Sameer Sadruddin - 310748 - Group 2 - Lab 11

February 5, 2022

Machine Learning Lab

Lab 11

0.0.1 Importing Packages

[1]:
```
import re                                                    #Importing Regular␣
 ↪Expression
import string                                                #Importing String
import numpy as np                                           #Importing Numpy
import pandas as pd                                          #Importing Pandas
import random                                                #Importing Random
from sklearn.datasets import fetch_20newsgroups              #Importing News Dataset
from sklearn.svm import SVC                                  #Importing SVM
from sklearn.model_selection import GridSearchCV             #Importing Grid Search
from sklearn.metrics import accuracy_score                   #Importing Accuracy␣
 ↪Score Metric
from nltk.corpus import stopwords                            #Importing Stopwords
from nltk.tokenize import word_tokenize                      #Importing NLTK Word␣
 ↪Tokenizer
import itertools                                             #Importing Iterator
import warnings                                              #Importing Warnings
warnings.filterwarnings('ignore')
```

0.0.2 Exercise 0: Preprocessing Text Data

Reading the Dataset and taking Subset with two categories named as sci.med and comp.graphics

[2]:
```
#Initializing the value for Random seed
random_seed = 3116
```

[3]:
```
news = fetch_20newsgroups(subset='all', categories=['sci.med','comp.graphics'],␣
 ↪shuffle=True, random_state=random_seed)
```

Preprocessing textual data to remove punctuation, stop-words

Function to Preprocess the data by removing Stopwords, punctuations and tokenizing the data

[4]:
```python
def preprocess_data(news_string):
    #Extracting the English stopwords and converting it into a set
    english_stop_words = set(stopwords.words('english'))

    #Making the data into the lower case string and then tokenizing the data into word list
    news_string = word_tokenize(news_string.lower())

    #Removing stopwords and punctuations from the word list
    news_string = [word for word in news_string if word not in english_stop_words and word.isalpha()]

    #Returning the final processed data list
    return news_string
```

Extracting News and its Target label

[5]:
```python
news_items = news['data']
news_target = news['target']
```

Applying Preprocessing step on all the News items

[6]:
```python
#Initializing an empty list to store processed news items
processed_news = []

#Iterating for each news items
for n_item in news_items:

    #Applying preprocessing on current news item
    processed_news.append(preprocess_data(n_item))
```

Implementing a bag-of-words feature representation for each text sample

Function to create a Word Frequency Dictionary from the Provided Documents

[7]:
```python
def update_word_freq(data, freq_dict):
    #Iterating for all words in the document
    for word in data:
        #If word is already present in the dictionary than we add 1
        if word in freq_dict:
            freq_dict[word] += 1
        #If word is not present, then we create a new key and assign 1 as a value
        else:
            freq_dict[word] = 1

    #Returning the created word frequency dictionary
    return freq_dict
```

Function to create a Binary vector for a document based on Bag of Word Representation

[8]:
```
def bag_of_words(data, freq_dict):
    #Initializing a vector with zeros having the length equal to total unique␣
 ↪words in the corpus
    sentence_vector = np.zeros(shape=(len(freq_dict.keys()),))

    #Iterating over all words in the document
    for word in data:

        #Placing 1 in the vector based on index assigned to that word in the␣
 ↪dictionary
        if word in freq_dict.keys():
            sentence_vector[list(freq_dict.keys()).index(word)] = 1

    #Returning the document vector
    return sentence_vector
```

Converting each document in the dataset into a Bag of Word Representation

[9]:
```
#Total Features to consider for Processing
features = 1000
```

[10]:
```
#Creating a dictionary to store all unique words and there count in the entire␣
 ↪corpus
corpus_word_freq = {}

#Iterating for each document in the dataset
for doc in processed_news:

    #Updating the dictionary for each document
    corpus_word_freq = update_word_freq(doc, corpus_word_freq)

#Sorting the dictionary in Descending Order
corpus_word_freq = dict(sorted(corpus_word_freq.items(), key=lambda item:␣
 ↪item[1], reverse=True))

#Extracting only required words based on feature value
corpus_word_freq = dict(itertools.islice(corpus_word_freq.items(), features))
```

[11]:
```
#Initializing a empty list to store the Bag of Word representation for each␣
 ↪document
news_bog_vectors = []

#Iterating for each document in the dataset
for doc in processed_news:
```

```
    #Creating a bag of word representation vector and appending it into the␣
↪final list
    news_bog_vectors.append(bag_of_words(doc, corpus_word_freq))
```

Displaying the created Bag of Word Representation in the form of Dataframe

[12]:
```
news_bog_df = pd.DataFrame(news_bog_vectors, columns=corpus_word_freq.keys())
news_bog_df.head()
```

[12]:
```
   subject  lines  organization  one  would  image  university  writes  \
0      1.0    1.0           1.0  0.0    0.0    0.0         0.0     0.0
1      1.0    1.0           1.0  0.0    0.0    0.0         0.0     0.0
2      1.0    1.0           1.0  1.0    0.0    0.0         0.0     0.0
3      1.0    1.0           1.0  0.0    1.0    0.0         0.0     0.0
4      1.0    1.0           1.0  1.0    0.0    0.0         0.0     0.0

   article  also  …  media  dan  exercise  request  den  goes  terms  shown  \
0      0.0   0.0  …    0.0  0.0       0.0      0.0  0.0   0.0    0.0    0.0
1      0.0   0.0  …    0.0  0.0       0.0      0.0  0.0   0.0    0.0    0.0
2      0.0   0.0  …    0.0  0.0       0.0      0.0  0.0   1.0    0.0    0.0
3      0.0   0.0  …    0.0  0.0       0.0      0.0  0.0   0.0    0.0    0.0
4      0.0   1.0  …    0.0  0.0       0.0      0.0  0.0   0.0    0.0    0.0

   centers  dangerous
0      0.0        0.0
1      0.0        0.0
2      0.0        0.0
3      0.0        0.0
4      0.0        0.0

[5 rows x 1000 columns]
```

Implementing a TF-IDF feature representation for each text sample

Function to calculate the Term Frequency (TF) for a word in a document

[13]:
```
def term_frequency(document, word):
    #tf = (total frequency of word in a document) / (total words in a document)
    return document[word]/sum(document.values())
```

Function to calculate the Inverse Document Frequency (IDF) for a word in the entire Corpus

[14]:
```
def inverse_document_frequency(total_doc_freq, word, total_documents):
    #idf = log(total documents / total frequency of word in all documents)
    return np.log(total_documents/total_doc_freq[word] + 1)
```

Function to calculate the TF-IDF of a all the words individually in the entire corpus

[15]:
```python
def tf_idf(document, total_doc_freq, total_documents):
    #Initializing a vector of zeros with a lenght of total unique words in the␣
↪entire corpus
    document_vector = np.zeros(shape=(len(total_doc_freq.keys()),))

    #Iterating for each word in the document
    for word in document.keys():

        #Checking if word exist in our feature set
        if word in total_doc_freq.keys():

            # tf-idf = tf * idf
            tf_idf = term_frequency(document, word) *␣
↪inverse_document_frequency(total_doc_freq, word, total_documents)

            #Inserting the calculated tf-idf for that word in the vector
            document_vector[list(total_doc_freq.keys()).index(word)] = tf_idf

    #Returning the final vector containing tf-idf values
    return document_vector
```

Converting each document in the dataset into a TF-IDF Representation

[16]:
```python
#Initializing a vector to store tf-idf vectors for each document
news_tfidf_vectors = []

#Iterating for each documents
for doc in processed_news:
    #Creating a word frequency dictionary for current document
    current_doc_dict = update_word_freq(doc, {})

    #Calculating and Appending the tf-idf vector in the final vector
    news_tfidf_vectors.append(tf_idf(current_doc_dict, corpus_word_freq,␣
↪len(processed_news)))
```

Displaying the created TF-IDF Representation in the form of Dataframe

[17]:
```python
news_tfidf_df = pd.DataFrame(news_tfidf_vectors, columns=corpus_word_freq.
↪keys())
news_tfidf_df.head()
```

[17]:

	subject	lines	organization	one	would	image	university	\
0	0.013108	0.013391	0.013782	0.000000	0.000000	0.0	0.0	
1	0.011704	0.011956	0.012305	0.000000	0.000000	0.0	0.0	
2	0.011300	0.011544	0.011881	0.030896	0.000000	0.0	0.0	
3	0.008739	0.008927	0.009188	0.000000	0.012308	0.0	0.0	
4	0.007364	0.007523	0.007743	0.010067	0.000000	0.0	0.0	

```
   writes  article      also  …  media  dan  exercise  request  den  \
0     0.0      0.0  0.000000  …    0.0  0.0       0.0      0.0  0.0
1     0.0      0.0  0.000000  …    0.0  0.0       0.0      0.0  0.0
2     0.0      0.0  0.000000  …    0.0  0.0       0.0      0.0  0.0
3     0.0      0.0  0.000000  …    0.0  0.0       0.0      0.0  0.0
4     0.0      0.0  0.011768  …    0.0  0.0       0.0      0.0  0.0

       goes  terms  shown  centers  dangerous
0  0.000000    0.0    0.0      0.0        0.0
1  0.000000    0.0    0.0      0.0        0.0
2  0.062734    0.0    0.0      0.0        0.0
3  0.000000    0.0    0.0      0.0        0.0
4  0.000000    0.0    0.0      0.0        0.0

[5 rows x 1000 columns]
```

Splitting the dataset randomly into train/validation/test splits according to ratios 80%:10%:10%

[18]:
```python
def split_dataset(news_vectors, targets, train_ratio, validation_ratio):

    #Combining the Feature columns and the target column into a single list
    combined = list(zip(news_vectors, targets))

    #Randomly shuffle the rows in the list
    random.shuffle(combined)

    #Calculating the training rows and validation rows
    train_rows = int(len(combined) * train_ratio)
    validation_rows = int(len(combined) * validation_ratio)

    #Extracting X matrix and Y matrix from the combined list
    X , y = list(zip(*combined))
    X, y = list(X), list(y)

    #Splitting X and y matrices into Training, Validation and Test set
    X_train, X_val, X_test = X[:train_rows], X[train_rows:
↪train_rows+validation_rows], X[train_rows+validation_rows:]
    y_train, y_val, y_test = y[:train_rows], y[train_rows:
↪train_rows+validation_rows], y[train_rows+validation_rows:]

    #Returning all the subsets
    return X_train, X_val, X_test, y_train, y_val, y_test
```

0.0.3 Exercise 1: Implementing Naive Bayes Classifier for Text Data

Class to Represent the Naive Bayes Algorithm

[19]:
```
class Naive_bayes:
    #Contructor function
    def __init__(self,feature_representation, dataset, target):

        #Checking if the feature representation value is valid otherwise␣
↪Raising Exception
        if feature_representation not in ['bog', 'tfidf']:
            raise Exception('Invalid value provided for Feature Representation')

        #Feature Representation type - Bag of Words (bog) or TF-IDF (tfidf)
        self.feature_representation = feature_representation

        #Feature Columns of the Dataset
        self.dataset = np.array(dataset)

        #Target/Label Column of the Dataset
        self.target = np.array(target)

        #Unique Target/Label values
        self.unique_target = list(set(target))

        #Current Accuarcy of the Model
        self.accuracy = 0

        #Combined Dataset containing both, the Feature Columns and the Target␣
↪Column
        self.combined = np.concatenate((self.dataset, self.target.
↪reshape(-1,1)), axis = 1)

    #Function to do Predictions on the dataset provided and calculate the␣
↪Accuracy
    def predict(self):

        #Iterating over all the different documents in the dataset
        for index, doc in enumerate(self.combined):

            #Initializing an empty list to store the predicted probability for␣
↪each target value
            tar_prob = []

            #Iterating over all unique target values for calculating there␣
↪probabilities
            for tar in self.unique_target:

                #Based on Feature representation type, calculating the␣
↪probabiltiy
```

```
                if self.feature_representation == 'bog':
                    tar_prob.append(self.__calculate_prob_bog(doc, tar))
                elif self.feature_representation == 'tfidf':
                    tar_prob.append(self.__calculate_prob_tfidf(doc, tar))

            #Normalizing each probability so that it sums to 1
            tar_prob = list(map(lambda x : x / (sum(tar_prob) + 1),tar_prob))

            #Extracting the class with the maximum Probability
            predicted_class = self.unique_target[tar_prob.index(max(tar_prob))]

            #Checking if the predicted class is equal to the actual class
            if predicted_class == self.target[index]:
                self.accuracy += 1

        #Calculating its final accuracy
        self.accuracy /= len(self.combined)

    #A private function to calculate the Probability for a document represented␣
↪using Bag of Words
    def __calculate_prob_bog(self, document, target):

        #Calculating the probability for a class itself
        # P(target) = (Number of times that class appears in the dataset) /␣
↪(total documents)
        prob_class = len(self.target[np.where(self.target == target)]) /␣
↪len(self.target)

        #Initializing the prior probability for each Word in the document
        words_prior_prob = 1

        #Iterating over each word in the document
        for i in range(len(document) - 1):

            #Only calculating the probability if the word exist in the document
            if document[i] == 1:

                #Calculating the Prior probability of the word given the class
                # P(w1 | target) = (Number of times the word occurs in all the␣
↪document given the class) / (Number of time word occurs in all documents)
                p_word_num = len(self.combined[np.where((self.combined[:,i] ==␣
↪1) & (self.combined[:,-1] == target))])
                p_word_den = len(self.combined[np.where(self.combined[:,-1] ==␣
↪target)])

                #Multiplying the current word prior probability with other words
```

```python
            words_prior_prob *= (p_word_num / p_word_den)

        #Returning the final probability -> P(class) * P(W | class)
        return words_prior_prob * prob_class

    #A private function to calculate the Probability for a document represented using TF-IDF
    def __calculate_prob_tfidf(self, document, target):

        #Calculating the probability for a class itself
        # P(target) = (Number of times that class appears in the dataset) / (total documents)
        prob_class = len(self.target[np.where(self.target == target)]) / len(self.target)

        #Initializing the prior probability for each Word in the document
        words_prior_prob = 1

        #Iterating over each word in the document
        for i in range(len(document) - 1):

            #Only calculating the probability if the word exist in the document
            if document[i] == 1:

                #Calculating the Prior probability of the word given the class
                # P(w1 | target) = (Number of times the word occurs in all the document given the class) / (Number of time word occurs in all documents)
                p_word_num = sum(self.combined[np.where((self.combined[:,i] == 1) & (self.combined[:,-1] == target))])
                p_word_den = sum(self.combined[np.where(self.combined[:,-1] == target)])

                #Multiplying the current word prior probability with other words
                words_prior_prob *= (p_word_num / p_word_den)

        #Returning the final probability -> P(class) * P(W | class)
        return words_prior_prob * prob_class

    #Function to display the Accuracy of the Model
    def score(self):
        feature_rep = 'Bag of Words' if self.feature_representation == 'bog' else 'TF-IDF'
        return 'The Accuracy for Naive Bayes using {} Representation is {:.2f}%'.format(feature_rep, self.accuracy * 100)
```

Using Bag of Word Representation

Splitting the Dataset with Bag of Word Representation into Train, Validation and Test sets

[20]:
```
X_train, X_val, X_test, y_train, y_val, y_test =␣
↪split_dataset(news_bog_vectors, news_target, 0.8, 0.1)
```

Creating and Fitting the Naive Bayes model on the training, Validation and Test Sets

[21]:
```
nb_train = Naive_bayes('bog', X_train, y_train)
nb_train.predict()

nb_validation = Naive_bayes('bog', X_val, y_val)
nb_validation.predict()

nb_test = Naive_bayes('bog', X_test, y_test)
nb_test.predict()
```

Calculating and Displaying the Training, Validation and Test Accuracies

[22]:
```
print('Training Accuracy: \n{}'.format(nb_train.score()))
print('\nValidation Accuracy: \n{}'.format(nb_validation.score()))
print('\nTest Accuracy: \n{}'.format(nb_test.score()))
```

```
Training Accuracy:
The Accuracy for Naive Bayes using Bag of Words Representation is 95.73%

Validation Accuracy:
The Accuracy for Naive Bayes using Bag of Words Representation is 97.45%

Test Accuracy:
The Accuracy for Naive Bayes using Bag of Words Representation is 98.98%
```

Using TF-IDF Representation

Splitting the Dataset with TF-IDF Representation into Train, Validation and Test sets

[23]:
```
X_train, X_val, X_test, y_train, y_val, y_test =␣
↪split_dataset(news_tfidf_vectors, news_target, 0.8, 0.1)
```

Creating and Fitting the Naive Bayes model on the training, Validation and Test Sets

[24]:
```
nb_train = Naive_bayes('tfidf', X_train, y_train)
nb_train.predict()

nb_validation = Naive_bayes('tfidf', X_val, y_val)
nb_validation.predict()

nb_test = Naive_bayes('tfidf', X_test, y_test)
nb_test.predict()
```

Calculating and Displaying the Training, Validation and Test Accuracies

[25]:
```python
print('Training Accuracy: \n{}'.format(nb_train.score()))
print('\nValidation Accuracy: \n{}'.format(nb_validation.score()))
print('\nTest Accuracy: \n{}'.format(nb_test.score()))
```

```
Training Accuracy:
The Accuracy for Naive Bayes using TF-IDF Representation is 50.06%

Validation Accuracy:
The Accuracy for Naive Bayes using TF-IDF Representation is 51.53%

Test Accuracy:
The Accuracy for Naive Bayes using TF-IDF Representation is 52.28%
```

The Accuracy of TF-IDF is low because we have only taken 1000 features.If we increase the number of features, the accuracy can improve subsequently.

0.0.4 Exercise 2: Implementing SVM Classifier via Scikit-Learn

Defining Hyperparameter Grid for Grid Search

[26]:
```python
hyperparameter_grid = {'C' : [0.01,0.02,0.03],
                       'kernel': ['linear', 'rbf'],
                       'gamma': ['scale','auto']}
```

Using Bag of Word Representation

Splitting the Dataset with Bag of Word Representation into Train, Validation and Test sets

[27]:
```python
X_train, X_val, X_test, y_train, y_val, y_test =␣
 ↪split_dataset(news_bog_vectors, news_target, 0.8, 0.1)
```

Creating and Fitting the SVM model on the training set using Grid Search and different Hyperparameter combination

[28]:
```python
#Initializing a SVM model with the random seed
svm = SVC(random_state=random_seed)

#Creating a Grid Seach object with the SVM model and K-fold Cross validation
grid_model = GridSearchCV(svm, hyperparameter_grid, n_jobs=-1, cv=5,␣
 ↪scoring='accuracy', return_train_score=True)

#Fitting the training dataset on SVM with different Hyperparameters
grid_model.fit(X_train, y_train)
```

[28]:
```
GridSearchCV(cv=5, estimator=SVC(random_state=3116), n_jobs=-1,
             param_grid={'C': [0.01, 0.02, 0.03], 'gamma': ['scale', 'auto'],
                         'kernel': ['linear', 'rbf']},
             return_train_score=True, scoring='accuracy')
```

[29]:
```
print('Best Hyperparameter combination found for Bag of Word Representation␣
↪after applying Grid Search: \n{}'.format(grid_model.best_params_))
```

```
Best Hyperparameter combination found for Bag of Word Representation after
applying Grid Search:
{'C': 0.03, 'gamma': 'scale', 'kernel': 'linear'}
```

Calculating and Displaying the Validation and Test Accuracies

[30]:
```
print('Validation Accuracy on best Hyperparameters: {:.2f}'.
↪format(accuracy_score(y_val, grid_model.predict(X_val)) * 100))
```

```
Validation Accuracy on best Hyperparameters: 95.41
```

[31]:
```
print('Test Accuracy on best Hyperparameters: {:.2f}'.
↪format(accuracy_score(y_test, grid_model.predict(X_test)) * 100))
```

```
Test Accuracy on best Hyperparameters: 96.45
```

Using TF-IDF Representation

Splitting the Dataset with TF-IDF Representation into Train, Validation and Test sets

[32]:
```
X_train, X_val, X_test, y_train, y_val, y_test =␣
↪split_dataset(news_tfidf_vectors, news_target, 0.8, 0.1)
```

Creating and Fitting the SVM model on the training set using Grid Search and different Hyperparameter combination

[33]:
```
#Initializing a SVM model with the random seed
svm = SVC(random_state=random_seed)

#Creating a Grid Seach object with the SVM model and K-fold Cross validation
grid_model = GridSearchCV(svm, hyperparameter_grid, n_jobs=-1, cv=5,␣
↪return_train_score=True)

#Fitting the training dataset on SVM with different Hyperparameters
grid_model.fit(X_train, y_train)
```

[33]:
```
GridSearchCV(cv=5, estimator=SVC(random_state=3116), n_jobs=-1,
             param_grid={'C': [0.01, 0.02, 0.03], 'gamma': ['scale', 'auto'],
                         'kernel': ['linear', 'rbf']},
             return_train_score=True)
```

[34]:
```
print('Best Hyperparameter combination found for TF-IDF Representation after␣
↪applying Grid Search: \n{}'.format(grid_model.best_params_))
```

```
Best Hyperparameter combination found for TF-IDF Representation after applying
Grid Search:
{'C': 0.03, 'gamma': 'scale', 'kernel': 'rbf'}
```

Calculating and Displaying the Validation and Test Accuracies

[35]:
```
print('Validation Accuracy on best Hyperparameters: {:.2f}'.
↪format(accuracy_score(y_val, grid_model.predict(X_val)) * 100))
```

```
Validation Accuracy on best Hyperparameters: 74.49
```

[36]:
```
print('Test Accuracy on best Hyperparameters: {:.2f}'.
↪format(accuracy_score(y_test, grid_model.predict(X_test)) * 100))
```

```
Test Accuracy on best Hyperparameters: 68.02
```