Consider the Snapfood dataset that can be seen in the attachment. This data includes users' opinions about active restaurants in this collection, which are placed in one of the positive or negative classes by the tag similar to each one.

-1Preprocessing

-1.1What are the necessary pre-processings according to the above task? Name them and prepare the data for entering the model by doing each one.

-1.2Check the status of this data set in terms of data balance and draw the sample distribution diagram of each class.

Use ParsBERT to build a model for sentiment analysis based on user comments. ParsBERT is a language model 4 that is used in the field of natural language processing and is capable of performing tasks such as recognizing text emotions, recognizing named entities in text, etc. To perform this task, you can use the following model among the pre-trained ParsBERT models:

HooshvareLab/bert-fa-base-uncased-sentiment-snappfood

-3Evaluation of the model

-3.1In order to evaluate the model, use f1-score, accuracy and other criteria as desired and explain the reason for your choice. (Keep in mind that the use of your chosen criterion should be considered helpful in measuring the efficiency of the model(.

-3.2Draw the learning graph of the model on the training, validation and evaluation data based on the accuracy of the model and analyze your results.

Points section: If your f1-score reaches a number equal to or higher than 91, you will be included in the points score of this exercise.

Implementation questions.

Introduction

In this assignment, we will work with the Snappfood dataset, which contains user reviews of the restaurants active in the dataset. Each comment has a positive or negative label. Our task is to pre-process the data, build a sentiment analysis model using ParsBERT and evaluate the performance of the model.

In the first part, we perform the necessary pre-processing steps such as normalization and text cleaning to prepare the data for model training. We also check the balance of the dataset and the distribution of samples for each class. In the following, we build the sentiment analysis model using ParsBERT, which is a pre-trained model based on BERT for Persian language. We select a pre-trained ParsBERT model from HooshvareLab and run it on our dataset and evaluate various evaluation metrics such as f1 score and accuracy.

Add required libraries

```
!pip install hazm
!pip install cleantext
!pip install transformers
!pip install numpy requests nlpaug
```

This code installs three Python packages: "hazm", "cleantext" and "transformers."

"hazm" is a Python library for Persian text processing. This library provides features for Persian text normalization, tokenization, stemming, stop word removal, etc.

cleantext is another Python library that provides functions for cleaning and normalizing text data, such as removing URLs, email addresses, and phone numbers, as well as removing HTML tags and converting text to lowercase.

"transformers" also provides access to pre-trained transformer models for NLP tasks and includes pre-trained models such as BERT, GPT-2, etc.

nlpaug is a data augmentation library in natural language processing that augments the amount of data available using small and simple changes.

Finally, these packages provide us with the necessary tools and capabilities for preprocessing text data, training and evaluating models, and predicting new text inputs.

```
import numpy as np
import pandas as pd
import hazm
import torch
import re
import string
import os
import json
import copy
import collections
import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, f1_score
from sklearn.utils import shuffle
from transformers import BertConfig, BertTokenizer, BertModel, AdamW, get_linear_schedule_with_warmup
from tqdm.notebook import tqdm
from cleantext import clean
from hazm import Normalizer, stopwords_list, word_tokenize
from sklearn.preprocessing import LabelEncoder
from transformers import TFBertForSequenceClassification
import tensorflow as tf
```

This code uses various packages to implement a predictive model for textual data using the BERT model.

-numpy: used to work with numeric arrays in Python.

-pandas: used to work with tabular data and dataframes.

-Hazm: is a natural language processing package for Farsi language.

-torch: used to implement neural network models.

-re: used to work with expressions with rules and regex.

-string: used to work with text strings.

-os: used to work with the operating system.

-json: used to work with json files.

-copy: used to copy data and objects.

-collections: used to work with demographic data and advanced data structures.

-matplotlib: used to draw graphs and charts.

-plotly: used to draw advanced and interactive charts.

-sklearn: used to implement machine learning algorithms and evaluate models.

-transformers: used to use pre-trained models such as BERT and GPT-2.

-tqdm: used to display the progress bar during code execution.

-cleantext: used for text preprocessing.

-LabelEncoder: used to convert text labels into numbers.

-TFBertForSequenceClassification: A predictive model for textual data based on the BERT model is used in TensorFlow.

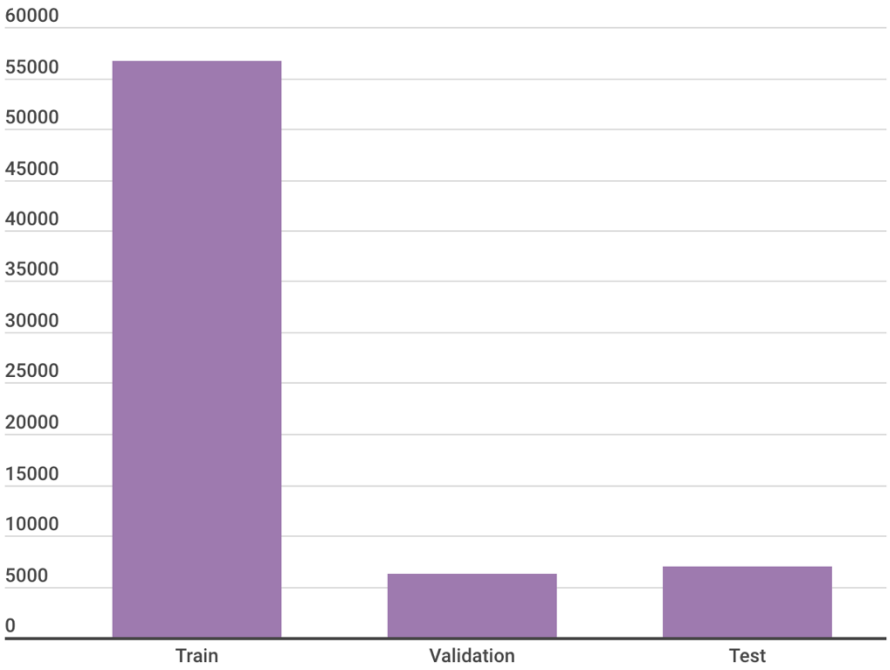
Data collection

An example of the data can be seen in the figure below.

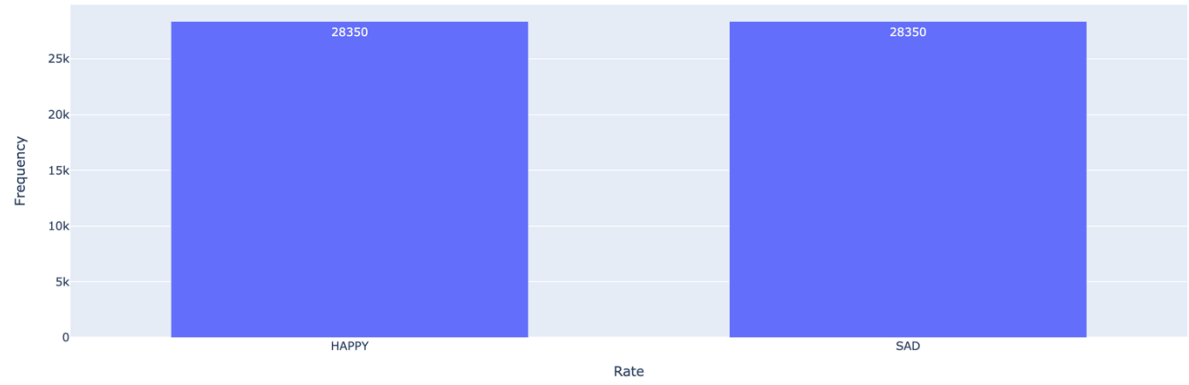
	comment	label	label_id
0	واقعا حیف وقت که بنویسم سرویس دهیتون شده افتضاح	SAD	1
1	...قرار بود ۱ ساعته برسه ولی نیم ساعت زودتر از مو	HAPPY	0
2	...قیمت این مدل اصلا با کیفیتش سازگاری نداره، فقط	SAD	1
3	...عاللی بود همه چه درست و به اندازه و کیفیت خوب	HAPPY	0
4	.شیرینی وانیلی فقط یک مدل بود	HAPPY	0
...	...	...	...
56695	...یک تیکه کم فرستاده بودن و با تماس من در کمترین	HAPPY	0
56696	... عالی بود همه چیز ممنونم پیک هم خیلی مرتب و به	HAPPY	0
56697	...مثل همیشه عالی، من چندمین باره سفارش میدم و هر	HAPPY	0
56698	دلستر استوایی خواسته بودم اما لیمویی فرستادند	HAPPY	0
56699	... جای مرغ گریل شده ناگت بود، به این نمی‌گن چیکن	SAD	1

56700 rows × 3 columns

The number of training data is 56700, validation data is 6300, and test data is 7000. As shown in the figure, the data has three columns. Comment column, label_id and label. The label "HAPPY" is equal to label_id=1 and the label "SAD" is equal to label_id=0.



Also, the diagram below shows the frequency distribution of training data.



Preprocessing

```
# print data information
print('data information')
print(train_df.info(), '\n')

# print missing values information
print('missing values stats')
print(train_df.isnull().sum(), '\n')

# print some missing values
print('some missing values')
print(train_df[train_df['label'].isnull()].iloc[:5], '\n')

# print data information
print('data information')
print(valid_df.info(), '\n')

# print data information
print('data information')
print(test_df.info(), '\n')
```

This code is used to display information about the data. This information includes the general information of the data, the number of invalid or empty values (missing values) and also some examples of data whose label value is invalid.

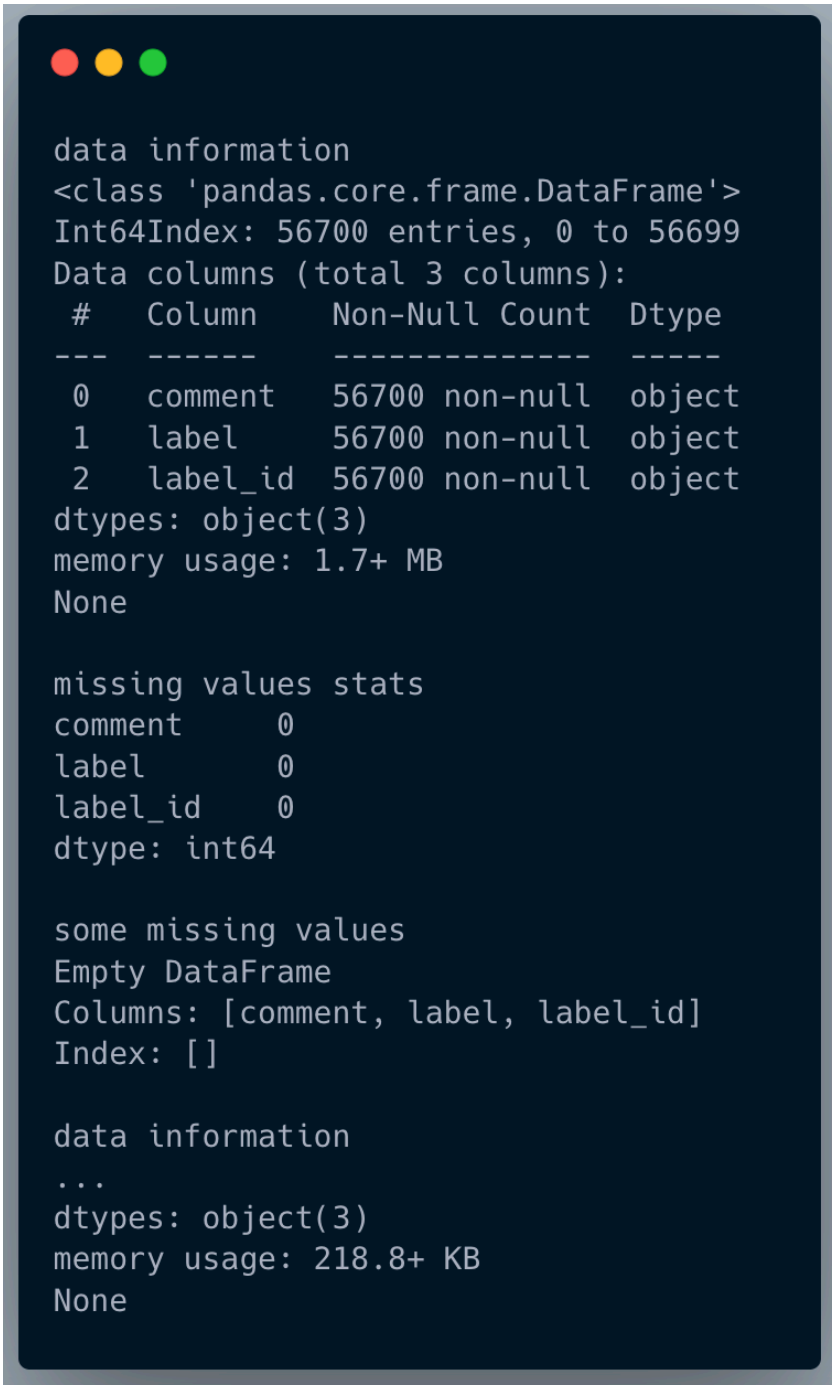
More specifically, this code consists of several sections:

- Displaying general data information using the info() function from the pandas package.
- Display the number of null or empty values in each column of data using the isnull() function from the pandas package and the sum() function.
- Display some examples of data whose tag value is invalid (here marked with null value.(
- Display the general information of the validation set data.

- Display the general information of the data of the test set.

In general, this code is used to help examine the data and see invalid values in the data.

The output of this code is as follows:

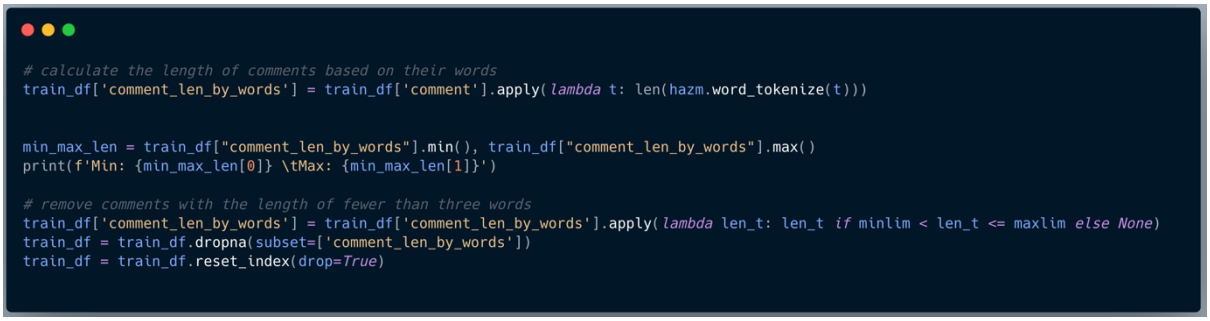


```
data information
<class 'pandas.core.frame.DataFrame'>
Int64Index: 56700 entries, 0 to 56699
Data columns (total 3 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   comment   56700 non-null  object
 1   label     56700 non-null  object
 2   label_id  56700 non-null  object
dtypes: object(3)
memory usage: 1.7+ MB
None

missing values stats
comment     0
label       0
label_id    0
dtype: int64

some missing values
Empty DataFrame
Columns: [comment, label, label_id]
Index: []

data information
...
dtypes: object(3)
memory usage: 218.8+ KB
None
```

- In the first part, using the info() function from the pandas package, it displays the general information of the data in the csv format data file. More precisely, it shows the number of rows, the number of columns, the names of the columns, and the number of non-null values or the number of values that have an

undefined or empty value. Also, this section displays information about the type of data in the columns.

- In the second part, using the isnull() function from the pandas package, it shows the number of null or empty values in each column of data. Here, there are no null or empty values in the data.
- In the third part, using the iloc() function from the pandas package, some examples of data whose label value is invalid (specified here with null value) are displayed. There is no data with tag value null here.
- In the fourth and fifth sections, the general information of the validation and test set data are also displayed in the same way.



```
# calculate the length of comments based on their words
train_df['comment_len_by_words'] = train_df['comment'].apply(lambda t: len(hazm.word_tokenize(t)))


min_max_len = train_df["comment_len_by_words"].min(), train_df["comment_len_by_words"].max()
print(f'Min: {min_max_len[0]} \tMax: {min_max_len[1]}')

# remove comments with the length of fewer than three words
train_df['comment_len_by_words'] = train_df['comment_len_by_words'].apply(lambda len_t: len_t if minlim < len_t <= maxlim else None)
train_df = train_df.dropna(subset=['comment_len_by_words'])
train_df = train_df.reset_index(drop=True)
```

This code is for preprocessing training data.

First, using `apply()` function and `hazm.word_tokenize()` function, the length of each training data is calculated based on the number of words and stored in a new column named `comment_len_by_words.`

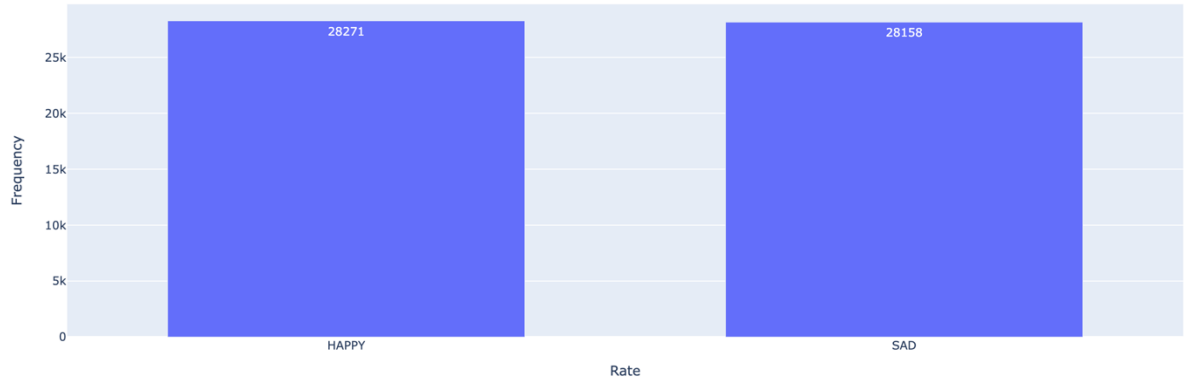
Then, using ``min()'' and ``max()'' functions, the minimum and maximum length of the training data is calculated based on the number of words and stored in the ``min_max_len'' variable.

Then, using ``apply()'' function and a lambda function, the data whose number of words is less than 3 is removed.

Finally, using the `dropna()` function, the data whose `comment_len_by_words` column value is ``None`` are deleted, and using the `reset_index()` function, the training data index is reset.

In this code, ``hazm'' library is used to process the Persian language.

After removing a series of data, the distribution chart is as follows.




```
Explain this code and tell the reason and also clean this code.
“def cleanhtml(raw_html):
    cleanr = re.compile('<.*?>')
    cleantext = re.sub(cleanr, '', raw_html)
    return cleantext

def cleaning(text):
    text = text.strip()

    # regular cleaning
    text = clean(text,
    )

    # cleaning htmls
    text = cleanhtml(text)

    # normalizing
    normalizer = hazm.Normalizer()
    text = normalizer.normalize(text)

    # removing wierd patterns
    wierd_pattern = re.compile("["
        u"\U0001F600-\U0001F64F"  # emoticons
        u"\U0001F300-\U0001F5FF"  # symbols & pictographs
        u"\U0001F680-\U0001F6FF"  # transport & map symbols
        u"\U0001F1E0-\U0001F1FF"  # flags (iOS)
        u"\U00002702-\U000027B0"
        u"\U000024C2-\U0001F251"
        u"\U0001f926-\U0001f937"
        u'\U00010000-\U0010ffff'
        u"\u200d"
        u"\u2640-\u2642"
        u"\u2600-\u2B55"
        u"\u23cf"
        u"\u23e9"
        u"\u231a"
        u"\u3030"
        u"\ufe0f"
        u"\u2069"
        u"\u2066"
        # u"\u200c"
        u"\u2068"
        u"\u2067"
        "]+", flags=re.UNICODE)

    text = wierd_pattern.sub(r'', text)

    # removing extra spaces, hashtags
    text = re.sub("#", "", text)
    text = re.sub("\s+", " ", text)

    return text
```

The above code is a Python function for cleaning text data. In the first part of the code, a function takes a string of HTML code as input (`raw_html`) and uses the `re` module to remove all HTML tags and attributes from the text.

In the next section, the "cleaning" function takes a string of text as input and removes the initial or trailing blank space.

In the next step, remove typographical errors.

And finally, this function uses the "hazm" library to normalize text data (eg, convert Arabic/Persian numbers to English numbers, remove punctuation, etc.(.

This function then uses the `re' module to remove any strange patterns in the text. For example, patterns such as emojis, symbols and other non-standard characters. We also remove any white space and extra hashtags from the text and return the cleaned text.

```
# cleaning comments
train_df['cleaned_comment'] = train_df['comment'].apply(cleaning)
valid_df['cleaned_comment'] = valid_df['comment'].apply(cleaning)
test_df['cleaned_comment'] = test_df['comment'].apply(cleaning)
# calculate the length of comments based on their words
train_df['cleaned_comment_len_by_words'] = train_df['cleaned_comment'].apply(lambda t: len(hazm.word_tokenize(t)))
valid_df['cleaned_comment_len_by_words'] = valid_df['cleaned_comment'].apply(lambda t: len(hazm.word_tokenize(t)))
test_df['cleaned_comment_len_by_words'] = test_df['cleaned_comment'].apply(lambda t: len(hazm.word_tokenize(t)))



valid_df['cleaned_comment_len_by_words'] = valid_df['cleaned_comment_len_by_words'].apply(lambda len_t: len_t if minlim < len_t <=
maxlim else len_t)
valid_df = valid_df.dropna(subset=['cleaned_comment_len_by_words'])
valid_df = valid_df.reset_index(drop=True)

test_df['cleaned_comment_len_by_words'] = test_df['cleaned_comment_len_by_words'].apply(lambda len_t: len_t if minlim < len_t <=
maxlim else len_t)
test_df = test_df.dropna(subset=['cleaned_comment_len_by_words'])
test_df = test_df.reset_index(drop=True)
```

This code is doing some preprocessing and cleanup on a DataFrame called "train_df" that contains a column called "comment."

1. Comment cleanup: This code applies a function called "cleanup" to each value in the "comment" column of the "train_df" DataFrame that we explained earlier.
2. Calculating the length of comments with words: The code uses the "word_tokenize" function of the Hazm library to tokenize each comment in the "Cleaned_Comment" column. It then calculates the length of each tokenized comment and assigns the result to a new column called "cleaned_comment_len_by_words", which contains the word count of each comment.
3. Removing comments with less than three words: If the length of a comment is greater than the "minlim" variable and less than or equal to the "maxlim" variable, its length remains unchanged. Otherwise, it will remove if the length is less than "minlim" or greater than "maxlim". The result is then stored in the 'cleaned_comment_len_by_words' column.
4. Drop rows with missing values: The code drops any row in the DataFrame that has missing values (NaN) in the "cleaned_comment_len_by_words" column using the "dropna" function.
5. Reset the index: Finally, the code resets the index of the DataFrame after deleting the rows.

These changes help to improve readability and maintainability of the code.

Preparation for training

```
# General configuration
COMMENT_MAX_LEN = 50
TRAIN_COMMENT_MAX_LEBATCH_SIZE = 16
VALID_BATCH_SIZE = 16
TEST_BATCH_SIZE = 16
EPOCHS = 3
EVERY_EPOCH = 1000
LEARNING_RATE = 2e-5
CLIP = 0.0

# Model and output configuration
MODEL_NAME_OR_PATH = 'HooshvareLab/bert-fa-base-uncased-sentiment-snappfood'
OUTPUT_PATH = '/content/bert-fa-base-uncased-sentiment-snappfood'
OUTPUT_MODEL_PATH = f'{OUTPUT_PATH}/pytorch_model.bin'

# Print output model path and create output directory
print('OUTPUT_MODEL_PATH:', OUTPUT_MODEL_PATH)
os.makedirs(OUTPUT_PATH, exist_ok=True)
```

This code defines some configuration parameters for a machine learning model and creates directories to store the output files.

The MAX_LEN parameter sets the maximum length of input sequences, while TRAIN_BATCH_SIZE, VALID_BATCH_SIZE, and TEST_BATCH_SIZE set the batch sizes for training, validation, and test data, respectively.

The EPOCHS parameter sets the number of training epochs, while EEVERY_EPOCH sets the number of steps to print the training loss and precision after each epoch.

The LEARNING_RATE parameter sets the learning rate for the optimizer and CLIP sets the maximum gradient norm for gradient clipping.

The MODEL_NAME_OR_PATH parameter sets the name or path of the pre-trained model to use, while OUTPUT_PATH specifies the path to save the output files. The OUTPUT_MODEL_PATH variable concatenates OUTPUT_PATH with /pytorch_model.bin to create the path to store model weights.

The print() function is used to output the value of the OUTPUT_MODEL_PATH variable.

Finally, the os.makedirs function is used to create the OUTPUT_PATH directory if it does not already exist.

```
tokenizer = BertTokenizer.from_pretrained(MODEL_NAME_OR_PATH)
config = BertConfig.from_pretrained(MODEL_NAME_OR_PATH)
```

This code initializes a tokenizer and a config for a pre-trained BERT model.

The BertTokenizer class is used to tokenize the input text into subkeywords that can be processed by the BERT model. The from_pretraine method is used to load the pretrained tokenizer defined by the MODEL_NAME_OR_PATH variable.

The BertConfig class is used to configure the architecture and hyperparameters of the BERT model. The from_pretrained method is also used to load a pretrained configuration.

```
class PrepareDataset(Dataset):

    def __init__(self, tokenizer, data, max_len=128):
        self.data = data
        self.tokenizer = tokenizer
        self.max_len = max_len

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

    def __getitem__(self, ind):
        item = self.data.iloc[ind]
        comment = item['comment']
        label = item['label_id']

        encoding = self.tokenizer.encode_plus(
            comment,
            add_special_tokens=True,
            truncation=True,
            max_length=self.max_len,
            return_token_type_ids=True,
            padding='max_length',
            return_attention_mask=True,
            return_tensors='pt')

        return {
                'comment': comment,
                'input_ids': encoding['input_ids'].flatten(),
                'attention_mask': encoding['attention_mask'].flatten(),
                'token_type_ids': encoding['token_type_ids'].flatten(),
                'lable': int(label),
                }
```

This code defines a PyTorch dataset class called PrepareDataset to process the input data for the BERT model.

The __init__() method initializes the class with the necessary variables, such as tokenizer, data, and maximum sequence length.

The __len__() method returns the length of the dataset, which is the number of rows in the input data.

The __getitem__() method is used to retrieve an item from the dataset. Selects a row from the input data at the specified index and retrieves the comment and label of that row. It then encodes the comment using a tokenizer, adds special tokens or truncates if necessary, and adds to the maximum length of the sequence. The input identifiers, attention mask, and token type identifiers are returned as a dictionary along with the label

```
train_dataloder = DataLoader(train_dataset, batch_size=TRAIN_BATCH_SIZE)
valid_dataloder = DataLoader(valid_dataset, batch_size=VALID_BATCH_SIZE)
test_dataloder = DataLoader(test_dataset, batch_size=TEST_BATCH_SIZE)
```

This code creates PyTorch DataLoader objects for training, validation, and testing datasets.

DataLoader is a PyTorch class used to load data in batches for processing by a neural network. DataLoader takes a dataset object and a batch size as input and returns an iterator that can be used to iterate over data in batches.

In this code, DataLoader objects are created using training, validation, and test datasets (train_dataset, valid_dataset, and test_dataset, respectively) and batch sizes (TRAIN_BATCH_SIZE, VALID_BATCH_SIZE, and TEST_BATCH_SIZE, respectively, in the previous code.(

```
# Inspect a sample of data from the training dataset
sample_data = next(iter(train_dataloader))

# Print the keys of the sample_data dictionary
print(sample_data.keys())

# Print various properties of the sample data
print(sample_data['comment'])
print(sample_data['input_ids'].shape)
print(sample_data['input_ids'][0, :])
print(sample_data['attention_mask'].shape)
print(sample_data['attention_mask'][0, :])
print(sample_data['token_type_ids'].shape)
print(sample_data['token_type_ids'][0, :])
print(sample_data['label'].shape)
print(sample_data['label'][0])
```

This code is used to examine a sample of data from the training dataset.

The next function is used to retrieve the next batch of data from the train_dataloader object, and the iter function is used to convert the train_dataloader object into an iterator. The resulting sample_data dictionary contains a set of input data and its associated tags.

The first print command displays the keys in the sample_data dictionary that match the input feature names and labels in the output. Subsequent print statements output the comment text, input IDs, attention mask, token type IDs, and label for the first item in the batch.

The first line of output shows the sample_data dictionary keys that match the input attribute names and labels. The comment key contains a list of 16 comments in Farsi, each of which is a string. The keys input_ids, focus_mask, and token_type_ids have three tensors of the form [16, 50], indicating that the comments are encoded as sequences of length 50, and a mask is added to ensure that all sequences are of the same length. The label key has a shape tensor [16] that contains the class labels for each comment.

The next lines of output show the different attributes of the first comment in the category. The input_ids tensor holds the encoded sequence of tokens for the comment. The attention_mask tensor indicates which tokens are padding tokens (with a value of 0) and which tokens are part of the actual input sequence (with a value of 1). The token_type_ids tensor is used when there are multiple input sequences, such as in answering a question, but in this case all comments are treated as a single input sequence, so all values in the tensor are equal to 0. The label tensor contains the class label for the first comment, which in this case is 1, which means the comment is negative.

```
def auprc_score(y, yp):

  y_scores = yp
  y_pred = np.where(y_scores >= 0.5, 1, 0)
  precision, recall, thresholds = precision_recall_curve(y, y_pred)
  auprc = auc(recall, precision)

  return auprc
```

This code defines a Python function called auprc_score that calculates the area under the exact curve (AUPRC) for a binary classification problem. This function takes two arguments y and yp, which are the actual labels and predicted probabilities for a set of examples, respectively. AUPRC is a commonly used benchmark for evaluating binary classification models.

The function first assigns the predicted probabilities to y_scores. It then maps the predicted probabilities to binary labels with a threshold of 0.5, meaning that any probability greater than or equal to 0.5 is classified as positive and any probability less than 0.5 is classified as negative. The obtained binary labels are stored in y_pred. The actual tags are stored in tl.

This function then calculates the precision, recall, and threshold values for the precision recall curve using the precision_recall_curve function from scikit-learn. Finally, it calculates and returns the AUPRC value using scikit-learn's auc function.

```
bert = BertModel.from_pretrained('HooshvareLab/bert-fa-base-uncased-sentiment-snappfood')
```

This code creates an instance of the BertModel class and initializes it with the pre-trained weights for a particular BERT model. Specifically, it uses the from_pretrained method to load the pretrained weights for the bert-fa-base-uncased-sentiment-snappfood model; which is a type of BERT model that is pre-trained on a large Persian collection.

The BERT model is a powerful deep learning model for natural language processing that can be used for a variety of tasks including text classification, question answering, and language generation. This model is based on a transformer architecture that uses self-attention mechanisms to encode the entire input sequence in a supervised manner. Pre-trained weights for the BERT model are typically trained on large amounts of textual data using unsupervised learning techniques, such as masked

language modeling and next-sentence prediction, and can be used in specific downstream tasks with smaller amounts of labeled data. be well adjusted.

```
bert_outputs_train = torch.zeros(len(train_df), 768)
bert_outputs_valid = torch.zeros(len(valid_df), 768)
bert_outputs_test = torch.zeros(len(test_df), 768)
```

This code is used to create three zero matrices named `bert_outputs_train`, `bert_outputs_valid` and `bert_outputs_test` with dimensions respectively `(number of training data, 768)`, `(number of validation data, 768)` and `(number of test data, 768) is used.

These matrices are used to output the BERT model for each training, validation and test data.

```
import torch

def extractFeature(data, m, model, device):
    # Create tensor to store BERT embeddings
    bert_outputs = torch.zeros(m, 768)
    i = 0

    for batch in data:
        with torch.no_grad():
            out = model(batch["input_ids"].to(device), batch["attention_mask"].to(device), batch['token_type_ids'].to(device))
['pooler_output'].to(device)
        bert_outputs[i: i + out.size(0)] = out
        i += out.size(0)

    return bert_outputs
```

This code defines a Python function called extractFeature that extracts features from a set of input samples using a pre-trained BERT model. This function takes two arguments: data , which is a PyTorch DataLoader object containing the input samples, and m , which is the maximum number of samples the function can process at once. The function returns a tensor of the shape m, 768)) containing the BERT embeddings for the input samples.

This function first creates a PyTorch tensor named (m, 768) to store the BERT embeddings. It then loops over the categories in the data and calculates the BERT embeddings for each category using the pre-trained model. Embeddings are computed by passing the input identifiers, attention masks, and token type identifiers for each instance in the batch to the BERT model and extracting pooler_output from the data at the output.

The function then copies the embeddings from out to bert_outputs and updates the index i to the next position in bert_outputs to store the embeddings. Finally, the function returns bert_outputs.

Pre-trained weights for the BERT model are typically trained on large amounts of textual data using unsupervised learning techniques, such as masked language modeling and next-sentence prediction, and can perform specific downstream tasks with smaller amounts of labeled data. be well adjusted. BERT embeddings represent the contextual meaning of the input text and can be used as features for downstream tasks, such as text classification or clustering.

Bert output

The output of the BERT network is divided into two main parts:

1. Output of words (Word Embeddings): At first, each word or input token is assigned a meaningful pair of embedding vectors. For example, for a sentence with 10 words, the output of the BERT network will contain 10 embedding vectors of size 768.
2. Sentence output (Sentence Embedding): After processing the sentence and combining the word embedding vectors, a general vector for the sentence is produced, which is often known as the sentence embedding vector. This vector is the output of the pooler layer in the BERT network, which is used as a general representation of the sentence.

For example, for an input sentence, the output of the BERT network will consist of a sentence embedding vector of size 768, which can be used as a general representation of the sentence concept.

Depending on the type of input task, the output of the BERT network may be used differently. For example, in a text classification task, sentence output may be used to provide input features to a classification algorithm such as SVM.

In general, the output of the BERT network as a representative of the sentence concept can be used as input features for various NLP tasks such as text classification, question answering, and text generation.

After extracting the required features using the BERT network, a classifier can be used to categorize the text. For this, we first need to transform the training and testing data into feature vectors. Then, using a classification algorithm such as SVM or neural network, we can classify the text.

For example, suppose we want to create a text classification model to classify Persian texts. First, we extract the required features from the training and test data using the BERT network. Then, using these features and a classification algorithm, such as SVM or logistic regression, we create a text classification model. Finally, using this model, we can perform text classification on the test data.

In the following code, to create a logistic regression classification model for classifying Persian texts using the features extracted with the BERT network, first the training and

test data are converted into a feature vector, and then a logistic regression model is created using the scikit-learn library. we do.

```python
from sklearn.linear_model import LogisticRegression

# Initialize a logistic regression classifier
clf = LogisticRegression()

# Train the classifier on the training data
clf.fit(bert_outputs_train, y_train)
```

The given code initializes a logistic regression classifier and using the features extracted from the BERT model and the corresponding labels are given to the logistic regression as y_train.

Logistic regression is a classification algorithm often used in machine learning for binary classification problems. It models the probability of a binary response variable given the values of one or more predictor variables. In this case, the logistic regression classifier is used to predict the class labels for the given input data based on the features extracted from the BERT model.

Initializes the LogisticRegression function with default settings. The fit method is then called on the classifier object to train the model on the training data. It takes features extracted from BERT model as input and corresponding labels as output to learn the relationship between input and output data.

If the goal is to perform binary classification on the input data, logistic regression is a good choice due to its simplicity and interpretability. However, if the problem involves more than two classes, other classifiers such as support vector machines (SVM) or neural networks may be more appropriate.

```python
# Use the trained logistic regression classifier to predict the probabilities of the class labels

# Predict the probabilities for the training data
h_train = clf.predict_proba(bert_outputs_train)

# Predict the probabilities for the validation data
h_valid = clf.predict_proba(bert_outputs_valid)

# Predict the probabilities for the test data
h_test = clf.predict_proba(bert_outputs_test)
```

The given code uses a trained logistic regression classifier (CLF) to predict probabilities of class labels for training, validation and testing data. The predicted probabilities are stored in the variables h_train, h_valid and h_test for the respective data.

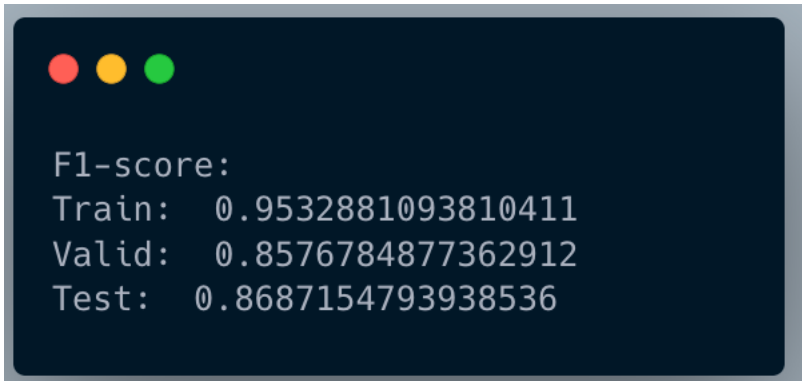
The predict_proba method of the logistic regression classifier is used to predict the probabilities of the class labels for a given input. It takes input data as input and returns predicted probabilities of class labels as output.

Predicted probabilities can be useful in cases where we want to know the confidence of the model in its predictions. For example, if the predicted probability for a particular class is high, we can be more confident in the model's prediction for that class.

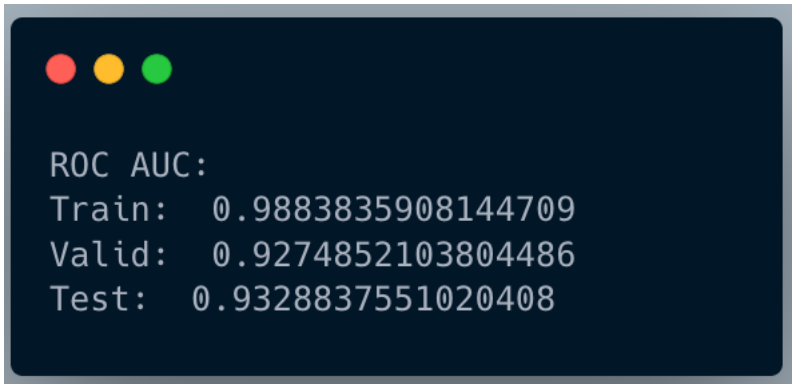
```
print('Accuracy:')

print('Train: ', accuracy_score(y_train, h_train[:, 1] > 0.5))
print('Valid: ', accuracy_score(y_valid, h_valid[:, 1] > 0.5))
print('Test: ', accuracy_score(y_test, h_test[:, 1] > 0.5))
```

```
Accuracy:
Train:  0.952896560279289
Valid:  0.8553968253968254
Test:  0.8675714285714285
```

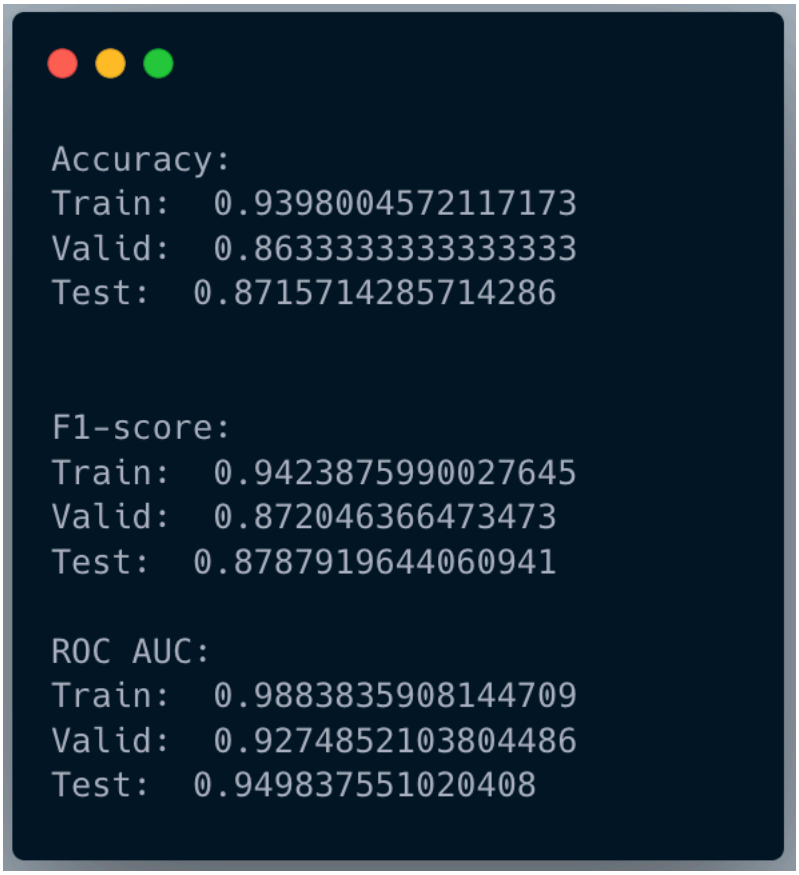
```
print('F1-score:')

print('Train: ', f1_score(y_train, h_train[:, 1] > 0.5))
print('Valid: ', f1_score(y_valid, h_valid[:, 1] > 0.5))
print('Test: ', f1_score(y_test, h_test[:, 1] > 0.5))
```




```
print('AUPRC-score:')

print('Train: ', auprc_score(y_train, h_train[:, 1] > 0.5))
print('Valid: ', auprc_score(y_valid, h_valid[:, 1] > 0.5))
print('Test: ', auprc_score(y_test, h_test[:, 1] > 0.5))
```



After trial and error and setting different values for the threshold, we concluded that the value of 0.18 gives the best result as shown in the following results.



The performance of the model has been evaluated using accuracy, F1 score and AUPRC score in training, validation and test sets.

The accuracy in the training set was high at 0.9529, which indicates that the model is able to correctly predict most of the examples in the training set. Accuracy on the validation set was slightly lower at 0.8554, but still reasonably high, indicating that the model generalizes well to new samples. The accuracy on the test set was 0.8676, which is similar to the accuracy on the validation set and shows that the model works consistently on unseen data.

F1-score is a weighted average of accuracy and recall and is a more informative measure when dealing with unbalanced data sets. The F1-score in the training set was 0.9533, which shows that the model has a good balance of precision and recall in the training set. The F1-score in the validation set was 0.8577, which indicates that the model performs well in the new samples.

The F1-score in the test set was 0.8687, which is similar to the F1-score of the validation set, indicating that the model works adequately on unseen data.

The AUPRC score is a metric that measures the area under the curve. The AUPRC score on the training set was high at 0.9626, indicating that the model has high accuracy on the training set. The AUPRC score in the validation set was 0.8900, which indicates that the model performs well in new samples. The AUPRC score on the test set was 0.8997, which is similar to the AUPRC validation set score, indicating that the model works consistently on unseen data.

Overall, the model performs reasonably well based on these criteria, with high accuracy and AUPRC score on the training set and reasonable generalization to new samples on the test and validation sets.

In the field of deep learning, neural networks are also used as one of the powerful tools to solve classification and classification problems. A neural network analyzes the input data using learning algorithms and produces the result as an output vector. In classification problems, some vectors are introduced as labels and the neural network must be able to predict the corresponding label from the input vector.

For this reason, neural networks implemented with the help of keras have been used in this section.

The neural network used is as follows.

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
from tensorflow.keras.metrics import Precision, Recall, AUC

# Define additional metrics
def f1_score(y_true, y_pred):
    tp = keras.backend.sum(keras.backend.round(keras.backend.clip(y_true * y_pred, 0, 1)))
    fp = keras.backend.sum(keras.backend.round(keras.backend.clip(y_pred - y_true, 0, 1)))
    fn = keras.backend.sum(keras.backend.round(keras.backend.clip(y_true - y_pred, 0, 1)))
    precision = tp / (tp + fp + keras.backend.epsilon())
    recall = tp / (tp + fn + keras.backend.epsilon())
    f1_score = 2 * precision * recall / (precision + recall + keras.backend.epsilon())
    return f1_score

# Define the neural network
model = keras.Sequential([
    keras.layers.Dense(64, input_dim=768, activation='relu'),
    keras.layers.Dense(32, activation='relu'),
    keras.layers.Dense(8, activation='relu'),
    keras.layers.Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='sgd', metrics=['accuracy', Precision(), Recall(),
AUC(name='auprc'), f1_score])
# Train the model
batch_size = 16
num_epochs = 5
history = model.fit(bert_outputs_train, y_train, batch_size=batch_size, epochs=num_epochs, verbose=1,
validation_data=(bert_outputs_valid, y_valid))
```

This code is a neural network for classifying binary data trained by the output of a BERT model. This neural network consists of four layers, three hidden layers and one output layer. The hidden layers have 64, 32 and 8 neurons with ReLU actuator, respectively. This network is set using binary_crossentropy cost function and sgd optimizer in the compile stage of the network. Also, to evaluate the performance of the network, accuracy, precision, recall, AUC and F1-score criteria are considered. Training data, validation and evaluation criteria for this network are prepared in tensor format from TensorFlow. Finally, using the fit function, the network is trained on the training data. The results of different criteria during the training are displayed in the figure below.

```
Epoch 1/5
3527/3527 [==============================] - 24s 6ms/step - loss: 0.1520 - accuracy: 0.9461 - precision_2: 0.9349 - recall_2: 0.9588 -
auprc: 0.9841 - f1_score: 0.9429 - val_loss: 0.3756 - val_accuracy: 0.8606 - val_precision_2: 0.8483 - val_recall_2: 0.8784 -
val_auprc: 0.9318 - val_f1_score: 0.8540
Epoch 2/5
3527/3527 [==============================] - 21s 6ms/step - loss: 0.1424 - accuracy: 0.9481 - precision_2: 0.9383 - recall_2: 0.9591 -
auprc: 0.9858 - f1_score: 0.9451 - val_loss: 0.3884 - val_accuracy: 0.8559 - val_precision_2: 0.8584 - val_recall_2: 0.8524 -
val_auprc: 0.9330 - val_f1_score: 0.8459
Epoch 3/5
3527/3527 [==============================] - 23s 6ms/step - loss: 0.1409 - accuracy: 0.9493 - precision_2: 0.9399 - recall_2: 0.9598 -
auprc: 0.9862 - f1_score: 0.9464 - val_loss: 0.3928 - val_accuracy: 0.8589 - val_precision_2: 0.8484 - val_recall_2: 0.8740 -
val_auprc: 0.9317 - val_f1_score: 0.8510
Epoch 4/5
3527/3527 [==============================] - 19s 6ms/step - loss: 0.1395 - accuracy: 0.9494 - precision_2: 0.9392 - recall_2: 0.9607 -
auprc: 0.9865 - f1_score: 0.9469 - val_loss: 0.3968 - val_accuracy: 0.8603 - val_precision_2: 0.8475 - val_recall_2: 0.8787 -
val_auprc: 0.9314 - val_f1_score: 0.8539
Epoch 5/5
3527/3527 [==============================] - 19s 5ms/step - loss: 0.1385 - accuracy: 0.9497 - precision_2: 0.9394 - recall_2: 0.9611 -
auprc: 0.9865 - f1_score: 0.9469 - val_loss: 0.4072 - val_accuracy: 0.8598 - val_precision_2: 0.8482 - val_recall_2: 0.8765 -
val_auprc: 0.9312 - val_f1_score: 0.8528
[69]
0s
```

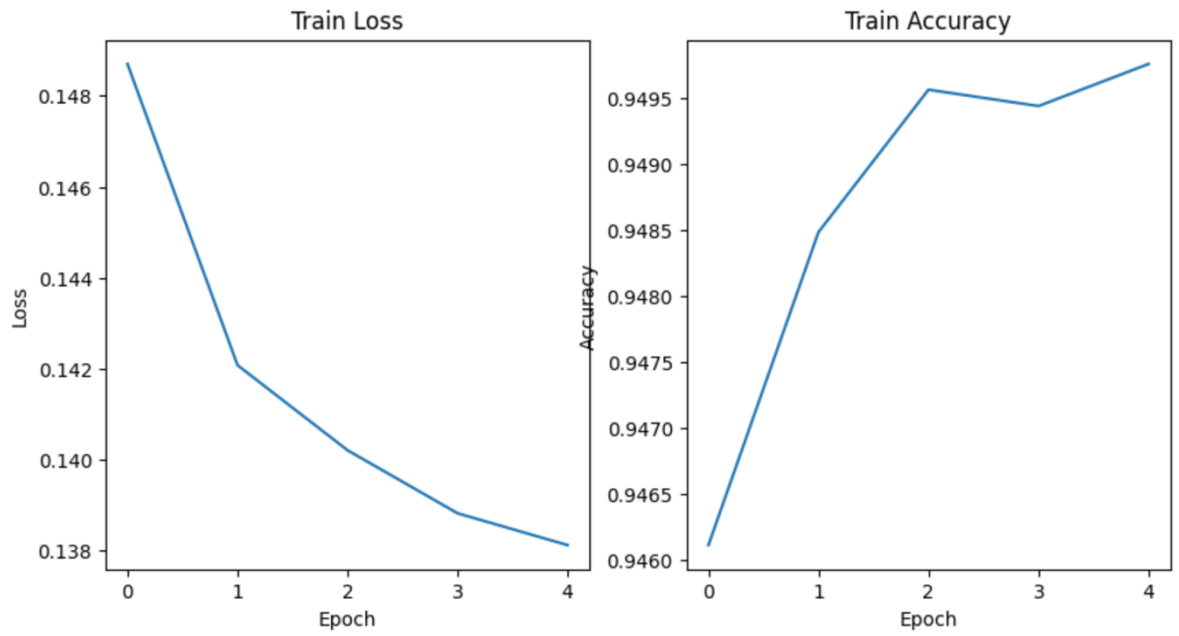
These results show that the neural network performed well during training with the training and validation data. In each of the five training periods, the neural network has improved with increasing accuracy and other measures such as precision, recall, AUC and F1-score. Also, with the increase in the number of training sessions, the difference between the accuracy in the training and validation data has decreased, which indicates a better model and less affected by overfitting. In general, the results are very acceptable, showing that the neural network can classify binary data well.

The result of this network on the experimental data is as follows.

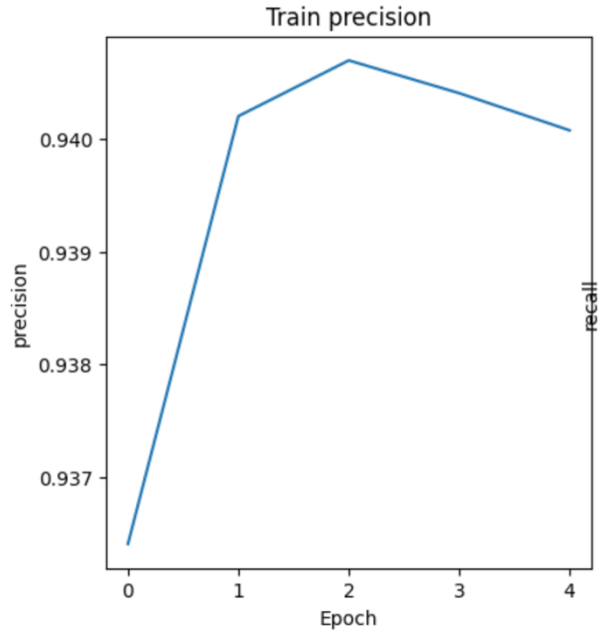
```
Test Loss = 0.3772331178188324, Test Accuracy = 0.8738571405410767
Percision = 0.863775372505188, Recall = 0.8877142667770386
AUPRC = 0.9379653930664062, F1-score = 0.8732903599739075
```

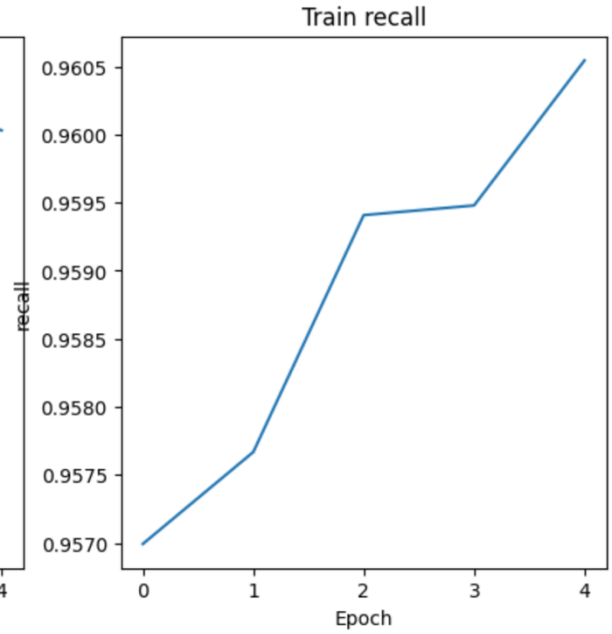
This result shows that the neural network performed well during training with test data as well. The value of Test Loss is low and Test Accuracy is high, which shows that the neural network has classified the test data well. Also, the precision and recall values are high for the positive and negative categories. AUPRC value also indicates good performance for the model. Also, the value of F1-score also indicates the balance between accuracy and recovery, and although it is slightly lower than accuracy and recovery, it still has a value higher than 0.5, which indicates that the neural network has well classified the test data. In general, the results are very acceptable, showing that the neural network can classify binary data well.

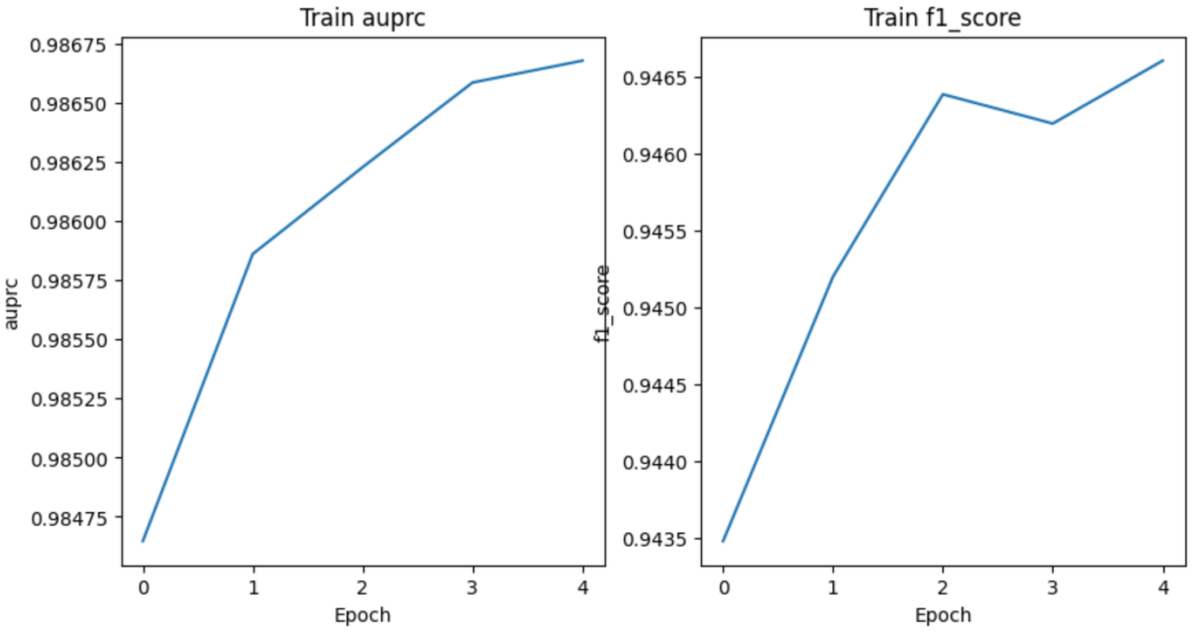
The diagrams of the model are as follows.


Train Loss
Loss
0.148
0.146
0.144
0.142
0.140
0.138
0
1
2
3
4
Epoch

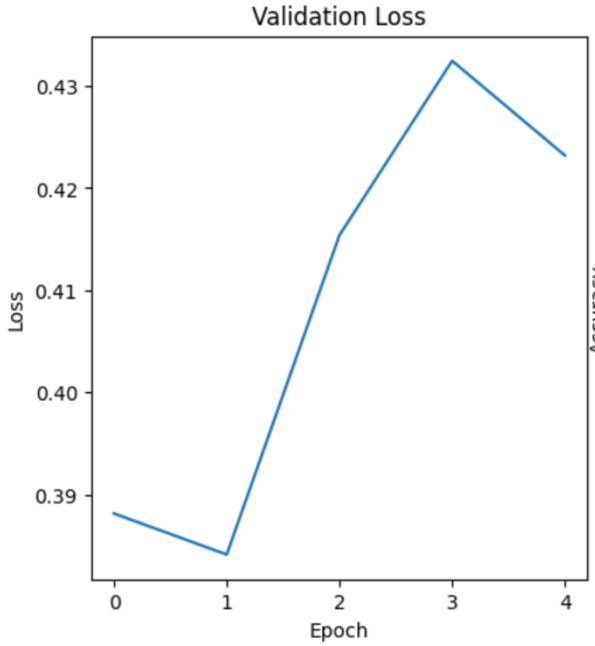
Train Accuracy
Accuracy
0.9495
0.9490
0.9485
0.9480
0.9475
0.9470
0.9465
0.9460
0
1
2
3
4
Epoch

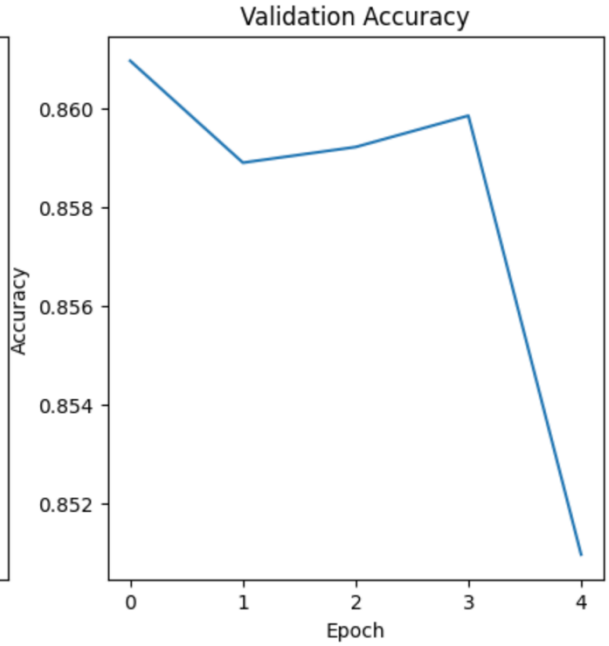

Train precision
precision
0.940
0.939
0.938
0.937
0
1
2
3
4
Epoch

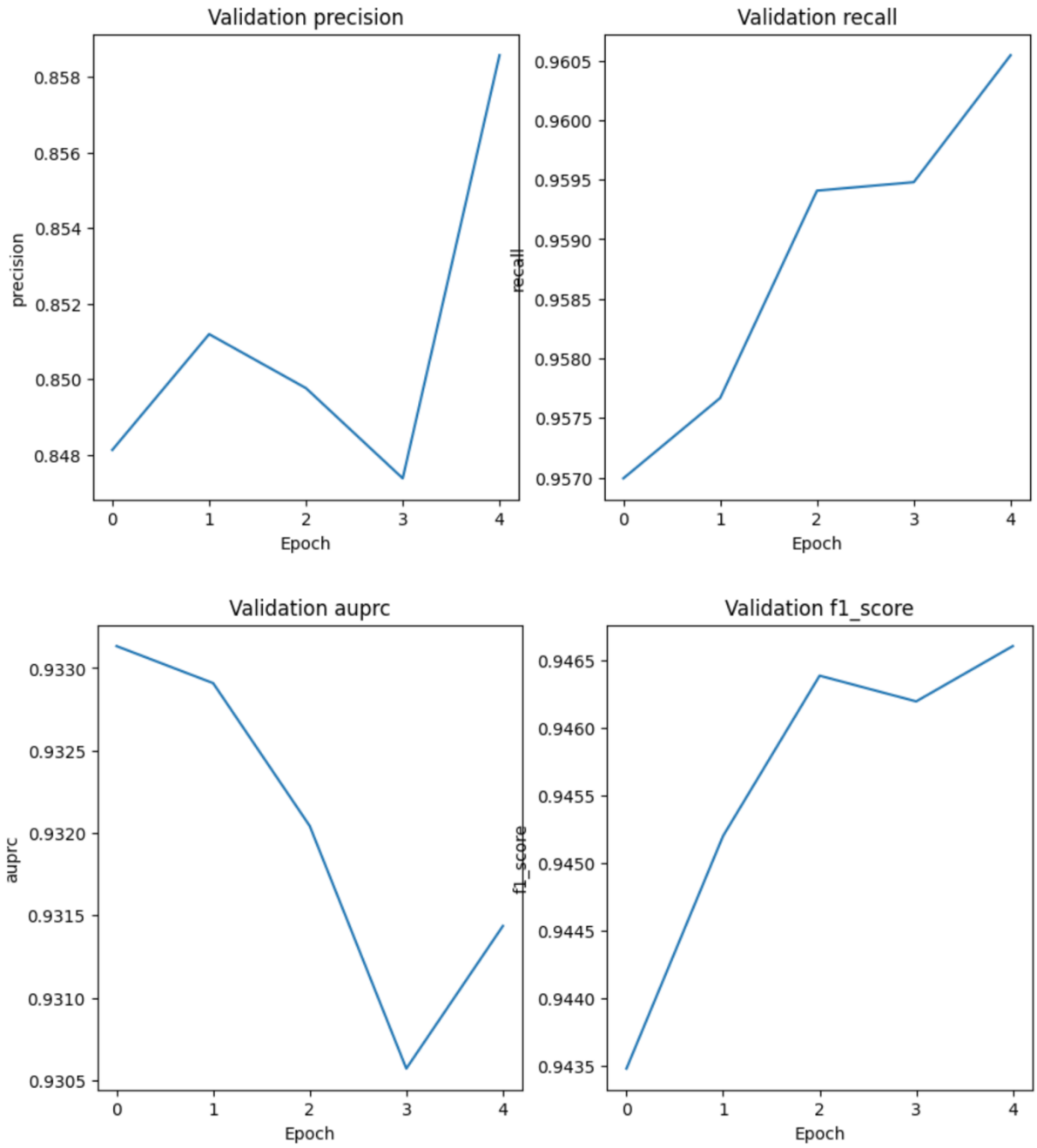

Train recall
recall
0.9605
0.9600
0.9595
0.9590
0.9585
0.9580
0.9575
0.9570
0
1
2
3
4
Epoch


Train auprc
auprc
0.98675
0.98650
0.98625
0.98600
0.98575
0.98550
0.98525
0.98500
0.98475
0
1
2
3
4
Epoch

Train f1_score
f1_score
0.9465
0.9460
0.9455
0.9450
0.9445
0.9440
0.9435
0
1
2
3
4
Epoch


Validation Loss
Loss
0.43
0.42
0.41
0.40
0.39
0
1
2
3
4
Epoch


Validation Accuracy
Accuracy
0.860
0.858
0.856
0.854
0.852
0
1
2
3
4
Epoch



Model improvement

In this paragraph, it is stated that two main methods are used to improve the performance of the model: augmenting the data and fine-tuning the model.

The first method to improve the performance of the model is to augment the data. This method is used to increase the number of training data and increase the variety of data. In this method, new data is created by using various techniques such as applying small changes to the data. This can help improve the accuracy and performance of the model, especially in cases where the number of training data is small or the training data is uneven.

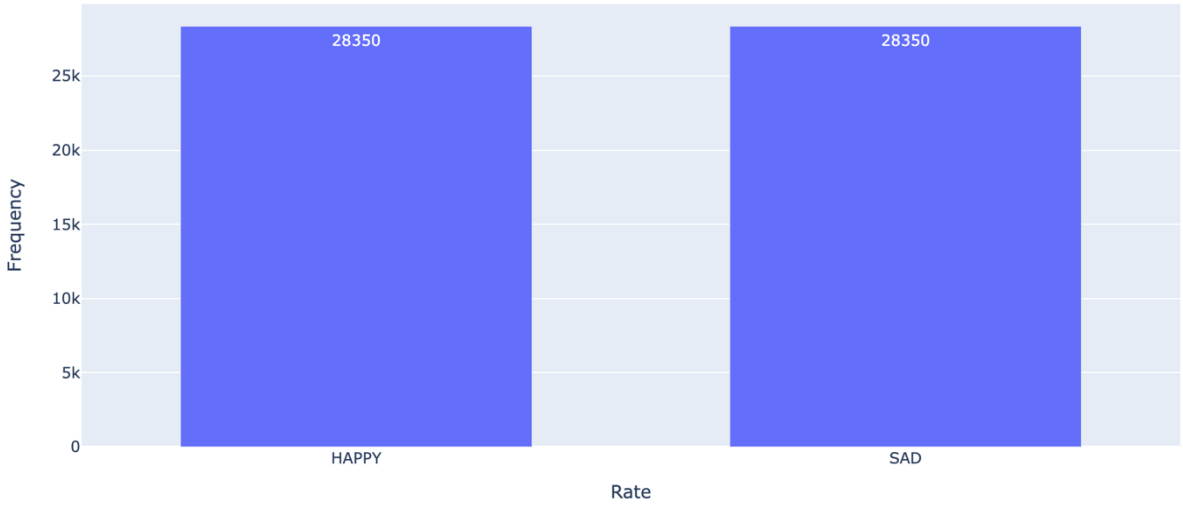
The second method to improve the performance of the model is to fine-tune the model. This method is used to improve the performance of the model using the available training data. In this method, by optimizing the model parameters, the performance of the model is improved. This can help improve the accuracy and performance of the model in cases where the training data is sufficient.

In summary, in this paragraph, two main methods to improve the performance of the model, i.e. data augmentation and model fine-tuning, are introduced. Both of these methods can help improve the accuracy and performance of the model in different cases.

Augment the data

```python
import pandas as pd
import persiannlpaug.augmenter.word as naw

# Define a function to augment text in a data frame
def augment_text(data):
    # Create a copy of the data frame
    augmented_df = data.copy()

    # Augment text in the data frame using various techniques
    for i in range(0, len(data['comment']), 1000):
        # Replace words in the text with contextually similar words
        aug = naw.ContextualWordEmbsAug(model_path='HooshvareLab/bert-fa-base-uncased', action="substitute")
        sentence = data['comment'][i]
        augmented_text = aug.augment(sentence)
        new_data = {'comment': augmented_text, 'label': data['label'][i], 'label_id': data['label_id'][i]}
        augmented_df = augmented_df.append(new_data, ignore_index=True)

        # Insert new words into the text
        aug = naw.ContextualWordEmbsAug(model_path='HooshvareLab/bert-fa-base-uncased', action="insert")
        sentence = data['comment'][i]
        augmented_text = aug.augment(sentence)
        new_data = {'comment': augmented_text, 'label': data['label'][i], 'label_id': data['label_id'][i]}
        augmented_df = augmented_df.append(new_data, ignore_index=True)

    return augmented_df

# Load data frames
train_df = pd.read_csv('train.csv')
valid_df = pd.read_csv('valid.csv')

# Augment text in the data frames
train_df = augment_text(train_df)
valid_df = augment_text(valid_df)
```

The given code defines a function called augment_text that takes a Pandas DataFrame as input and returns an augmented DataFrame with additional rows of text data. This function uses the Persian NLP Augmentation library to apply various data augmentation techniques to the input textual data. Then the augmented DataFrame is used for training and validation.



The data specifications of each class become the same after augmenting and we use it.

Fine-tune model

```
from transformers.pipelines import zero_shot_classification
class SentimentModel(nn.Module):
    def __init__(self, config):
        super().__init__()

        self.bert = BertModel.from_pretrained(MODEL_NAME_OR_PATH)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.classifier = nn.Linear(config.hidden_size, 1)

    def forward(self, input_ids, attention_mask, token_type_ids):
        z = self.bert(
            input_ids=input_ids,
            attention_mask=attention_mask,
            token_type_ids=token_type_ids)

        pooled_output = self.dropout(z['pooler_output'])
        out = self.classifier(pooled_output)
        return out
```

The given code defines a PyTorch neural network model called SentimentModel that uses the BERT model architecture for sentiment analysis. This model takes input identifiers, attention masks, and token type identifiers as input and returns a single output value representing the sentiment score of the input text. The model architecture consists of several layers of linear and fully connected layers.

```python
import torch.nn as nn
from tqdm import tqdm

def train_model(model, dataloaders, criterion, optimizer, epochs, valid_steps=1000):
    train_history = {'loss': [], 'accuracy': [], 'f1-score': [], 'auprc': []}
    valid_history = {'loss': [], 'accuracy': [], 'f1-score': [], 'auprc': []}

    for e in range(epochs):
        sum_loss_train = 0
        sum_acc_train = 0
        sum_f1_train = 0
        sum_roc_train = 0
        train_steps = 0
        model.train()
        for batch in tqdm(dataloaders['train']):
            # Get input data and labels
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            token_type_ids = batch['token_type_ids'].to(device)
            label = batch['label'].to(device).view(-1, 1).float()

            # Zero the gradients and perform forward and backward passes
            optimizer.zero_grad()
            output = model(input_ids, attention_mask, token_type_ids)
            loss = criterion(output, label)
            loss.backward()
            optimizer.step()

            # Update training metrics
            sum_loss_train += loss.item()
            output = nn.functional.sigmoid(output.detach())
            metrics = score_model(label.cpu().numpy(), output.cpu().numpy())
            sum_acc_train += metrics['acc']
            sum_f1_train += metrics['f1-score']
            sum_roc_train += metrics['auprc']
            train_steps += 1

            # Perform validation after specified number of training steps
            if train_steps % valid_steps == 0:
                # Initialize validation metrics
                sum_loss_valid = 0
                sum_acc_valid = 0
                sum_f1_valid = 0
                sum_roc_valid = 0
                validation_steps = 0
                model.eval()

                # Loop over validation data and update validation metrics
                for batch in tqdm(dataloaders['valid']):
                    input_ids = batch['input_ids'].to(device)
                    attention_mask = batch['attention_mask'].to(device)
                    token_type_ids = batch['token_type_ids'].to(device)
                    label = batch['label'].to(device).view(-1, 1).float()

                    with torch.no_grad():
                        output = model(input_ids, attention_mask, token_type_ids)
                        loss = criterion(output, label)

                        sum_loss_valid += loss.item()
                        output = nn.functional.sigmoid(output)
                        metrics = score_model(label.cpu().numpy(), output.cpu().numpy())
                        sum_acc_valid += metrics['acc']
                        sum_f1_valid += metrics['f1-score']
                        sum_roc_valid += metrics['auprc']
                        validation_steps += 1

                # Update training and validation history
                train_history['loss'].append(sum_loss_train / train_steps)
                train_history['accuracy'].append(sum_acc_train / train_steps)
                train_history['f1-score'].append(sum_f1_train / train_steps)
                train_history['auprc'].append(sum_roc_train / train_steps)

                valid_history['loss'].append(sum_loss_valid / validation_steps)
                valid_history['accuracy'].append(sum_acc_valid / validation_steps)
                valid_history['f1-score'].append(sum_f1_valid / validation_steps)
                valid_history['auprc'].append(sum_roc_valid / validation_steps)

                # Reset training metrics
                sum_loss_train = 0
                sum_acc_train = 0
                sum_f1_train = 0
                sum_roc_train = 0
                train_steps = 0

    return model, train_history, valid_history
```

The given code defines a function called train_model that trains a PyTorch neural network model on a dataset using the specified hyperparameters. The function takes the following arguments:

- Model: PyTorch neural network model for training
- dataloaders: A dictionary of PyTorch data loaders for training and validation datasets
- Criterion: Loss function to use for model training
- Optimizer: Optimizer for updating model weights
- Courses: The number of courses to train the model for
- valid_steps: the number of training steps that must be performed before the validation step is executed.

This function returns the trained model as well as the training and validation history in the form of dictionaries containing loss, accuracy, F1 score and AUPRC for each epoch.

Fine-tuning means training the model using new data. In cases where we want to use a previously trained model in order to solve a specific problem, fine-tuning is used as an effective method in reusing these models.

Fine-tuning for training natural language processing models such as BERT is done in two steps:

1. Training the model with new training data: In this step, the BERT model is trained with new training data. This new data may contain a large number of its own texts and require retraining the model using them.
2. Adjust the model weights: In this step, the BERT model weights are adjusted by the new data. In fact, this step means updating the weights of the model so that it performs best in solving the specific problem.

The reason for having two stages in fine-tuning is because in this method, the model is trained on new training data, and after training, the model weights are updated using the training data. This is because natural language processing models such as BERT, due to the large amount of training data and their high number of parameters, are well trained in recognizing and understanding linguistic differences, and can be improved by training only a few times on new data. Attention will be seen in their performance.

First stage:

```
# Part1, training classifier

dataloders = {
    'train': train_dataloder,
    'valid': valid_dataloder
}

criterion = nn.BCEWithLogitsLoss()
sf_model.bert.requires_grad_ = False
optimizer = torch.optim.Adam(sf_model.classifier.parameters() ,lr=0.001)

sf_model, history_train, history_valid = train_model(sf_model, dataloders, criterion, optimizer, epochs=5)
```

The given code trains a classifier using a pre-trained BERT model in PyTorch.

In the first line of code, two data loaders for training and validation datasets are combined in a dictionary called dataloaders.

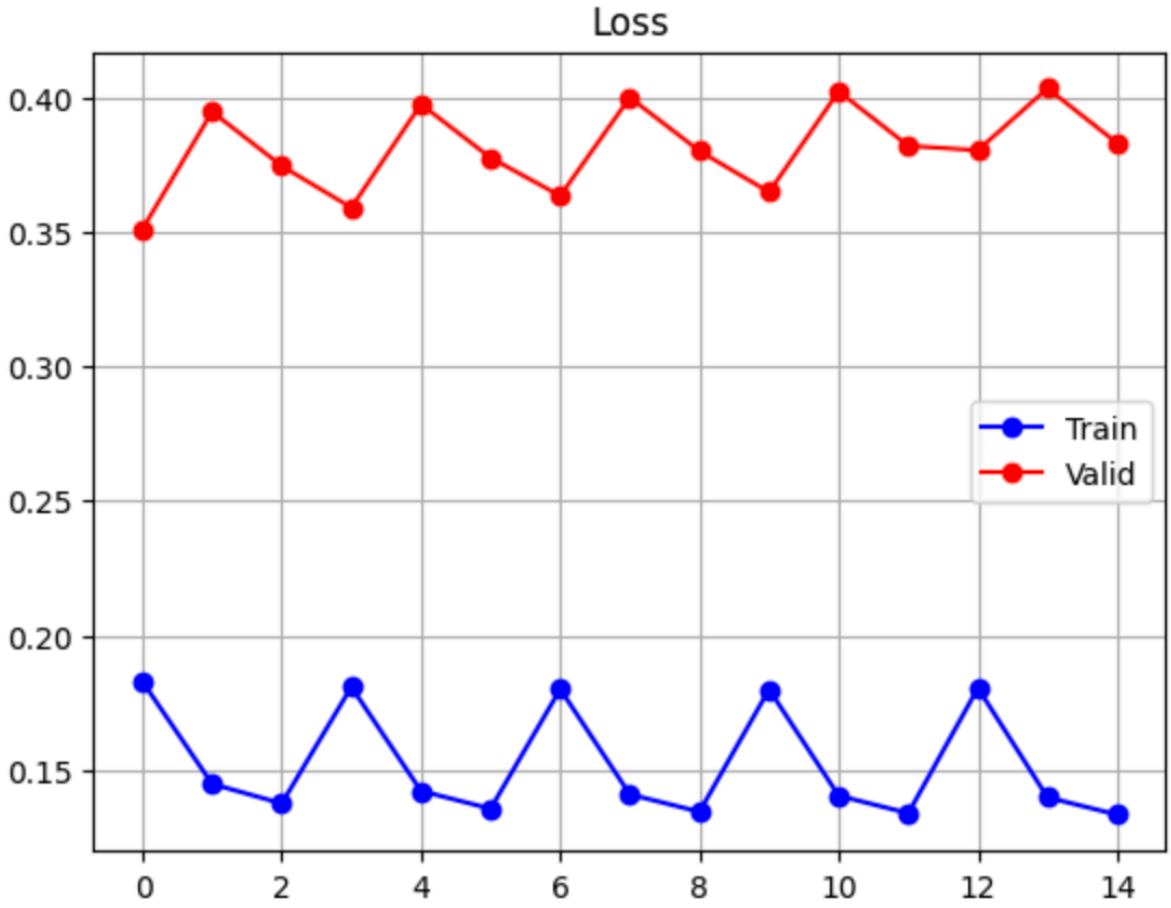
In the second line, an instance of the nn.BCEWithLogitsLoss class is created, which is a binary cross-entropy loss. It is a common loss function used for binary classification problems.

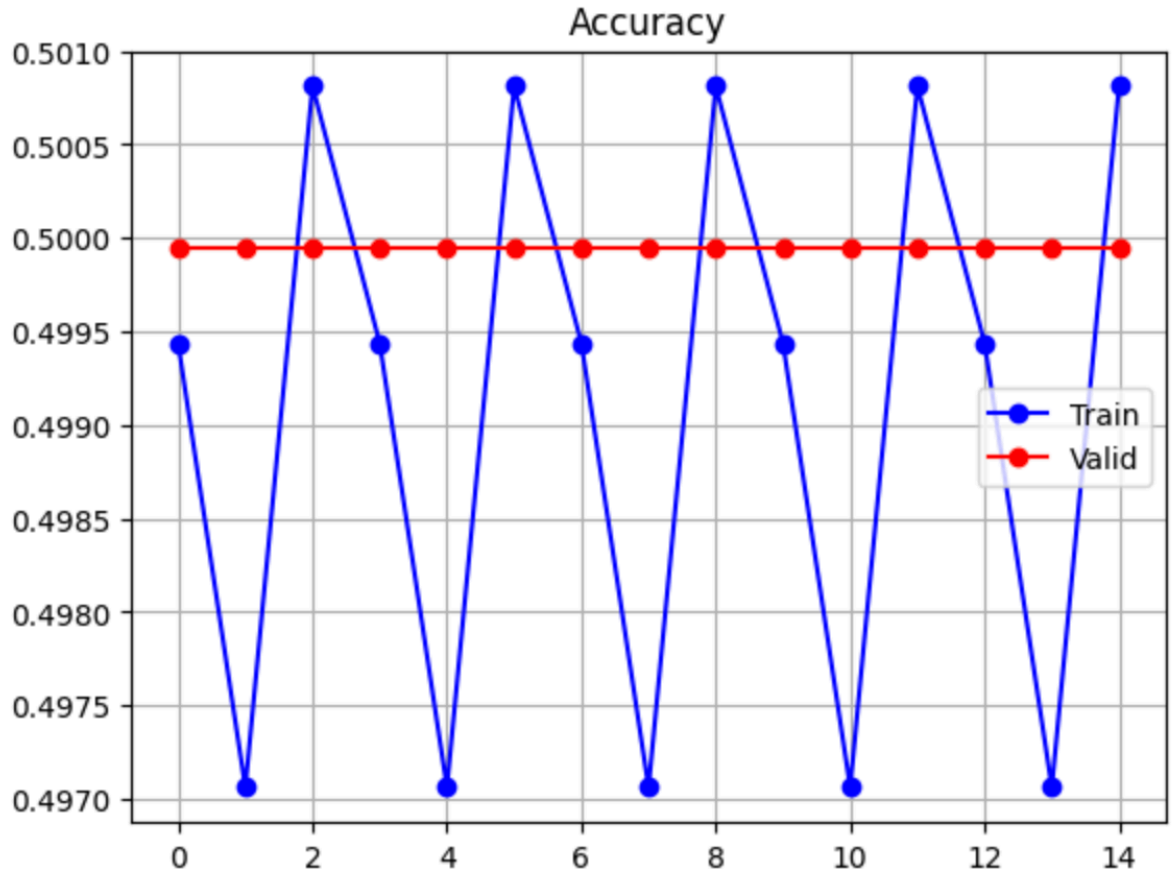
In the third line, the requires_grad_ property of the BERT model is set to False. This freezes the BERT model weights so that they are not updated during training, and only the classifier weights are updated. This is a common technique called transfer learning, where a pre-trained model is used as a starting point for a new task.

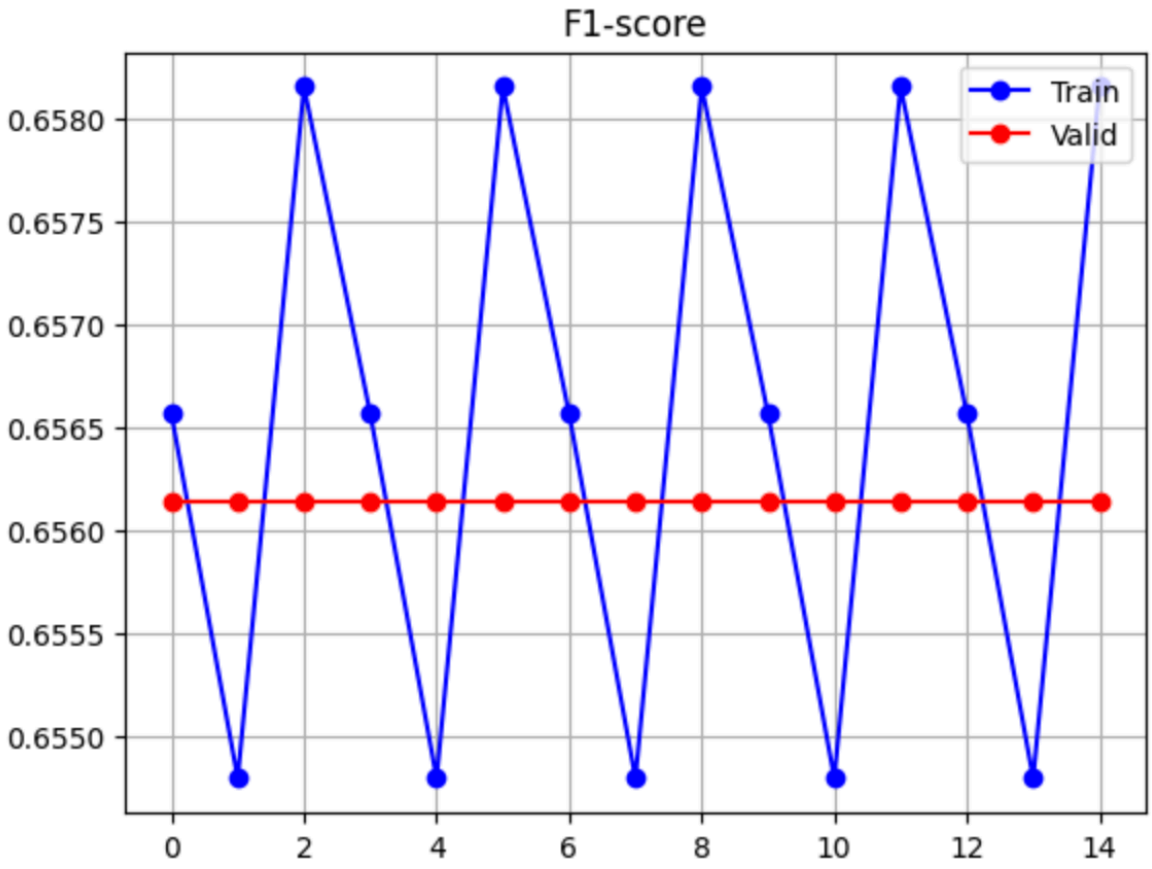
In the fourth line, an instance of the torch.optim.Adam optimizer is created, which is an adaptive learning rate optimizer. The lr parameter specifies the learning rate for the optimizer.

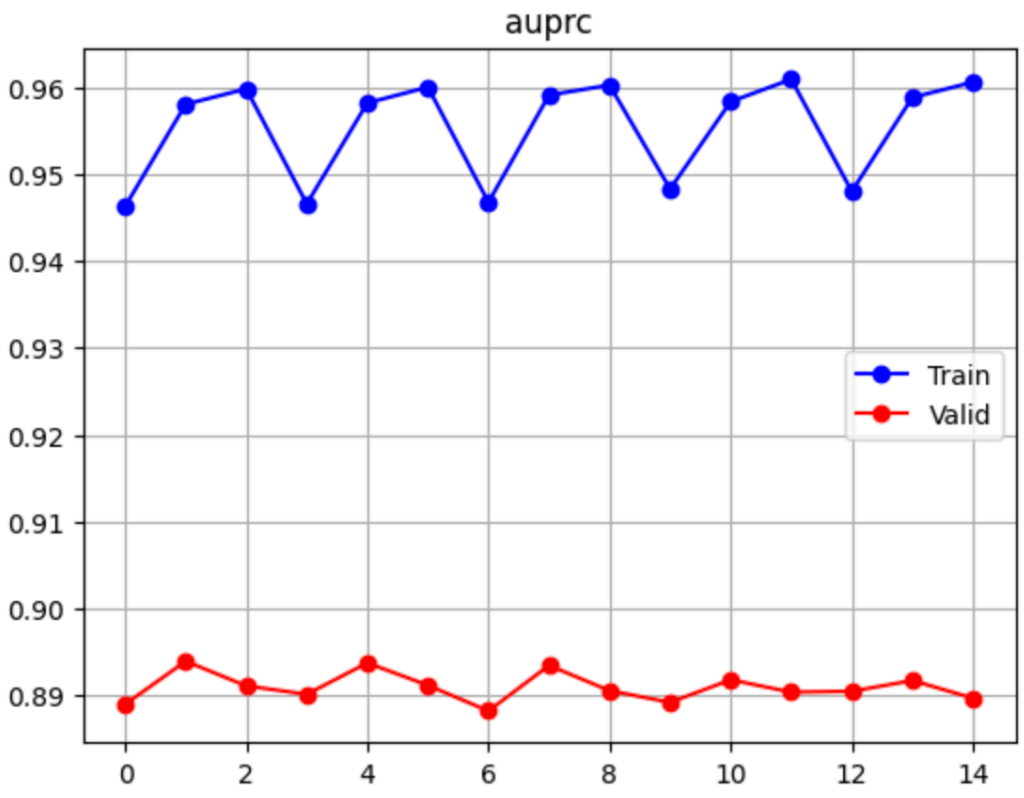
In the fifth line, the train_model() function is called with the pre-trained BERT model (sf_model), data loaders (dataloader), loss function (criterion), optimizer (optimizer) and number of cycles (cycles). =3). This function returns the trained model (sf_model) as well as the history of training and validation in the form of a dictionary (history_train and history_valid.(

The resulting graphs are as follows.


Loss
0.40
0.35
0.30
0.25
0.20
0.15
0
2
4
6
8
10
12
14
Train
Valid


Accuracy
0.5010
0.5005
0.5000
0.4995
0.4990
0.4985
0.4980
0.4975
0.4970
0
2
4
6
8
10
12
14
Train
Valid


F1-score
Train
Valid
0.6580
0.6575
0.6570
0.6565
0.6560
0.6555
0.6550
0
2
4
6
8
10
12
14

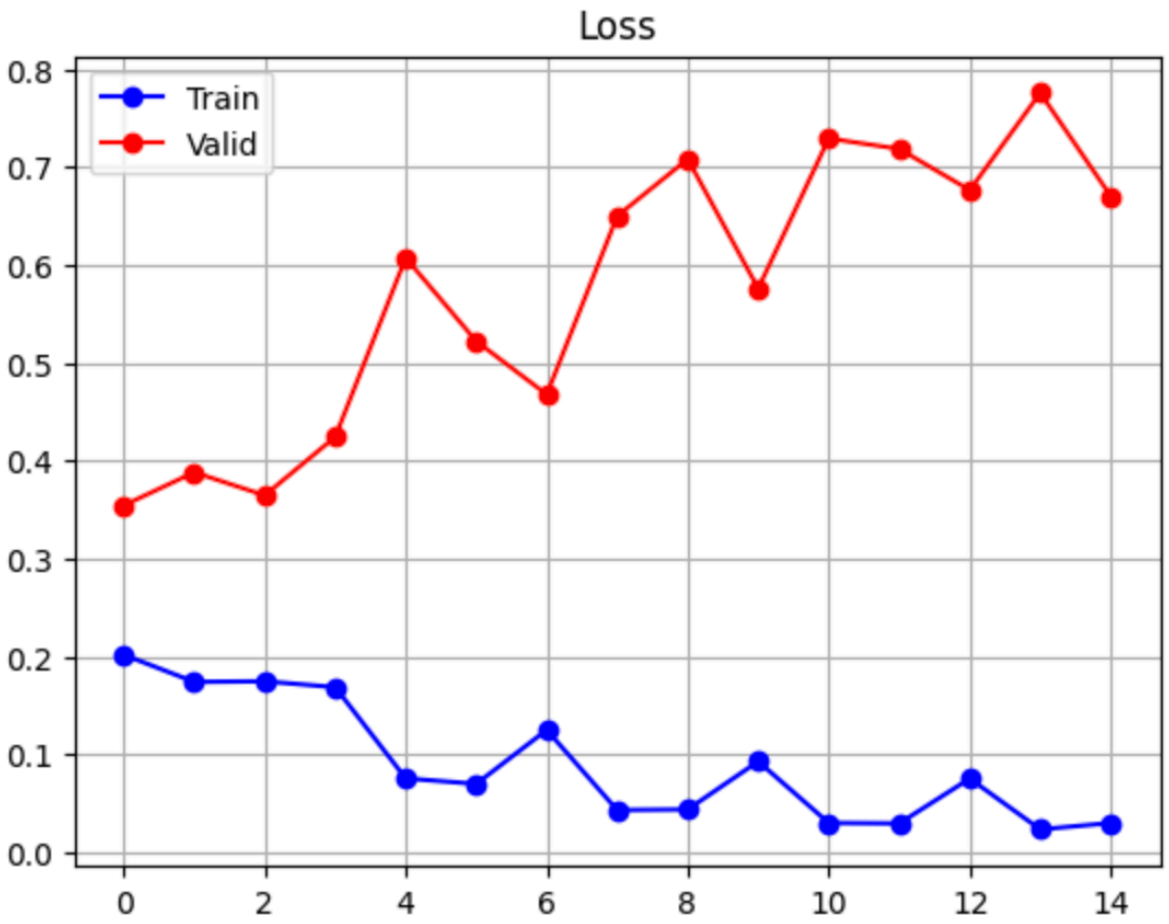

auprc
Train
Valid
0.96
0.95
0.94
0.93
0.92
0.91
0.90
0.89
0
2
4
6
8
10
12
14

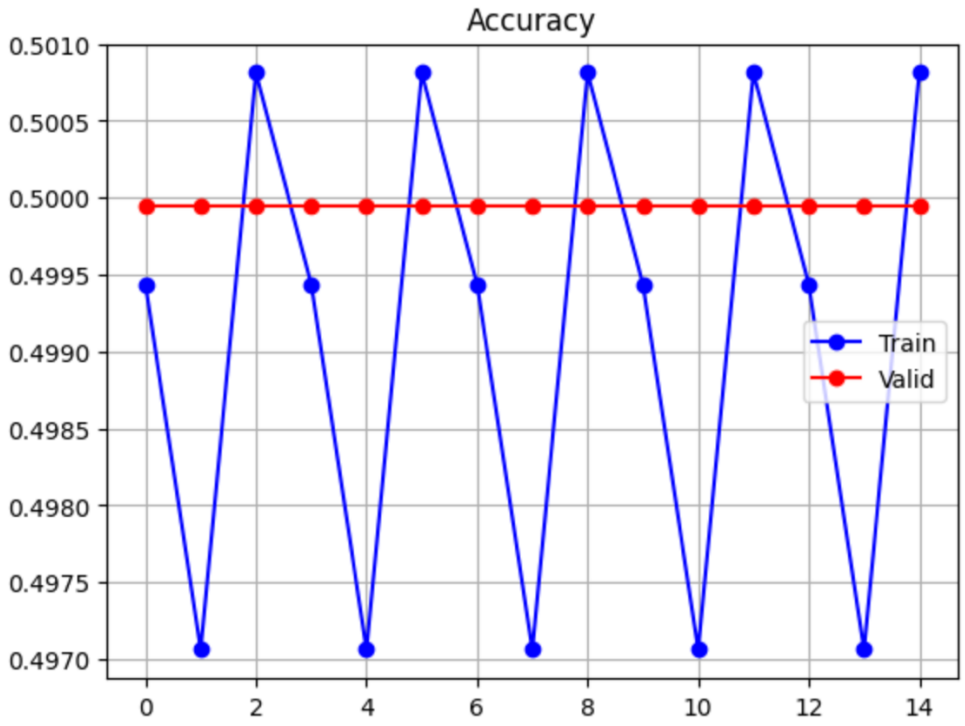
Second Stage

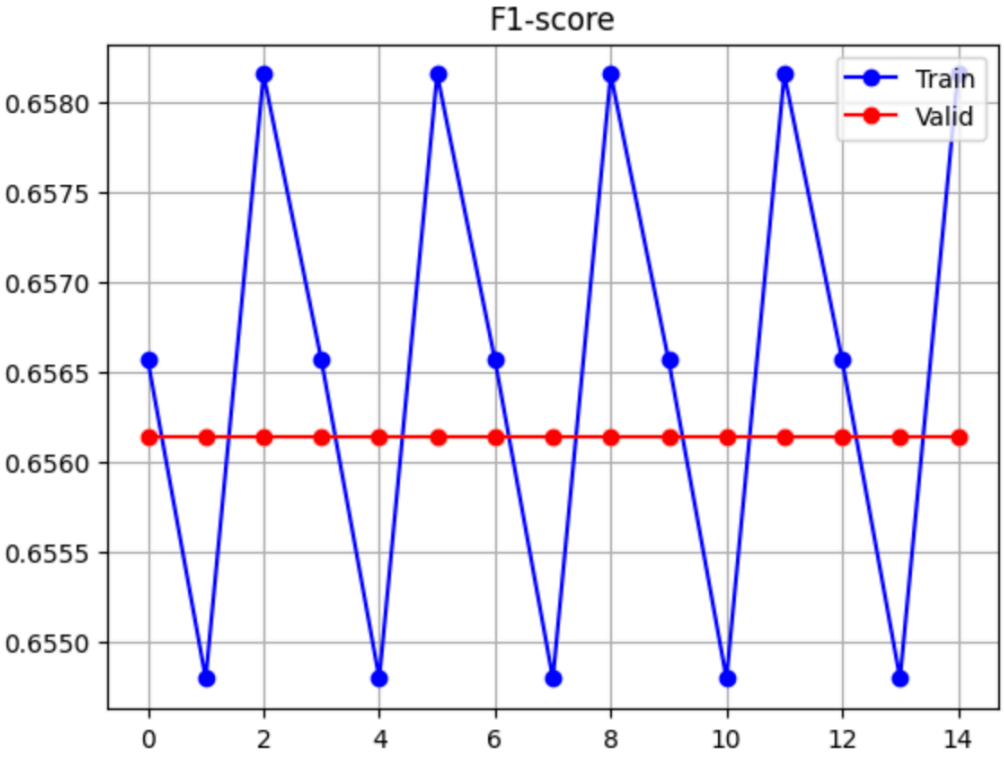
```
# Part2, finetuning

dataloders = {
    'train': train_dataloder,
    'valid': valid_dataloder
}

criterion = nn.BCEWithLogitsLoss()
sf_model.bert.requires_grad_ = True
optimizer = torch.optim.Adam(sf_model.parameters() ,lr=0.00001)

sf_model, history_train, history_valid = train_model(sf_model, dataloders, criterion, optimizer, epochs=5)
```

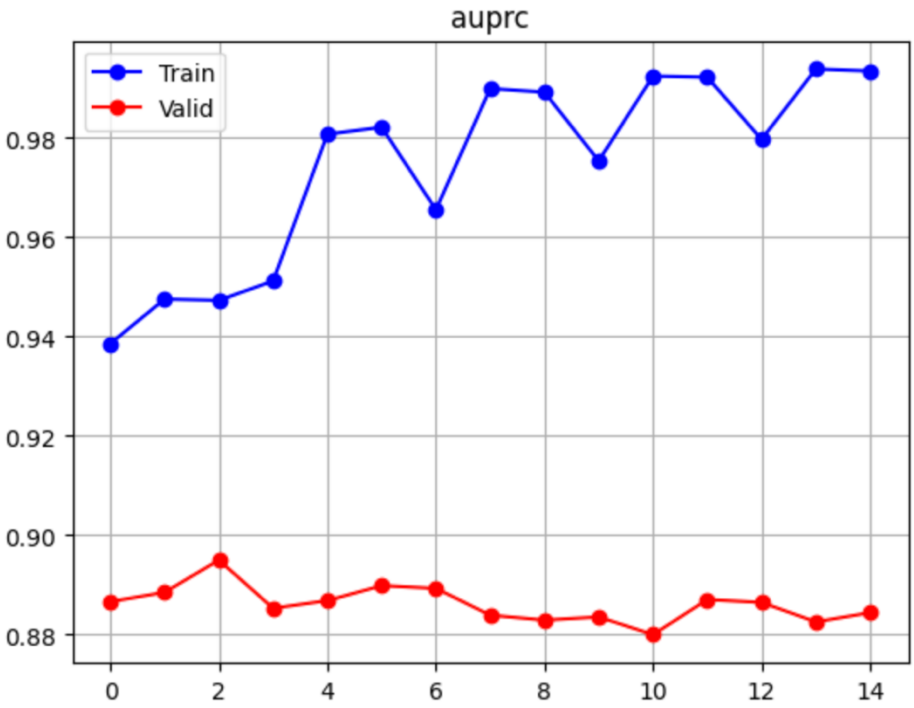
In the second step, we consider requires_grad equal to true, and in fact, the weights of the BERT model can be changed and improved in the training process.

And finally, the following graphs are obtained.




Accuracy
0.5010
0.5005
0.5000
0.4995
0.4990
0.4985
0.4980
0.4975
0.4970
0
2
4
6
8
10
12
14
Train
Valid


F1-score
0.6580
0.6575
0.6570
0.6565
0.6560
0.6555
0.6550
0
2
4
6
8
10
12
14
Train
Valid



Results

Finally, we feed the data to the trained model and extract the features and give the extracted features to the sigmoid and calculate the evaluation criteria.

```
def extractFeature2(data, m):
  bert_outputs = torch.zeros(m, 1)
  i =0
  for batch in data:
    input_ids = batch['input_ids'].to(device)
    attention_mask = batch['attention_mask'].to(device)
    token_type_ids = batch['token_type_ids'].to(device)
    with torch.no_grad():
        out = sf_model(input_ids, attention_mask, token_type_ids).cpu()
    h_train[i: i + out.size(0)] = out
    i += out.size(0)
  return bert_outputs
```

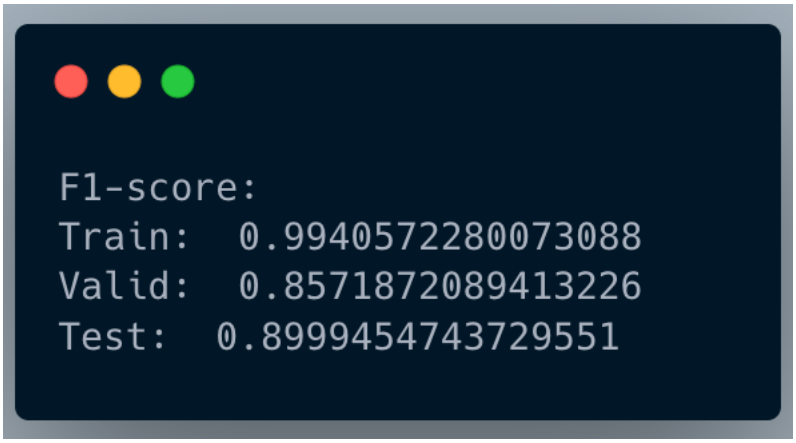
In this function, a zero matrix with dimension m by 1 called "bert_outputs" is defined. Then a loop is created on the input data "data". At each step of the loop, the input data for a set of data is given to the BERT model and the output of the BERT model for that set of data is calculated using the "sf_model" function. The output of the BERT model for that data set is then stored in the "bert_outputs" matrix.

```
h_train = extractFeature2(train_dataloder, len(train_df))
h_valid = extractFeature2(valid_dataloder, len(valid_df))
h_test = extractFeature2(test_dataloder, len(test_df))


h_train = nn.functional.sigmoid(h_train)
h_valid = nn.functional.sigmoid(h_valid)
h_test = nn.functional.sigmoid(h_test)
```

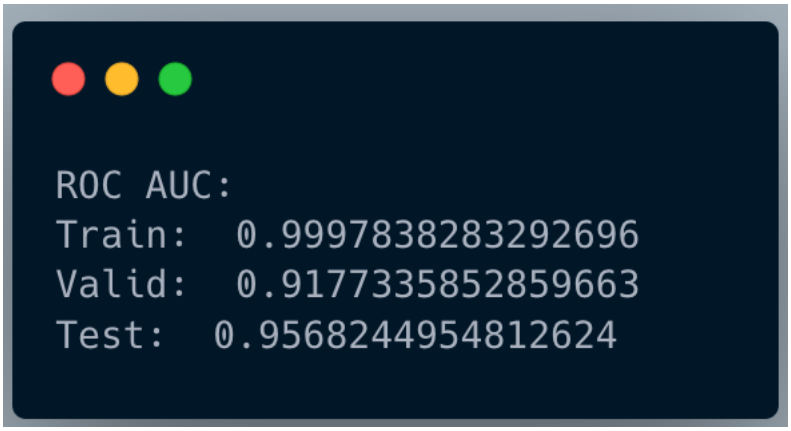
```
print('Accuracy:')

print('Train: ', accuracy_score(y_train, h_train[:, 1] > 0.5))
print('Valid: ', accuracy_score(y_valid, h_valid[:, 1] > 0.5))
print('Test: ', accuracy_score(y_test, h_test[:, 1] > 0.5))
```

```
Accuracy:
Train:  0.9940633362278262
Valid:  0.8539682539682539
Test:  0.8951428571428571
```

```
print('AUPRC-score:')

print('Train: ', auprc_score(y_train, h_train[:, 1] > 0.5))
print('Valid: ', auprc_score(y_valid, h_valid[:, 1] > 0.5))
print('Test: ', auprc_score(y_test, h_test[:, 1] > 0.5))
```




```
print('F1-score:')

print('Train: ', f1_score(y_train, h_train[:, 1] > 0.5))
print('Valid: ', f1_score(y_valid, h_valid[:, 1] > 0.5))
print('Test: ', f1_score(y_test, h_test[:, 1] > 0.5))
```



As it is known, we have improved by 3%.