

# Chapter 5: Fine-Tuning Language Models for Text Classification



In this chapter, we will learn how to configure a pre-trained model for text classification and how to fine-tune it to any text classification downstream task, such as sentiment analysis or multi-class classification. We will also discuss how to handle sentence-pair and regression problems by covering an implementation. We will work with well-known datasets such as GLUE, as well as our own custom datasets. We will then take advantage of the Trainer class, which deals with the complexity of processes for training and fine-tuning.

First, we will learn how to fine-tune single-sentence binary sentiment classification with the Trainer class. Then, we will train for sentiment classification with native PyTorch without the Trainer class. In multi-class classification, more than two classes will be taken into consideration. We will have seven class classification fine-tuning tasks to perform. Finally, we will train a text regression model to predict numerical values with sentence pairs.

The following topics will be covered in this chapter:

- Introduction to text classification

- Fine-tuning the BERT model for single-sentence binary classification

- Training a classification model with native PyTorch

- Fine-tuning BERT for multi-class classification with custom datasets

- Fine-tuning BERT for sentence-pair regression

- Utilizing `run_glue.py` to fine-tune the models

---

## Technical requirements

We will be using Jupyter Notebook to run our coding exercises. You will need Python 3.6+ for this. Ensure that the following packages are installed:

`sklearn`

`Transformers 4.0+`

`datasets`

All the notebooks for the coding exercises in this chapter will be available at the following GitHub link: <https://github.com/PacktPublishing/Mastering-Transformers/tree/main/CH05>.

Check out the following link to see the Code in Action video:

<https://bit.ly/3y5Fe6R>

---

## Introduction to text classification

Text classification (also known as text categorization) is a way of mapping a document (sentence, Twitter post, book chapter, email content, and so on) to a category out of a predefined list (classes). In the case of two classes that have positive and negative labels, we call this **binary classification** – more specifically, **sentiment analysis**. For more than two classes, we call this **multi-class classification**, where the classes are mutually exclusive, or **multi-label classification**, where the classes are not mutually exclusive, which means a document can receive more than one label. For instance, the content of a news article may be related to sport and politics at the same time. Beyond this classification, we may want to score the documents in a range of  $[-1,1]$  or rank them in a range of  $[1-5]$ . We can solve this kind of problem with a regression model, where the type of the output is numeric, not categorical.

Luckily, the transformer architecture allows us to efficiently solve these problems. For sentence-pair tasks such as document similarity or textual entailment, the input is not a single sentence, but rather two sentences, as illustrated in the following diagram. We can score to what degree two sentences are semantically similar or

predict whether they are semantically similar. Another sentence-pair task is **textual entailment**, where the problem is defined as multi-class classification. Here, two sequences are consumed in the GLUE benchmark: entail/contradict/neutral:

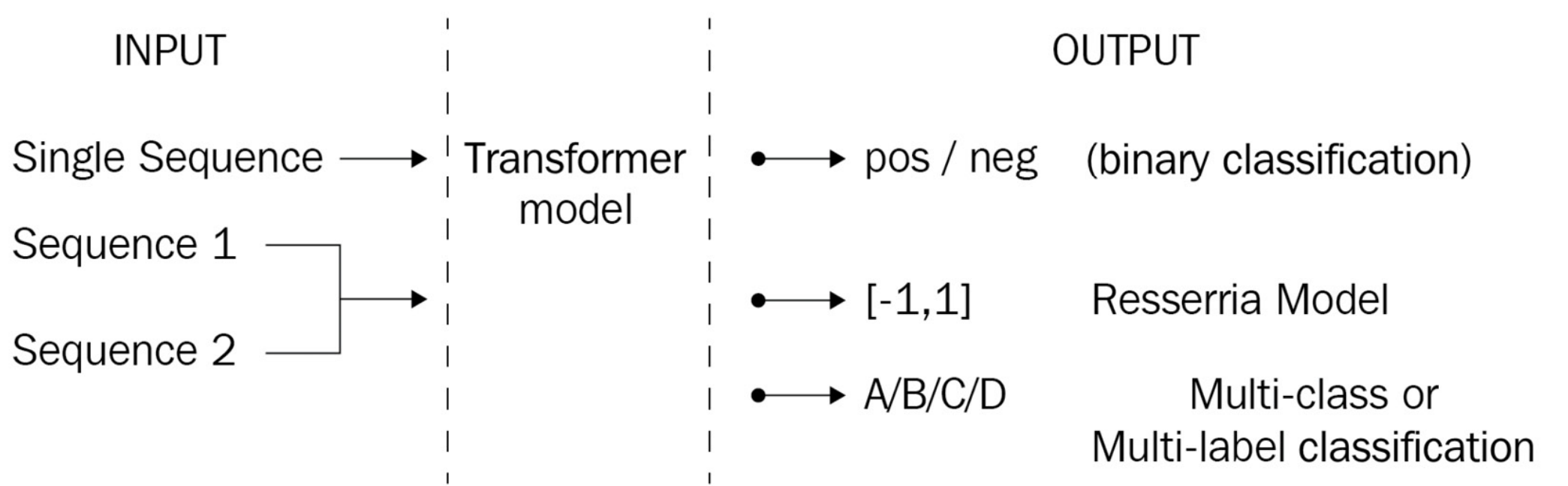


Figure 5.1 – Text classification scheme

Let's start our training process by fine-tuning a pre-trained BERT model for a common problem: sentiment analysis.

## Fine-tuning a BERT model for single-sentence binary classification

In this section, we will discuss how to fine-tune a pre-trained BERT model for sentiment analysis by using the popular **IMDb sentiment** dataset. Working with a GPU will speed up our learning process, but if you do not have such resources, you can work with a CPU as well for fine-tuning. Let's get started:

1. To learn about and save our current device, we can execute the following lines of code:

Copy

Explain

```
from torch import cuda
device = 'cuda' if cuda.is_available() else 'cpu'
```

2. We will use the **DistilBertForSequenceClassification** class here, which is inherited from the **DistilBert** class, with a special sequence classification head at the top. We can utilize this *classification head* to train the classification model, where the number of classes is **2** by default:

Copy

Explain

```
from transformers import DistilBertTokenizerFast,
DistilBertForSequenceClassification
model_path= 'distilbert-base-uncased'
tokenizer = DistilBertTokenizerFast.from_pre-trained(model_path)
model = \ DistilBertForSequenceClassification.from_pre-trained(model_path,
id2label={0:"NEG", 1:"POS"}, label2id={"NEG":0, "POS":1})
```

3. Notice that two parameters called **id2label** and **label2id** are passed to the model to use during inference. Alternatively, we can instantiate a particular **config** object and pass it to the model, as follows:

Copy

Explain

```
config = AutoConfig.from_pre-trained(...)
SequenceClassification.from_pre-trained(... config=config)
```

4. Now, let's select a popular sentiment classification dataset called **IMDB Dataset**. The original dataset consists of two sets of data: 25,000 examples for training and 25 examples for testing. We will split the dataset into test and validation sets. Note that the examples for the first half of the dataset are positive, while the second half's examples are all negative. We can distribute the examples as follows:

Copy

Explain

```
from datasets import load_dataset
imdb_train= load_dataset('imdb', split="train")
imdb_test= load_dataset('imdb', split="test[:6250]+test[-6250:]")
imdb_val= \
load_dataset('imdb', split="test[6250:12500]+test[-12500:-6250:]")
```

5. Let's check the shape of the dataset:

Copy

Explain

```
>>> imdb_train.shape, imdb_test.shape, imdb_val.shape
((25000, 2), (12500, 2), (12500, 2))
```

6. You can take a small portion of the dataset based on your computational resources. For a smaller portion, you should run the following code to select 4,000 examples for training, 1,000 for testing, and 1,000 for validation, like so:

Copy

Explain

```
imdb_train= load_dataset('imdb', split="train[:2000]+train[-2000:]")
imdb_test= load_dataset('imdb', split="test[:500]+test[-500:]")
imdb_val= load_dataset('imdb', split="test[500:1000]+test[-1000:-500:]")
```



7. Now, we can pass these datasets through the `tokenizer` model to make them ready for training:

Copy

Explain

```
enc_train = imdb_train.map(lambda e: tokenizer( e['text'], padding=True,
truncation=True), batched=True, batch_size=1000)
enc_test =  imdb_test.map(lambda e: tokenizer( e['text'], padding=True,
truncation=True), batched=True, batch_size=1000)
enc_val =   imdb_val.map(lambda e: tokenizer( e['text'], padding=True,
truncation=True), batched=True, batch_size=1000)
```

8. Let’s see what the training set looks like. The attention mask and input IDs were added to the dataset by the tokenizer so that the BERT model can process:

Copy

Explain

```
import pandas as pd
pd.DataFrame(enc_train)
```

The output is as follows:

	attention_mask	input_ids	label	text
0	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 22953, 2213, 4381, 2152, 2003, 1037, 947...	1	Bromwell High is a cartoon comedy. It ran at t...
1	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 11573, 2791, 1006, 2030, 2160, 24913, 20...	1	Homelessness (or Houselessness as George Carli...
2	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 8235, 2058, 1011, 3772, 2011, 23920, 575...	1	Brilliant over-acting by Lesley Ann Warren. Be...
3	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 2023, 2003, 4089, 1996, 2087, 2104, 9250...	1	This is easily the most underrated film inn th...
4	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 2023, 2003, 2025, 1996, 5171, 11463, 837...	1	This is not the typical Mel Brooks film. It wa...
...	...	...	...	...
24995	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 2875, 1996, 2203, 1997, 1996, 3185, 1010...	0	Towards the end of the movie, I felt it was to...
24996	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 2023, 2003, 1996, 2785, 1997, 3185, 2008...	0	This is the kind of movie that my enemies cont...
24997	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 1045, 2387, 1005, 6934, 1005, 2197, 2305...	0	I saw 'Descent' last night at the Stockholm Fi...
24998	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 2070, 3152, 2008, 2017, 4060, 2039, 2005...	0	Some films that you pick up for a pound turn o...
24999	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 2023, 2003, 2028, 1997, 1996, 12873, 435...	0	This is one of the dumbest films, I've ever se...

25000 rows × 4 columns

Figure 5.2 – Encoded training dataset

At this point, the datasets are ready for training and testing. The `Trainer` class (`TFTTrainer` for TensorFlow) and the `TrainingArguments` class (`TFTTrainingArguments` for TensorFlow) will help us with much of the training complexity. We will define our argument set within the `TrainingArguments` class, which will then be passed to the `Trainer` object.

Let’s define what each training argument does:

Argument	Definition
output_dir	Points to the model checkpoints and where the predictions will be saved at the end.
do_train and do_eval	Options for monitoring the model's performance during training.
logging_strategy	The options here are no, epoch, and steps (by default).
logging_steps (default value: 500)	The number of steps between two logs to be saved into the logging_dir director.
save_strategy	This is used to save the model checkpoint. The options are no, epoch, and steps (default).
save_steps (def. 500 )	The number of steps between two checkpoints.
fp16	This is for mixed precision and uses both 16-bit and 32-bit floating-point types to make the model train faster and use less memory.
load_best_model_at_end	As the name suggests, this will load the best model checkpoint in terms of validation loss at the end of training.
logging_dir	TensorBoard log directory.

Table 1 – Table of different training argument definitions

9. For more information, please check the API documentation of **TrainingArguments** or execute the following code in a Python notebook:
- Copy

Explain

TrainingArguments?
10. Although deep learning architectures such as LSTM need many epochs, sometimes more than 50, for transformer-based fine-tuning, we will typically be satisfied with an epoch number of 3 due to transfer learning. Most of the time, this number is enough for fine-tuning, as a pre-trained model learns a lot about the language during the pre-training phase, which takes about 50 epochs on average. To determine the correct number of epochs, we need to monitor training and evaluation loss. We will learn how to track training in [Chapter 11, Attention Visualization and Experiment Tracking](#).
11. This will be enough for many downstream task problems, as we will see here. During the training process, our model checkpoints will be saved under the **./MyIMDBModel** folder for every 200 steps:

```

from transformers import TrainingArguments, Trainer
training_args = TrainingArguments(
    output_dir='./MyIMDBModel',
    do_train=True,
    do_eval=True,
    num_train_epochs=3,
    per_device_train_batch_size=32,
    per_device_eval_batch_size=64,
    warmup_steps=100,
    weight_decay=0.01,
    logging_strategy='steps',
    logging_dir='./logs',
    logging_steps=200,
    evaluation_strategy= 'steps',
    fp16= cuda.is_available(),
    load_best_model_at_end=True
)

```

12. Before instantiating a **Trainer** object, we will define the **compute\_metrics()** method, which helps us monitor the progress of the training in terms of particular metrics for whatever we need, such as Precision, RMSE, Pearson correlation, BLEU, and so on. Text classification problems (such as sentiment classification or multi-class classification) are mostly evaluated with **micro-averaging** or **macro-averaging F1**. While the macro-averaging method gives equal weight to each class, micro-averaging gives equal weight to each per-text or per-token classification decision. Micro-averaging is equal to the ratio of the number of times the model decides correctly to the total number of decisions that have been made. On the other hand, the macro-averaging method computes the average score of Precision, Recall, and F1 for each class. For our classification problem, macro-averaging is more convenient for evaluation since we want to give equal weight to each label, as follows:

```

from sklearn.metrics import accuracy_score, Precision_Recall_fscore_support
def compute_metrics(pred):
    labels = pred.label_ids
    preds = pred.predictions.argmax(-1)
    Precision, Recall, f1, _ = \
    Precision_Recall_fscore_support(labels, preds, average='macro')
    acc = accuracy_score(labels, preds)
    return {
        'Accuracy': acc,
        'F1': f1,
        'Precision': Precision,
        'Recall': Recall
    }

```

13. We are almost ready to start the training process. Now, let's instantiate the **Trainer** object and start it. The **Trainer** class is a very powerful and



optimized tool for organizing complex training and evaluation processes for PyTorch and TensorFlow (**TFT**ainer for TensorFlow) thanks to the **transformers** library:

Copy Explain

```
trainer = Trainer(  
    model=model,  
    args=training_args,  
    train_dataset=enc_train,  
    eval_dataset=enc_val,  
    compute_metrics= compute_metrics  
)
```

14. Finally, we can start the training process:

Copy Explain

```
results=trainer.train()
```

The preceding call starts logging metrics, which we will discuss in more detail in [Chapter 11, Attention Visualization and Experiment Tracking](#). The entire IMDB dataset includes 25,000 training examples. With a batch size of 32, we have 25K/32 ~782 steps, and 2,346 (782 x 3) steps to go for 3 epochs, as shown in the following progress bar:

[2346/2346 21:13, Epoch 3/3]

Step	Training Loss	Validation Loss	Accuracy	F1	Precision	Recall	Runtime	Samples Per Second
200	0.417800	0.239647	0.900160	0.899943	0.903660	0.900160	58.657100	213.103000
400	0.251100	0.207064	0.918960	0.918960	0.918960	0.918960	58.724400	212.859000
600	0.237300	0.188785	0.926560	0.926554	0.926707	0.926560	58.727300	212.848000
800	0.209200	0.234559	0.923680	0.923621	0.924982	0.923680	58.750400	212.764000
1000	0.128500	0.248400	0.927280	0.927280	0.927286	0.927280	58.717100	212.885000
1200	0.137400	0.251818	0.920000	0.919869	0.922771	0.920000	58.713500	212.898000
1400	0.125900	0.186671	0.930720	0.930707	0.931054	0.930720	58.724900	212.857000
1600	0.111800	0.230385	0.932960	0.932959	0.932980	0.932960	58.695400	212.964000
1800	0.051300	0.255035	0.933440	0.933440	0.933440	0.933440	58.840300	212.440000
2000	0.045200	0.269209	0.934800	0.934795	0.934927	0.934800	58.819400	212.515000
2200	0.053700	0.242861	0.934640	0.934639	0.934661	0.934640	58.836100	212.455000


Figure 5.3 – The output produced by the Trainer object

15. The **T**ainer object keeps the checkpoint whose validation loss is the smallest at the end. It selects the checkpoint at step 1,400 since the validation loss at this step is the minimum. Let's evaluate the best checkpoint on three (train/test/validation) datasets:



```
>>> q=[trainer.evaluate(eval_dataset=data) for data in [enc_train, enc_val,
enc_test]]
>>> pd.DataFrame(q, index=["train","val","test"]).iloc[:,5]
```

The output is as follows:



[391/391 10:03]

	eval_loss	eval_accuracy	eval_f1	eval_precision	eval_recall
train	0.057059	0.98320	0.983199	0.983259	0.98320
val	0.186671	0.93072	0.930707	0.931054	0.93072
test	0.213239	0.92616	0.926128	0.926904	0.92616

Figure 5.4 – Classification model's performance on the train/validation/test dataset

16. Well done! We have successfully completed the training/testing phase and received 92.6 accuracy and 92.6 F1 for our macro-average. To monitor your training process in more detail, you can call advanced tools such as TensorBoard. These tools parse the logs and enable us to track various metrics for comprehensive analysis. We've already logged the performance and other metrics under the `./logs` folder. Just running the `tensorboard` function within our Python notebook will be enough, as shown in the following code block (we will discuss TensorBoard and other monitoring tools in [Chapter 11, Attention Visualization and Experiment Tracking](#), in detail):

```
%reload_ext tensorboard
%tensorboard --logdir logs
```

17. Now, we will use the model for inference to check if it works properly. Let's define a prediction function to simplify the prediction steps, as follows:

```
def get_prediction(text):
    inputs = tokenizer(text, padding=True, truncation=True,
max_length=250, return_tensors="pt").to(device)
    outputs = \
model(inputs["input_ids"].to(device), inputs["attention_mask"].to(device))
    probs = outputs[0].softmax(1)
    return probs, probs.argmax()
```

18. Now, run the model for inference:

[Copy](#)[Explain](#)

```
>>> text = "I didn't like the movie it bored me "  
>>> get_prediction(text)[1].item()  
0
```

19. What we got here is **0**, which is a negative. We have already defined which ID refers to which label. We can use this mapping scheme to get the label. Alternatively, we can simply pass all these boring steps to a dedicated API, namely Pipeline, which we are already familiar with. Before instantiating it, let's save the best model for further inference:

[Copy](#)[Explain](#)

```
model_save_path = "MyBestIMDBModel"  
trainer.save_model(model_save_path)  
tokenizer.save_pre-trained(model_save_path)
```

The Pipeline API is an easy way to use pre-trained models for inference. We load the model from where we saved it and pass it to the Pipeline API, which does the rest. We can skip this saving step and instead directly pass our **model** and **tokenizer** objects in memory to the Pipeline API. If you do so, you will get the same result.

20. As shown in the following code, we need to specify the task name argument of Pipeline as **sentiment-analysis** when we perform binary classification:

[Copy](#)[Explain](#)

```
>>> from transformers import pipeline, \ DistilBertForSequenceClassification,  
DistilBertTokenizerFast  
>>> model = \ DistilBertForSequenceClassification.from_pre-  
trained("MyBestIMDBModel")  
>>> tokenizer= \ DistilBertTokenizerFast.from_pre-trained("MyBestIMDBModel")  
>>> nlp= pipeline("sentiment-analysis", model=model, tokenizer=tokenizer)  
>>> nlp("the movie was very impressive")  
Out:  [{'label': 'POS', 'score': 0.9621992707252502}]  
>>> nlp("the text of the picture was very poor")  
Out:  [{'label': 'NEG', 'score': 0.9938313961029053}]
```

Pipeline knows how to treat the input and somehow learned which ID refers to which (**POS** or **NEG**) label. It also yields the class probabilities.

Well done! We have fine-tuned a sentiment prediction model for the IMDb dataset using the **Trainer** class. In the next section, we will do the same binary classification training but with native PyTorch. We will also use a different dataset.

# Training a classification model with native PyTorch

The **Trainer** class is very powerful, and we have the HuggingFace team to thank for providing such a useful tool. However, in this section, we will fine-tune the pre-trained model from scratch to see what happens under the hood. Let's get started:

1. First, let's load the model for fine-tuning. We will select **DistilBERT** here since it is a small, fast, and cheap version of BERT:

```
from transformers import DistilBertForSequenceClassification
model = DistilBertForSequenceClassification.from_pretrained('distilbert-base-uncased')
```

[Copy](#)[Explain](#)

2. To fine-tune any model, we need to put it into training mode, as follows:

```
model.train()
```

[Copy](#)[Explain](#)

3. Now, we must load the tokenizer:

```
from transformers import DistilBertTokenizerFast
tokenizer = DistilBertTokenizerFast.from_pretrained('bert-base-uncased')
```

[Copy](#)[Explain](#)

4. Since the **Trainer** class organized the entire process for us, we did not deal with optimization and other training settings in the previous IMDb sentiment classification exercise. Now, we need to instantiate the optimizer ourselves. Here, we must select **AdamW**, which is an implementation of the Adam algorithm but with a weight decay fix. Recently, it has been shown that **AdamW** produces better training loss and validation loss than models trained with Adam. Hence, it is a widely used optimizer within many transformer training processes:

```
from transformers import AdamW
optimizer = AdamW(model.parameters(), lr=1e-3)
```

[Copy](#)[Explain](#)

To design the fine-tuning process from scratch, we must understand how to implement a single step forward and backpropagation. We can pass a single batch through the transformer layer and get the output, which is called **forward propagation**. Then, we must compute the loss using the output and ground truth label and update the model weight based on the loss. This is called **backpropagation**.

The following code receives three sentences associated with the labels in a single batch and performs forward propagation. At the end, the model automatically computes the loss:

Copy Explain

```
import torch
texts= ["this is a good example","this is a bad example","this is a good one"]
labels= [1,0,1]
labels = torch.tensor(labels).unsqueeze(0)
encoding = tokenizer(texts, return_tensors='pt', padding=True,
truncation=True, max_length=512)
input_ids = encoding['input_ids']
attention_mask = encoding['attention_mask']
outputs = \
model(input_ids, attention_mask=attention_mask, labels=labels)
loss = outputs.loss
loss.backward()
optimizer.step()
Outputs
SequenceClassifierOutput(
  (['loss', tensor(0.7178, grad_fn=<NllLossBackward>)], ('logits',tensor([[ 0.0664,
-0.0161],[ 0.0738, 0.0665], [ 0.0690, -0.0010]], grad_fn=<AddmmBackward>))))
```

The model takes `input_ids` and `attention_mask`, which were produced by the tokenizer, and computes the loss using ground truth labels. As we can see, the output consists of both `loss` and `logits`. Now, `loss.backward()` computes the gradient of the tensor by evaluating the model with the inputs and labels. `optimizer.step()` performs a single optimization step and updates the weight using the gradients that were computed, which is called backpropagation. When we put all these lines into a loop shortly, we will also add `optimizer.zero_grad()`, which clears the gradient of all the parameters. It is important to call this at the beginning of the loop; otherwise, we may accumulate the gradients from multiple steps. The second tensor of the output is **logits**. In the context of deep learning, the term logits (short for **logistic units**) is the last layer of the neural architecture and consists of prediction



values as real numbers. Logits need to be turned into probabilities by the softmax function in the case of classification. Otherwise, they are simply normalized for regression.

5. If we want to manually calculate the loss, we must not pass the labels to the model. Due to this, the model only yields the logits and does not calculate the loss. In the following example, we are computing the cross-entropy loss manually:

Copy Explain

```
from torch.nn import functional
labels = torch.tensor([1,0,1])
outputs = model(input_ids, attention_mask=attention_mask)
loss = functional.cross_entropy(outputs.logits, labels)
loss.backward()
optimizer.step()
loss
Output: tensor(0.6101, grad_fn=<NllLossBackward>)
```

6. With that, we've learned how batch input is fed in the forward direction through the network in a single step. Now, it is time to design a loop that iterates over the entire dataset in batches to train the model with several epochs. To do so, we will start by designing the `Dataset` class. It is a subclass of `torch.Dataset`, inherits member variables and functions, and implements `__init__()` and `__getitem__()` abstract functions:

Copy Explain

```
from torch.utils.data import Dataset
class MyDataset(Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels
    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in
self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item
    def __len__(self):
        return len(self.labels)
```

7. Let's fine-tune the model for sentiment analysis by taking another sentiment analysis dataset called the SST-2 dataset; that is, **Stanford Sentiment Treebank v2 (SST2)**. We will also load the corresponding metric for SST-2 for evaluation, as follows:

Copy

Explain

```
import datasets
from datasets import load_dataset
sst2= load_dataset("glue","sst2")
from datasets import load_metric
metric = load_metric("glue", "sst2")
```

8. We will extract the sentences and the labels accordingly:

Copy

Explain

```
texts=sst2['train']['sentence']
labels=sst2['train']['label']
val_texts=sst2['validation']['sentence']
val_labels=sst2['validation']['label']
```

9. Now, we can pass the datasets through the tokenizer and instantiate the **MyDataset** object to make the BERT models work with them:

Copy

Explain

```
train_dataset= MyDataset(tokenizer(texts, truncation=True, padding=True), labels)
val_dataset= MyDataset(tokenizer(val_texts, truncation=True, padding=True),
val_labels)
```

10. Let's instantiate a **DataLoader** class that provides an interface to iterate through the data samples by loading order. This also helps with batching and memory pinning:

Copy

Explain

```
from torch.utils.data import DataLoader
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=16, shuffle=True)
```

11. The following lines detect the device and define the **AdamW** optimizer properly:

Copy

Explain

```
from transformers import AdamW
device = \
torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
model.to(device)
optimizer = AdamW(model.parameters(), lr=1e-3)
```

So far, we know how to implement forward propagation, which is where we process a batch of examples. Here, batch data is fed in the forward direction through the neural network. In a single step, each layer from the first to the final

one is processed by the batch data, as per the activation function, and is passed to the successive layer. To go through the entire dataset in several epochs, we designed two nested loops: the outer loop is for the epoch, while the inner loop is for the steps for each batch. The inner part is made up of two blocks; one is for training, while the other one is for evaluating each epoch. As you may have noticed, we called `model.train()` at the first training loop, and when we moved the second evaluation block, we called `model.eval()`. This is important as we put the model into training and inference mode.

12. We have already discussed the inner block. Note that we track the model's performance by means of the corresponding the `metric` object:

Copy Explain

```
for epoch in range(3):
    model.train()
    for batch in train_loader:
        optimizer.zero_grad()
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)
        outputs = \
model(input_ids, attention_mask=attention_mask, labels=labels)
        loss = outputs[0]
        loss.backward()
        optimizer.step()
    model.eval()
    for batch in val_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)
        outputs = \
model(input_ids, attention_mask=attention_mask, labels=labels)
        predictions=outputs.logits.argmax(dim=-1)
        metric.add_batch(
            predictions=predictions,
            references=batch["labels"],
        )
    eval_metric = metric.compute()
    print(f"epoch {epoch}: {eval_metric}")
OUTPUT:
epoch 0: {'accuracy': 0.9048165137614679}
epoch 1: {'accuracy': 0.8944954128440367}
epoch 2: {'accuracy': 0.9094036697247706}
```

Well done! We've fine-tuned our model and got around 90.94 accuracy. The remaining processes, such as saving, loading, and inference, will be similar to what we did with the `Trainer` class.

With that, we are done with binary classification. In the next section, we will learn how to implement a model for multi-class classification for a language other than English.

---

## Fine-tuning BERT for multi-class classification with custom datasets

In this section, we will fine-tune the Turkish BERT, namely **BERTurk**, to perform seven-class classification downstream tasks with a custom dataset. This dataset has been compiled from Turkish newspapers and consists of seven categories. We will start by getting the dataset. Alternatively, you can find it in this book's GitHub respository or get it from <https://www.kaggle.com/savasy/ttc4900>:

1. First, run the following code to get data within a Python notebook:

Copy

Explain

```
!wget
https://raw.githubusercontent.com/savasy/TurkishTextClassification/master/TTC4900.csv
```

2. Start by loading the data:

Copy

Explain

```
import pandas as pd
data= pd.read_csv("TTC4900.csv")
data=data.sample(frac=1.0, random_state=42)
```

3. Let's organize the IDs and labels with **id2label** and **label2id** to make the model figure out which ID refers to which label. We will also pass the number of labels, **NUM\_LABELS**, to the model to specify the size of a thin classification head layer on top of the BERT model:

Copy

Explain

```
labels=["teknoloji","ekonomi","saglik","siyaset","kultur","spor","dunya"]
NUM_LABELS= len(labels)
id2label={i:l for i,l in enumerate(labels)}
label2id={l:i for i,l in enumerate(labels)}
data["labels"]=data.category.map(lambda x: label2id[x.strip()])
data.head()
```



The output is as follows:

	category	text	labels
4657	teknoloji	acıların kedisi sam çatık kaşlı kedi sam in i...	0
3539	spor	g saray a git santos van_persie den forma ala...	5
907	dunya	endonezya da çatışmalar 14 ölü endonezya da i...	6
4353	teknoloji	emniyetten polis logolu virüs uyarısı telefon...	0
3745	spor	beni türk yapın cristian_baroni yıldırım dan ...	5

Figure 5.5 – Text classification dataset – TTC 4900

4. Let’s count and plot the number of classes using a pandas object:

Copy

Explain

```
data.category.value_counts().plot(kind='pie')
```

As shown in the following diagram, the dataset classes have been fairly distributed:

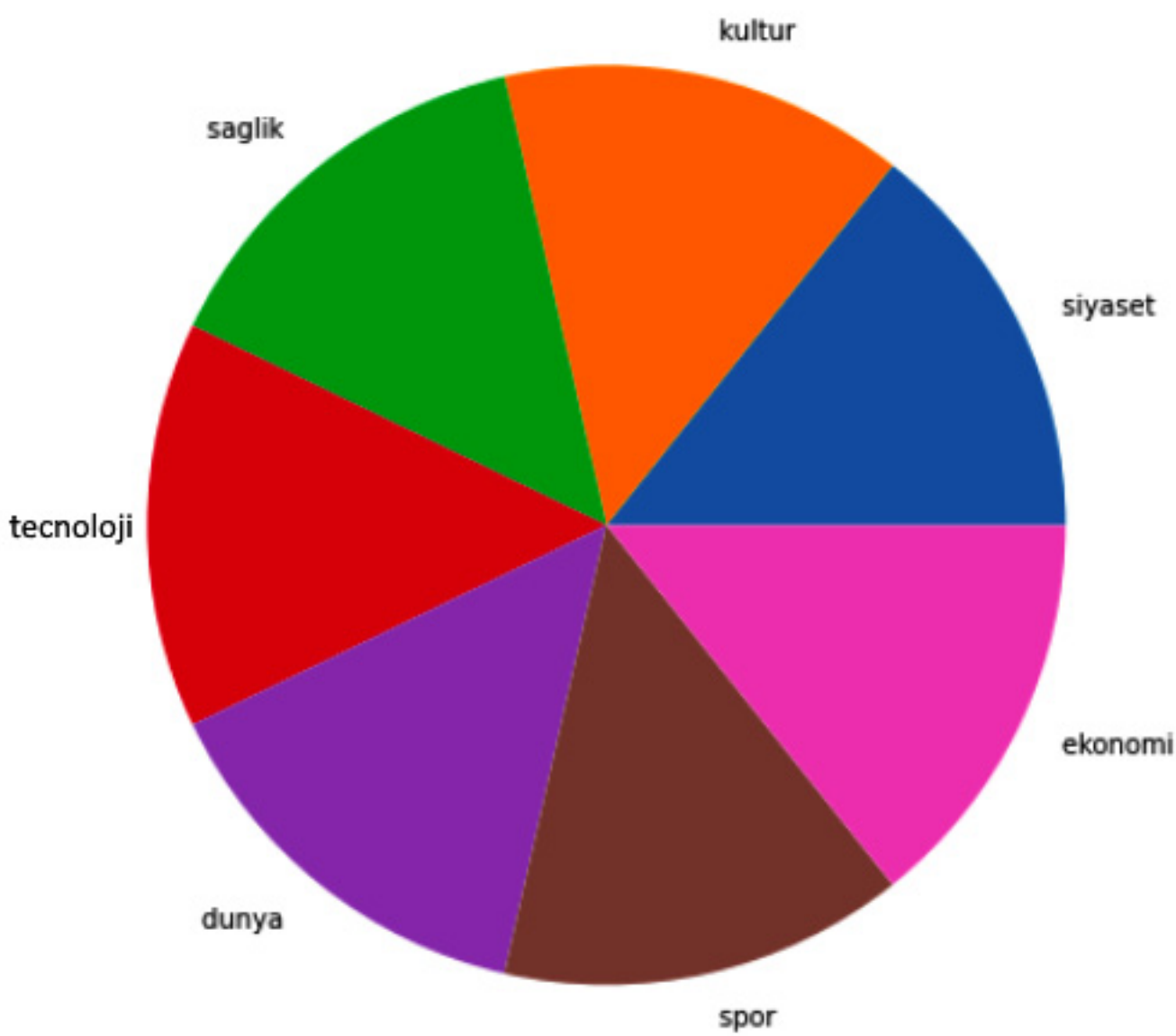


Figure 5.6 – The class distribution

5. The following execution instantiates a sequence classification model with the number of labels (7), label ID mappings, and a Turkish BERT model (`dbmdz/bert-base-turkish-uncased`), namely BERTurk. To check this, execute the following:

[Copy](#)[Explain](#)

```
>>> model
```

6. The output will be a summary of the model and is too long to show here. Instead, let's turn our attention to the last layer by using the following code:

[Copy](#)[Explain](#)

```
(classifier): Linear(in_features=768, out_features=7, bias=True)
```

7. You may have noticed that we did not choose **DistilBert** as there is no pre-trained *uncased* **DistilBert** for the Turkish language:

[Copy](#)[Explain](#)

```
from transformers import BertTokenizerFast
tokenizer = BertTokenizerFast.from_pretrained("dbmdz/bert-base-turkish-uncased",
max_length=512)
from transformers import BertForSequenceClassification
model = BertForSequenceClassification.from_pretrained("dbmdz/bert-base-turkish-
uncased", num_labels=NUM_LABELS, id2label=id2label, label2id=label2id)
model.to(device)
```

8. Now, let's prepare the training (%50), validation (%25), and test (%25) datasets, as follows:

[Copy](#)[Explain](#)

```
SIZE= data.shape[0]
## sentences
train_texts= list(data.text[:SIZE//2])
val_texts= list(data.text[SIZE//2:(3*SIZE)//4 ])
test_texts= list(data.text[(3*SIZE)//4:])
## labels
train_labels= list(data.labels[:SIZE//2])
val_labels= list(data.labels[SIZE//2:(3*SIZE)//4])
test_labels= list(data.labels[(3*SIZE)//4:])
## check the size
len(train_texts), len(val_texts), len(test_texts)
(2450, 1225, 1225)
```

9. The following code tokenizes the sentences of three datasets and their tokens and converts them into integers (**input\_ids**), which are then fed into the BERT model:

Copy

Explain

```
train_encodings = tokenizer(train_texts, truncation=True, padding=True)
val_encodings = tokenizer(val_texts, truncation=True, padding=True)
test_encodings = tokenizer(test_texts, truncation=True, padding=True)
```

10. We have already implemented the `MyDataset` class (please see page 14). The class inherits from the abstract `Dataset` class by overwriting the `__getitem__` and `__len__()` methods, which are expected to return the items and the size of the dataset using any data loader, respectively:

Copy

Explain

```
train_dataset = MyDataset(train_encodings, train_labels)
val_dataset = MyDataset(val_encodings, val_labels)
test_dataset = MyDataset(test_encodings, test_labels)
```

11. We will keep batch size as `16` since we have a relatively small dataset. Notice that the other parameters of `TrainingArguments` are almost the same as they were for the previous sentiment analysis experiment:

Copy

Explain

```
from transformers import TrainingArguments, Trainer
training_args = TrainingArguments(
    output_dir='./TTC4900Model',
    do_train=True,
    do_eval=True,
    num_train_epochs=3,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=32,
    warmup_steps=100,
    weight_decay=0.01,
    logging_strategy='steps',
    logging_dir='./multi-class-logs',
    logging_steps=50,
    evaluation_strategy="steps",
    eval_steps=50,
    save_strategy="epoch",
    fp16=True,
    load_best_model_at_end=True
)
```

12. Sentiment analysis and text classification are objects of the same evaluation metrics; that is, macro-averaging macro-averaged F1, Precision, and Recall. Therefore, we will not define the `compute_metric()` function again. Here is the code for instantiating a `Trainer` object:

```
trainer = Trainer(  
    model=model,  
    args=training_args,  
    train_dataset=train_dataset,  
    eval_dataset=val_dataset,  
    compute_metrics= compute_metrics  
)
```

13. Finally, let's start the training process:

```
trainer.train()
```

The output is as follows:

[462/462 12:27, Epoch 3/3]

Step	Training Loss	Validation Loss	Accuracy	F1	Precision	Recall	Runtime	Samples Per Second
50	1.874100	1.706715	0.377143	0.379416	0.553955	0.383715	20.982000	58.383000
100	0.842900	0.327575	0.915102	0.913738	0.914565	0.915279	20.981700	58.384000
150	0.358200	0.281808	0.911020	0.910288	0.912012	0.911213	20.997000	58.342000
200	0.233500	0.366845	0.905306	0.905313	0.916948	0.903440	20.980800	58.387000
250	0.222700	0.292270	0.922449	0.921374	0.921567	0.923131	20.981300	58.385000
300	0.257700	0.280120	0.924898	0.923810	0.924427	0.924510	20.979800	58.390000
350	0.115200	0.292410	0.925714	0.924946	0.924752	0.925454	20.982700	58.381000
400	0.064900	0.322697	0.925714	0.924944	0.925674	0.925265	20.988300	58.366000
450	0.080400	0.297606	0.929796	0.929267	0.929170	0.929497	20.985100	58.375000

Figure 5.7 – The output of the Trainer class for text classification

14. To check the trained model, we must evaluate the fine-tuned model on three dataset splits, as follows. Our best model is fine-tuned at step 300 with a loss of 0.28012:

```
q=[trainer.evaluate(eval_dataset=data) for data in [train_dataset, val_dataset,  
test_dataset]]  
pd.DataFrame(q, index=["train","val","test"]).iloc[:, :5]
```

The output is as follows:



	eval_loss	eval_Accuracy	eval_F1	eval_Precision	eval_Recall
train	0.091844	0.975510	0.97546	0.975942	0.975535
val	0.280120	0.924898	0.92381	0.924427	0.924510
test	0.280038	0.926531	0.92542	0.927410	0.925425

Figure 5.8 – The text classification model's performance on the train/validation/test dataset

The classification accuracy is around 92.6, while the F1 macro-average is around 92.5. In the literature, many approaches have been tested on this Turkish benchmark dataset. They mostly followed TF-IDF and linear classifier, word2vec embeddings, or an LSTM-based classifier and got around 90.0 F1 at best. Compared to those approaches, other than transformer, the fine-tuned BERT model outperforms them.

15. As with any other experiment, we can track the experiment via TensorBoard:

```
%load_ext tensorboard
%tensorboard --logdir multi-class-logs/
```

16. Let's design a function that will run the model for inference. If you want to see a real label instead of an ID, you can use the `config` object of our model, as shown in the following `predict` function:

```
def predict(text):
    inputs = tokenizer(text, padding=True, truncation=True, max_length=512,
return_tensors="pt").to("cuda")
    outputs = model(**inputs)
    probs = outputs[0].softmax(1)
    return probs, probs.argmax(),model.config.id2label[probs.argmax().item()]
```

17. Now, we are ready to call the `predict` function for text classification inference. The following code classifies a sentence about a football team:

```
text = "Fenerbahçeli futbolcular kısa paslarla hazırlık çalışması yaptılar"
predict(text)
(tensor([[5.6183e-04, 4.9046e-04, 5.1385e-04, 9.9414e-04, 3.4417e-04, 9.9669e-01,
4.0617e-04]], device='cuda:0', grad_fn=<SoftmaxBackward>), tensor(5,
device='cuda:0'), 'spor')
```

18. As we can see, the model correctly predicted the sentence as sports (**spor**). Now, it is time to save the model and reload it using the **from\_pre-trained()** function. Here is the code:

```
model_path = "turkish-text-classification-model"
trainer.save_model(model_path)
tokenizer.save_pre-trained(model_path)
```

[Copy](#)[Explain](#)

19. Now, we can reload the saved model and run inference with the help of the **pipeline** class:

```
model_path = "turkish-text-classification-model"
from transformers import pipeline, BertForSequenceClassification,
BertTokenizerFast
model = BertForSequenceClassification.from_pre-trained(model_path)
tokenizer= BertTokenizerFast.from_pre-trained(model_path)
nlp= pipeline("sentiment-analysis", model=model, tokenizer=tokenizer)
```

[Copy](#)[Explain](#)

20. You may have noticed that the task's name is **sentiment-analysis**. This term may be confusing but this argument will actually return **TextClassificationPipeline** at the end. Let's run the pipeline:

```
>>> nlp("Sinemada hangi filmler oynuyor bugün")
[{'label': 'kültür', 'score': 0.9930670261383057}]
>>> nlp("Dolar ve Euro bugün yurtiçi piyasalarda yükseldi")
[{'label': 'ekonomi', 'score': 0.9927696585655212}]
>>> nlp("Bayern Münih ile Barcelona bugün karşı karşıya geliyor. Maçı İngiliz hakem James Watts yönetecek!")
[{'label': 'spor', 'score': 0.9975664019584656}]
```

[Copy](#)[Explain](#)

That's our model! It has predicted successfully.

So far, we have implemented two single-sentence tasks; that is, sentiment analysis and multi-class classification. In the next section, we will learn how to handle sentence-pair input and how to design a regression model with BERT.

---

# Fine-tuning the BERT model for sentence-pair regression

The regression model is considered to be for classification, but the last layer only contains a single unit. This is not processed by softmax logistic regression but normalized. To specify the model and put a single-unit head layer at the top, we can either directly pass the `num_labels=1` parameter to the `BERT.from_pre-trained()` method or pass this information through a `Config` object. Initially, this needs to be copied from the `config` object of the pre-trained model, as follows:

Copy Explain

```
from transformers import DistilBertConfig, DistilBertTokenizerFast,
DistilBertForSequenceClassification
model_path='distilbert-base-uncased'
config = DistilBertConfig.from_pre-trained(model_path, num_labels=1)
tokenizer = DistilBertTokenizerFast.from_pre-trained(model_path)
model = \
DistilBertForSequenceClassification.from_pre-trained(model_path, config=config)
```

Well, our pre-trained model has a single-unit head layer thanks to the `num_labels=1` parameter. Now, we are ready to fine-tune the model with our dataset. Here, we will use the **Semantic Textual Similarity-Benchmark (STS-B)**, which is a collection of sentence pairs that have been drawn from a variety of content, such as news headlines. Each pair has been annotated with a similarity score from 1 to 5. Our task is to fine-tune the BERT model to predict these scores. We will evaluate the model using the Pearson/Spearman correlation coefficients while following the literature. Let's get started:

1. The following code loads the data. The original data was splits into three. However, the test split has no label so that we can divide the validation data into two parts, as follows:

```
import datasets
from datasets import load_dataset
stsb_train= load_dataset('glue','stsb', split="train")
stsb_validation = load_dataset('glue','stsb', split="validation")
stsb_validation=stsb_validation.shuffle(seed=42)
stsb_val= datasets.Dataset.from_dict(stsb_validation[:750])
stsb_test= datasets.Dataset.from_dict(stsb_validation[750:])
```

2. Let’s make the `stsb_train` training data neat by wrapping it with pandas:

```
pd.DataFrame(stsb_train)
```

Here is what the training data looks like:

	idx	label	sentence1	sentence2
0	0	5.00	A plane is taking off.	An air plane is taking off.
1	1	3.80	A man is playing a large flute.	A man is playing a flute.
2	2	3.80	A man is spreading shredded cheese on a pizza.	A man is spreading shredded cheese on an uncoo...
3	3	2.60	Three men are playing chess.	Two men are playing chess.
4	4	4.25	A man is playing the cello.	A man seated is playing the cello.
...	...	...	...	...
5744	5744	0.00	Severe Gales As Storm Clodagh Hits Britain	Merkel pledges NATO solidarity with Latvia
5745	5745	0.00	Dozens of Egyptians hostages taken by Libyan t...	Egyptian boat crash death toll rises as more b...
5746	5746	0.00	President heading to Bahrain	President Xi: China to continue help to fight ...
5747	5747	0.00	China, India vow to further bilateral ties	China Scrambles to Reassure Jittery Stock Traders
5748	5748	0.00	Putin spokesman: Doping charges appear unfounded	The Latest on Severe Weather: 1 Dead in Texas ...

5749 rows × 4 columns

Figure 5.9 – STS-B training dataset

3. Run the following code to check the shape of the three sets:

```
stsb_train.shape, stsb_val.shape, stsb_test.shape
((5749, 4), (750, 4), (750, 4))
```

4. Run the following code to tokenize the datasets:



```
enc_train = stsb_train.map(lambda e: tokenizer( e['sentence1'],e['sentence2'],
padding=True, truncation=True), batched=True, batch_size=1000)
enc_val = stsb_val.map(lambda e: tokenizer( e['sentence1'],e['sentence2'],
padding=True, truncation=True), batched=True, batch_size=1000)
enc_test = stsb_test.map(lambda e: tokenizer( e['sentence1'],e['sentence2'],
padding=True, truncation=True), batched=True, batch_size=1000)
```

5. The tokenizer merges two sentences with a [SEP] delimiter and produces single `input_ids` and an `attention_mask` for a sentence pair, as shown here:

```
pd.DataFrame(enc_train)
```

The output is as follows:

	attention_mask	idx	input_ids	label	sentence1	sentence2
0	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	0	[101, 1037, 4946, 2003, 2635, 2125, 1012, 102,...	5.00	A plane is taking off.	An air plane is taking off.
1	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	1	[101, 1037, 2158, 2003, 2652, 1037, 2312, 8928...	3.80	A man is playing a large flute.	A man is playing a flute.
2	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	2	[101, 1037, 2158, 2003, 9359, 14021, 5596, 209...	3.80	A man is spreading shredded cheese on a pizza.	A man is spreading shredded cheese on an uncoo...
3	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	3	[101, 2093, 2273, 2024, 2652, 7433, 1012, 102,...	2.60	Three men are playing chess.	Two men are playing chess.
4	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	4	[101, 1037, 2158, 2003, 2652, 1996, 10145, 101...	4.25	A man is playing the cello.	A man seated is playing the cello.
...	...	...	...	...	...	...
5744	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	5744	[101, 5729, 14554, 2015, 2004, 4040, 18856, 13...	0.00	Severe Gales As Storm Clodagh Hits Britain	Merkel pledges NATO solidarity with Latvia
5745	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	5745	[101, 9877, 1997, 23437, 19323, 2579, 2011, 19...	0.00	Dozens of Egyptians hostages taken by Libyan t...	Egyptian boat crash death toll rises as more b...
5746	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	5746	[101, 2343, 5825, 2000, 15195, 102, 2343, 8418...	0.00	President heading to Bahrain	President Xi: China to continue help to fight ...
5747	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	5747	[101, 2859, 1010, 2634, 19076, 2000, 2582, 177...	0.00	China, India vow to further bilateral ties	China Scrambles to Reassure Jittery Stock Traders
5748	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	5748	[101, 22072, 14056, 1024, 23799, 5571, 3711, 4...	0.00	Putin spokesman: Doping charges appear unfounded	The Latest on Severe Weather: 1 Dead in Texas ...

5749 rows × 6 columns

Figure 5.10 – Encoded training dataset

Similar to other experiments, we follow almost the same scheme for the `TrainingArguments` and `Trainer` classes. Here is the code:

[Copy](#)[Explain](#)

```
from transformers import TrainingArguments, Trainer
training_args = TrainingArguments(
    output_dir='./stsb-model',
    do_train=True,
    do_eval=True,
    num_train_epochs=3,
    per_device_train_batch_size=32,
    per_device_eval_batch_size=64,
    warmup_steps=100,
    weight_decay=0.01,
    logging_strategy='steps',
    logging_dir='./logs',
    logging_steps=50,
    evaluation_strategy="steps",
    save_strategy="epoch",
    fp16=True,
    load_best_model_at_end=True
)
```

6. Another important difference between the current regression task and the previous classification tasks is the design of `compute_metrics`. Here, our evaluation metric will be based on the **Pearson Correlation Coefficient** and the **Spearman's Rank Correlation** following the common practice provided in the literature. We also provide the **Mean Squared Error (MSE)**, **Root Mean Square Error (RMSE)**, and **Mean Absolute Error (MAE)** metrics, which are commonly used, especially for regression models:

[Copy](#)[Explain](#)

```
import numpy as np
from scipy.stats import pearsonr
from scipy.stats import spearmanr
def compute_metrics(pred):
    preds = np.squeeze(pred.predictions)
    return {"MSE": ((preds - pred.label_ids) ** 2).mean().item(),
            "RMSE": (np.sqrt((preds - pred.label_ids) ** 2).mean()).item(),
            "MAE": (np.abs(preds - pred.label_ids)).mean().item(),
            "Pearson" : pearsonr(preds, pred.label_ids)[0],
            "Spearman's Rank":spearmanr(preds, pred.label_ids)[0]
    }
```

7. Now, let's instantiate the `Trainer` object:

CopyExplain

```
trainer = Trainer(  
    model=model,  
    args=training_args,  
    train_dataset=enc_train,  
    eval_dataset=enc_val,  
    compute_metrics=compute_metrics,  
    tokenizer=tokenizer  
)
```

Run the training, like so:

CopyExplain

```
train_result = trainer.train()
```

The output is as follows:

Step	Training Loss	Validation Loss	Mse	Rmse	Mae	Pearson	Spearman's rank	Runtime	Samples Per Second
50	4.973200	2.242550	2.242550	1.497515	1.261815	0.140489	0.138228	0.943900	794.538000
100	1.447300	0.801587	0.801587	0.895314	0.735321	0.808588	0.809430	0.933300	803.602000
150	0.940400	0.693730	0.693730	0.832904	0.675787	0.843234	0.842421	0.930700	805.838000
200	0.736300	0.679696	0.679696	0.824437	0.662136	0.846722	0.843393	0.934700	802.407000
250	0.585400	0.590002	0.590002	0.768116	0.618677	0.859470	0.854824	0.931600	805.067000
300	0.513800	0.584674	0.584674	0.764640	0.610141	0.861033	0.856779	0.942900	795.438000
350	0.488000	0.604512	0.604512	0.777504	0.611338	0.865844	0.861726	0.939600	798.174000
400	0.362900	0.555219	0.555219	0.745130	0.582900	0.868366	0.863372	0.938200	799.379000
450	0.298500	0.544973	0.544973	0.738223	0.576751	0.868145	0.864209	0.938200	799.407000
500	0.270100	0.546966	0.546966	0.739571	0.575326	0.867538	0.864035	0.941900	796.240000

Figure 5.11 – Training result for text regression

8. The best validation loss that's computed is **0.544973** at step **450**. Let's evaluate the best checkpoint model at that step, as follows:

CopyExplain

```
q=[trainer.evaluate(eval_dataset=data) for data in [enc_train, enc_val, enc_test]]  
pd.DataFrame(q, index=["train","val","test"]).iloc[:,5]
```

The output is as follows:

[90/90 00:30]

	eval_loss	eval_MSE	eval_RMSE	eval_MAE	eval_Pearson	eval_Spearman's Rank
train	0.232471	0.232471	0.482152	0.372915	0.944844	0.935578
val	0.544973	0.544973	0.738223	0.576751	0.868145	0.864209
test	0.537752	0.537752	0.733316	0.567489	0.875409	0.872858

## Figure 5.12 – Regression performance on the training/validation/test dataset

The Pearson and Spearman correlation scores are around 87.54 and 87.28 on the test dataset, respectively. We did not get a SoTA result, but we did get a comparable result for the STS-B task based on the GLUE Benchmark leaderboard. Please check the leaderboard!

9. We are now ready to run the model for inference. Let's take the following two sentences, which share the same meaning, and pass them to the model:

Copy Explain

```
s1,s2="A plane is taking off. ","An air plane is taking off."
encoding = tokenizer(s1,s2, return_tensors='pt', padding=True, truncation=True,
max_length=512)
input_ids = encoding['input_ids'].to(device)
attention_mask = encoding['attention_mask'].to(device)
outputs = model(input_ids, attention_mask=attention_mask)
outputs.logits.item()
OUTPUT: 4.033723831176758
```

10. The following code consumes the negative sentence pair, which means the sentences are semantically different:

Copy Explain

```
s1,s2="The men are playing soccer. ","A man is riding a motorcycle."
encoding = tokenizer("hey how are you there","hey how are you",
return_tensors='pt', padding=True, truncation=True, max_length=512)
input_ids = encoding['input_ids'].to(device)
attention_mask = encoding['attention_mask'].to(device)
outputs = model(input_ids, attention_mask=attention_mask)
outputs.logits.item()
OUTPUT: 2.3579328060150146
```

11. Finally, we will save the model, as follows:

Copy Explain

```
model_path = "sentence-pair-regression-model"
trainer.save_model(model_path)
tokenizer.save_pre-trained(model_path)
```

Well done! We can congratulate ourselves since we have successfully completed three tasks: sentiment analysis, multi-class classification, and sentence pair regression.

---



# Utilizing run\_glue.py to fine-tune the models

So far, we have designed a fine-tuning architecture from scratch using both native PyTorch and the `Trainer` class. The HuggingFace community also provides another powerful script called `run_glue.py` for GLUE benchmark and GLUE-like classification downstream tasks. This script can handle and organize the entire training/validation process for us. If you want to do quick prototyping, you should use this script. It can fine-tune any pre-trained models on the HuggingFace hub. We can also feed it with our own data in any format.

Please go to the following link to access the script and to learn more:

<https://github.com/huggingface/transformers/tree/master/examples>.

The script can perform nine different GLUE tasks. With the script, we can do everything that we have done with the `Trainer` class so far. The task name could be one of the following GLUE tasks: `cola`, `sst2`, `mrpc`, `stsb`, `qqp`, `mnli`, `qnli`, `rte`, or `wnli`.

Here is the script scheme for fine-tuning a model:

Copy Explain

```
export TASK_NAME= "My-Task-Name"
python run_glue.py \
  --model_name_or_path bert-base-cased \
  --task_name $TASK_NAME \
  --do_train \ --do_eval \
  --max_seq_length 128 \
  --per_device_train_batch_size 32 \
  --learning_rate 2e-5 \
  --num_train_epochs 3 \
  --output_dir /tmp/$TASK_NAME/
```

The community provides another script called `run_glue_no_trainer.py`. The main difference between the original script and this one is that this no-trainer script gives us more chances to change the options for the optimizer, or add any customization that we want to do.

---

# Summary

In this chapter, we discussed how to fine-tune a pre-trained model for any text classification downstream task. We fine-tuned the models using sentiment analysis, multi-class classification, and sentence-pair classification – more specifically, sentence-pair regression. We worked with a well-known IMDb dataset and our own custom dataset to train the models. While we took advantage of the `Trainer` class to cope with much of the complexity of the processes for training and fine-tuning, we learned how to train from scratch with native libraries to understand forward propagation and backpropagation with the `transformers` library. To summarize, we discussed and conducted fine-tuning single-sentence classification with `Trainer`, sentiment classification with native PyTorch without `Trainer`, single-sentence multi-class classification, and fine-tuning sentence-pair regression.

In the next chapter, we will learn how to fine-tune a pre-trained model to any token classification downstream task, such as parts-of-speech tagging or named-entity recognition.

---

[Previous Chapter](#)[Next Chapter](#)