Spam/Ham Classification Report

Sakthe Balan G K

November 3, 2024

1 Running the code

Firstly, make sure the necessary libraries are available

1. numpy
2. pandas
3. re (for preprocessing the features)
4. sklearn (for SVC)

The submission consists of a .py file (source code), a .txt file (Details), a .csv file named (**train dataset.csv**) which contains the training data and this report pdf. When the code is run directly,

1. It automatically generates a test folder and generates a number of emails in that folder (in the form of email1.txt, email2.txt ...). Next, it takes these mails in the 'test' folder and predicts corresponding labels for them.
2. It also creates a .csv file (**y_true.csv**) which contains the actual labels for the test mails (This is required to calculate the accuracy score).
3. Also, four seperate models are used for predictions. The predictions of all these four models are automatically stored in a .csv file named **results.csv** in the necessary format (spam/ham)

If you don't want to automatically generate these test mails (or) if you have a seperate test folder to test the code, you can simply comment the part of the code which generates the test mails and run it. The part to be commented is given below. While commenting, please make sure that a test folder is available with emails named in same format and actual labels as a .csv file under the same name in the directory for the code to run smoothly.

```
### This code is used to create the test dataset in the form of text files ###

master_df_test = pd.DataFrame(master_df).iloc[5000:].reset_index(drop = True, inplace =
    False).rename(columns={"Category":"label", "Message":"message"})

if not os.path.exists("test"):
    os.makedirs("test")

for index, row in master_df_test.iterrows():

    row_data = str(row[1])
    filename = os.path.join("test", f"email{index+1}.txt")
    with open(filename, "w", encoding = "ISO-8859-1") as f:
        f.write(row_data)

y_true = master_df_test["label"]
y_true.to_csv("y_true.csv", index=False)
```

Listing 1: Test Data Generation Code

2 Dataset Selection and Preparation

The dataset, named **train dataset.csv**, was initially read and split into training and testing portions. The first 5,000 entries were used for training, and the rest were allocated to the test set. Each email in the test set was saved as an individual text file in a directory named `test`, following the assignment's requirements.

```
# Read the CSV dataset and drop unwanted columns from the file
master_df = pd.read_csv("train dataset.csv", encoding="ISO-8859-1")

dataset = pd.DataFrame(master_df).iloc[0:5000].reset_index(drop = True, inplace = False).
    rename(columns={"Category":"label", "Message":"message"})
```

Listing 2: Data Preparation Code

3 Algorithms Implemented

The code includes implementations of multiple machine learning algorithms from scratch. Here's a detailed explanation of each:

3.1 Naive Bayes Classifier

The Naive Bayes classifier is based on Bayes' theorem and assumes conditional independence between features. This classifier calculates the probability of a message being spam or ham by using the frequencies of words in each class (spam or ham). The implementation follows these steps:

- **Prior Probabilities**: The probabilities of each class (spam and ham) are computed as the proportion of spam and ham messages in the training data.
- **Word Probabilities**: For each word, the conditional probability of it appearing in spam and ham messages is calculated. Laplace smoothing is applied to avoid zero probabilities.
- **Prediction**: For a given message, the log-probabilities of it being spam or ham are calculated by summing the log-probabilities of each word in the message. The class with the higher log-probability is chosen as the prediction.

The Naive Bayes classifier is effective for text classification because it handles high-dimensional data well and is computationally efficient. The corresponding code for the algorithm is given below :

```
class NaiveBayesClassifier:
    def __init__(self):
        self.word_probs = {}
        self.class_probs = {}

    def fit(self, X, y):
        # Calculate prior probabilities
        self.class_probs[1] = np.mean(y == 1)  # Probability of spam
        self.class_probs[0] = np.mean(y == 0)  # Probability of ham

        # Separate spam and ham messages
        spam_messages = X[y == 1]
        ham_messages = X[y == 0]

        # Calculate total word counts for each class
        total_spam_words = np.sum(spam_messages)
        total_ham_words = np.sum(ham_messages)

        # Calculate word probabilities given each class
        self.word_probs[1] = (np.sum(spam_messages, axis=0) + 1) / (total_spam_words + X.
            shape[1])
        self.word_probs[0] = (np.sum(ham_messages, axis=0) + 1) / (total_ham_words + X.shape
            [1])
```

```
    def predict(self, X):
        predictions = []
        for message in X:
            # Calculate log-probabilities for spam and ham
            log_spam_prob = np.log(self.class_probs[1])
            log_ham_prob = np.log(self.class_probs[0])
            for i, word_count in enumerate(message):
                if word_count > 0:  # Only consider words that appear in the message
                    log_spam_prob += word_count * np.log(self.word_probs[1][i])
                    log_ham_prob += word_count * np.log(self.word_probs[0][i])
            # Choose the class with higher log-probability
            predictions.append(1 if log_spam_prob > log_ham_prob else 0)
        return np.array(predictions)
```

Listing 3: Naive Bayes Algorithm from scratch

3.2 K-Nearest Neighbors (KNN) Classifier

The KNN classifier assigns a label based on the labels of the k nearest neighbors of a given data point in the feature space. The Euclidean distance is used to find the nearest neighbors. Here's how the KNN classifier works in this implementation:

- **Distance Calculation**: For each test point, the Euclidean distance to all training points is computed.
- **Neighbor Selection**: The k closest neighbors are selected.
- **Majority Voting**: The class labels of the k neighbors are examined, and the most common label is chosen as the prediction.

KNN is simple and effective for smaller datasets. For my case, I have already tuned the hyper parameter k (I took some values for k and run the code) and found that k=3 provides the best performance. So, k=3 is taken. The code for the algorithm is given below :

```
class KNearestNeighborsClassifier:
    def __init__(self, k=3):
        self.k = k

    def fit(self, X, y):
        self.X_train = X
        self.y_train = y

    def predict(self, X):
        predictions = []
        for x in X:
            distances = np.sqrt(np.sum((self.X_train - x) ** 2, axis=1))  # Calculate
                Euclidean distance
            k_nearest_indices = distances.argsort()[:self.k]  # Get indices of k nearest
                neighbors
            k_nearest_labels = self.y_train[k_nearest_indices]  # Get the labels of those
                neighbors
            majority_vote = Counter(k_nearest_labels).most_common(1)[0][0]  # Majority vote
            predictions.append(majority_vote)
        return np.array(predictions)
```

Listing 4: KNN algorithm from scratch

3.3 Logistic Regression Classifier

Logistic Regression is a linear classifier that models the probability that a given input belongs to a particular class using a sigmoid function. The implementation includes:

- **Sigmoid Function**: The sigmoid function is used to convert the linear combination of features into a probability.

- **Gradient Descent Optimization**: The weights and bias are updated using gradient descent, minimizing the error between the predicted and actual labels over several epochs.
- **Prediction**: For each input, the sigmoid function's output is thresholded at 0.5 to assign a class label (spam or ham).

Logistic Regression is suitable for binary classification and provides interpretable results as the weights indicate feature importance. Max_iterations is set to 1000 and learning rate is set to 0.01 (found optimal by trail and error.)The code for the algorithm is given below :

```
class LogisticRegressionClassifier:
    def __init__(self, lr=0.01, epochs=1000):
        self.lr = lr
        self.epochs = epochs

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

    def fit(self, X, y):
        self.weights = np.zeros(X.shape[1])
        self.bias = 0
        for _ in range(self.epochs):
            linear_model = np.dot(X, self.weights) + self.bias
            y_pred = self.sigmoid(linear_model)
            # Gradient descent updates
            dw = (1 / len(y)) * np.dot(X.T, (y_pred - y))
            db = (1 / len(y)) * np.sum(y_pred - y)
            self.weights -= self.lr * dw
            self.bias -= self.lr * db

    def predict(self, X):
        linear_model = np.dot(X, self.weights) + self.bias
        y_pred = self.sigmoid(linear_model)
        return [1 if i > 0.5 else 0 for i in y_pred]
```

Listing 5: Logistic regression algorithm from scratch

3.4 Support Vector Machine (SVM) Classifier

An SVM classifier is implemented using Scikit-Learn's built-in `SVC` function. SVMs attempt to find the hyperplane that maximally separates the two classes in the feature space.

A basic SVM model is used with default parameters as this itself produces high accuracy of 0.99.

4 Feature preproccesing and extraction

4.1 Text Preprocessing done

In this code, text preprocessing is done using the `re` library, which provides functions for handling regular expressions. This preprocessing is important for converting raw email text into a consistent and cleaner format before feeding it into the classifier. Here's a breakdown of each preprocessing step:

```
def preprocess_text(text):
    text = text.lower()  # Convert to lowercase
    text = re.sub(r'\W', ' ', text)  # Remove special characters and punctuation
    text = re.sub(r'\s+', ' ', text).strip()  # Remove extra spaces
    return text
```

Listing 6: Text Preprocessing Code

The preprocessing function performs the following operations:

- **Convert to Lowercase**: The text is converted to lowercase using `text.lower()`. This step ensures that words like "Free" and "free" are treated the same by the classifier, which is especially important in case-sensitive languages.
- **Remove Special Characters and Punctuation**: The code `re.sub(r'\W', ' ', text)` removes any character that is not a letter, number, or underscore. This includes punctuation marks and special characters. By replacing these characters with spaces, the function eliminates noise in the data, which can improve the model's focus on meaningful words.
- **Remove Extra Spaces**: The code `re.sub(r'\s+', ' ', text).strip()` replaces multiple spaces with a single space and removes leading or trailing spaces. This step is useful for ensuring that words are properly separated and no unnecessary spaces are present, leading to cleaner and more consistent input data.

4.2 Purpose of Preprocessing Steps

These preprocessing steps are essential for text-based machine learning tasks because they:

- **Improve Consistency**: Converting text to lowercase and removing extra spaces ensures that the same word appears consistently across all samples.
- **Reduce Noise**: Removing special characters and punctuation helps the model focus on actual words that contribute to the meaning of the text, reducing irrelevant noise.
- **Enhance Model Performance**: By cleaning the data, these preprocessing steps improve the quality of the input features, which can lead to better model accuracy and generalization.

4.3 Addressing Class Imbalance through Oversampling

In this dataset, there is an imbalance between the number of spam and ham emails, with one class (spam class) having significantly fewer samples than the other (ham class). This imbalance can lead to poor model performance, particularly for the minority class, as the model might become biased towards the majority class. To address this issue, **oversampling** is used.

4.3.1 Why Address Class Imbalance?

Class imbalance poses several challenges in classification:

- **Bias Toward Majority Class**: When the dataset is imbalanced, the model tends to predict the majority class more often, as this minimizes the overall error. This bias leads to low recall for the minority class, which, in this case, could mean failing to detect spam emails.
- **Reduced Sensitivity to Minority Class**: Misclassifying a spam email as ham could result in unwanted or harmful messages reaching users. A balanced dataset allows the model to learn features from both classes adequately, improving its sensitivity to spam.
- **Fairer Evaluation Metrics**: When the classes are balanced, evaluation metrics such as accuracy, precision, and recall provide a fairer measure of model performance, as they reflect the model's ability to detect both classes.

4.3.2 Oversampling Implementation

To address the imbalance, the minority class (spam emails) is **oversampled** by randomly duplicating samples until the number of spam samples matches the number of ham samples. This results in a balanced dataset that improves the model's ability to recognize patterns in both classes.

The code snippet for oversampling is as follows:

```
# Separate the dataset into spam and ham
spam_data = dataset[dataset['label'] == 1]
ham_data = dataset[dataset['label'] == 0]

# Oversample spam data to match the size of ham data
oversampled_spam_data = spam_data.sample(len(ham_data), replace=True)
balanced_data = pd.concat([ham_data, oversampled_spam_data])
```

Listing 7: Oversampling implementation

This technique effectively creates a balanced dataset by increasing the representation of the minority class, allowing the model to learn features from both classes equally. With this approach, the model can achieve better accuracy for both spam and ham, improving its overall performance.

4.4 Why Use a Vectorizer Instead of Manual Feature Extraction?

Using a vectorizer, such as `TfidfVectorizer`, is advantageous over manual feature extraction for several reasons:

- **Efficiency**: A vectorizer quickly transforms text data into numerical features, making the process more efficient than manually counting word frequencies.
- **Consistency**: It maintains a standardized vocabulary and ensures consistent feature representation across all samples.
- **Enhanced Performance**: Techniques like TF-IDF can weigh important words more heavily, helping to improve the model's ability to differentiate between classes.

Manual feature extraction would require more time and may result in inconsistencies, especially for large datasets. A vectorizer provides a robust, optimized approach, essential for text-based classification tasks.

4.5 Feature extraction done in the code

In the code, tfidf vectorizer was used to vectorize the messages. I have taken `max_features` = 1000 and `stop_words` = 'english' as parameters (Found optimal).

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(max_features=1000, stop_words='english')
X_train = vectorizer.fit_transform(X_train).toarray()
X_test = vectorizer.transform(X_test).toarray()
```

Listing 8: Vectorizing the messages

A simple mapping has been done in the label to convert spam, ham to binary output (0 for ham and 1 for spam). Code below :

```
dataset['label'] = dataset['label'].map({'spam': 1, 'ham': 0})
```

Listing 9: Convertting label to binary format

5 Taking the test data as required

The following lines of code automatically checks the presence of test folder (which supposedly should contain the test mails in the form email1.txt , email2.txt ...)in the same directory. If present, the code reads each email in the folder as input in order, and stores into a dataframe (This is the `X_test`) Now, the test data is available which is then preprocessed and vectorized for prediction. Code snippet is given below for the following :

```
folder_path = "test"
data = []

if not os.path.exists("test"):
    print("test folder does not exist")

if not os.path.exists(folder_path):
    print(f"{folder_path} folder does not exist")
else:
    # Sort filenames numerically by extracting the numeric part
    filenames = sorted(os.listdir(folder_path), key=lambda x: int(x.split('.')[0].replace('
        email', '')))

    # Loop through each sorted file
    for filename in filenames:
        if filename.endswith(".txt"):  # Check for .txt files
            file_path = os.path.join(folder_path, filename)  # Full path to the file

            # Read the file content
            with open(file_path, "r", encoding="ISO-8859-1") as file:
                content = file.read().strip()  # Read and strip whitespace/newlines

            # Append the content to the list
            data.append({"message": content})
```

Listing 10: Code snippet to take the test data

6 Results and Evaluation

The result is the classification of each email in the test folder into spam or ham. After models are trained on the train data, These are fitten on the test data and the predicitons are made. As there are 4 models implemented, there are 4 outputs for a test mail, each one is a classification (spam/ham) on a particular model. These outputs are automatically stores in a .csv file named results.csv.

6.1 Accuracy score

A simple function was created which manually calculates the accuracy score (Ratio of total correct predictions to Total number of predictions). Code snippet for that is given below:

```
def accuracy_score(y_true, y_pred):
    pred = np.sum(y_true == y_pred)
    acc = pred / len(y_true)
    return acc
```

Listing 11: Accuracy score funtion

6.2 Observations

The classifier was evaluated on the test set of emails. Each email was labeled as +1 (spam) or 0 (ham). The accuracy achieved on this test set was

Classifier	Accuracy Score
Naive Bayes	0.98
SVM	0.99
K-Nearest Neighbors (k=3)	0.97
Logistic Regression	0.98

Table 1: Performance Comparison of Classifiers

This shows that all the models performs well for the test dataset