PRML Assignment – 3 Report
Custom Spam Classifier

ED20B058 Settur Anantha Sai Jithamanyu

Dataset:

In the custom spam classifier that has been built from scratch, we used the spam assassin data that was taken from Kaggle. This is a cleaned dataset from the original one available on SpamAssassin. The dataset contains thousands of emails with metadata such as the sender's email address, subject line, and email content. Each email is assigned a label indicating whether it is spam or non-spam. This labeling is typically done by human experts who manually review each email and classify it accordingly. It comprises almost 6000 emails with a spam-to-ham ratio of 31%. We will take the dataset available on Kaggle and perform a preprocessing of the dataset and make a new CSV file of it which is what we will be using as the train data.

Preprocessing:

The preprocessing that we do on the data is rather simple. We read the body of the email and removed all characters from it which were not alphanumeric using regex. Then we perform text normalization using lemmatization. The most popular text normalization techniques used are lemmatization and stemming. In the notebook, we have mentioned a note on why we chose to use lemmatization over stemming. Lemmatization considers the context of the word rather than simply looking at the word, as is the case in stemming. This context-based text normalization is what makes lemmatization better. This information about which is better was taken from this source.

Feature extraction:

I utilized the feature extraction module of the sklearn library to extract features from my text data. Specifically, I employed the CountVectorizer and TfidfVectorizer techniques to transform my text data into a matrix of word counts and term frequency-inverse document frequency values, respectively. We will see which feature extraction technique works better for the data and use that for feature extraction. The working of both these methods has been briefly described in the notebook. From a search on the internet, I could see that TfidfVectorizer would perform better, but we can use it with Naive Bayes Classifier as it's not binary data, but we can still use Gaussian Discriminant Analysis. The better performance of TfidfVectorizer is attributed to its ability to serve more information about the word in terms of semantic meaning compared to CountVectorizer. There were also results on the internet that suggested that TfidfVectorizer would not always perform better. This is something that we actually observe in our case also. We will realize that binary encoding works better than TfidfVectorizer. One of the hyperparameters associated with feature extraction is the total number of words to be considered controlled by the hyperparameters max_features. Although I haven't fully examined what the ideal value of max_features would be. But it makes sense to say that more features would make it easier to learn the underlying structure, altho not true in most cases. So as to find the best hyperparameter for max_feature and what technique to use for extracting the features, we will do some hyperparameter tuning. Before that, we will discuss the various models we have tried for training and find the best-suited model based on validation accuracy.

Models used for training:
For training, I have tried using Naive Bayes, Gaussian Naive Bayes, Logistic Regression, K-nearest neighbors, and SVM(not so thoroughly examined). After training these models for multiple feature extraction methods(those discussed above), and saw that Logistic Regression performs the best. The code for all the algorithms was coded from scratch and is given in the notebook.

Logistic regression;
Since explaining all the models will take a lot of time, I won't be doing that I will just explain the working and code of Logistic regression. The code defines a Python class named "LogisticRegression" that uses the logistic regression algorithm to perform binary classification on a given dataset. The class takes in two arguments, input data X and output labels Y, and optional hyperparameters such as the number of iterations and learning rate.
In the constructor method (init), the input data X is first modified by concatenating a column of ones to account for the bias term. Then, the data is split into training and testing sets using the train_test_split function from the popular machine learning library, scikit-learn (sklearn). Furthermore, a weight vector W is randomly initialized, and an empty list called cost is created to store the cost function values during training.
The primary logistic regression algorithm is executed within a loop that iterates for a specified number of iterations. In each iteration, the sigmoid function is applied to the dot product of the training input data X_train and the weight vector W to compute the predicted output values Y_hat. Next, the cost function is calculated using these predicted output values and the actual output labels Y_train. Then, the gradient of the cost function with respect to the weight vector W is computed using the grad method, and the weight vector W is updated using the learning rate lr and the computed gradient.
After the loop completes, the trained weight vector W is utilized to make predictions on the testing data using the prediction method. The sigmoid function is applied to the dot product of the testing input data X_test and the weight vector W to obtain the predicted output values y_test_pred. These predicted output values are then thresholded at 0.5 to obtain binary predictions. The accuracy of these predictions is computed by comparing them to the actual output labels Y_test, and the average accuracy is stored as the test_acc attribute of the class.

To make the work easier, several helper functions were defined, such as sigmoid, cost, gradient, etc. So right now, we have code for performing Logistic regression on any data that we feed into it.

Hyperparameter Tuning

We run the logistic regression over a wide range of values for max_features with CountVectorizer, CountVectorizer with binary true, and TfIdfVectorizer for a fixed learning rate and find out the best value for max_features and best feature extraction model. From the seen hyperparameter tuning, it was found that the best results were obtained for max_features=4000 and binary=True for CountVectorizer. Contrary to what we thought before that having more features would make the model better we find that we attain the highest validation accuracy at only 4000 features that are selected based on the most frequent of most seen words in the dataset. If we don't fix the max_features, we will end up with 60k features, as there are about 60k unique words in that corpus.

Testing:

After concluding that Logistic regression performs the best among the models I have tried and choosing the feature extraction process to be CountVectorizer with max_features=4000 and binary=True, we have a model with a decent feature extraction part. Combining all these, I wrote a function called 'run_this', which takes in the path of the test dataset with the format mentioned in the assignment. To test it on a test dataset, use the path of test data as input to 'run_this' and then make sure that the train data, which is the 'spam' folder with 2 CSV files named 'completeSpamAssassin.csv' and 'processed_data.csv,' is in the same directory. I will be submitting this dataset along with the code and report.

So to test on the test data, pass in the path of the test data folder as input to 'run_this'. This function returns a dictionary with file names as keys and predictions as their values. Now using this dictionary, one can find out the test accuracy.

Conclusion:

I have built a custom spam classification model using Logistic regression for classification and CountVectorizer for feature extraction, which gave a validation accuracy of >98%. I have also provided you with a function that will take test data as input and train on the training data that I have, and give predictions on the test data.

Submission:

In the submission, you will see 5 things.

- One notebook named 'Spam_classifier.ipynb' comprises all the codes I have tried and tested, along with explanations. This includes the run_this function that you can use for the testing on test data. **The last cell would be all that you will be needed to run for testing.**

- One folder named **spam** with 2 CSV files named ‘completeSpamAssassin.csv’ and ‘processed_data.csv.’ This folder is the one with training data. We will be using processed_data.csv for training the models.
- One folder called **TestDataset**, a dummy folder with 3 emails in Txt format to illustrate the working of the ‘run_this’ function.
- One text file named Participant_details, as asked in the assignment.
- One pdf named Report is this file.