Mid_Sem

September 22, 2021

1 Mid-Semester Assignment

1.1 Name = P. V. Sriram

1.2 Roll No. = 1801CS37

2 Setup

[1]:
```
# Import Libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
```

[2]:
```
# Read File
file = open("data.txt", 'r')

lines = file.readlines()
```

[3]:
```
# Extract Data
labels = []
data = []

for line in lines:
    labels.append(line.split('\t')[0])
    data.append(line.split('\t')[1])
```

[4]:
```
X_train, Y_train = data, labels
```

3 Multi-Nomial Naive Bayes

[5]:
```
import string
# import nltk
# nltk.download('stopwords')
from nltk.corpus import stopwords
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score
stop_words = set(stopwords.words('english'))
```

```
from sklearn.feature_extraction.text import CountVectorizer

# Multi-Nomial Naive Bayes Class
class MNB():
    def __init__(self, alpha = 0.001, cutoff_freq = 0):
        # Initialise vectorisers
        self.vectorizer_x = CountVectorizer()
        self.vectorizer_y = CountVectorizer()
        self.cutoff_freq = cutoff_freq
        self.alpha = alpha

    # Preprocess vocabulary
    def pre_process(self):
        self.vocab = {}
        self.features = []
        for i in range(len(self.X_train)):
            for word in self.X_train[i].split():
                word_new  = word.strip(string.punctuation).lower()
                if (len(word_new) > 2)  and (word_new not in stop_words):
                    if word_new in self.vocab:
                        self.vocab[word_new] += 1
                    else:
                        self.vocab[word_new] = 1

        for key in self.vocab:
            if self.vocab[key] >= self.cutoff_freq:
                self.features.append(key)

    # Vectorise each sentences
    # Using count vectorisers
    def encode(self):
        self.vectorizer_x.fit(self.features)
        self.X_cv_train = self.vectorizer_x.transform(self.X_train).toarray()

        self.vectorizer_y.fit(self.classes)
        self.Y_cv_train = self.vectorizer_y.transform(self.Y_train).toarray()

    # Train the Model
    # Calculate conditional Probabilities
    def fit(self, X_train, Y_train):
        self.X_train = X_train
        self.Y_train = Y_train

        self.classes, self.class_probs = np.unique(self.Y_train, return_counts␣
↪= True)
        self.class_probs = self.class_probs / np.sum(self.class_probs)
```

```
        self.pre_process()
        self.encode()
        self.index_matrix = np.dot(self.X_cv_train.T, self.Y_cv_train)
        temp1 = np.tile(np.sum(clf1.index_matrix, axis = 1), (2, 1)).T + np.
↪tile(np.sum(clf1.index_matrix, axis = 0), (29, 1))
        self.prob_matrix = (self.index_matrix + self.alpha) / temp1

    # Predict class using
    # conditional probabilites
    def predict(self, X_test):
        X_cv_test = self.vectorizer_x.transform(X_test).toarray()
        prediction = []
        for sample in X_cv_test:
            current = -1 * np.inf
            for i, y in enumerate(self.prob_matrix.T):
                temp = np.multiply(sample.T, y)
                prob = np.sum(np.log(temp[np.where(temp != 0)])) + np.log(self.
↪class_probs[i])
                if (prob > current):
                    current = prob
                    class_ = self.classes[i]
            prediction.append(class_)
        return (prediction)

    # Evaluate Performance
    def evaluate(self, X_val, Y_val):
        pred = self.predict(X_val)
        return (accuracy_score(pred, Y_val))
```

[6]:
```
# Initialise Model
clf1 = MNB()

# Train Model
clf1.fit(X_train, Y_train)
```

[7]:
```
# Visualise the Index matrix
clf1.index_matrix
```

[7]:
```
array([[0, 1],
       [0, 1],
       [1, 1],
       [0, 3],
       [0, 1],
       [0, 1],
       [0, 1],
       [1, 1],
       [0, 1],
```

```
       [0, 1],
       [0, 1],
       [0, 2],
       [0, 1],
       [0, 1],
       [1, 2],
       [0, 1],
       [0, 1],
       [1, 1],
       [0, 1],
       [0, 1],
       [0, 1],
       [0, 1],
       [0, 1],
       [1, 0],
       [0, 1],
       [0, 1],
       [1, 1],
       [1, 0],
       [0, 1]])
```

[8]:
```
# Accuracy
print("Accuracy of model on train dataset is: ", clf1.evaluate(X_train,␣
 ↪Y_train))
```

```
Accuracy of model on train dataset is:  1.0
```

[9]:
```
# Vocabulary of dataset
print("The model considers following vocabulary (along with their occurences):␣
 ↪")
clf1.vocab
```

```
The model considers following vocabulary (along with their occurences):
```

[9]:
```
{'recent': 1,
 'years': 1,
 'researchers': 1,
 'computer': 2,
 'vision': 2,
 'proposed': 1,
 'many': 1,
 'deep': 3,
 'learning': 2,
 'methods': 3,
 'various': 1,
 'tasks': 1,
 'facial': 2,
```

```
 'recognition': 2,
 'made': 1,
 'enormous': 1,
 'leap': 1,
 'using': 1,
 'techniques': 1,
 'systems': 1,
 'benefit': 1,
 'hierarchical': 1,
 'architecture': 1,
 'learn': 1,
 'discriminative': 1,
 'face': 1,
 'representation': 1,
 'widely': 1,
 'used': 1}
```

[10]:
```
# Perfrom prediction on Test Sample
ans = clf1.predict(["Deep learning based computer vision methods have been used␣
 ↪for facial recognition."])
print("The Test Sample belongs to class: ", ans[0])
```

```
The Test Sample belongs to class:  CV
```

4 Multi-Variate Naive Bayes

[11]:
```
# Multi-Nomial Naive Bayes Class
class MVB():
    def __init__(self, alpha = 0.001, cutoff_freq = 0):
        # Initialise vectorisers
        self.vectorizer_x = CountVectorizer(binary = True)
        self.vectorizer_y = CountVectorizer(binary = True)
        self.cutoff_freq = cutoff_freq
        self.alpha = alpha

    # Preprocess vocabulary
    def pre_process(self):
        self.vocab = {}
        self.features = []
        for i in range(len(self.X_train)):
            for word in self.X_train[i].split():
                word_new  = word.strip(string.punctuation).lower()
                if (len(word_new)>2)  and (word_new not in stop_words):
                    if word_new in self.vocab:
                        self.vocab[word_new]+=1
                    else:
```

```
                        self.vocab[word_new]=1

        for key in self.vocab:
            if self.vocab[key] >= self.cutoff_freq:
                self.features.append(key)

    # Vectorise each sentences
    # Using count vectorisers
    def encode(self):
        self.vectorizer_x.fit(self.features)
        self.X_cv_train = self.vectorizer_x.transform(self.X_train)

        self.vectorizer_y.fit(self.classes)
        self.Y_cv_train = self.vectorizer_y.transform(self.Y_train)

    # Train the Model
    # Calculate conditional Probabilities
    def fit(self, X_train, Y_train):
        self.X_train = X_train
        self.Y_train = Y_train

        self.classes, self.class_probs = np.unique(self.Y_train, return_counts␣
↪= True)
        self.class_probs = self.class_probs / np.sum(self.class_probs)

        self.pre_process()
        self.encode()

        self.index_matrix = np.dot(self.X_cv_train.T, self.Y_cv_train).toarray()
        temp1 = np.tile(np.sum(clf1.index_matrix, axis = 1), (2, 1)).T + np.
↪tile(np.sum(clf1.index_matrix, axis = 0), (29, 1))
        self.prob_matrix = (self.index_matrix + self.alpha) / temp1

    # Predict class using
    # conditional probabilites
    def predict(self, X_test):
        X_cv_test = self.vectorizer_x.transform(X_test).toarray()
        prediction = []
        for sample in X_cv_test:
            current = -1 * np.inf
            for i, y in enumerate(self.prob_matrix.T):
                temp = np.multiply(sample.T, y)
                prob = np.sum(np.log(temp[np.where(temp != 0)])) + np.log(self.
↪class_probs[i])
                if (prob > current):
                    current = prob
                    class_ = self.classes[i]
```

```
            prediction.append(class_)
        return (prediction)

    # Evaluate Performance
    def evaluate(self, X_val, Y_val):
        pred = self.predict(X_val)
        return (accuracy_score(pred, Y_val))
```

[12]:
```
# Initialise Model
clf2 = MVB()

# Train Model
clf2.fit(X_train, Y_train)
```

[13]:
```
# Visualise the Index matrix
clf2.index_matrix
```

[13]:
```
array([[0, 1],
       [0, 1],
       [1, 1],
       [0, 2],
       [0, 1],
       [0, 1],
       [0, 1],
       [1, 1],
       [0, 1],
       [0, 1],
       [0, 1],
       [0, 2],
       [0, 1],
       [0, 1],
       [1, 2],
       [0, 1],
       [0, 1],
       [1, 1],
       [0, 1],
       [0, 1],
       [0, 1],
       [0, 1],
       [0, 1],
       [1, 0],
       [0, 1],
       [0, 1],
       [1, 1],
       [1, 0],
       [0, 1]])
```

[14]:
```
# Accuracy
print("Accuracy of model on train dataset is: ", clf2.evaluate(X_train,␣
↪Y_train))
```

```
Accuracy of model on train dataset is:  1.0
```

[15]:
```
# Vocabulary of dataset
print("The model considers following vocabulary (along with their occurences):␣
↪")
clf2.vocab
```

```
The model considers following vocabulary (along with their occurences):
```

[15]:
```
{'recent': 1,
 'years': 1,
 'researchers': 1,
 'computer': 2,
 'vision': 2,
 'proposed': 1,
 'many': 1,
 'deep': 3,
 'learning': 2,
 'methods': 3,
 'various': 1,
 'tasks': 1,
 'facial': 2,
 'recognition': 2,
 'made': 1,
 'enormous': 1,
 'leap': 1,
 'using': 1,
 'techniques': 1,
 'systems': 1,
 'benefit': 1,
 'hierarchical': 1,
 'architecture': 1,
 'learn': 1,
 'discriminative': 1,
 'face': 1,
 'representation': 1,
 'widely': 1,
 'used': 1}
```

[16]:
```
# Perfrom prediction on Test Sample
ans = clf2.predict(["Deep learning based computer vision methods have been used␣
↪for facial recognition."])
print("The Test Sample belongs to class: ", ans[0])
```

```
The Test Sample belongs to class:  CV
```