DMT2023_HW4

June 8, 2023

0.1 Group composition:

————YOUR TEXT STARTS HERE————

Aur, Marina Iuliana, 1809715

Balestrucci, Sophia, 1713638

0.2 Homework 4

The homework consists of two parts:

1. Text Representation

and

2. Deep Learning

   Ensure that the notebook can be faithfully reproduced by anyone (hint: pseudo random number generation).

   If you need to set a random seed, set it to `709`.

   If multiple code cells are provided for a single part, it is **NOT** mandatory to use them all.

1 Part 1

In this part of the homework, you have to deal with Text Representation.

[ ]:
```
#REMOVE_OUTPUT#
!pip install --upgrade --no-cache-dir gdown
#YOUR CODE STARTS HERE#
import numpy as np, pandas as pd, nltk, re, time, matplotlib.pyplot as plt,
 ↪tabulate
!pip install langdetect
from langdetect import detect
nltk.download('punkt')
nltk.download('stopwords')
from nltk.corpus import stopwords
!pip install -U sentence-transformers
from sklearn.feature_extraction.text import CountVectorizer
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
from sentence_transformers import SentenceTransformer
from sklearn.model_selection import train_test_split, cross_val_score, KFold
from sklearn.naive_bayes import MultinomialNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score,
 ↪precision_score, recall_score, f1_score, log_loss
#YOUR CODE ENDS HERE#
#THIS IS LINE 20#
```

1.1 Part 1.1

The company **F**antastic **S**olution sells products. Customers can leave product reviews on their platform. The company wants to classify the reviews into positive and negative.

Their requirements are unclear: they mention both accuracy and calculation time, but it is not known which is more important to them. :'(

They also forbid you to do a hyper-parameter optimisation. (why? :O)

To help you (?), they have already pre-processed the data. They have translated each text into a random language.

The best thing to do is to provide them with a list of models that can best meet their (unclear) requirements.

1.1.1 1.1.1

Download the data from the Drive link (code already provided).

[ ]:
```
#REMOVE_OUTPUT#
!gdown 1X6QnCcOgnNEBQ1xnilmPWqDIs7bRrQof
```

1.1.2 1.1.2

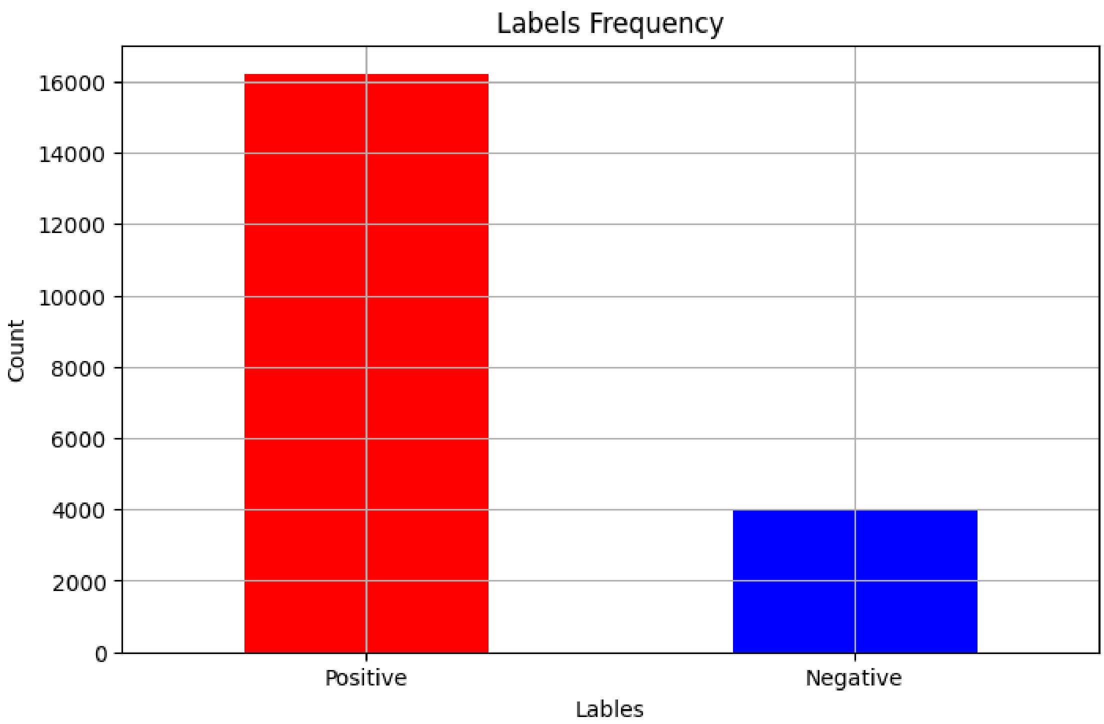
Understand (!) and pre-process *(general term!)* the data. Divide the data according to your needs.

No specific request

[38]:
```
#YOUR CODE STARTS HERE#

# Exploratory Data Analysis (EDA)  1

FS_reviews = pd.read_json('FS_reviews.jsonl',  lines = True)
FS_reviews.to_csv ('FS_reviews.csv', index = None)

FS_reviews.info()

print("")
print("Raiting Frequency: ")
print(FS_reviews["rating"].value_counts())

print("")
FS_reviews['label'] = 0
FS_reviews.loc[FS_reviews['rating'].isin([1, 2]), 'label'] = 1 # Negative␣
↪samples
FS_reviews.loc[FS_reviews['rating'].isin([4, 5]), 'label'] = 0 # Positive␣
↪samples

plt.figure(figsize=(8, 5))
ax = FS_reviews["label"].value_counts().plot(kind = "bar", color=["red",␣
↪"blue"])
ax.set_title("Labels Frequency")
ax.set_xlabel("Lables")
ax.set_xticklabels(["Positive", "Negative"], rotation=360)
```

```python
ax.set_ylabel("Count")
ax.grid()
plt.show()











#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20210 entries, 0 to 20209
Data columns (total 10 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   unique_id          20210 non-null  object
 1   product_name       20210 non-null  object
 2   product_type       20210 non-null  object
 3   helpful            20210 non-null  object
 4   rating             20210 non-null  int64
 5   title              20210 non-null  object
 6   date               20200 non-null  datetime64[ns]
 7   review_text        20210 non-null  object
 8   reviewer           20210 non-null  object
 9   reviewer_location  20210 non-null  object
dtypes: datetime64[ns](1), int64(1), object(8)
memory usage: 1.5+ MB

Raiting Frequency:
5    12102
4     4108
1     2568
2     1432
Name: rating, dtype: int64
```



[39]:

```
#YOUR CODE STARTS HERE#

# Exploratory Data Analysis (EDA) 2
review_text_len = FS_reviews['review_text'].apply(lambda x: len(x.split()))
print("Average number of words in the 'review_text'column before pre-processing:
↪", review_text_len.mean())
title_len = FS_reviews['title'].apply(lambda x: len(x.split()))
print("Average number of words in the 'title' column before pre-processing:",␣
↪title_len.mean())
print("Check if the review text consists entirely of alphabetic characters:",␣
↪FS_reviews['review_text'].str.isalpha().all())

languages = []
for text in FS_reviews['review_text']:
  language = detect(text)
  if language not in languages:
    languages.append(language)

print(" ")
print("Languages found in the reviews: ")
for lang in languages:
  print(lang)
```

```python
print(" ")
print("Languages supported by ntlk library: ")
for lang in stopwords.fileids():
  print(lang)

languages_dict = {
    'en': 'english',
    'ro': 'romanian',
    'da': 'danish',
    'es': 'spanish',
    'de': 'german',
    'sl': 'slovene',
    'it': 'italian',
    'sv': 'swedish',
    'pt': 'portuguese',
    'nl': 'dutch'
}

#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

```
Average number of words in the 'review_text'column before pre-processing:
75.01573478476001
Average number of words in the 'title' column before pre-processing:
4.221326076199901
Check if the review text consists entirely of alphabetic characters: False

Languages found in the reviews:
en
ro
es
af
de
sl
no
ca
it
so
ko
pl
sv
pt
fr

Languages supported by ntlk library:
arabic
azerbaijani
```

```
basque
bengali
catalan
chinese
danish
dutch
english
finnish
french
german
greek
hebrew
hinglish
hungarian
indonesian
italian
kazakh
nepali
norwegian
portuguese
romanian
russian
slovene
spanish
swedish
tajik
turkish
```

[40]:
```
#YOUR CODE STARTS HERE#

# Pre-processing Part
def clean_text(text, languages_name):
    '''
    This function performs text cleaning on the input by removing␣
  ↪non-alphabetic characters and stopwords.
    It then returns a string of pre-processed tokens in lowercase.
    '''
    tokens = nltk.word_tokenize(text.lower()) # lower casing + tokenization
    language = detect(text)
    if language in languages_dict.keys():
        stop_words = set(stopwords.words(languages_dict[language]))
    else:
      stop_words = []

    # remove non-alphabetic characters and stopwords
    preprocessed_text = [re.sub(r'[^a-zA-z\s]', '', token) for token in tokens␣
  ↪if token not in stop_words]
```

```python
    return ' '.join(preprocessed_text)

FS_reviews["review_text_preprocessed"]  = FS_reviews["review_text"].
 ↪apply(lambda text: clean_text(text, languages_dict))

FS_reviews_reduced = FS_reviews[["label", "review_text_preprocessed"]]

review_text_len = FS_reviews_reduced['review_text_preprocessed'].apply(lambda x:
 ↪ len(x.split()))
print("Average number of words in the 'review_text_preprocessed' column after␣
 ↪pre-processing:", review_text_len.mean())
print(" ")

print("Shape of the final dataframe:", FS_reviews_reduced.shape)
print(" ")

print("First 10 lines of the final dataframe: ")

display(FS_reviews_reduced.head(10))




#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

```
Average number of words in the 'review_text_preprocessed' column after pre-
processing: 40.761850569025235

Shape of the final dataframe: (20210, 2)

First 10 lines of the final dataframe:

   label                            review_text_preprocessed
0      1  expensive  three months daily use ca nt say he…
1      1              meditation thing  buy tape take class
2      1  got pedometer  found instructions nt clear  ne…
3      1  pedometer arrive held prisoner difficulttoopen…
4      1  offered one cycling thought tasty  good  fact …
5      1  best thing say test took  days receive results…
6      1  oregon scientific pedometer inaccurate waste m…
7      1  pedometer much cheaper      typical decent one…
8      1  monitor works really well  functions bit limit…
9      1  nt know works found would need wear chest band…
```

————-YOUR TEXT STARTS HERE————-

First, to obtain a better understanding of our data, we conducted an **Exploratory Data Anal-**

ysis (EDA) and identified the *review_text column* (as the review title does not provide enough information about the review) and the *rating column* as relevant data for our sentiment analysis.

In the original dataset, the ratings range from 1 to 5, with the absence of rating 3 (likely indicating neutrality): in order to use the data for sentiment analysis, we need to label ratings 1 and 2 as 1 **(Negative samples)**, and ratings 4 and 5 as 0 **(Positive samples)**.

During our analysis of the *review_text column*, we noticed the presence of non-alphanumeric characters and excessively long reviews: as a result, we have decided to **clean the text** by converting it to lowercase, removing non-alphanumeric characters and eliminating stopwords in multiple languages.

1.1.3 1.1.3

Choose at least 1 and a maximum of 3 encodings. Encode the data.

P.S. If you need it, Word2Vec has a version for Documents

[41]:
```
#YOUR CODE STARTS HERE#

start_time = time.time()

count_vect = CountVectorizer() # default parameters for reproducibility
sentences = FS_reviews_reduced["review_text_preprocessed"].to_numpy()
count_vect_X = count_vect.fit_transform(sentences)
count_vect_word_matrix = count_vect_X.todense()
count_vect_word_matrix

end_time = time.time()
pipeline1_time = end_time - start_time




























#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

[42]:
```
#YOUR CODE STARTS HERE#

start_time = time.time()

list_of_words = FS_reviews_reduced["review_text_preprocessed"].apply(lambda x :␣
↪x.split())
documents = [TaggedDocument(doc, [i]) for i, doc in enumerate(list_of_words)]
doc2vec = Doc2Vec(documents) # default parameters for reproducibility
doc2vec_X = [list(doc2vec.infer_vector(doc.words)) for doc in documents]
doc2vec_X = np.array(doc2vec_X)
doc2vec_X

end_time = time.time()
pipeline2_time = end_time - start_time
























#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

[43]:
```
#YOUR CODE STARTS HERE#

start_time = time.time()

model = SentenceTransformer('paraphrase-albert-small-v2')
```

```
sentences = FS_reviews_reduced["review_text_preprocessed"].to_numpy()
sentence_embeddings_X = model.encode(sentences) # default parameters for
↪reproducibility
sentence_embeddings_X

end_time = time.time()
pipeline3_time = end_time - start_time

#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

———-YOUR TEXT STARTS HERE———-

The first chosen encoder, **Count Vectorizer** or Bag-of-Words (BoW), represents the text using a frequency vector of each word in it, allowing it to effectively capture the significance of words.

The second chosen encoder, **Doc2Vec**, performs well in capturing the context and semantics of words.

The last chosen encoder, **BERT** (Bidirectional Encoder Representations from Transformers), represents a powerful choice for sentiment analysis due to its transformer-based architecture.

1.1.4 1.1.4

Choose **ONE** classifier for **EACH** encoding. Train the classifiers.

[44]:
```
#YOUR CODE STARTS HERE#

y = FS_reviews_reduced["label"].to_numpy()

# CountVectorizer + MultinomialNaiveBayes
mnb_clf = MultinomialNB() # default parameters for reproducibility

# Split in train, test and validation
X_train_mnb, X_test_mnb, y_train_mnb, y_test_mnb =␣
 ↪train_test_split(count_vect_X,
                                                                    y,
                                                                    test_size =␣
 ↪0.15,
                                                                    # seed for␣
 ↪reproducibility
                                                                   ␣
 ↪random_state = 709)

# 0.85 represents the remaining 85% after the initial test split
X_train_mnb, X_val_mnb, y_train_mnb, y_val_mnb = train_test_split(X_train_mnb,
                                                                  y_train_mnb,
                                                                  test_size = 0.
 ↪17/0.85,
                                                                  # seed for␣
 ↪reproducibility
                                                                  random_state␣
 ↪= 709)


# Train
start_time = time.time()

mnb_clf.fit(X_train_mnb, y_train_mnb)

end_time = time.time()
pipeline1_time += (end_time - start_time)
```

```
#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

[45]:

```
#YOUR CODE STARTS HERE#

# Doc2Vec + RandomForest
rf_clf = RandomForestClassifier(random_state = 709) # default parameters and␣
 ↪seed for reproducibility

# Split in train, test and validation
X_train_rf, X_test_rf, y_train_rf, y_test_rf = train_test_split(doc2vec_X,
                                                                y,
                                                                test_size = 0.
 ↪15,
                                                                # seed for␣
 ↪reproducibility
                                                                random_state =␣
 ↪709)

# 0.85 represents the remaining 85% after the initial test split
X_train_rf, X_val_rf, y_train_rf, y_val_rf = train_test_split(X_train_rf,
                                                              y_train_rf,
                                                              test_size = 0.17/
 ↪0.85,
                                                              # seed for␣
 ↪reproducibility
                                                              random_state =␣
 ↪709)

# Train
start_time = time.time()

rf_clf.fit(X_train_rf, y_train_rf)

end_time = time.time()
pipeline2_time += (end_time - start_time)
```

```
#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

[46]:
```
#YOUR CODE STARTS HERE#

# SVC + BERT
svc_clf = SVC(random_state = 709) # default parameters and seed for␣
↪reproducibility

# Split in train, test and validation
X_train_svc, X_test_svc, y_train_svc, y_test_svc =␣
↪train_test_split(sentence_embeddings_X,
                                                                    y,
                                                                    test_size =␣
↪0.15,
                                                                    # seed for␣
↪reproducibility
                                                                   ␣
↪random_state = 709)

# 0.85 represents the remaining 85% after the initial test split
X_train_svc, X_val_svc, y_train_svc, y_val_svc = train_test_split(X_train_svc,
                                                                  y_train_svc,
                                                                  test_size = 0.
↪17/0.85,
                                                                  # seed for␣
↪reproducibility
                                                                  random_state␣
↪= 709)

# Train
start_time = time.time()

svc_clf.fit(X_train_svc, y_train_svc)

end_time = time.time()
pipeline3_time += (end_time - start_time)
```

```
#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

———-YOUR TEXT STARTS HERE———-

Due to an imbalance between the minority class (1) and the majority class (0), we decided to use classifiers that are robust to **class imbalance**. Furthermore, to ensure reproducibility and avoid hyperparameter optimization, we selected classifiers that demonstrate strong performance using their **default parameters**.

Specifically, **Multinomial Naive Bayes Classifier** is suitable for count-based data, **Random Forest Classifier** is good in capturing intricate word-class relationships from Doc2Vec embeddings and **Support Vector Classifier (SVC)** performs well with high-dimensional feature vectors produced by BERT.

1.1.5 1.1.5

Obtain the metrics you want to show the company.

[47]:
```
#YOUR CODE STARTS HERE#
y_pred_mnb = mnb_clf.predict(X_val_mnb)

report_mnb = classification_report(y_val_mnb, y_pred_mnb)
accuracy_mnb = accuracy_score(y_val_mnb, y_pred_mnb)
precision_mnb = precision_score(y_val_mnb, y_pred_mnb)
recall_mnb = recall_score(y_val_mnb, y_pred_mnb)
F1_mnb = f1_score(y_val_mnb, y_pred_mnb)


kfold = KFold(n_splits = 5, shuffle = True, random_state = 709) # seed for␣
 ↪reproducibility
scores = cross_val_score(mnb_clf, X_val_mnb, y_val_mnb, cv=kfold)
mean_score_mnb = scores.mean()
std_mnb = scores.std()
























#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

[48]:
```
#YOUR CODE STARTS HERE#
y_pred_rf = rf_clf.predict(X_val_rf)
```

```
report_rf = classification_report(y_val_rf, y_pred_rf)
accuracy_rf = accuracy_score(y_val_rf, y_pred_rf)
precision_rf = precision_score(y_val_rf, y_pred_rf)
recall_rf = recall_score(y_val_rf, y_pred_rf)
F1_rf = f1_score(y_val_rf, y_pred_rf)


scores = cross_val_score(rf_clf, X_val_rf, y_val_rf, cv=kfold)
mean_score_rf = scores.mean()
std_rf = scores.std()

#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

[49]:

```
#YOUR CODE STARTS HERE#
y_pred_svc = svc_clf.predict(X_val_svc)

report_svc = classification_report(y_val_svc, y_pred_svc)
accuracy_svc = accuracy_score(y_val_svc, y_pred_svc)
precision_svc = precision_score(y_val_svc, y_pred_svc)
recall_svc = recall_score(y_val_svc, y_pred_svc)
F1_svc = f1_score(y_val_svc, y_pred_svc)
```

```
scores = cross_val_score(svc_clf, X_val_svc, y_val_svc, cv=kfold)
mean_score_svc = scores.mean()
std_svc = scores.std()

#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

———-YOUR TEXT STARTS HERE———-

The company is primarily interested in the **accuracy metric and calculation time**, but accuracy metric may not be sufficient when dealing with an *imbalanced dataset* like ours.

In fact, while accuracy measures how well a model performs overall, **precision, recall, and F1 score** provide insight into where and how the model makes mistakes.

Additionaly, **Cross-validation score** provide an estimation of the model's performance and its ability to generalize predictions on unseen data.

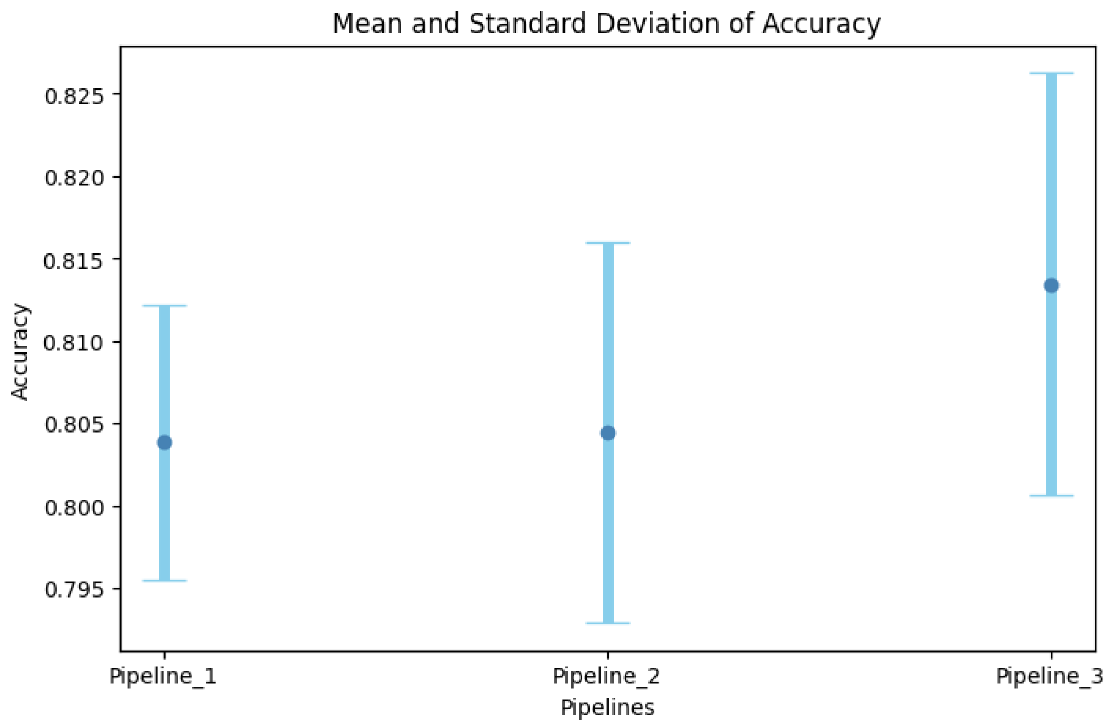
1.1.6 1.1.6

Provide the company with all the information it needs to choose the pipeline it prefers.

[70]:
```
#YOUR CODE STARTS HERE#

rows = [
    ['Pipeline_1', 'MultinomialNB', 'Count Vectorizer', accuracy_mnb,␣
 ↪precision_mnb, recall_mnb, F1_mnb, pipeline1_time],
    ['Pipeline_2', 'Random Forest', 'Doc2Vec', accuracy_rf, precision_rf,␣
 ↪recall_rf, F1_rf, pipeline2_time],
    ['Pipeline_3', 'SVC', 'BERT', accuracy_svc, precision_svc, recall_svc,␣
 ↪F1_svc, pipeline3_time]
]

columns = ['Pipeline', 'Classifier', 'Encoder', 'Accuracy', 'Precision',␣
 ↪'Recall', 'F1 score', 'Calculation Time (in seconds)']
table = pd.DataFrame(rows, columns=columns)

display(table)
print(" ")
print("Classification Report for the Pipeline_1")
print(report_mnb, "\n")
print("Classification Report for the Pipeline_2")
print(report_rf, "\n")
print("Classification Report for the Pipeline_3")
print(report_svc)
print(" ")

mean_scores = [mean_score_mnb, mean_score_rf, mean_score_svc ]
std_scores = [std_mnb, std_rf, std_svc]
configurations = ['Pipeline_1', 'Pipeline_2', 'Pipeline_3']

plt.figure(figsize=(8, 5))
plt.errorbar(configurations, mean_scores, yerr = std_scores, fmt = 'o', color =␣
 ↪'steelblue',
ecolor = 'skyblue', elinewidth = 5, capsize=10)
plt.xlabel('Pipelines')
plt.ylabel('Accuracy')
plt.title('Mean and Standard Deviation of Accuracy')
plt.show()
```

```
#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

```
    Pipeline     Classifier           Encoder  Accuracy  Precision    Recall  \
0  Pipeline_1  MultinomialNB  Count Vectorizer  0.837893   0.646226  0.402349
1  Pipeline_2  Random Forest           Doc2Vec  0.809953   0.818182  0.052863
2  Pipeline_3            SVC              BERT  0.834109   0.872483  0.190896

   F1 score  Calculation Time (in seconds)
0  0.495928                       1.613990
1  0.099310                      67.644440
2  0.313253                     108.249977


Classification Report for the Pipeline_1
              precision    recall  f1-score   support

           0       0.86      0.95      0.90      2755
           1       0.65      0.40      0.50       681

    accuracy                           0.84      3436
   macro avg       0.76      0.67      0.70      3436
weighted avg       0.82      0.84      0.82      3436



Classification Report for the Pipeline_2
              precision    recall  f1-score   support

           0       0.81      1.00      0.89      2755
           1       0.82      0.05      0.10       681

    accuracy                           0.81      3436
   macro avg       0.81      0.52      0.50      3436
weighted avg       0.81      0.81      0.74      3436



Classification Report for the Pipeline_3
              precision    recall  f1-score   support

           0       0.83      0.99      0.91      2755
           1       0.87      0.19      0.31       681

    accuracy                           0.83      3436
   macro avg       0.85      0.59      0.61      3436
weighted avg       0.84      0.83      0.79      3436
```



———-YOUR TEXT STARTS HERE———-

We have supplied to the company a **summary table** that presents the different pipelines we used for sentiment analysis and their corresponding performance.

We have also provided a **classification report** for each pipeline, which allows us to observe how the pipeline performs on both the Negative samples (1) and the Positive samples (0).

Additionally, we have generated a **Mean and Standard Deviation of Accuracy plot** that provides insights into the consistency and stability of the pipeline's performance.

———-YOUR TEXT STARTS HERE———-

If the company prioritizes **only accuracy**, Pipeline 3 shows in the plot a higher mean and standard deviation of accuracy compared to the other two pipelines (but very high calculation time).

Otherwise, if the company priorities **only calculation time** or **accuracy and calculation time**, Pipeline 1 achieves optimal calculation time and consistent accuracy.

Overall, both pipeline result in a robust outcome that also handles well class imbalance of our dataset, while Pipeline 2, despite having good accuracy and calculation time, does not handle class imbalance well (as we can observe from the F1 score).

1.2 Part 1.2

1.2.1 1.2.1

Consider a scenario in which you have a set of words.

These must be transformed into a representation suitable for Machine Learning.

However, each representation has a fixed limit K.

Comment on how 3 word representations would behave in this scenario.

Use at most 3 sentences.

———-YOUR TEXT STARTS HERE———-

CountVectorizer manages to handle only situations in which K is smaller than the non-fixed length of word representation vectors: by setting the **max_features parameter** to K we limit the vocabulary to the top K most frequent words and we can indirectly control the length of the vectors.

In *Doc2Vec*, the **vector_size parameter** determines the dimensionality of the word representation vectors or embeddings: we can set it to K in order to respond to the need to have a fixed limit.

Similary, *BERT trasformer* employs **padding technique** to address the fixed word representation vector size limitation: if the length is bigger than K padding tokens are added to the end until it reaches length K, otherwise the vector is truncated to length K.

2 Part 2

In this part of the homework, you have to deal with Deep Learning.

[ ]:
```python
#REMOVE_OUTPUT#
#YOUR CODE STARTS HERE#
import torch
from torch.utils.data import DataLoader
from torch.utils.data.dataset import random_split
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator
from torchtext.data.functional import to_map_style_dataset
from sklearn.model_selection import train_test_split
import time
torch.manual_seed(709) # seed for reproducibility








#YOUR CODE ENDS HERE#
#THIS IS LINE 20#
```

2.1 Part 2.1

You have to use the same data as in Part 1, but you can use whatever adjustments you have made to it (only Part 1.1.2).

2.1.1 2.1.1

Prepare the data structures you will need.

[52]:
```python
#YOUR CODE STARTS HERE#

# Create a tokenizer
tokenizer = get_tokenizer('basic_english')

def yield_tokens(data_iter):
    for _, text in data_iter:
        yield tokenizer(text)

# Split the reduced dataset into train and test
train_data, test_data = train_test_split(FS_reviews_reduced, test_size=0.2,
 ↪random_state=709)

# Build the vocabulary from the tokens yielded by train_data
```

```
vocab = build_vocab_from_iterator(yield_tokens(train_data.values),␣
 ↪specials=["<unk>"])
vocab.set_default_index(vocab["<unk>"])

# Create the pipelines
text_pipeline = lambda x: vocab(tokenizer(x))
label_pipeline = lambda x: int(x)


#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

[53]:
```
#YOUR CODE STARTS HERE#

# Use the gpu
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def collate_batch(batch):
    label_list, text_list, offsets = [], [], [0]
    for (_label, _text) in batch:
         label_list.append(label_pipeline(_label))
         processed_text = torch.tensor(text_pipeline(_text), dtype=torch.int64)
         text_list.append(processed_text)
         offsets.append(processed_text.size(0))
    label_list = torch.tensor(label_list, dtype=torch.int64)
    offsets = torch.tensor(offsets[:-1]).cumsum(dim=0)
    text_list = torch.cat(text_list)
    return label_list.to(device), text_list.to(device), offsets.to(device)

# Create a dataloader object
```

```
dataloader = DataLoader(train_data.values, batch_size=8, shuffle=True,␣
↪collate_fn=collate_batch)
















#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

[54]:
```
#YOUR CODE STARTS HERE#
```

```
#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

———-YOUR TEXT STARTS HERE———-

The **yield_tokens function** is defined to iterate over the data and yield tokens from the text column using the tokenizer and is used to separates words from the sentence: it takes text as input and returns tokens/words as output.

The **build_vocab_from_iterator() function** returns an instance of Vocab that has a mapping from word to their indexes according.

We created a helper function to tokenize and vectorize text in order to process train datasets loaded into the DataLoader() constructor and applied to each batch using the collate_fn argument during looping.

2.1.2 2.1.2

Define your model

[55]:
```
#YOUR CODE STARTS HERE#
class TextClassificationModel(torch.nn.Module):
    def __init__(self, vocab_size, embed_dim, num_ratings):
        super(TextClassificationModel, self).__init__()

        # Embedding layer
        self.embedding = torch.nn.EmbeddingBag(vocab_size, embed_dim,
 ↪sparse=False)

        # Linear layer
        self.fc = torch.nn.Linear(embed_dim, num_ratings)

        self.init_weights()

    def init_weights(self):
        initrange = 0.5
        self.embedding.weight.data.uniform_(-initrange, initrange)
        self.fc.weight.data.uniform_(-initrange, initrange)
        self.fc.bias.data.zero_()

    def forward(self, text, offsets):
        embedded = self.embedding(text, offsets)
        return self.fc(embedded)

















#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

[56]:
```
#YOUR CODE STARTS HERE#
all_ratings = set([label for (label, text) in train_data.values])
```

```
print("Ratings",all_ratings)
num_ratings = len(all_ratings)
print("Number ratings",num_ratings)
FS_reviews_label = {0: "Negative",
                    1: "Positive"}

vocab_size = len(vocab)
emb_size = 64
model = TextClassificationModel(vocab_size, emb_size, num_ratings).to(device)


#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

```
Ratings {0, 1}
Number ratings 2
```

———-YOUR TEXT STARTS HERE———-

This code defines a **text classification model** with an Embedding layer followed by a Linear layer: the *forward method* performs the forward pass for mapping input text to predicted ratings, and the *init_weights method* initializes the weights of the model's layers.

The **EmbeddingBag layer**, self.embedding, generates a vector of values (embed_dim) for each word and computes the mean of word embeddings in ‘bags’, taking *vocab_size* and *embed_dim* as input parameters.

The **Linear layer** is defined to perform the final classification, taking *embed_dim* as input and producing *num_ratings* as output, which is used to map the output of the embedding layer to the predicted ratings.

2.1.3 2.1.3

Train and optimize your model

[57]:
```
#YOUR CODE STARTS HERE#
def train(dataloader):
    model.train()
    total_acc, total_count = 0, 0
    log_interval = 500
    start_time = time.time()

    for idx, (label, text, offsets) in enumerate(dataloader):
        optimizer.zero_grad()
        predicted_label = model(text, offsets)
        loss = criterion(predicted_label, label)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 0.1)
        optimizer.step()
        total_acc += (predicted_label.argmax(1) == label).sum().item()
        total_count += label.size(0)
        if idx % log_interval == 0 and idx > 0:
            elapsed = time.time() - start_time
            print('| epoch {:3d} | {:5d}/{:5d} batches '
                  '| accuracy {:8.3f}'.format(epoch, idx, len(dataloader),
                                              total_acc/total_count))
            total_acc, total_count = 0, 0
            start_time = time.time()

def evaluate(dataloader):
    model.eval()
    total_acc, total_count = 0, 0

    with torch.no_grad():
        for idx, (label, text, offsets) in enumerate(dataloader):
            predicted_label = model(text, offsets)
            loss = criterion(predicted_label, label)
            total_acc += (predicted_label.argmax(1) == label).sum().item()
            total_count += label.size(0)
    return total_acc/total_count



#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

[58]:
```
#YOUR CODE STARTS HERE#
```

```
#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

[59]:
```
#YOUR CODE STARTS HERE#
```

```
#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

———-YOUR TEXT STARTS HERE———-

The **train function** trains the model by iterating over batches of labeled text data provided by the dataloader, using the model to predict labels, calculating loss with a criterion function, and accumulating accuracy.

The **evaluate function** evaluates the model's accuracy on a validation or test set by iterating over batches of data, calculating predicted labels, computing loss, and accumulating accuracy and the total number of examples.

In particular, **accuracy** is calculated by comparing the predicted labels with the true labels.

2.1.4 2.1.4

Show the performance of your model

[60]:
```
#YOUR CODE STARTS HERE#
# Hyperparameters
EPOCHS = 10 # epoch
LR = 5  # learning rate
BATCH_SIZE = 64 # batch size for training

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=LR)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 1.0, gamma=0.1)
total_accu = None

train_dataset = to_map_style_dataset(train_data.values) # Convert␣
 ↪iterable-style dataset to map-style dataset.
test_dataset = to_map_style_dataset(test_data.values) # Convert iterable-style␣
 ↪dataset to map-style dataset.
num_train = int(len(train_dataset) * 0.95)
split_train_, split_valid_ = \
    random_split(train_dataset, [num_train, len(train_dataset) - num_train])

train_dataloader = DataLoader(split_train_, batch_size=BATCH_SIZE,
                              shuffle=True, collate_fn=collate_batch)
valid_dataloader = DataLoader(split_valid_, batch_size=BATCH_SIZE,
                              shuffle=True, collate_fn=collate_batch)
test_dataloader = DataLoader(test_dataset, batch_size=BATCH_SIZE,
                             shuffle=True, collate_fn=collate_batch)

for epoch in range(1, EPOCHS + 1):
    epoch_start_time = time.time()
    train(train_dataloader)
    accu_val = evaluate(valid_dataloader)
    if total_accu is not None and total_accu > accu_val:
      scheduler.step()
    else:
       total_accu = accu_val
    print('-' * 59)
    print('| end of epoch {:3d} | time: {:5.2f}s | '
          'valid accuracy {:8.3f} '.format(epoch,
                                           time.time() - epoch_start_time,
                                           accu_val))
    print('-' * 59)
#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

```
-----------------------------------------------------------
| end of epoch   1 | time:  1.72s | valid accuracy    0.792
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   2 | time:  1.16s | valid accuracy    0.823
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   3 | time:  1.17s | valid accuracy    0.800
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   4 | time:  1.15s | valid accuracy    0.843
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   5 | time:  1.16s | valid accuracy    0.843
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   6 | time:  1.16s | valid accuracy    0.842
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   7 | time:  1.18s | valid accuracy    0.844
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   8 | time:  1.14s | valid accuracy    0.844
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   9 | time:  1.14s | valid accuracy    0.844
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch  10 | time:  1.44s | valid accuracy    0.845
-----------------------------------------------------------
```

[61]:

```
#YOUR CODE STARTS HERE#
print('Checking the results of test dataset.')
accu_test = evaluate(test_dataloader)
print('Test accuracy {:8.3f}'.format(accu_test))
```

```
#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

```
Checking the results of test dataset.
Test accuracy    0.843
```

[62]:
```
#YOUR CODE STARTS HERE#
```

```
#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

———-YOUR TEXT STARTS HERE———-

We split the training dataset into train/valid sets with a split ratio of 0.95 (train) and 0.05 (valid)using random_split() function in PyTorch core library.

CrossEntropyLoss criterion combines nn.LogSoftmax() and nn.NLLLoss() in a single class and it's useful when training a classification problem with C classes, **SGD** implements stochastic gradient descent method as the optimizer, the initial learning rate is set to 5, **StepLR** is used here to adjust the learning rate through epochs.

2.1.5 2.1.5

Provide an ablation study on at least one and at most three parameters.

[63]:
```python
#YOUR CODE STARTS HERE#
# Hyperparameters
EPOCHS = 10
LR =[0.5, 2, 5]  # Learning rate to try
BATCH_SIZE = 64

train_dataset = to_map_style_dataset(train_data.values) #Convert iterable-style dataset to map-style dataset.
test_dataset = to_map_style_dataset(test_data.values) #Convert iterable-style dataset to map-style dataset.
num_train = int(len(train_dataset) * 0.95)
split_train_, split_valid_ = \
    random_split(train_dataset, [num_train, len(train_dataset) - num_train])
for lr in LR:
    print(f"Training with learning rate: {lr}")
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=lr)
    scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 1.0, gamma=0.1)
    total_accu = None

    for epoch in range(1, EPOCHS + 1):
        epoch_start_time = time.time()
        train(train_dataloader)
        accu_val = evaluate(valid_dataloader)
        if total_accu is not None and total_accu > accu_val:
            scheduler.step()
        else:
            total_accu = accu_val
        print('-' * 59)
        print('| end of epoch {:3d} | time: {:5.2f}s | '
              'valid accuracy {:8.3f} '.format(epoch,
                                               time.time() - epoch_start_time,
                                               accu_val))
        print('-' * 59)






#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

```
Training with learning rate: 0.5
-----------------------------------------------------------
| end of epoch   1 | time:  1.77s | valid accuracy    0.847
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   2 | time:  1.34s | valid accuracy    0.847
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   3 | time:  1.13s | valid accuracy    0.849
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   4 | time:  1.15s | valid accuracy    0.854
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   5 | time:  1.17s | valid accuracy    0.853
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   6 | time:  1.14s | valid accuracy    0.853
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   7 | time:  1.14s | valid accuracy    0.853
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   8 | time:  1.14s | valid accuracy    0.853
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   9 | time:  1.14s | valid accuracy    0.853
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch  10 | time:  1.14s | valid accuracy    0.853
-----------------------------------------------------------
Training with learning rate: 2
-----------------------------------------------------------
| end of epoch   1 | time:  1.72s | valid accuracy    0.854
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   2 | time:  1.80s | valid accuracy    0.850
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   3 | time:  1.17s | valid accuracy    0.860
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   4 | time:  1.15s | valid accuracy    0.860
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   5 | time:  1.14s | valid accuracy    0.860
-----------------------------------------------------------
-----------------------------------------------------------
```

```
| end of epoch   6 | time:  1.16s | valid accuracy    0.862
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   7 | time:  1.14s | valid accuracy    0.860
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   8 | time:  1.13s | valid accuracy    0.860
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   9 | time:  1.15s | valid accuracy    0.860
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch  10 | time:  1.15s | valid accuracy    0.860
-----------------------------------------------------------
Training with learning rate: 5
-----------------------------------------------------------
| end of epoch   1 | time:  1.31s | valid accuracy    0.858
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   2 | time:  1.75s | valid accuracy    0.864
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   3 | time:  1.62s | valid accuracy    0.817
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   4 | time:  1.15s | valid accuracy    0.860
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   5 | time:  1.14s | valid accuracy    0.859
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   6 | time:  1.16s | valid accuracy    0.860
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   7 | time:  1.16s | valid accuracy    0.860
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   8 | time:  1.14s | valid accuracy    0.860
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   9 | time:  1.13s | valid accuracy    0.860
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch  10 | time:  1.14s | valid accuracy    0.860
-----------------------------------------------------------
```

[64]:

```
#YOUR CODE STARTS HERE#
# Hyperparameters
EPOCHS = 10
LEARNING_RATE = 5
batch_sizes = [32, 64, 128]  # Batch sizes to try

train_dataset = to_map_style_dataset(train_data.values)
test_dataset = to_map_style_dataset(test_data.values)
num_train = int(len(train_dataset) * 0.95)
split_train_, split_valid_ = random_split(train_dataset, [num_train,␣
 ↪len(train_dataset) - num_train])

valid_dataloader = DataLoader(split_valid_, batch_size=BATCH_SIZE,␣
 ↪shuffle=True, collate_fn=collate_batch)

for batch_size in batch_sizes:
    print(f"Training with batch size: {batch_size}")
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING_RATE)
    scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 1.0, gamma=0.1)
    total_accu = None

    train_dataloader = DataLoader(split_train_, batch_size=batch_size,␣
 ↪shuffle=True, collate_fn=collate_batch)

    for epoch in range(1, EPOCHS + 1):
        epoch_start_time = time.time()
        train(train_dataloader)
        accu_val = evaluate(valid_dataloader)
        if total_accu is not None and total_accu > accu_val:
            scheduler.step()
        else:
            total_accu = accu_val
        print('-' * 59)
        print(f"| end of epoch {epoch:3d} | time: {time.time() -␣
 ↪epoch_start_time:.2f}s | valid accuracy {accu_val:8.3f}")
        print('-' * 59)





#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

```
Training with batch size: 32
-----------------------------------------------------------
```

```
| end of epoch   1 | time: 1.60s | valid accuracy    0.910
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   2 | time: 2.20s | valid accuracy    0.867
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   3 | time: 2.01s | valid accuracy    0.907
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   4 | time: 1.57s | valid accuracy    0.905
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   5 | time: 1.62s | valid accuracy    0.905
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   6 | time: 1.55s | valid accuracy    0.905
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   7 | time: 1.55s | valid accuracy    0.905
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   8 | time: 1.55s | valid accuracy    0.905
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   9 | time: 1.57s | valid accuracy    0.905
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch  10 | time: 2.18s | valid accuracy    0.905
-----------------------------------------------------------
Training with batch size: 64
-----------------------------------------------------------
| end of epoch   1 | time: 1.58s | valid accuracy    0.907
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   2 | time: 1.46s | valid accuracy    0.867
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   3 | time: 1.79s | valid accuracy    0.899
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   4 | time: 1.44s | valid accuracy    0.901
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   5 | time: 1.16s | valid accuracy    0.901
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   6 | time: 1.16s | valid accuracy    0.901
-----------------------------------------------------------
```

```
-----------------------------------------------------------
| end of epoch   7 | time: 1.17s | valid accuracy    0.901
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   8 | time: 1.17s | valid accuracy    0.901
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   9 | time: 1.59s | valid accuracy    0.901
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch  10 | time: 1.79s | valid accuracy    0.901
-----------------------------------------------------------
Training with batch size: 128
-----------------------------------------------------------
| end of epoch   1 | time: 1.09s | valid accuracy    0.863
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   2 | time: 0.95s | valid accuracy    0.862
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   3 | time: 0.93s | valid accuracy    0.895
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   4 | time: 0.93s | valid accuracy    0.896
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   5 | time: 0.94s | valid accuracy    0.896
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   6 | time: 0.95s | valid accuracy    0.891
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   7 | time: 0.93s | valid accuracy    0.894
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   8 | time: 0.98s | valid accuracy    0.894
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch   9 | time: 0.94s | valid accuracy    0.894
-----------------------------------------------------------
-----------------------------------------------------------
| end of epoch  10 | time: 0.92s | valid accuracy    0.894
-----------------------------------------------------------
```

[35]:
```
#YOUR CODE STARTS HERE#
```

```
#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

———-YOUR TEXT STARTS HERE———-

The validation accuracy in the training with **different learning rates** shows a somewhat stable behavior, with a slight fluctuation but no significant improvement or degradation over the epochs.

The batch size determines how many samples are processed simultaneously before updating the model's weights based on the computed gradients. Also here the validation accuracy in training with **different batch sizes** shows a relatively stable behavior, with minor fluctuations similar to the previous case.

2.2 Part 2.2

2.2.1 2.2.1

How would a Deep Learning model (of the kind we have seen) behave in the case where a word was never seen during training? Answer on both practical and theoretical aspects.

Use at most 3 sentences.

———-YOUR TEXT STARTS HERE———-

When encountering **unknown words** in the test data that do not appear in the training data or vocabulary the approach is to ignore these unknown words, this means that these words are not considered during the evaluation or classification process.

The reason for ignoring unknown words is that the model does not have any information or knowledge about these words and building a specific model for unknown words is not generally helpful because simply knowing which class has more unknown words does not provide useful information for classification.

From a **practical standpoint**, deep learning models would assign low probabilities or scores to unknown words that are encountered during testing, this is because the model has no prior knowledge or training on these words and cannot accurately estimate their relevance or classification.