# hw2p1

October 5, 2021

### 0.0.1 HW2 - Part 1

[2]:

```
'''
Importing all libraries
'''
from copy import deepcopy
from numpy import argmax
import contractions
from bs4 import BeautifulSoup
import re
import pandas as pd
import numpy as np
import nltk
import torch
import gensim
import warnings
from sklearn.metrics import accuracy_score
from numpy import vstack
from torchvision import transforms, utils
from torch.utils.data import Dataset, DataLoader
import torch.optim as optim
import torch.nn.functional as F
import torch.nn as nn
import gensim.downloader as api
from sklearn.svm import LinearSVC as SVC
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import confusion_matrix as cm
from sklearn.linear_model import Perceptron
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords

nltk.download('wordnet')
warnings.filterwarnings('ignore')
CUDA_LAUNCH_BLOCKING = 1
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     /home/darkghost/nltk_data…
```

```
[nltk_data]   Package wordnet is already up-to-date!
```

### 0.0.2 Creating a class "DataTransformation" to manage preprocessing of data

Usage of functions:

1. read_file(): reads the tsv file and returns the dataframe
2. df_formation(): reads the dataframe and picks 50k reviews of each star rating and returns the final combined df
3. label() and apply_label(): To apply 1, 2 or 3 label to the reviews
4. remove_html_url(): removes the HTML and URL from the reviews
5. tokenize(): tokenizes the reviews
6. without_preprocess(): returns df without doing all preprocessing, just tokenized
7. with_preprocess(): returns preprocessed and tokenized reviews
8. train_test_split(): splits the df into 80%-20% train-test split

[4]:
```
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 df_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.df_formation()
        dataset['label'] = dataset.apply(lambda row: self.label(row), axis=1)

        return dataset

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

    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
```

### 0.0.3 Creating class Vectorization to generate feature vectors of the words based on the requirements

Functions are as follows:

1. get_mean_vector(): returns feature vector vlues for every word in the review
2. feature_extraction(): Either pads or takes first 10 vectors or calculate mean vector for full review
3. pad_review(): pads the reviews to the desired length
4. join_words(): list of words in converted to back to one sentence

[5]:
```
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)
```

```python
    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
```

Sentence class returns one review at a time from the dataset through the use of **iter**.

[6]:
```
class Sentence(object):
    def __init__(self, dataset):
        self.dataset = dataset
    def __iter__(self):
        for row in self.dataset:
            yield row
```

### 0.0.4 Class to train and evaluate the Perceptron

[23]:
```
class Percept(object):

    def __init__(self, X_train, Y_train, X_test, Y_test, max_iter=100,␣
↪random_state=20, eta0=0.01, verbose=0):
        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], " %.6f" % score_list[1],
              " %.6f" % score_list[2], " %.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()
        Y_train_pred = percept.predict(self.X_train)
        train_score = self.metrics(self.Y_train, Y_train_pred)
        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
```

### 0.0.5 Class to train and evaluate the SVM

[22]:

```
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], " %.6f" % score_list[1],
              " %.6f" % score_list[2], " %.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()
        Y_train_pred = svc_model.predict(self.X_train)
        train_score = self.metrics(self.Y_train, Y_train_pred)

        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
```

### 0.0.6 Reading the file and carrying out preprocessing

[11]:

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

```
Preproces: True
```

### 0.0.7 Splitting data and generating pretrained and self-trained word2vec models

[12]:

```
train, test = dt.train_test_split()

sentences = Sentence(train['review_body'])

pretrained_model = api.load('word2vec-google-news-300')
model = gensim.models.Word2Vec(sentences, vector_size = 300, min_count = 10,␣
 ↪window = 11, seed = 200)
```

#### Semantic similarities in pretrained model

[13]:

```
print(pretrained_model.most_similar(positive=['woman', 'king'],␣
 ↪negative=['man'], topn=1))
print(pretrained_model.similarity('excellent', 'outstanding'))
```

```
[('queen', 0.7118193507194519)]
0.5567486
```

#### Semantic Similarities in Self-Trained Model

[14]:

```
print(model.wv.most_similar(positive=['woman', 'king'], negative=['man'],␣
 ↪topn=1))
print(model.wv.similarity('excellent', 'outstanding'))
```

```
[('arthur', 0.536632239818573)]
0.7561389
```

From the obervation, it looks like finding most similar word works better in pretrained model an it works better, but similarities between two words in some cases are better in our self-trained model

### 0.0.8 Self-trained model feature extraction

[15]:

```
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()
```

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

### 0.0.9 Pre-trained model feature extraction

[16]:

```
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()
```

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

### 0.0.10 TF-IDF feature extraction

[17]:

```
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
```

[18]:

```
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']
```

### 0.0.11 Training Perceptron on all three types of feature vectors

[24]:

```
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…
Training Score
0.828443  0.793716  0.886345  0.837477
Testing Score
0.829329  0.798938  0.885727  0.840097
Fitting the Model…
Training Score
0.763225  0.690646  0.951428  0.800329
Testing Score
0.767361  0.698484  0.950958  0.805398
Fitting the Model…
Training Score
0.783854  0.956979  0.593071  0.732307
Testing Score
0.778406  0.954277  0.590347  0.729439
```

From the observation, perceptron with self-trained feature vector model performed the best in terms of accuracy on current dataset, whereas pretrained model and tf-idf one performed similar on the basis of accuracy.

The results may vary according to the number of iterations and learning rate.

### 0.0.12 Training SVM on three types of Feature-vectors

```
[25]: 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
Training Score
0.865810  0.852399  0.883987  0.867906
Testing Score
0.867006  0.858566  0.882659  0.870446
Fitting the SVM
Training Score
0.829827  0.812739  0.856039  0.833827
Testing Score
0.830619  0.818565  0.854902  0.836339
Fitting the SVM
Training Score
0.892099  0.888155  0.896438  0.892277
Testing Score
0.886275  0.885772  0.890021  0.887891
```

From the obervation, TF-IDF SVM performs the best in terms of accuracy, followed by Self-trained and Pretrained SVM. Results may vary according to type of kernel used and number of iterations.

### 0.0.13 For conclusion, Self-trained Word2Vec gives decent performance on average compared to other two feature extraction models for Perceptron and SVM.

# hw2p2

October 5, 2021

## 0.1 HW-2 Part-2 - consists of Q4 of hw2

[1]:

```
'''
Importing all libraries
'''
from copy import deepcopy
from numpy import argmax
import contractions
from bs4 import BeautifulSoup
import re
import pandas as pd
import numpy as np
import nltk
import torch
import gensim
import warnings
from sklearn.metrics import accuracy_score
from numpy import vstack
from torchvision import transforms, utils
from torch.utils.data import Dataset, DataLoader
import torch.optim as optim
import torch.nn.functional as F
import torch.nn as nn
import gensim.downloader as api
from sklearn.svm import LinearSVC as SVC
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import confusion_matrix as cm
from sklearn.linear_model import Perceptron
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords

nltk.download('wordnet')
warnings.filterwarnings('ignore')
CUDA_LAUNCH_BLOCKING = 1
```

```
/home/darkghost/anaconda3/envs/ml/lib/python3.7/site-
packages/gensim/similarities/__init__.py:15: UserWarning: The
```

```
gensim.similarities.levenshtein submodule is disabled, because the optional
Levenshtein package <https://pypi.org/project/python-Levenshtein/> is
unavailable. Install Levenhstein (e.g. `pip install python-Levenshtein`) to
suppress this warning.
  warnings.warn(msg)
[nltk_data] Downloading package wordnet to
[nltk_data]     /home/darkghost/nltk_data…
[nltk_data]   Package wordnet is already up-to-date!
```

### 0.1.1 Transformation class for pre-processing

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

        return dataset

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

    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
```

### 0.1.2 Vectorization Class for feature Extraction

[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
        self.classification = classification
        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

    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
```

[4]:

```
class Sentence(object):
    def __init__(self, dataset):
        self.dataset = dataset

    def __iter__(self):
        for row in self.dataset:
            yield row
```

### 0.1.3 Multi-layer Perceptron using average word2Vec similar to "Simple Models"

[5]:

```
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
```

### 0.1.4 Multi-layer Perceptron using first 10 Word2Vec features as input features

[6]:

```
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
```

**Data Loaders for Train Data and Test Data, which supplies reviews one by one from the batches to the model.**

[7]:

```
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)
```

[8]:

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

train, test = dt.train_test_split()

sentences = Sentence(train['review_body'])

pretrained_model = api.load('word2vec-google-news-300')
model = gensim.models.Word2Vec(
    sentences, vector_size=300, min_count=10, window=11, seed=200)
```

```
Preproces: True
```

Mean feature extraction

[9]:

```
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()
```

```
Vectorization Completed
```

MEAN MULTI-CLASS FEATURES EXTRACTION

[10]:

```
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()
```

```
Vectorization Completed
```

TEN FEATURES IN A SINGLE ROW FEATURE EXTRACTION

[11]:
```
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()
```

```
Vectorization Completed
```

TEN FEATURES IN A SINGLE ROW MULTI-CLASS FEATURES EXTRACTION

[12]:
```
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()
```

```
Vectorization Completed
```

PRETRAINED MODEL FEATURES EXTRACTION

[13]:
```
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()
```

```
Vectorization Completed
```

PRETRAINED MODEL MULTI-CLASS FEATURES EXTRACTION

[14]:
```
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()
```

```
Vectorization Completed
```

PRETRAINED MODE VEC BINARY FEATURES EXTRACTION

[15]:

```
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()
```

```
Vectorization Completed
```

PRETRAINED MDOE VEC MULTI-CLASS FEATURES EXTRACTION

[16]:

```
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()
```

```
Vectorization Completed
```

### 0.1.5 TRAINING FUNCTION

[17]:

```
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))
            # y_batch.unsqueeze(1) (label.unsqueeze(1)).to(device)
            loss = criterion(output, label.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')
```

### 0.1.6 TEST FUNCTION

[25]:

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

    device = torch.device('cpu')
    testing_data = testData(torch.FloatTensor(
        dataset_x), torch.LongTensor(dataset_y))
    test_loader = DataLoader(dataset=testing_data, batch_size=16)
    tmp = 0
    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)
        if acc > tmp:
            tmp = acc
    print('Accuracy: %.3f' % tmp)
```

[21]:

```
device = torch.device('cuda')
```

[31]:

```
""" 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")
```

```
Accuracy: 0.788
```

[32]:

```
""" 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")
```

```
Accuracy: 0.629
```

[33]:

```
""" 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")
```

```
Accuracy: 0.716
```

[34]:

```
""" 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")
```

```
Accuracy: 0.568
```

[35]:

```
""" 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")
```

```
Accuracy: 0.838
```

[36]:

```
""" 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")
```

```
Accuracy: 0.679
```

[37]:
```
""" 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")
```

```
Accuracy: 0.755
```

[38]:
```
""" 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")
```

```
Accuracy: 0.608
```

## 0.2 Observation:

MLP trained using self-trained Word2Vec feature vectors produces better accuracy compared to pre-trained ones.

# hw2p3

October 5, 2021

### 0.1 HW2 - Part 3: Q5 (RNN and GRU)

[1]:

```
'''
Importing all libraries
'''
from copy import deepcopy
from numpy import argmax
import contractions
from bs4 import BeautifulSoup
import re
import pandas as pd
import numpy as np
import nltk
import torch
import gensim
import warnings
from sklearn.metrics import accuracy_score
from numpy import vstack
from torchvision import transforms, utils
from torch.utils.data import Dataset, DataLoader
import torch.optim as optim
import torch.nn.functional as F
import torch.nn as nn
import gensim.downloader as api
from sklearn.svm import LinearSVC as SVC
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import confusion_matrix as cm
from sklearn.linear_model import Perceptron
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords

nltk.download('wordnet')
warnings.filterwarnings('ignore')
CUDA_LAUNCH_BLOCKING = 1
```

```
/home/darkghost/anaconda3/envs/ml/lib/python3.7/site-
packages/gensim/similarities/__init__.py:15: UserWarning: The
```

```
gensim.similarities.levenshtein submodule is disabled, because the optional
Levenshtein package <https://pypi.org/project/python-Levenshtein/> is
unavailable. Install Levenhstein (e.g. `pip install python-Levenshtein`) to
suppress this warning.
  warnings.warn(msg)
[nltk_data] Downloading package wordnet to
[nltk_data]     /home/darkghost/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):
        s = re.sub(
            r'(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]*)', '', s,␣
↪flags=re.MULTILINE)
        soup = BeautifulSoup(s, 'html.parser')
        s = soup.get_text()
        return s

    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
```

[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
        self.classification = classification
        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

    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 = []
        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
```

[4]:
```
class Sentence(object):
    def __init__(self, dataset):
        self.dataset = dataset

    def __iter__(self):
        for row in self.dataset:
            yield row
```

[5]:
```
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])
        return X, Y
```

[6]:

```
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

        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)
        # Initializing hidden state for first input using method defined below
        hidden = self.init_hidden(batch_size)
        # Passing in the input and hidden state into the model and obtaining␣
 ↪outputs
        out, hidden = self.layer(x, hidden)
        # Reshaping the outputs such that it can be fit into the fully␣
 ↪connected layer
        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
```

[7]:

```
filename = "./amazon_reviews_us_Kitchen_v1_00.tsv"
dt = DataTranformation(filename, True)
train, test = dt.train_test_split()
sentences = Sentence(train['review_body'])
```

```
Preproces: True
```

[8]:

```
pretrained_model = api.load('word2vec-google-news-300')
model = gensim.models.Word2Vec(
    sentences, vector_size=300, min_count=10, window=11, seed=200)
```

[9]:

```
def my_collate(batch):
    '''
     collate_fn is your callable/function that processes the batch you want to␣
 ↪return from your dataloader
    '''
    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):

    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)
    tmp = 0
    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('cuda'))
            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)
        print('Accuracy: %.3f' % acc)
```

```
[10]: device = torch.device("cuda")
```

RNN and GRU with binary and Self Trained Model

```
[11]: 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
```

```
Vectorization Completed
Accuracy: 0.757
Accuracy: 0.781
```

RNN and GRU with multi-classification self trained w2v model

[12]:
```
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()
print("RNN: ")
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)
print("GRU: ")
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
```

```
Vectorization Completed
RNN:
Accuracy: 0.581
GRU:
Accuracy: 0.601
```

RNN and GRU with binary and pre-trained w2v model

[16]:
```
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()

print("RNN:")
rnn_train(rnn_bin, 10, X_rnn_pre_train, Y_rnn_pre_train, name = "rnn_pre_model")
rnn_test(rnn_bin, 10, X_rnn_pre_test, Y_rnn_pre_test, name = "rnn_pre_model")

print("GRU: ")
```

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

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
```

```
Vectorization Completed
RNN:
Accuracy: 0.822
GRU:
Accuracy: 0.871
```

RNN and GRU with multi-class and Pretrained w2v model

[13]:

```
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()
print("RNN:")
rnn_train(rnn, 10, X_rnn_pre_multi_train, Y_rnn_pre_multi_train, name =␣
 ↪"rnn_pre_model_multi")
rnn_test(rnn, 10, X_rnn_pre_multi_test, Y_rnn_pre_multi_test, name =␣
 ↪"rnn_pre_model_multi")
print("GRU: ")
rnn_train(gru_model, 10, X_rnn_pre_multi_train, Y_rnn_pre_multi_train, name =␣
 ↪"gru_pre_model_multi")
rnn_test(gru_model, 10, X_rnn_pre_multi_test, Y_rnn_pre_multi_test, name =␣
 ↪"gru_pre_model_multi")

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
```

```
Vectorization Completed
RNN:
Accuracy: 0.702
GRU:
Accuracy: 0.738
```

## 0.2 Observation:

GRU gives better accuracy compare to RNN in all cases with this data.