HW2

October 5, 2021

[1]:
```
import pandas as pd
import numpy as np
import nltk
nltk.download('wordnet')
import re
from bs4 import BeautifulSoup
import contractions
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from sklearn.linear_model import Perceptron
from sklearn.metrics import confusion_matrix as cm
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import LinearSVC as SVC
import warnings
import gensim
import gensim.downloader as api
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, utils
from numpy import vstack
from sklearn.metrics import accuracy_score
from numpy import argmax
from copy import deepcopy
# device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# torch.cuda.set_device(device)
warnings.filterwarnings('ignore')
CUDA_LAUNCH_BLOCKING=1
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\mishr\AppData\Roaming\nltk_data…
[nltk_data]   Package wordnet is already up-to-date!
```

```
[2]: class DataTranformation(object):

         def __init__(self, filename, preprocess):
             self.filename = filename
             self.random_state = 10
             self.n = 50000
             self.preprocess = preprocess
             print("Preproces: " + str(preprocess))

         def read_file(self, error_bad_lines = False, warn_bad_lines = False, sep =␣
     ↪"\t"):
             df = pd.read_csv(self.filename, sep = sep, error_bad_lines =␣
     ↪error_bad_lines, warn_bad_lines = warn_bad_lines)
             df = df.dropna()
             return df

         def formation(self, row1 = 'review_body', row2 = 'star_rating', ):
             df = self.read_file()
             df = df[[row1, row2]]
             df = df.dropna()

             dataset = pd.concat([df[df['star_rating'] == 1].sample(n = 50000,␣
     ↪random_state = 10),
                                  df[df['star_rating'] == 2].sample(n = 50000, random_state =␣
     ↪10),
                                  df[df['star_rating'] == 3].sample(n = 50000, random_state =␣
     ↪10),
                                  df[df['star_rating'] == 4].sample(n = 50000, random_state =␣
     ↪10),
                                  df[df['star_rating'] == 5].sample(n = 50000, random_state =␣
     ↪10)])

             dataset = dataset.reset_index(drop = True)

             return dataset

         def label(self, rows):
             if rows.star_rating > 3:
                 return 1
             elif rows.star_rating < 3:
                 return 2
             else:
                 return 3

         def apply_label(self):
             dataset = self.formation()
```

```
        dataset['label'] = dataset.apply(lambda row : self.label(row), axis = 1)

        return dataset

    def remove_html_and_url(self, s):
        a = re.sub(r'(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]*)',␣
↪'', s, flags=re.MULTILINE)
        soup = BeautifulSoup(a, 'html.parser')
        a = soup.get_text()
        return a

    def tokenize(self, s):
        text_tokens = word_tokenize(s)
        return text_tokens

    def without_preprocess(self):
        dataset = self.apply_label()
        dataset.review_body = dataset.review_body.apply(self.tokenize)
        return dataset

    def with_preprocess(self):
        dataset = self.apply_label()
        dataset.review_body = dataset.review_body.str.lower()

        dataset.review_body = dataset.review_body.apply(lambda s: self.
↪remove_html_and_url(s))
        dataset.review_body = dataset.review_body.apply(lambda s: re.
↪sub("[^a-zA-Z']+", " ", s))
        dataset.review_body = dataset.review_body.apply(lambda s: re.sub(' +',␣
↪' ', s))

        dataset.review_body = dataset.review_body.apply(self.tokenize)

        dataset.dropna()
        return dataset

    def train_test_split(self):

        if self.preprocess:
            dataset = self.with_preprocess()
        else:
            dataset = self.without_preprocess()

        train = dataset.sample(frac = 0.8, random_state = 200)
        test = dataset.drop(train.index)
        train = train.reset_index(drop = True)
        test = test.reset_index(drop = True)
        return train, test
```

Here I made a class for data processing and splitting the dataset into train and test dataset. So for constructor **init** method will take filename from where we have to read the files, preprocess is a boolean variable which tells if we want to process the dataset. By processing the dataset means to clean the dataset (removing html and urls markups, removing non-alphabetic characters, converting all the strings to lowercase).

Methods: read_file: args: error_bad_lines -> boolean {to remove all the lines if they mismatch the size of the dataframe} warn_bad_lines -> boolean {to remove the warning if while reading csv file we found any null cells or bad rows or columns} sep {which sperator is used by the file. For eg csv files are seperated by ',', in tsv files the rows are sepearted by tabs '/t'} Objective: to read the file using pandas and dropping all the rows if it contains null or empty cells. Return: dataframe

formation: args: row1, row2 -> string {mentioning which columns are required for training} Objective: as the question suggested we need to 50000 samples for every star rating so in this method we just store random 50k samples for each star rating and push it into new dataframe. Return: dataframe

label: args: None Objective : a method to map the star ratings to a particular class label based on the conditions given in the question Condition: if star rating is greater than 3 then map it to class label '1'. if star rating is less than 3 then map it to class label '2'. else all the star rating equal to 3 map it to class label '3'. Return: None

apply_label: args: None Objective: to apply above method onto the dataframe generated from formation method. Return: dataframe

remove_html_and_url: args: s -> string Objective: to remove all the html markups and url from a given string Return: string

tokenize: args: s -> string Objective: to tokenize the string (Converting a given string into list of words or tokens) Return: List

without_preprocess: args: None Objective: To return the dataframe without preprocessing the dataset. Return: dataframe

with_preprocess: args: None Objective: To return the dataframe with preprocessed data. We will remove all the html, urls, extra spaces and non-alphabetic characters from the dataset. Return: dataframe

train_test_split: args: None Objective: to split the dataframe into training and testing dataframe. We are using 80-20 split over here. Return: dataframe

[3]:
```
class Vectorization(object):

    def __init__(self, model, dataset, model_type = "model", classification =␣
 ↪"binary", mode = "mean", pad = False):
        self.model = model
        self.dataset = dataset
        self.model_type = model_type # our own model or pretrained
        self.classification = classification # binary or multi-class
        if self.model_type == "pretrained":
            self.vocab = self.model
        if self.model_type == "model":
            self.vocab = self.model.wv

        self.mode = mode
        self.pad =pad

        print("Vectorizing training dataset....")
        print("Model Type: " + self.model_type)
        print("Classification: " + self.classification)

    def get_mean_vector(self, data_review_body, data_label):

        if self.classification == "binary":
            if data_label != 3:
                if self.model_type == "model":
                    words = [word for word in data_review_body if word in self.
↪vocab.index_to_key]
                    if len(words) >= 1:
                        rev = []
                        for word in words:
                            rev.append(np.array(self.vocab[word]))

                        if type(data_label) is not int : print("Found")
                        return rev, data_label
                else:
                    words = [word for word in data_review_body if word in self.
↪vocab]
                    if len(words) >= 1:
                        rev = []
                        for word in words:
                            rev.append(np.array(self.vocab[word]))

                        if type(data_label) is not int : print("Found")
                        return rev, data_label

        else:
            if self.model_type == "mode":
                words = [word for word in data_review_body if word in self.
↪vocab.index_to_key]
                if len(words) >= 1:
                    rev = []
                    for word in words:
                        rev.append(np.array(self.vocab[word]))
                    return rev, data_label
            else:
                words = [word for word in data_review_body if word in self.
↪vocab]
                if len(words) >= 1:
                    rev = []
                    for word in words:
                        rev.append(np.array(self.vocab[word]))
                    return rev, data_label


    def feature_extraction(self):
        feature = []
        y_label = []
        # print(self.vocab.index_to_key)
        for data_review_body, data_label in zip(self.dataset.review_body, self.
↪dataset.label):
            try:
                x, y = self.get_mean_vector(data_review_body, data_label)
                if self.pad:
                    if len(x) >= 50:
                        feature.append(x[:50])
                        y_label.append(y)
                    else:
                        feature.append(x)
                        y_label.append(y)
                else:
                    if self.mode == "vec":
                        if len(x) >= 10:
                            feature.append(x[:10])
                            y_label.append(y)
                    else:
                        feature.append(np.mean(x, axis = 0))
                        y_label.append(y)
            except:
                pass
        print("Vectorization Completed")
        return feature, y_label

    def pad_review(self, review, seq_len):

        features = np.zeros((seq_len, 300), dtype=float)
        features[-len(review):] = np.array(review)[:seq_len]

        return features

    def join_words(self, x):
        y = ""
        for ele in x:
            y = ' '.join(ele)
        return y
```

In this class we generate training and testing dataset for different model. Constructor take several arguments: model: Either our general model or pretrained model dataset: training or testing model_type: Either our general model or pretrained model -> string classification: binary or multiclass. -> string mode: mean or vectors (Given in the question we need to either form mean model which is simple model or to generate 10-vec/50-vec dataset for MLP, RNN, and GRU) -> string pad: So in the given question for RNN and GRU we need to padding for the dataset whose vector size is less than 50. -> boolean

Methods: get_mean_vector: args: data_review_body, data_label: dataframes column Objective: extracting the feature vectors for calcualting mean for specific models Return: corresponding feature data vectors and its label

feature_extraction: args: None Objective: Based on the cases it generates the dataset. For eg for a simple binary model it will generate mean vectors for each row of data review body, for rnn it will generate 50 x 300 feature vector. Return: List of features and label

pad_review: args: review, seq_len : review -> list of words for a particualr row, seq_len-> len of the list Objective: This method is for RNN and GRU data generation. As the question stated we need to pad zeros in feature map which have seq_len less than 50. Return: feature map

join_words: args: x->list of words Objective: to convert list of words into string Return: string

[4]:
```
class Sentence(object):
    def __init__(self, dataset):
        self.dataset = dataset
    def __iter__(self):
        for row in self.dataset:
            yield row
```

Sentence class is an iterator class which yields single row of the dataset.

[5]:
```
class Percept(object):

    def __init__(self, X_train, Y_train, X_test, Y_test, max_iter = 100,␣
 ↪random_state = 20, eta0 = 0.01, verbose = 1):
        self.X_train = X_train
        self.Y_train = Y_train
        self.X_test = X_test
        self.Y_test = Y_test
        self.max_iter = max_iter
        self.random_state = random_state
        self.eta0 = eta0
        self.verbose = verbose

    def metrics(self, true, pred):
        tn, fp, fn, tp = cm(true, pred).ravel()
        acc = (tp + tn)/(tn + fp + fn + tp)
        prec = tp/(tp + fp)
        rec = tp / (tp + fn)
        f1 = 2*(rec * prec) / (rec + prec)
        return [acc, prec, rec, f1]

    def print_seq(self, score_list):
        print("%.6f" % score_list[0], "\n%.6f" % score_list[1], "\n%.6f" %␣
↪score_list[2], "\n%.6f" % score_list[3])

    def perceptron_model(self):
        percept = Perceptron(max_iter = self.max_iter, random_state = self.
↪random_state, eta0 = self.eta0, verbose=self.verbose)

        print("Fitting the Model...")
        percept.fit(self.X_train, self.Y_train)
        return percept

    def evaluation(self):
        percept = self.perceptron_model()

        print("Evaluating the model on training dataset..")
        Y_train_pred = percept.predict(self.X_train)

        train_score = self.metrics(self.Y_train, Y_train_pred)

        print("Evaluating the model on testing dataset...")
        Y_test_pred = percept.predict(self.X_test)
        test_score = self.metrics(self.Y_test, Y_test_pred)

        print("Training Score")
        self.print_seq(train_score)

        print("Testing Score")
        self.print_seq(test_score)

        return test_score
```

Perceptron class Contructor takes dataset, number of epochs, random_state, learning rate and verbose as argument This class is used for training preceptron for different model type and mode of classification

Methods: metrics: args: true, pred {true means true label and pred means predicted label by perceptron} Objective: To calculate accuracy, precision, recall and F-score with the help of confusion matrix Return: list of metrics

print_seq: args: list of metrics Objective: printing the score Return: None

preceptron_model: args: None Objective: intitalizing preceptron model and fitting the model on training dataset Return: preceptron model

Evaluation: args: None Objective: predicting the preceptron model on testing dataset and calulating the performance of the trained perceptron model Return: Scores

[6]:
```python
class SVM(object):

    def __init__(self, X_train, Y_train, X_test, Y_test, max_iter = 500):
        self.X_train = X_train
        self.Y_train = Y_train
        self.X_test = X_test
        self.Y_test = Y_test
        self.max_iter = max_iter

    def intitalize_model(self):
        # Linear SVM
        svc = SVC(max_iter = self.max_iter)

        print("Fitting the SVM")
        svc_model = svc.fit(self.X_train, self.Y_train)
        return svc_model

    def print_seq(self, score_list):
        print("%.6f" % score_list[0], "\n%.6f" % score_list[1], "\n%.6f" %␣
 ↪score_list[2], "\n%.6f" % score_list[3])

    def metrics(self, true, pred):
        tn, fp, fn, tp = cm(true, pred).ravel()
        acc = (tp + tn)/(tn + fp + fn + tp)
        prec = tp/(tp + fp)
        rec = tp / (tp + fn)
        f1 = 2*(rec * prec) / (rec + prec)
        return [acc, prec, rec, f1]

    def evaluation(self):
        svc_model = self.intitalize_model()

        print("Evaluating the model on training dataset..")
        Y_train_pred = svc_model.predict(self.X_train)
        train_score = self.metrics(self.Y_train, Y_train_pred)

        print("Evaluating the model on testing dataset...")
        Y_test_pred = svc_model.predict(self.X_test)
        test_score = self.metrics(self.Y_test, Y_test_pred)

        print("Training Score")
        self.print_seq(train_score)
        print("Testing Score")
        self.print_seq(test_score)

        return test_score
```

SVM class Contructor takes dataset and number of epochs as argument This class is used for training Linear SVM for different model type and mode of classification

Methods: metrics: args: true, pred {true means true label and pred means predicted label by perceptron} Objective: To calculate accuracy, precision, recall and F-score with the help of confusion matrix Return: list of metrics

print_seq: args: list of metrics Objective: printing the score Return: None

initialize_model: args: None Objective: intitalizing SVM model and fitting the model on training dataset Return: svm model

Evaluation: args: None Objective: predicting the svm model on testing dataset and calulating the performance of the trained svm model Return: Scores

[7]:
```
class MLP(nn.Module):
    def __init__(self, classification = "binary", vocab_size = 300):
        super(MLP, self).__init__()
        hidden_1 = 50
        hidden_2 = 10
        if classification == "binary":
            self.fc3 = nn.Linear(hidden_2, 3)
        else:
            # For multi-classification
            self.fc3 = nn.Linear(hidden_2, 4)
        self.fc1 = nn.Linear(vocab_size, hidden_1)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        self.sig = nn.Sigmoid()
        self.soft = nn.Softmax(dim = 1)


    def forward(self, x):
        x = x.view(-1, x.shape[1])
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

This FNN class is for simple mean model Constructor takes classification, vocab_size as argument The objective of this class is to generate a Feed Forward Neural Network with 50 neurons as first hidden layers ans 10 neurons as second hidden layer (given in the question)

[8]:
```
class MLP_vec(nn.Module):
    def __init__(self, classification = "binary", vocab_size = 300):
        super(MLP_vec, self).__init__()
        hidden_1 = 50
        hidden_2 = 10
        if classification == "binary":
            self.fc3 = nn.Linear(hidden_2, 3)
        else:
            # For multi-classification
            self.fc3 = nn.Linear(hidden_2, 4)
        self.prod = 10
        self.fc1 = nn.Linear(vocab_size * self.prod, hidden_1)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        self.sig = nn.Sigmoid()
        self.soft = nn.Softmax(dim = 1)


    def forward(self, x):
        x = x.view(-1, x.shape[1] * x.shape[2])
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

This FNN class is for vector model like FNN part(b) where we need 10 x 300 feature vectors Constructor takes classification, vocab_size as argument Similar Objective as the simple mean FNN class.

[9]:
```
class trainData(Dataset):

    def __init__(self, X_data, y_data):
        self.X_data = X_data
        self.y_data = y_data

    def __getitem__(self, index):
        return self.X_data[index], self.y_data[index]

    def __len__(self):
        return len(self.X_data)


class testData(Dataset):

    def __init__(self, X_data, Y_data):
        self.X_data = X_data
        self.Y_data = Y_data

    def __getitem__(self, index):
        return self.X_data[index], self.Y_data[index]
    def __len__ (self):
        return len(self.X_data)
```

DataLoader for Feed Forward Neural Network

```
[10]: class RNN_Data(Dataset):

          def __init__(self, X_data, Y_data):

              self.X_data = X_data
              self.Y_data = Y_data

          def __len__(self):

              return len(self.X_data)

          def __getitem__(self, index):
              pad = np.zeros((50, 300), dtype = float)
              pad[-len(self.X_data[index]):] = np.array(self.X_data[index])[:50]
              X = torch.FloatTensor(pad)
              Y = torch.tensor(self.Y_data[index])
              #print(Y, " <=> ", self.Y_data[index], ' <=> ', index)

              return X, Y
```

DataLoader for RNN and GRU models

```
[11]: class Model(nn.Module):
          def __init__(self, input_size, output_size, hidden_dim, n_layers,␣
       ↪model_type = "rnn"):
              super(Model, self).__init__()

              # Defining some parameters
              self.hidden_dim = hidden_dim
              self.n_layers = n_layers
              self.model_type = model_type

              #Defining the layers
              if self.model_type == "gru":
                  self.layer = nn.GRU(input_size, hidden_dim, n_layers,␣
       ↪batch_first=True)
              else:
                  self.layer = nn.RNN(input_size, hidden_dim, n_layers,␣
       ↪batch_first=True)
              # Fully connected layer
              self.fc = nn.Linear(2500, output_size)

          def forward(self, x):
```

```
        batch_size = x.size(0)

        hidden = self.init_hidden(batch_size)

        out, hidden = self.layer(x, hidden)

        out = out.contiguous().view(-1, out.shape[1] * out.shape[2])
        out = self.fc(out)

        return out, hidden

    def init_hidden(self, batch_size):
        hidden = torch.zeros(self.n_layers, batch_size, self.hidden_dim).cuda()
        return hidden
```

Model class for generating models for RNN and GRU based on which model_type we want to train Arguments: input_size: 300 vectors output_size: depends upon classification type (binary classification or multiclass) hidden_dim: 50 hidden dimensions n_layers: number of RNN or GRU layers you want to add to the network model_type: States which model (RNN or GRU) we want to intitalize

[13]:
```
""" READING THE DATA FILES"""
print("Reading datafile...")

filename = "./amazon_reviews_us_Kitchen_v1_00.tsv"
dt = DataTranformation(filename, True)

print("Data Preprocessing Initiated...")
train, test = dt.train_test_split()
print("Train-Test Split Completed...")

print("Dataset Iterator formation...")
sentences = Sentence(train['review_body'])
```

```
Reading datafile…
Preproces: True
Data Preprocessing Initiated…
Train-Test Split Completed…
Dataset Iterator formation…
```

[14]:
```
# Pretrained Model
print("Generating Pretrained Model...")
pretrained_model = api.load('word2vec-google-news-300')

# Calculating Pretrained Model's perforamance on given examples
print(pretrained_model.most_similar(positive=['woman', 'king'],␣
↪negative=['man'], topn=1))
```

```
print(pretrained_model.similarity('excellent', 'outstanding'))
```

```
Generating Pretrained Model…
[('queen', 0.7118193507194519)]
0.5567486
```

[15]:
```
# General Model
print("General Model Generation...")
model = gensim.models.Word2Vec(sentences, vector_size = 300, min_count = 10,␣
 ↪window = 11, seed = 200)

# Calculating General Model's performance on given examples
print(model.wv.most_similar(positive=['woman', 'king'], negative=['man'],␣
 ↪topn=1))
print(model.wv.similarity('excellent', 'outstanding'))
```

```
General Model Generation…
[('arthur', 0.47155138850212097)]
0.79131085
```

[15]:
```
# Creating an instance of vectorizer to extract features
""" MEAN FEATURES EXTRACTION """
vec_train = Vectorization(model = model, dataset = train)
vec_test = Vectorization(model, test)

X_train_model, Y_train_model = vec_train.feature_extraction()
X_test_model, Y_test_model = vec_test.feature_extraction()

""" TO BE USED FOR PERCEPTRON, SVM AND FNN """
```

```
Vectorizing training dataset…
Model Type: model
Classification: binary
Vectorizing training dataset…
Model Type: model
Classification: binary
Vectorization Completed
Vectorization Completed
```

[15]:
```
' TO BE USED FOR PERCEPTRON, SVM AND FNN '
```

[16]:
```
""" MEAN MULTI-CLASS FEATURES EXTRACTION """
vec_multi_train = Vectorization(model, train, classification = "multi-class")
vec_multi_test = Vectorization(model, test, classification = "multi-class")

X_train_multi, Y_train_multi = vec_multi_train.feature_extraction()
X_test_multi, Y_test_multi = vec_multi_test.feature_extraction()
```

```
""" TO BE USED FOR FNN """
```

```
Vectorizing training dataset…
Model Type: model
Classification: multi-class
Vectorizing training dataset…
Model Type: model
Classification: multi-class
Vectorization Completed
Vectorization Completed
```

[16]: ' TO BE USED FOR FNN '

[17]:
```
""" TEN FEATURES IN A SINGLE ROW FEATURE EXTRACTION """
vec_mode_train = Vectorization(model, train, classification="binary",␣
 ↪mode="vec")
vec_mode_test = Vectorization(model, test, classification="binary", mode="vec")

X_train_mode, Y_train_mode = vec_mode_train.feature_extraction()
X_test_mode, Y_test_mode = vec_mode_test.feature_extraction()

""" TO BE USED FOR FNN """
```

```
Vectorizing training dataset…
Model Type: model
Classification: binary
Vectorizing training dataset…
Model Type: model
Classification: binary
Vectorization Completed
Vectorization Completed
```

[17]: ' TO BE USED FOR FNN '

[18]:
```
""" TEN FEATURES IN A SINGLE ROW MULTI-CLASS FEATURES EXTRACTION """
vec_mode_train_multi = Vectorization(model, train,␣
 ↪classification="multi-class", mode="vec")
vec_mode_test_multi = Vectorization(model, test, classification="multi-class",␣
 ↪mode="vec")

X_train_mode_multi, Y_train_mode_multi = vec_mode_train_multi.
 ↪feature_extraction()
X_test_mode_multi, Y_test_mode_multi = vec_mode_test_multi.feature_extraction()

""" TO BE USED FOR FNN """
```

```
Vectorizing training dataset…
Model Type: model
Classification: multi-class
Vectorizing training dataset…
Model Type: model
Classification: multi-class
Vectorization Completed
Vectorization Completed
```

```
[18]: ' TO BE USED FOR FNN '
```

```
[19]: """PRETRAINED MODEL FEATURES EXTRACTION"""
vec2_train = Vectorization(model = pretrained_model, dataset = train,␣
 ↪model_type = "pretrained")
vec2_test = Vectorization(model = pretrained_model, dataset = test, model_type␣
 ↪= "pretrained")

X_train_pre, Y_train_pre = vec2_train.feature_extraction()
X_test_pre, Y_test_pre = vec2_test.feature_extraction()

""" TO BE USED FOR PERCEPTRON AND SVM """
```

```
Vectorizing training dataset…
Model Type: pretrained
Classification: binary
Vectorizing training dataset…
Model Type: pretrained
Classification: binary
Vectorization Completed
Vectorization Completed
```

```
[19]: ' TO BE USED FOR PERCEPTRON AND SVM '
```

```
[23]: """ PRETRAINED MODEL MULTI-CLASS FEATURES EXTRACTION """

vec2_multi_train = Vectorization(model = pretrained_model, dataset = train,␣
 ↪classification = "multi-class", model_type="pretrained")
vec2_multi_test = Vectorization(model = pretrained_model, dataset = test,␣
 ↪classification = "multi-class", model_type = "pretrained")

X_train_multi_pre, Y_train_multi_pre = vec2_multi_train.feature_extraction()
X_test_multi_pre, Y_test_multi_pre = vec2_multi_test.feature_extraction()

""" TO BE USED FOR FNN """
```

```
Vectorizing training dataset…
Model Type: pretrained
Classification: multi-class
```

```
Vectorizing training dataset…
Model Type: pretrained
Classification: multi-class
Vectorization Completed
Vectorization Completed
```

```
[23]: ' TO BE USED FOR FNN '
```

```
[20]: """ PRETRAINED MODE VEC BINARY FEATURES EXTRACTION """

vec_mode_train_pre = Vectorization(model = pretrained_model, dataset = train,␣
 ↪model_type="pretrained", mode="vec")
vec_mode_test_pre = Vectorization(model = pretrained_model, dataset = test,␣
 ↪model_type="pretrained", mode = "vec")

X_train_mode_pre, Y_train_mode_pre = vec_mode_train_pre.feature_extraction()
X_test_mode_pre, Y_test_mode_pre = vec_mode_test_pre.feature_extraction()

""" TO BE USED FOR FNN """
```

```
Vectorizing training dataset…
Model Type: pretrained
Classification: binary
Vectorizing training dataset…
Model Type: pretrained
Classification: binary
Vectorization Completed
Vectorization Completed
```

```
[20]: ' TO BE USED FOR FNN '
```

```
[21]: """ PRETRAINED MDOE VEC MULTI-CLASS FEATURES EXTRACTION """

vec_mode_train_multi_pre = Vectorization(model = pretrained_model, dataset =␣
 ↪train, classification = "multi-class", model_type="pretrained", mode = "vec")
vec_mode_test_multi_pre = Vectorization(model = pretrained_model, dataset =␣
 ↪test, classification = "multi-class", model_type = "pretrained", mode =␣
 ↪"vec")

X_train_mode_multi_pre, Y_train_mode_multi_pre = vec_mode_train_multi_pre.
 ↪feature_extraction()
X_test_mode_multi_pre, Y_test_mode_multi_pre = vec_mode_test_multi_pre.
 ↪feature_extraction()

""" TO BE USED FOR FNN """
```

```
Vectorizing training dataset…
Model Type: pretrained
```

```
Classification: multi-class
Vectorizing training dataset…
Model Type: pretrained
Classification: multi-class
Vectorization Completed
Vectorization Completed
```

[21]:
```
' TO BE USED FOR FNN '
```

[22]:
```
"""" TFIDF FEATURES EXTRACTION """
    # TFIDF
def get_tfidf(train, test):
    train_x = train.apply(lambda x: " ".join(ele for ele in x))
    test_x = test.apply(lambda x: " ".join(ele for ele in x))
    tfidf_vect = TfidfVectorizer(min_df = 0.001)
    train_x_vectors = tfidf_vect.fit_transform(train_x)
    train_x_vectors = pd.DataFrame(train_x_vectors.toarray(), columns =␣
 ↪tfidf_vect.get_feature_names())
    test_x_vectors = tfidf_vect.transform(test_x)
    test_x_vectors = pd.DataFrame(test_x_vectors.toarray(), columns =␣
 ↪tfidf_vect.get_feature_names())
    return train_x_vectors, test_x_vectors

train_tfidf = train[train.label != 3].reset_index(drop = True)
test_tfidf = test[test.label != 3].reset_index(drop = True)
X_train_tfidf, X_test_tfidf = get_tfidf(train_tfidf.review_body, test_tfidf.
 ↪review_body)
Y_train_tfidf = train_tfidf['label']
Y_test_tfidf = test_tfidf['label']

""" TO BE USED FOR PERCEPTRON AND SVM """
```

[22]:
```
' TO BE USED FOR PERCEPTRON AND SVM '
```

[24]:
```
# PERCEPTRON

per = Percept(X_train = X_train_model, Y_train = Y_train_model, X_test =␣
 ↪X_test_model, Y_test = Y_test_model)
model_test_score = per.evaluation()

per2 = Percept(X_train = X_train_pre, Y_train = Y_train_pre, X_test =␣
 ↪X_test_pre, Y_test = Y_test_pre)
model_pre_test_score = per2.evaluation()

per3 = Percept(X_train = X_train_tfidf, Y_train = Y_train_tfidf, X_test =␣
 ↪X_test_tfidf, Y_test = Y_test_tfidf)
model_tfidf_test_score = per3.evaluation()
```

```
Fitting the Model…
-- Epoch 1
Norm: 1.29, NNZs: 300, Bias: 0.000000, T: 159848, Avg. loss: 0.036654
Total training time: 0.10 seconds.
-- Epoch 2
Norm: 1.36, NNZs: 300, Bias: -0.050000, T: 319696, Avg. loss: 0.036680
Total training time: 0.21 seconds.
-- Epoch 3
Norm: 1.44, NNZs: 300, Bias: 0.010000, T: 479544, Avg. loss: 0.036638
Total training time: 0.32 seconds.
-- Epoch 4
Norm: 1.51, NNZs: 300, Bias: -0.070000, T: 639392, Avg. loss: 0.036496
Total training time: 0.41 seconds.
-- Epoch 5
Norm: 1.54, NNZs: 300, Bias: -0.030000, T: 799240, Avg. loss: 0.036516
Total training time: 0.50 seconds.
-- Epoch 6
Norm: 1.52, NNZs: 300, Bias: -0.020000, T: 959088, Avg. loss: 0.036951
Total training time: 0.58 seconds.
Convergence after 6 epochs took 0.59 seconds
Evaluating the model on training dataset..
Evaluating the model on testing dataset…
Training Score
0.819854
0.845115
0.782099
0.812387
Testing Score
0.816328
0.846522
0.778234
0.810943
Fitting the Model…
-- Epoch 1
Norm: 0.38, NNZs: 300, Bias: -0.010000, T: 159831, Avg. loss: 0.002503
Total training time: 0.08 seconds.
-- Epoch 2
Norm: 0.38, NNZs: 300, Bias: -0.020000, T: 319662, Avg. loss: 0.002502
Total training time: 0.16 seconds.
-- Epoch 3
Norm: 0.41, NNZs: 300, Bias: -0.020000, T: 479493, Avg. loss: 0.002507
Total training time: 0.24 seconds.
-- Epoch 4
Norm: 0.41, NNZs: 300, Bias: -0.020000, T: 639324, Avg. loss: 0.002511
Total training time: 0.33 seconds.
-- Epoch 5
Norm: 0.39, NNZs: 300, Bias: -0.020000, T: 799155, Avg. loss: 0.002505
Total training time: 0.41 seconds.
```

```
-- Epoch 6
Norm: 0.40, NNZs: 300, Bias: -0.020000, T: 958986, Avg. loss: 0.002495
Total training time: 0.49 seconds.
Convergence after 6 epochs took 0.49 seconds
Evaluating the model on training dataset..
Evaluating the model on testing dataset…
Training Score
0.763225
0.690646
0.951428
0.800329
Testing Score
0.767361
0.698484
0.950958
0.805398
Fitting the Model…
-- Epoch 1
Norm: 0.76, NNZs: 3104, Bias: 0.000000, T: 160035, Avg. loss: 0.001609
Total training time: 0.49 seconds.
-- Epoch 2
Norm: 0.83, NNZs: 3104, Bias: -0.010000, T: 320070, Avg. loss: 0.001623
Total training time: 0.96 seconds.
-- Epoch 3
Norm: 0.85, NNZs: 3104, Bias: 0.000000, T: 480105, Avg. loss: 0.001630
Total training time: 1.43 seconds.
-- Epoch 4
Norm: 0.85, NNZs: 3104, Bias: 0.000000, T: 640140, Avg. loss: 0.001640
Total training time: 1.91 seconds.
-- Epoch 5
Norm: 0.86, NNZs: 3104, Bias: -0.010000, T: 800175, Avg. loss: 0.001629
Total training time: 2.39 seconds.
-- Epoch 6
Norm: 0.87, NNZs: 3104, Bias: -0.010000, T: 960210, Avg. loss: 0.001616
Total training time: 2.87 seconds.
Convergence after 6 epochs took 2.87 seconds
Evaluating the model on training dataset..
Evaluating the model on testing dataset…
Training Score
0.783854
0.956979
0.593071
0.732307
Testing Score
0.778406
0.954277
0.590347
0.729439
```

[25]:
```
# SVM
svm = SVM(X_train = X_train_model, Y_train = Y_train_model, X_test =␣
 ↪X_test_model, Y_test = Y_test_model)
svm_model_test_score = svm.evaluation()

svm2 = SVM(X_train = X_train_pre, Y_train = Y_train_pre, X_test = X_test_pre,␣
 ↪Y_test = Y_test_pre)
svm_pre_test_score = svm2.evaluation()

svm3 = SVM(X_train = X_train_tfidf, Y_train = Y_train_tfidf, X_test =␣
 ↪X_test_tfidf, Y_test = Y_test_tfidf)
svm_tfidf_test_score = svm3.evaluation()
```

```
Fitting the SVM
Evaluating the model on training dataset..
Evaluating the model on testing dataset…
Training Score
0.867518
0.860909
0.875845
0.868313
Testing Score
0.868509
0.866467
0.875087
0.870756
Fitting the SVM
Evaluating the model on training dataset..
Evaluating the model on testing dataset…
Training Score
0.829827
0.812739
0.856039
0.833827
Testing Score
0.830619
0.818565
0.854902
0.836339
Fitting the SVM
Evaluating the model on training dataset..
Evaluating the model on testing dataset…
Training Score
0.892099
0.888155
0.896438
0.892277
```

```
Testing Score
0.886275
0.885772
0.890021
0.887891
```

[26]:
```
""" CLEANING TO FREE SOME MEMORY """

del per3, svm3, model_tfidf_test_score, svm_tfidf_test_score,\
    train_tfidf, test_tfidf, X_train_tfidf, Y_train_tfidf, X_test_tfidf,\
    Y_test_tfidf
```

Observations: Perceptron: Trained Word2Vec Model's Accuracy -> 0.816 Pretrained Word2Vec Model's Accuracy -> 0.767 TFIDF Model's Accuracy -> 0.778 Conclusion: Trained Word2Vec Model performed better compare to pretrained and TFIDF models

```
SVM: Trained Word2Vec Model's Accuracy -> 0.868
     Pretrained Word2Vec Model's Accuracy ->0.830
     TFIDF Model's Accuracy -> 0.886
Conclusion: TFIDF model performed better compare to other models. Our trained model's F-score i
            TFIDF model.
```

[35]:
```
""" TRAINING FUNCTION """

def training(model, epoch, dataset_x, dataset_y, name = "model"):

    device = torch.device('cuda')
    print(model)

    model = model.to(device)

    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(mlp_model.parameters(), lr=0.01)

    criterion = criterion.to(device)

    training_data = trainData(torch.FloatTensor(dataset_x), torch.
 ↪LongTensor(dataset_y))
    train_loader = DataLoader(dataset = training_data, batch_size=16, shuffle =␣
 ↪True)

    for epoch in range(epoch):

        train_loss = 0.0

        mlp_model.train()
        for input_data, label in train_loader:
            optimizer.zero_grad()
            output = mlp_model(input_data.to(device))
            loss = criterion(output, label.to(device)) #y_batch.unsqueeze(1)␣
↪(label.unsqueeze(1)).to(device)
            loss.backward()
            optimizer.step()
            train_loss += loss.item() * input_data.size(1)

        train_loss = train_loss/len(train_loader.dataset)

        #print('Epoch: {} \tTraining Loss: {:.6f}'.format(epoch+1, train_loss))
        torch.save(mlp_model.state_dict(), name + str(epoch + 1) + '.pt')

""" TESTING FUNCTION """

def testing(model, epoch, dataset_x, dataset_y, name = "model"):
    max_acc = 0
    device = torch.device('cpu')
    testing_data = testData(torch.FloatTensor(dataset_x), torch.
↪LongTensor(dataset_y))
    test_loader = DataLoader(dataset = testing_data, batch_size=16)

    for i in range(1, epoch + 1):

        model.load_state_dict(torch.load(name +str(i) + '.pt'))
        model = model.to(device)

        predictions, actual = list(), list()
        for test_data, test_label in test_loader:

            pred = mlp_model(test_data.to(device))
            pred = pred.detach().numpy()
            pred = argmax(pred, axis= 1)
            target = test_label.numpy()
            target = target.reshape((len(target), 1))
            pred = pred.reshape((len(pred)), 1)
            pred = pred.round()
            predictions.append(pred)
            actual.append(target)

        predictions, actual = vstack(predictions), vstack(actual)
        acc = accuracy_score(actual, predictions)
        max_acc = max(max_acc, acc)
    print('Accuracy: %.3f' % max_acc)
```

[20]:
```
# MLP
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

[36]:
```
""" BINARY-MEAN MLP """

mlp_model = MLP() # binary classification
training(mlp_model, 10, X_train_model, Y_train_model, name = "mlp_model")
testing(mlp_model, 10, X_test_model, Y_test_model, name = "mlp_model")
```

```
MLP(
  (fc3): Linear(in_features=10, out_features=3, bias=True)
  (fc1): Linear(in_features=300, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (sig): Sigmoid()
  (soft): Softmax(dim=1)
)
Accuracy: 0.884
```

[37]:
```
""" MULTI-CLASS MEAN MLP """

mlp_model = MLP(classification = "multi-class")
training(mlp_model, 10, X_train_multi, Y_train_multi, name = "mlp_model_multi")
testing(mlp_model, 10, X_test_multi, Y_test_multi, name = "mlp_model_multi")
```

```
MLP(
  (fc3): Linear(in_features=10, out_features=4, bias=True)
  (fc1): Linear(in_features=300, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (sig): Sigmoid()
  (soft): Softmax(dim=1)
)
Accuracy: 0.721
```

[38]:
```
""" BINARY-VEC MLP """

mlp_model = MLP_vec()
training(mlp_model, 10, X_train_mode, Y_train_mode, name = "mlp_mode_vec")
testing(mlp_model, 10, X_test_mode, Y_test_mode, name = "mlp_mode_vec")
```

```
MLP_vec(
  (fc3): Linear(in_features=10, out_features=3, bias=True)
  (fc1): Linear(in_features=3000, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (sig): Sigmoid()
  (soft): Softmax(dim=1)
)
Accuracy: 0.770
```

[39]:
```
""" MULTI-VEC MLP """

mlp_model = MLP_vec(classification="multi-class")
```

```
training(mlp_model, 10, X_train_mode_multi, Y_train_mode_multi, name =␣
↪"mlp_mode_vec_multi")
testing(mlp_model, 10, X_test_mode_multi, Y_test_mode_multi, name =␣
↪"mlp_mode_vec_multi")
```

```
MLP_vec(
  (fc3): Linear(in_features=10, out_features=4, bias=True)
  (fc1): Linear(in_features=3000, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (sig): Sigmoid()
  (soft): Softmax(dim=1)
)
Accuracy: 0.617
```

[40]:
```
""" BINARY-MEAN PRETRAINED MLP """
mlp_model = MLP()
training(mlp_model, 10, X_train_pre, Y_train_pre, name = "mlp_model_pre")
testing(mlp_model, 10, X_test_pre, Y_test_pre, name = "mlp_model_pre")
```

```
MLP(
  (fc3): Linear(in_features=10, out_features=3, bias=True)
  (fc1): Linear(in_features=300, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (sig): Sigmoid()
  (soft): Softmax(dim=1)
)
Accuracy: 0.838
```

[41]:
```
""" MULTI-CLASS MEAN PRETRAINED MLP """
mlp_model = MLP(classification = "multi-class")
training(mlp_model, 10, X_train_multi_pre, Y_train_multi_pre, name =␣
↪"mlp_mode_multi_pre")
testing(mlp_model, 10, X_test_multi_pre, Y_test_multi_pre, name =␣
↪"mlp_mode_multi_pre")
```

```
MLP(
  (fc3): Linear(in_features=10, out_features=4, bias=True)
  (fc1): Linear(in_features=300, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (sig): Sigmoid()
  (soft): Softmax(dim=1)
)
Accuracy: 0.679
```

[42]:
```
""" BINARY VEC PRETRAINED MLP """

mlp_model = MLP_vec()
```

```
training(mlp_model, 10, X_train_mode_pre, Y_train_mode_pre, name =␣
 ↪"mlp_vec_pre")
testing(mlp_model, 10, X_test_mode_pre, Y_test_mode_pre, name = "mlp_vec_pre")
```

```
MLP_vec(
  (fc3): Linear(in_features=10, out_features=3, bias=True)
  (fc1): Linear(in_features=3000, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (sig): Sigmoid()
  (soft): Softmax(dim=1)
)
Accuracy: 0.755
```

[43]:
```
""" MULTI-CLASS VEC PRETRAINED MLP """

mlp_model = MLP_vec(classification = "multi-class")
training(mlp_model, 10, X_train_mode_multi_pre, Y_train_mode_multi_pre, name =␣
 ↪"mlp_vec_multi_pre")
testing(mlp_model, 10, X_test_mode_multi_pre, Y_test_mode_multi_pre, name =␣
 ↪"mlp_vec_multi_pre")

del mlp_model, X_train_mode_multi_pre, Y_train_mode_multi_pre,␣
 ↪X_test_mode_multi_pre, Y_test_mode_multi_pre,\
    X_train_mode_pre, Y_train_mode_pre, X_test_mode_pre, Y_test_mode_pre,␣
 ↪X_train_multi_pre, Y_train_multi_pre,\
    X_test_multi_pre, Y_test_multi_pre, X_train_pre, Y_train_pre, X_test_pre,␣
 ↪Y_test_pre, X_train_mode_multi,\
    Y_train_mode_multi, X_test_mode_multi, Y_test_mode_multi, X_train_mode,␣
 ↪Y_train_mode, X_test_mode, Y_test_mode,\
    X_train_multi, Y_train_multi, X_test_multi, Y_test_multi, X_train_model,␣
 ↪Y_train_model, X_test_model, Y_test_model

del vec_mode_train_multi_pre, vec_mode_test_multi_pre, vec_mode_train_pre,␣
 ↪vec_mode_test_pre, vec2_multi_train,\
    vec2_multi_test, vec2_train, vec2_test, vec_mode_train_multi,␣
 ↪vec_mode_test_multi, vec_mode_train, vec_mode_test,\
    vec_multi_train, vec_multi_test, vec_train, vec_test
```

```
MLP_vec(
  (fc3): Linear(in_features=10, out_features=4, bias=True)
  (fc1): Linear(in_features=3000, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (sig): Sigmoid()
  (soft): Softmax(dim=1)
)
Accuracy: 0.608
```

Observations: BINARY-MEAN FNN -> 0.884 MULTI-CLASS MEAN FNN -> 0.721 BINARY-VEC FNN -> 0.770 MULTI-VEC FNN -> 0.617

```
BINARY-MEAN PRETRAINED FNN -> 0.838
MULTI-CLASS MEAN PRETRAINED FNN -> 0.679
BINARY VEC PRETRAINED FNN -> 0.755
MULTI-CLASS VEC PRETRAINED FNN -> 0.608
```

Conclusion: As we can see here In all the cases our trained model ouperforms the pretrained model.

[22]:
```
def my_collate(batch):
    data = [item[0] for item in batch]
    target = [item[1] for item in batch]
    return data, target


def rnn_train(model, epoch, dataset_x, dataset_y, name):

    rnn_train = RNN_Data(dataset_x, dataset_y)
    train_loader_mode = DataLoader(dataset = rnn_train, batch_size=8, shuffle =␣
↪True, collate_fn=my_collate, drop_last=True)

    criterion = nn.CrossEntropyLoss()
    criterion = criterion.to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)

    for ep in range(1, epoch + 1):

        for input_data, label in train_loader_mode:
            optimizer.zero_grad()
            input_data = torch.stack(input_data)
            label = torch.stack(label)
            output, hidden = model(input_data.to(device))
            loss = criterion(output, label.to(device))
            loss.backward()
            optimizer.step()

        #print('Epoch: {} \tTraining Loss: {:.6f}'.format(ep, loss.item()))
        torch.save(model.state_dict(), name + str(ep) + '.pt')

def rnn_test(model, epoch, dataset_x, dataset_y, name):
    max_acc = 0
    rnn_test = RNN_Data(dataset_x, dataset_y)
    test_loader_mode = DataLoader(dataset = rnn_test, batch_size=8,␣
↪collate_fn=my_collate, drop_last=True)

    for i in range(1, epoch + 1):
        model.load_state_dict(torch.load(name +str(i) + '.pt'))
        model = model.to(device)

        predictions, actual = list(), list()
        for test_data, test_label in test_loader_mode:
            test_data = torch.stack(test_data)
            test_label = torch.stack(test_label)
            pred, hid = model(test_data.to(device))
            pred = pred.to('cpu')
            pred = pred.detach().numpy()
            pred = argmax(pred, axis= 1)
            target = test_label.numpy()
            target = target.reshape((len(target), 1))
            pred = pred.reshape((len(pred)), 1)
            pred = pred.round()
            predictions.append(pred)
            actual.append(target)

        predictions, actual = vstack(predictions), vstack(actual)
        acc = accuracy_score(actual, predictions)
        max_acc = max(max_acc, acc)
    print('Accuracy: %.3f' % max_acc)
```

Collate Function: a custom batch loader for DataLoader. It is used if the size of the dataset is volatile rnn_train and rnn_test are training and testing functions. Both functions take model, dataset, number of epochs and name as arguments. These both functions are being used to train and test both RNN and GRU models.

[48]:
```
"""
    RNN DATASET PREPARATION
    SAME FOR GRU...
"""
rnn_bin = Model(300, 3, 50, 1)
rnn_bin = rnn_bin.to(device)
gru_model_bin = Model(300, 3, 50, 1, model_type="gru")
gru_model_bin = gru_model_bin.to(device)

vec_rnn_train = Vectorization(model, train, classification = "binary", pad =␣
 ↪True)
vec_rnn_test = Vectorization(model, test, classification ="binary", pad = True)

X_rnn_train, Y_rnn_train = vec_rnn_train.feature_extraction()
X_rnn_test, Y_rnn_test = vec_rnn_test.feature_extraction()


rnn_train(rnn_bin, 10, X_rnn_train, Y_rnn_train, name = "rnn_model")
rnn_test(rnn_bin, 10, X_rnn_test, Y_rnn_test, name = "rnn_model")
```

```
rnn_train(gru_model_bin, 10, X_rnn_train, Y_rnn_train, name = "gru_model")
rnn_test(gru_model_bin, 10, X_rnn_test, Y_rnn_test, name = "gru_model")

del vec_rnn_train, vec_rnn_test, X_rnn_train, X_rnn_test, Y_rnn_train,␣
 ↪Y_rnn_test
```

```
Accuracy: 0.766
Accuracy: 0.770
```

[49]:
```
rnn = Model(300, 4, 50, 1)
rnn = rnn.to(device)
vec_rnn_multi_train = Vectorization(model, train, classification =␣
 ↪"multi-class", pad = True)
vec_rnn_multi_test = Vectorization(model, test, classification = "multi-class",␣
 ↪pad = True)

X_rnn_multi_train, Y_rnn_multi_train = vec_rnn_multi_train.feature_extraction()
X_rnn_multi_test, Y_rnn_multi_test = vec_rnn_multi_test.feature_extraction()

rnn_train(rnn, 10, X_rnn_multi_train, Y_rnn_multi_train, name =␣
 ↪"rnn_multi_model")
rnn_test(rnn, 10, X_rnn_multi_test, Y_rnn_multi_test, name = "rnn_multi_model")

gru_model = Model(300, 4, 50, 1, model_type="gru")
gru_model = gru_model.to(device)

rnn_train(gru_model, 10, X_rnn_multi_train, Y_rnn_multi_train, name =␣
 ↪"gru_multi_model")
rnn_test(gru_model, 10, X_rnn_multi_test, Y_rnn_multi_test, name =␣
 ↪"gru_multi_model")

del vec_rnn_multi_train, vec_rnn_multi_test, Y_rnn_multi_train,␣
 ↪X_rnn_multi_test, Y_rnn_multi_test
```

```
Vectorizing training dataset…
Model Type: model
Classification: multi-class
Vectorizing training dataset…
Model Type: model
Classification: multi-class
Vectorization Completed
Vectorization Completed
Accuracy: 0.605
Accuracy: 0.628
```

```
[23]: vec_rnn_pre_train = Vectorization(model = pretrained_model, dataset = train,␣
       ↪model_type="pretrained", classification = "binary", mode = "vec", pad = True)
      vec_rnn_pre_test = Vectorization(model = pretrained_model, dataset = test,␣
       ↪model_type = "pretrained", classification = "binary", mode = "vec", pad =␣
       ↪True)

      X_rnn_pre_train, Y_rnn_pre_train = vec_rnn_pre_train.feature_extraction()
      X_rnn_pre_test, Y_rnn_pre_test = vec_rnn_pre_test.feature_extraction()

      rnn_bin = Model(300, 3, 50, 1)
      rnn_bin = rnn_bin.to(device)
      gru_model_bin = Model(300, 3, 50, 1, model_type="gru")
      gru_model_bin = gru_model_bin.to(device)

      rnn_train(rnn_bin, 10, X_rnn_pre_train, Y_rnn_pre_train, name =␣
       ↪"rnn_pre_model_bin")
      rnn_test(rnn_bin, 10, X_rnn_pre_test, Y_rnn_pre_test, name =␣
       ↪"rnn_pre_model_bin")

      rnn_train(gru_model_bin, 10, X_rnn_pre_train, Y_rnn_pre_train, name =␣
       ↪"gru_pre_model_bin")
      rnn_test(gru_model_bin, 10, X_rnn_pre_test, Y_rnn_pre_test, name =␣
       ↪"gru_pre_model_bin")

      del vec_rnn_pre_train, vec_rnn_pre_test,  X_rnn_pre_train, Y_rnn_pre_train,␣
       ↪X_rnn_pre_test, Y_rnn_pre_test
```

```
Accuracy: 0.820
Accuracy: 0.871
```

```
[26]: vec_rnn_pre_multi_train = Vectorization(model = pretrained_model, dataset =␣
       ↪train, model_type = "pretrained", classification = "multi-class", mode =␣
       ↪"vec", pad = True)
      vec_rnn_pre_multi_test = Vectorization(model = pretrained_model, dataset =␣
       ↪test, model_type = "pretrained", classification = "multi-class", mode =␣
       ↪"vec", pad = True)

      X_rnn_pre_multi_train, Y_rnn_pre_multi_train = vec_rnn_pre_multi_train.
       ↪feature_extraction()
      X_rnn_pre_multi_test, Y_rnn_pre_multi_test = vec_rnn_pre_multi_test.
       ↪feature_extraction()

      rnn = Model(300, 4, 50, 1)
      rnn = rnn.to(device)
      gru_model = Model(300, 4, 50, 1, model_type="gru")
      gru_model = gru_model.to(device)
```

```
rnn_train(rnn, 10, X_rnn_pre_multi_train, Y_rnn_pre_multi_train, name =␣
↪"rnn_pre_model")
rnn_test(rnn, 10, X_rnn_pre_multi_test, Y_rnn_pre_multi_test, name =␣
↪"rnn_pre_model")

rnn_train(gru_model, 10, X_rnn_pre_multi_train, Y_rnn_pre_multi_train, name =␣
↪"gru_pre_model")
rnn_test(gru_model, 10, X_rnn_pre_multi_test, Y_rnn_pre_multi_test, name =␣
↪"gru_pre_model")

del vec_rnn_pre_multi_train, vec_rnn_pre_multi_test,  X_rnn_pre_multi_train,␣
↪Y_rnn_pre_multi_train, X_rnn_pre_multi_test, Y_rnn_pre_multi_test
```

```
Accuracy: 0.672
Accuracy: 0.715
```

Observation: RNN: Binary model -> 0.766 (Our trained model) Multi-class -> 0.605 (Our trained model) Binary model -> 0.820 (pretrained model) Multi-class -> 0.871 (pretrained model)

```
GRU:
    Binary model -> 0.770 (Our Trained Model)
    Multi-class -> 0.628 (Our Trained Model)
    Binary model -> 0.672 (Pretrained Model)
    Multi-class -> 0.715 (Pretrained Model)
```