# **Question One: Hugging Face (CODE)**

import torch

from transformers import BertTokenizer, BertForSequenceClassification, Trainer, TrainingArguments

from datasets import Dataset, DatasetDict

from sklearn.model\_selection import train\_test\_split

import numpy as np

from datasets import load\_metric

# Set up the device

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

# Load a pre-trained tokenizer and model

tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')

model = BertForSequenceClassification.from\_pretrained('bert-base-uncased', num\_labels=3)

model.to(device)

# Create or Load the Dataset (for example purposes, let's create a small sample dataset)

texts = [

"This is an easy sentence to understand.",

"The mitochondrion is a double-membrane-bound organelle found in most eukaryotic organisms.",

"The cat sat on the mat."

]

labels = [0, 2, 0] # 0 = easy, 1 = medium, 2 = difficult

# Tokenize the data

def tokenize\_function(texts):

return tokenizer(texts, padding='max\_length', truncation=True)

train\_texts, test\_texts, train\_labels, test\_labels = train\_test\_split(texts, labels, test\_size=0.2, random\_state=42)

train\_encodings = tokenize\_function(train\_texts)

test\_encodings = tokenize\_function(test\_texts)

# Convert the tokenized data to a Dataset

train\_dataset = Dataset.from\_dict({

'input\_ids': train\_encodings['input\_ids'],

'attention\_mask': train\_encodings['attention\_mask'],

'labels': train\_labels

})

test\_dataset = Dataset.from\_dict({

'input\_ids': test\_encodings['input\_ids'],

'attention\_mask': test\_encodings['attention\_mask'],

'labels': test\_labels

})

# Define a function for computing metrics (accuracy in this case)

def compute\_metrics(eval\_pred):

from datasets import load\_metric

# Simulate some predictions and true labels

predictions = np.array([0, 2, 0, 0, 1])

true\_labels = np.array([0, 2, 0, 1, 2])

# Load the metric

accuracy\_metric = load\_metric("accuracy")

f1\_metric = load\_metric("f1")

# Compute the metrics

accuracy = accuracy\_metric.compute(predictions=predictions, references=true\_labels)

f1 = f1\_metric.compute(predictions=predictions, references=true\_labels, average="macro")

print(f"Accuracy: {accuracy['accuracy']:.2f}")

print(f"F1-Score: {f1['f1']:.2f}")

# Define the training arguments

training\_args = TrainingArguments(

output\_dir='./results',

num\_train\_epochs=3,

per\_device\_train\_batch\_size=2,

per\_device\_eval\_batch\_size=2,

warmup\_steps=500,

weight\_decay=0.01,

logging\_dir='./logs',

logging\_steps=10,

evaluation\_strategy="epoch"

)

# Create the Trainer

trainer = Trainer(

model=model,

args=training\_args,

train\_dataset=train\_dataset,

eval\_dataset=test\_dataset,

compute\_metrics=compute\_metrics

)

# Fine-tune the model

trainer.train()

# Evaluate the model on the test dataset

eval\_result = trainer.evaluate()

print(f"Test Accuracy: {eval\_result['eval\_accuracy']}")

# **Question Two: Finetuning Large Language Models**

import torch

from transformers import BertTokenizer, BertForSequenceClassification, Trainer, TrainingArguments

from datasets import load\_dataset, DatasetDict

from sklearn.model\_selection import train\_test\_split

from datasets import load\_metric

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

# Set up the device

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

# 1. Install Necessary Libraries and Tools

# Already done: torch, transformers, datasets, sklearn, matplotlib

# 2. Load the Dataset and Perform Basic EDA

# Assuming a dataset of health-related articles with labels for "Prevention", "Treatment", and "Diagnosis"

dataset = load\_dataset("health\_fact")

# Simplify the dataset for demonstration purposes

df = dataset['train'].to\_pandas()

df = df[['text', 'label']]

# Perform Basic EDA

# Analyze the distribution of sentiment classes

print(df['label'].value\_counts())

# Check for missing values

print(df.isnull().sum())

# Understand the length of text articles

df['text\_length'] = df['text'].apply(len)

print(df['text\_length'].describe())

# Visualization: Distribution of sentiment classes and text lengths

sns.countplot(x=df['label'])

plt.title("Distribution of Article Categories")

plt.show()

sns.histplot(df['text\_length'], bins=20, kde=True)

plt.title("Distribution of Article Lengths")

plt.show()

# 3. Dataset Preparation

# Preprocessing: Tokenization

tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')

def tokenize\_function(texts):

return tokenizer(texts, padding='max\_length', truncation=True, max\_length=128)

df['label'] = df['label'].map({"Prevention": 0, "Treatment": 1, "Diagnosis": 2})

dataset = DatasetDict({

'train': df[['text', 'label']]

})

tokenized\_datasets = dataset.map(lambda x: tokenize\_function(x['text']), batched=True)

tokenized\_datasets = tokenized\_datasets.remove\_columns(["text"])

tokenized\_datasets.set\_format("torch")

# Train-test split

train\_dataset, test\_dataset = train\_test\_split(tokenized\_datasets['train'], test\_size=0.2, random\_state=42)

# Convert to Dataset objects

train\_dataset = DatasetDict({'train': train\_dataset})

test\_dataset = DatasetDict({'test': test\_dataset})

# 4. Model Selection

# Choosing and Loading the Pre-trained Model

model = BertForSequenceClassification.from\_pretrained('bert-base-uncased', num\_labels=3)

model.to(device)

# 5. Fine-tuning Process

# Define Training Arguments

training\_args = TrainingArguments(

output\_dir='./results',

num\_train\_epochs=3,

per\_device\_train\_batch\_size=8,

per\_device\_eval\_batch\_size=8,

warmup\_steps=500,

weight\_decay=0.01,

logging\_dir='./logs',

logging\_steps=10,

evaluation\_strategy="epoch"

)

# Train the Model

trainer = Trainer(

model=model,

args=training\_args,

train\_dataset=train\_dataset['train'],

eval\_dataset=test\_dataset['test'],

compute\_metrics=lambda p: load\_metric("accuracy").compute(predictions=np.argmax(p.predictions, axis=1), references=p.label\_ids)

)

trainer.train()

# 6. Evaluation

# Test the Model and Compare Before and After Fine-tuning

eval\_results\_before = trainer.evaluate(eval\_dataset=test\_dataset['test'])

print(f"Test Accuracy Before Fine-tuning: {eval\_results\_before['eval\_accuracy']}")

# Fine-tune the model

trainer.train()

# Evaluate the fine-tuned model

eval\_results\_after = trainer.evaluate(eval\_dataset=test\_dataset['test'])

print(f"Test Accuracy After Fine-tuning: {eval\_results\_after['eval\_accuracy']}")

# Comparison

improvement = eval\_results\_after['eval\_accuracy'] - eval\_results\_before['eval\_accuracy']

print(f"Accuracy Improvement After Fine-tuning: {improvement}")