = \log\left(\frac{P(\text{word} \mid \text{spam}) \cdot (1 - P(\text{word} \mid \text{not spam}))}{(1 - P(\text{word} \mid \text{spam})) \cdot P(\text{word} \mid \text{not spam})}\right) + \log\left(\frac{1 - P(\text{word} \mid \text{spam})}{1 - P(\text{word} \mid \text{not spam})}\right)$$

3. **Final Prediction**:
   - After processing all words, the classifier determines the label based on `result`:
     - If `result >= 0`, the email is classified as spam (`1`).
     - Otherwise, it's classified as not spam (`0`).

4. Logistic Regression

Logistic regression estimates the probability of a binary outcome (0 or 1) by applying a sigmoid function to a linear combination of input features. This implementation trains the model using **gradient descent**, for optimizing the weights to minimize prediction error. Below, I provided a detailed breakdown of each section of the code.

4.1 Class Initialization: `__init__`

```
class LogisticRegression:
    def __init__(self, learning_rate=0.01, num_iterations=1000):
        self.learning_rate = learning_rate
        self.num_iterations = num_iterations
        self.weights = None
```

The `LogisticRegression` class is initialized with:

- `learning_rate`: Controls the step size in gradient descent updates. A small learning rate leads to smaller updates, slowing convergence but potentially achieving higher accuracy.
- `num_iterations`: Sets the number of iterations in the gradient descent loop.
- `weights`: Placeholder for the model's weight vector, initialized in the `fit` method.

4.2 Adding a Bias Term: `_add_bias`

```
def _add_bias(self, X):
    # Add bias column of ones to given data
    bias = np.ones((X.shape[0], 1))
  return np.hstack([bias, X])
```

Logistic regression requires a **bias term** (intercept) to account for a constant offset in predictions. This method adds a column of ones to the input feature matrix `X`, effectively increasing its dimensionality by 1.

Let X be the input matrix of shape `(n,m)` where n is the number of samples and m is the number of features. After adding the bias term, the resulting shape of `X` becomes `(n,m+1)`.

4.3 Sigmoid Function: `_sigmoid`

```
def _sigmoid(self, z):
    return 1 / (1 + np.exp(-z))
```

The **sigmoid function** transforms a real-valued input zzz into a value between 0 and 1. This is useful in logistic regression for converting the linear output of the model to a probability.

Mathematically, the sigmoid function is given by:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

where z is the linear combination of input features and weights.

4.4 Model Training: `fit`

```
def fit(self, X, y):
    X = self._add_bias(X)
    n_samples, n_features = X.shape
    self.weights = np.zeros(n_features)

    for i in range(self.num_iterations):
        linear_model = np.dot(X, self.weights)
        y_pred = self._sigmoid(linear_model)

        gradient = np.dot(X.T, (y_pred - y)) / n_samples
        self.weights -= self.learning_rate * gradient
```

The `fit` method trains the logistic regression model by using **gradient descent**. Here's a breakdown of each step:

Step-by-Step Explanation

1. **Adding Bias**: Calls `_add_bias` to add a bias term to X.
2. **Weight Initialization**: Initializes the weights vector to zeros. The length of this vector matches the number of features (including bias).
3. **Gradient Descent Loop**:
    - **Compute Linear Model**: Calculates the dot product of X and `self.weights`, which represents the linear model:

      $$z = X \cdot w$$

    - **Apply Sigmoid Function**: Transforms the linear output z to predicted probabilities $\hat{y}$:

      $$\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}}$$

    - **Compute Gradient**: Calculates the gradient of the logistic loss function with respect to the weights:

      $$\nabla L = \frac{1}{n}\sum_{i=1}^{n} X_i^T(\hat{y}_i - y_i)$$

        - X^T is the transpose of the input matrix X,
        - $\hat{y} - y$ is the difference between the predicted probabilities and the true labels,
        - n is the number of samples.
    - **Update Weights**: Updates the weights by subtracting the scaled gradient from the current weights. This is the gradient descent step:

$$w = w - \alpha \cdot \nabla L$$

where α is the learning rate. This iterative update helps minimize the prediction error by moving the weights in the direction of the negative gradient.

4.5 Predicting Probabilities: `predict_proba`

```
def predict_proba(self, X):
    X = self._add_bias(X)
    linear_model = np.dot(X, self.weights)
    return self._sigmoid(linear_model)
```

The `predict_proba` function calculates the probability of each input sample belonging to the positive class (1). It works by:

1. Adding the bias term to X.
2. Calculating the linear model, $z = X \cdot w$
3. Applying the sigmoid function to convert z into a probability:

$$P(y = 1 \mid X) = \sigma(X \cdot w) = \frac{1}{1 + e^{-X \cdot w}}$$

This results in a probability value between 0 and 1 for each input sample.

4.6 Making Predictions: `predict`

```
def predict(self, X):
    y_pred_proba = self.predict_proba(X)
    return [1 if i > 0.5 else 0 for i in y_pred_proba]
```

The `predict` function makes binary predictions based on a **0.5 threshold**:

- If the predicted probability is greater than 0.5, it classifies the sample as 1 (positive class).
- If the probability is less than or equal to 0.5, it classifies the sample as 0 (negative class).

This threshold-based prediction is represented as:

$$\hat{y} = \begin{cases} 1 & \text{if } P(y = 1|X) > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

5. Support Vector Machine (SVM)

Support vector machine(SVM) was used from `sklearn` module in python. It was implemented in `main.py` file. Code for the same will be:

```
from sklearn.svm import SVC
svm_model = SVC()
svm_model.fit(x_train_vect2, y_train2)
```

Here `SVC` is support vector classifier which is used for classifying data.

6. Predictions and Model Performances

This is done in `main.py` file where all the models are called from there respective files and training data is fitted in them. All the models predictions for unseen data(data from `test` folder) will be printed when `main.py` file is run.

6.1 Data Preparation

```
x_train2, x_test2, y_train2, y_test2 = data_pre.X_train, data_pre.X_test, data_pre.y_t
x_train_vect2, x_test_vect2 = data_pre.X_train_vect, data_pre.X_test_vect

final_input_test2 = test_data.final_input_test
final_input_test_vect2 = test_data.final_input_test_vect
```

The data for training and testing is imported from two modules:

- `data_preprocessing.py`: Contains preprocessed training and testing data split into `X_train`, `X_test`, `y_train`, `y_test` for feature matrices and labels.
- `test_data.py`: Provides an additional dataset, `final_input_test`, used for making predictions on completely unseen data.

The code loads:

- **Vectorized data** (`x_train_vect2` and `x_test_vect2`): Preprocessed and vectorized text data suitable for models like logistic regression and SVM.
- **Labels** (`y_train2`, `y_test2`): The true classifications for each training and test sample.

6.2 Naive Bayes Model

```
naive_model = NaiveBayes()
naive_model.fit(x_train2, y_train2)

naive_test_predict = naive_model.predict(x_test2)
naive_train_predict = naive_model.predict(x_train2)

final_naive_predict = naive_model.predict(final_input_test2)

naive_test_accuracy = sum(naive_test_predict == y_test2) / len(naive_test_predict)
naive_train_accuracy = sum(naive_train_predict == y_train2) / len(naive_train_predict)
```

After training, the model is evaluated on the test data, and predictions are made for the unseen `final_input_test2` dataset.

Accuracy of Naive Bayes on Trained data is: 99.67%
Accuracy of Naive Bayes on Test data is: 99.17%

6.3 Logistic Regression Model

```
logReg_model = LogisticRegression()
logReg_model.fit(x_train_vect2, y_train2)

logReg_test_predict = logReg_model.predict(x_test_vect2)
```

```
logReg_train_predict = logReg_model.predict(x_train_vect2)

final_logReg_predict = logReg_model.predict(final_input_test_vect2)

logReg_test_accuracy = sum(logReg_test_predict == y_test2) / len(logReg_test_predict)
logReg_train_accuracy = sum(logReg_train_predict == y_train2) / len(logReg_train_predi
```

Accuracy of Logistic Regression on Trained data is: 97.71%
Accuracy of Logistic Regression on Test data is: 98.00%

6.4 Support Vector Machine (SVM)

```
svm_model = SVC()
svm_model.fit(x_train_vect2, y_train2)

svm_test_predict = svm_model.predict(x_test_vect2)
svm_train_predict = svm_model.predict(x_train_vect2)

final_svm_predict = svm_model.predict(final_input_test_vect2)

svm_train_accuracy = sum(svm_train_predict == y_train2) / len(svm_train_predict)
svm_test_accuracy = sum(svm_test_predict == y_test2) / len(svm_test_predict)
```

Accuracy of SVM on Trained data is: 96.16%
Accuracy of SVM on Test data is: 94.17%