BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

BITS F464 Machine Learning

Name	ID No.
Donkada Vishal Dheeraj	2018A7PS0239H
Thakur Shivank Singh	2018A7PS0439H
Pranav Reddy Pesaladinne	2018A7PS0238H

Assignment 1B

Naive Bayes Classifier

Design Procedure and Implementation

The Naive Bayes classifier is a simple classifier that classifies based on probabilities of events. It is the applied commonly to text classification. Though it is a simple algorithm, it performs well in many text classification problems.

We have implemented a Naïve Bayes Classifier model which classifies a given mail as spam or not using pandas and NumPy libraries. The code mainly consists of the following parts –

- Loading the Data (.txt to .csv)

  Converting the given .txt file into a .csv file with mail text as one column and its label (spam or not) in next column. We read the text file and based on the format and whitespaces (label at the end of each line), make a dictionary with the column names and convert it to a .csv file using pandas library.

```
file = open(pathdir + '/dataset_NB.txt','r')    # opening .txt file
dataset = file.readlines()    # reads text file and returns a list containing each line in the file as a list item
dataset[-1] = dataset[-1]+'\n'
file.close()
```

```
text = []
spam = []
for data in dataset:
    text.append(data[:-2].rstrip())    # appending mail text to list
    spam.append(data[-2])              # appending spam label to list
```

```
dictionary = {
    'mail':text,
    'spam':spam
}
df = pd.DataFrame(dictionary)    # convert dict to pandas dataframe
df.to_csv(pathdir + '/dataset_NB.csv',index=False)    # saving dataframe as .csv file
```

- Data Analysis and Pre-processing

  First, we check the different statistics of the dataset like word count, number of unique/distinct words and topmost frequently occurring words. This gives us an overall idea of the text data we are dealing with and gives us an idea of the data pre-processing steps which may help the model.

```
print("Distribution of labels in given data")
print(df['spam'].value_counts())    # count of unique labels in the 'spam' column
# df.groupby('spam').spam.count()

count = 0
for mail in df['mail']:
    count += len(mail.split())
print("Total number of words:", count)

word_dict = {}
for mail in df['mail']:
    for word in mail.split():
        word = word.lower()
        if word not in word_dict:
            word_dict[word] = 1
        else:
            word_dict[word] += 1
print("Total distinct words:", len(word_dict))
print("Top frequent words:-")
dict(sorted(word_dict.items(), key=lambda item: item[1], reverse=True))    # prints a sorted dict with topmost freq words
```

The following steps help in cleaning and filtering the text data to extract only important features (words) which may help the model in classifying the mails better.

- lower_case() : to lowercase text
- remove_punct(): to remove punctuations and other symbols
- remove_stopwords(): to remove a defined list of stopwords (and other frequently occurring words)

```
# function to lowercase text
def lower_case(text):
    clean_text = str(text).lower()
    return clean_text

# function to remove puctuations and other symbols
def remove_punct(text):
    punctuations = '''!#$%&'"()*+,-./:;?@[\]^_`{|}~0123456789'''
    clean_text = ""
    for char in text:
        if char in punctuations:
            clean_text += " "
        else:
            clean_text += char
    return " ".join(clean_text.strip().split())

# function to remove stopwords
def remove_stopwords(text):
    tmp = []
    for word in text.split():
        if word not in stopwords:
            tmp.append(word)
    clean_text = " ".join(tmp)
    return clean_text
```

```
def preprocessing(df):
    for i,data in enumerate(df.iloc[:,0]):
        data = lower_case(data)
        data = remove_punct(data)
        data = remove_stopwords(data)
        df.iloc[i] = data
    return df
```

- ## Cross Validation and Splitting Data

  We use 7-fold Cross Validation and split data into training and test data based on the iterations.

```
k = 7    # for 7-Fold Cross Validation
for _ in range(0,k):
    # K-FOLD SPLIT
    start = int((_*len(df))/k)
    end = int(((_+1)*len(df))/k)
    # print(start, end)
    ### TEST data
    X_test = df.iloc[start:end+1, :-1]    # all cols except last col as features (X)
    Y_test = df.iloc[start:end+1, -1]     # last col as labels/classes/categories (Y)
    ### TRAINING data
    X_train = df.iloc[:start, :-1]
    X_train = X_train.append(df.iloc[end+1:, :-1])
    Y_train = df.iloc[:start, -1]
    Y_train = Y_train.append(df.iloc[end+1:, -1])

    X_train = X_train.reset_index(drop=True)
    X_test = X_test.reset_index(drop=True)
    Y_train = Y_train.reset_index(drop=True)
    Y_test = Y_test.reset_index(drop=True)
```

- ## Finding unique words and Bag of words

  We find unique words and store it in a dictionary, with key as the word and value as its count. Then we use a list to represent bag of words. We make a list for every mail and fill the index where the unique word is present in that mail. This list is then appended to the overall “bow” list and then converted to a NumPy array.

```
    # UNIQUE WORDS DICTIONARY
    uniq_dict = {}
    for mail in X_train['mail']:
        # print(mail)
        for word in mail.split():
            # print(word)
            if word not in uniq_dict:
                uniq_dict[word] = 1
            else:
                uniq_dict[word] += 1
```

```
    # BAG OF WORDS
    bow = []    # bag of words
    for mail in X_train['mail']:
        ohe_sent = [0] * len(uniq_dict)   # one hot encoding of words in a sentence
        for word in mail.split():
            ohe_sent[list(uniq_dict.keys()).index(word)] = 1
        bow.append(ohe_sent)

    bow = np.asarray(bow)
    print("Bag of Words shape:", bow.shape)
```

- ## Calculating probabilities and testing

  We now implement the Naïve Bayes model and calculate probabilities based on the standard Bayes formula.

$$p(C_k|x) = \frac{p(C_k)p(x|C_k)}{p(x)} \propto p(C_k)p(x|C_k) = p(C_k, x_1, x_2, ..., x_n)$$

We also apply Laplace Smoothening to solve the problem of zero probability for new unique words in the test data.

$$p_\lambda(C_k) = p_\lambda(Y = C_k) = \frac{\sum_{i=1}^{N} I(y_i = C_k) + \lambda}{N + K\lambda}$$

$$p(x_1 = a_j | y = C_k) = \frac{\sum_{i=1}^{N} I(x_{1i} = a_j, y_i = C_k) + \lambda}{\sum_{i=1}^{N} I(y_i = C_k) + A\lambda}$$

Where,

K is the number of classes/labels

A is the number of features (total number of unique words).

We have used λ=1 (ALPHA=1)

Firstly, we calculate prior probabilities and store in prior_y1 and prior_y0.

Then, we calculate the conditional probabilities for every word based on Two cases – y=0 and y=1.

- conditional_probs: A numpy array which stores the conditional probabilities of all unique words. The first row is for the case when y=0 and the second row when y=1

```
# CALCULATING PROBABILITIES
ALPHA = 1    # Laplace Smoothing (to handle missing words in dictionary)

#calculate prior probabilities => P(y=1) , P(y=0)
num_y1 = sum(Y_train)
num_y0 = len(Y_train) - num_y1
prior_y1 = (num_y1+ALPHA)/(len(Y_train) + (len(Y_train.unique())*ALPHA))
prior_y0 = 1-prior_y1

#calculate likelihoods/conditional prob => P(x/y=0) , P(x/y=1)
# Case Y=0:
pw_y0 = [0] * len(uniq_dict)
for j in range(len(uniq_dict)):
    sumw = 0
    for i in range(len(boW)):
        if boW[i][j] == 1 and Y_train[i] == 0:
            sumw += 1
    pw_y0[j] = sumw

pw_y0 = np.asarray(pw_y0)

conditional_probs = np.zeros((len(uniq_dict)))    # array of conditional probabilities calculated for each word in dict
conditional_probs = (pw_y0 + ALPHA)/(num_y0 + len(uniq_dict)*ALPHA)
conditional_probs = np.reshape(conditional_probs, (1,len(uniq_dict)))
```

```
    # Case Y=1:
    pw_y1 = [0] * len(uniq_dict)
    for j in range(len(uniq_dict)):
        sumw = 0
        for i in range(len(boW)):
            if boW[i][j] == 1 and Y_train[i] == 1:
                sumw += 1
        pw_y1[j] = sumw

    pw_y1 = np.asarray(pw_y1)
    # print(pw_y1)

    arr = (pw_y1 + ALPHA)/(num_y1 + len(uniq_dict)*ALPHA)
    arr = np.reshape(arr,(1, len(uniq_dict)))
    conditional_probs = np.append(conditional_probs, arr, axis=0)
```

Now, we begin classifying the test data. Firstly, we preprocess the data and then based on the text data, we substitute already calculated probabilities (during training) in the formula to obtain py0_w and py1_w. Based on which probability is higher, we classify the mail as spam(1) or not spam(0).

- Y_pred: list which stores the predictions of the model

```
    # TEST ACCURACY
    X_test_pp = preprocessing(X_test)

    Y_pred = [0] * len(X_test_pp)
    for i,mail in enumerate(X_test_pp['mail']):
        py0_w = prior_y0
        py1_w = prior_y1
        for w in mail.split():
            if w in uniq_dict:
                py0_w *= conditional_probs[0][list(uniq_dict.keys()).index(w)]
                py1_w *= conditional_probs[1][list(uniq_dict.keys()).index(w)]
            else:
                py0_w *= ALPHA/(num_y0 + len(uniq_dict)*ALPHA)
                py1_w *= ALPHA/(num_y1 + len(uniq_dict)*ALPHA)

        # print(py1_w, py0_w)
        if py1_w >= py0_w:
            #y_pred.append(1)
            Y_pred[i] = 1
        else:
            #y_pred.append(0)
            Y_pred[i] = 0
```

- ## Measuring performance

Finally, we calculate the Accuracy and F1 Score comparing predictions (Y_pred) and test data actual labels (Y_test)

```
# calculating accuracy and f1 score
def score(Y_test, Y_pred):
    # f1_score = TP/(TP+0.5*(FP+FN))
    TP = 0
    FP = 0
    FN = 0
    TN = 0
    for i in range(len(Y_test)):
        if Y_pred[i] == 1 and Y_test[i] == 1:
            TP += 1
        elif Y_pred[i] == 1 and Y_test[i] == 0:
            FP += 1
        elif Y_pred[i] == 0 and Y_test[i] == 1:
            FN += 1
        else:
            TN += 1
    acc = (TP+TN)/(len(Y_test))
    f1_scr = TP/(TP+0.5*(FP+FN))
    return acc, f1_scr
```

Results

Accuracy and F1 Score of the model over each fold and the overall average accuracy scores -

```
1. Test Accuracy: 83.22 %
   F1 Score:      0.8537
2. Test Accuracy: 86.81 %
   F1 Score:      0.8742
3. Test Accuracy: 79.17 %
   F1 Score:      0.8171
4. Test Accuracy: 77.78 %
   F1 Score:      0.8
5. Test Accuracy: 78.47 %
   F1 Score:      0.777
6. Test Accuracy: 77.78 %
   F1 Score:      0.7895
7. Test Accuracy: 76.92 %
   F1 Score:      0.7626
Average Test Accuracy: 80.01998001998001
Average F1 Score:      0.8105637031831149
```

We were able to achieve 80% overall accuracy using the Naïve Bayes Classifier after Laplace Smoothing and Data Pre-processing to classify mails as spam or not.

Conclusion and Limitations

- The ‘naive’ nature of the Naïve Bayes Classifier comes from the strong assumption that all features are independent of each other. Although this belief, makes the model faster than most other algorithms, this case rarely happens in real life as there are a lot of interactions and relations between every feature which affect the overall outcome. The assumption that all features are independent makes Naive Bayes algorithm less accurate than other complicated algorithms. Hence, we see there is a trade off between speed and accuracy.
- The Naïve Bayes algorithm faces the ‘zero-frequency problem’ where it assigns zero probability to a variable in the test data set which wasn’t available in the training dataset. This can be overcome using a technique called Laplace Smoothing which we have implemented in the model.
- The probability estimations may be incorrect based on the skewness of the data. Hence performance is sensitive to skewed data based on its distribution.