```
# IMPORTANT: RUN THIS CELL IN ORDER TO IMPORT YOUR KAGGLE DATA SOURCES,
# THEN FEEL FREE TO DELETE THIS CELL.
# NOTE: THIS NOTEBOOK ENVIRONMENT DIFFERS FROM KAGGLE'S PYTHON
# ENVIRONMENT SO THERE MAY BE MISSING LIBRARIES USED BY YOUR
# NOTEBOOK.


sandeepchatterjee66_ml4crypto_path = kagglehub.dataset_download('sandeepchatterjee66/ml4crypto')

print('Data source import complete.')
```

Start coding or generate with AI.

```
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
```

```
data = pd.read_csv("/content/TrainingData.csv")
data
```

	BID	Bitstream	class
0	0	1000111010111101101100110111001111001000101111...	1
1	1	1101111100101011111111101101010001110110000010...	1
2	2	0011001010001010100100011101000111110100101111...	0
3	3	1101010110000110100001001100111101000000110001...	1
4	4	1010111100001001000101010010111010011101001100...	1
...	...	...	...
1995	1995	1110110011110100001111101111010110011000001110...	0
1996	1996	0100010100011110101110000110100101100000011001...	1
1997	1997	1100001010100011010001110001010010101010101100...	0
1998	1998	00111100000001110101101111110110100010010100011...	1
1999	1999	0100000010100101000000011010011011011111011011...	1

2000 rows × 3 columns

Next steps: Generate code with data View recommended plots New interactive sheet

```
len(data["Bitstream"][0])
```

```
1024
```

```
# import torch
# import numpy as np
# from torch.utils.data import Dataset, DataLoader
# from transformers import GPT2LMHeadModel, GPT2Tokenizer, AdamW, get_linear_schedule_with_warmup
# import os

# # Check if TPU is available
# import os
# import transformers

# if 'COLAB_TPU_ADDR' in os.environ:
#     TPU = True
#     resolver = transformers.TPUMembershipFilter()
#     transformers.utils.set_seed(42)
# else:
#     TPU = False

# # Load and preprocess the data
# class BitstreamDataset(Dataset):
#     def __init__(self, data, tokenizer, max_length=1024):
#         self.data = data
#         self.tokenizer = tokenizer
#         self.max_length = max_length
```

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

#     def __getitem__(self, idx):
#         bitstream = self.data['Bitstream'][idx]
#         label = self.data['class'][idx]
#         inputs = self.tokenizer.encode_plus(bitstream,
#                                             max_length=self.max_length,
#                                             pad_to_max_length=True,
#                                             return_tensors='pt')
#         return inputs, label

# # Fine-tune GPT on TPU
# if TPU:
#     import torch_xla
#     import torch_xla.core.xla_model as xm
#     import torch_xla.distributed.parallel_loader as pl

#     model = GPT2LMHeadModel.from_pretrained('gpt2')
#     tokenizer = GPT2Tokenizer.from_pretrained('gpt2')

#     dataset = BitstreamDataset(data, tokenizer)
#     train_loader = pl.ParallelLoader(dataset, [xm.xla_device()])

#     optimizer = AdamW(model.parameters(), lr=2e-5)
#     scheduler = get_linear_schedule_with_warmup(optimizer,
#                                                  num_warmup_steps=100,
#                                                  num_training_steps=len(train_loader) * 3)

#     model.train()
#     for epoch in range(3):
#         for inputs, labels in train_loader.per_device_loader(xm.xla_device()):
#             outputs = model(inputs, labels=labels)
#             loss = outputs.loss
#             xm.optimizer_step(optimizer)
#             scheduler.step()
#             xm.mark_step()

#     # Evaluate on test set
#     model.eval()
#     accuracy = 0
#     for inputs, labels in test_dataloader:
#         outputs = model(inputs)
#         predictions = outputs.logits.argmax(dim=1)
#         accuracy += (predictions == labels).float().mean()
#     print(f'Test accuracy: {accuracy / len(test_dataloader)}')
# else:
#     # Use CPU/GPU if TPU is not available
#     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
#     # Rest of the code remains the same as before
```

```
pip install torch
```

```
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (2.5.0+cu121)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch) (3.16.1)
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/python3.10/dist-packages (from torch) (4.12.2)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch) (3.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch) (3.1.4)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from torch) (2024.10.0)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.10/dist-packages (from torch) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from sympy==1.13.1->torch) (1.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch) (3.0.2)
```

```
pip install peft
```

```
Requirement already satisfied: peft in /usr/local/lib/python3.10/dist-packages (0.13.2)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-packages (from peft) (1.26.4)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from peft) (24.2)
Requirement already satisfied: psutil in /usr/local/lib/python3.10/dist-packages (from peft) (5.9.5)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.10/dist-packages (from peft) (6.0.2)
Requirement already satisfied: torch>=1.13.0 in /usr/local/lib/python3.10/dist-packages (from peft) (2.5.0+cu121)
Requirement already satisfied: transformers in /usr/local/lib/python3.10/dist-packages (from peft) (4.46.2)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from peft) (4.66.6)
Requirement already satisfied: accelerate>=0.21.0 in /usr/local/lib/python3.10/dist-packages (from peft) (1.1.1)
Requirement already satisfied: safetensors in /usr/local/lib/python3.10/dist-packages (from peft) (0.4.5)
Requirement already satisfied: huggingface-hub>=0.17.0 in /usr/local/lib/python3.10/dist-packages (from peft) (0.26.2)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from huggingface-hub>=0.17.0->peft) (3.16.1)
Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.10/dist-packages (from huggingface-hub>=0.17.0->peft) (202
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from huggingface-hub>=0.17.0->peft) (2.32.3)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.10/dist-packages (from huggingface-hub>=0.17.0-
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch>=1.13.0->peft) (3.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch>=1.13.0->peft) (3.1.4)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.10/dist-packages (from torch>=1.13.0->peft) (1.13.1)
```

```
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from sympy==1.13.1->torch>=1.13.0->pe
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/dist-packages (from transformers->peft) (2024.9.11)
Requirement already satisfied: tokenizers<0.21,>=0.20 in /usr/local/lib/python3.10/dist-packages (from transformers->peft) (0.20.3)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch>=1.13.0->peft) (3.0.2
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->huggingface-hub>=
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->huggingface-hub>=0.17.0->peft
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->huggingface-hub>=0.17.
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->huggingface-hub>=0.17.
```

```python
import torch

# Reload the best model
model.load_state_dict(torch.load(best_model_path))
model.to(device)

# Evaluation loop with reversed logits
model.eval()

# Assuming input_ids, attention_mask, and labels are already tensors:
# Ensure that all tensors are moved to the appropriate device (GPU or CPU)
input_ids = input_ids.to(device)
attention_mask = attention_mask.to(device)
labels = labels.to(device)

# Reverse the logits function
def reverse_logits(logits):
    return logits * -1  # Reverse logits by multiplying by -1

# Test the model without reversing logits
with torch.no_grad():
    outputs = model(input_ids, attention_mask=attention_mask)
    logits = outputs.logits
    predictions = torch.argmax(logits, dim=-1)

    # Calculate accuracy without reversing logits
    correct_predictions = (predictions == labels).sum().item()
    accuracy_without_reverse = correct_predictions / len(labels) * 100

print(f"Accuracy without reversing logits on the entire dataset: {accuracy_without_reverse:.2f}%")

# Test the model with reversed logits
with torch.no_grad():
    reversed_logits = reverse_logits(logits)
    reversed_predictions = torch.argmax(reversed_logits, dim=-1)

    # Calculate accuracy with reversed logits
    correct_predictions = (reversed_predictions == labels).sum().item()
    accuracy_with_reverse = correct_predictions / len(labels) * 100

print(f"Accuracy with reversed logits on the entire dataset: {accuracy_with_reverse:.2f}%")
```

```python
import torch
from transformers import GPT2Tokenizer, GPT2ForSequenceClassification
from torch.utils.data import DataLoader, TensorDataset
from torch.optim import AdamW
from sklearn.model_selection import train_test_split
import pandas as pd
import random
from peft import get_peft_model, LoraConfig, PeftModel

df = data

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Load the tokenizer and model
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
tokenizer.pad_token = tokenizer.eos_token  # Set pad_token to eos_token
model = GPT2ForSequenceClassification.from_pretrained("gpt2", num_labels=2)

# Ensure the model's config uses the pad_token
model.config.pad_token_id = tokenizer.pad_token_id

# Resize embeddings for new pad token
model.resize_token_embeddings(len(tokenizer))
model.to(device)

# LoRA configuration for efficient fine-tuning
lora_config = LoraConfig(
```

```
        r=8,  # rank of low-rank matrices (you can tune this)
        lora_alpha=32,  # scaling factor for LoRA layers
        target_modules=["attn.c_attn", "attn.c_proj"],  # which modules to apply LoRA to
        lora_dropout=0.1,  # dropout for LoRA layers
        bias="none",  # no bias term in LoRA layers
    )

    # Apply LoRA to the model
    model = get_peft_model(model, lora_config)
    model.to(device)

    # Load your dataset (adjust the file path and column names)
    #df = pd.read_csv('/kaggle/input/ml4crypto/TrainingData.csv')  # Replace with your dataset file path

    # Extract binary strings and labels
    binary_strings = df['Bitstream'].tolist()
    labels = df['class'].tolist()

    # Preprocess the binary strings by splitting them into halves and XOR'ing the halves
    def xor_preprocess(binary_string):
        # Split the string into two halves
        s1 = binary_string[:512]
        s2 = binary_string[512:]

        # XOR the halves
        s1_int = int(s1, 2)
        s2_int = int(s2, 2)
        xor_result = s1_int ^ s2_int

        # Convert XOR result back to binary string (512 bits)
        xor_binary_string = format(xor_result, '512b')
        return xor_binary_string

    # Apply preprocessing to all binary strings
    processed_binary_strings = [xor_preprocess(s) for s in binary_strings]

    # Tokenize the processed binary strings
    inputs = tokenizer(processed_binary_strings, padding=True, truncation=True, max_length=1024, return_tensors="pt")

    # Convert to tensors and move to the appropriate device
    input_ids = inputs['input_ids'].to(device)
    attention_mask = inputs['attention_mask'].to(device)
    labels = torch.tensor(labels).to(device)

    # Create a DataLoader for batching
    dataset = TensorDataset(input_ids, attention_mask, labels)
    train_dataset, val_dataset = train_test_split(dataset, test_size=0.2)
    train_dataloader = DataLoader(train_dataset, batch_size=2, shuffle=True)
    val_dataloader = DataLoader(val_dataset, batch_size=2)

    # Optimizer
    optimizer = AdamW(model.parameters(), lr=5e-5)

    # Training loop
    epochs = 15  # Update epoch count to 10 as per your request
    best_accuracy = 0.0
    best_model_path = "gpt_best_model.pth"

    for epoch in range(epochs):
        model.train()
        total_loss = 0
        for batch in train_dataloader:
            # Move the batch to the device
            input_ids, attention_mask, labels = [item.to(device) for item in batch]

            # Forward pass
            outputs = model(input_ids, attention_mask=attention_mask, labels=labels)
            loss = outputs.loss
            total_loss += loss.item()

            # Backward pass and optimization
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        print(f"Epoch {epoch + 1}/{epochs}, Loss: {total_loss / len(train_dataloader)}")

        # Save the best model based on validation accuracy
        model.eval()
        total_correct = 0
        total_samples = 0
        with torch.no_grad():
```

```
        for batch in val_dataloader:
            input_ids, attention_mask, labels = [item.to(device) for item in batch]

            # Forward pass
            outputs = model(input_ids, attention_mask=attention_mask)
            logits = outputs.logits
            predictions = torch.argmax(logits, dim=-1)

            # Calculate accuracy
            total_correct += (predictions == labels).sum().item()
            total_samples += labels.size(0)

    accuracy = total_correct / total_samples
    print(f"Validation Accuracy: {accuracy * 100:.2f}%")

    # Save the model if it's the best
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        torch.save(model.state_dict(), best_model_path)
        print(f"Best model saved with accuracy: {accuracy * 100:.2f}%")

# Reload the best model
model.load_state_dict(torch.load(best_model_path))
model.to(device)

# Evaluation loop with random sampling and reversed logits
model.eval()
```

```
Some weights of GPT2ForSequenceClassification were not initialized from the model checkpoint at gpt2 and are newly initialized: [
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
/usr/local/lib/python3.10/dist-packages/peft/tuners/lora/layer.py:1150: UserWarning: fan_in_fan_out is set to False but the targe
  warnings.warn(
Epoch 1/15, Loss: 0.7280282241478563
Validation Accuracy: 48.50%
Best model saved with accuracy: 48.50%
Epoch 2/15, Loss: 0.7053972987830639
Validation Accuracy: 53.00%
Best model saved with accuracy: 53.00%
Epoch 3/15, Loss: 0.706314246468246
Validation Accuracy: 47.00%
Epoch 4/15, Loss: 0.6977106180042029
Validation Accuracy: 49.75%
Epoch 5/15, Loss: 0.6942617348954081
Validation Accuracy: 49.75%
Epoch 6/15, Loss: 0.6937116514518857
Validation Accuracy: 49.00%
Epoch 7/15, Loss: 0.6957987089455128
Validation Accuracy: 49.75%
Epoch 8/15, Loss: 0.6938062854111194
Validation Accuracy: 51.25%
Epoch 9/15, Loss: 0.6907831660285592
Validation Accuracy: 49.25%
Epoch 10/15, Loss: 0.6924337783828378
Validation Accuracy: 49.00%
Epoch 11/15, Loss: 0.6779078487679362
Validation Accuracy: 49.50%
Epoch 12/15, Loss: 0.684359211884439
Validation Accuracy: 47.50%
Epoch 13/15, Loss: 0.6861557794362306
Validation Accuracy: 48.50%
Epoch 14/15, Loss: 0.679899048730731
Validation Accuracy: 46.25%
Epoch 15/15, Loss: 0.6790768676623702
Validation Accuracy: 46.75%
<ipython-input-14-7b21c4af369d>:133: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default val
  model.load_state_dict(torch.load(best_model_path))
PeftModel(
  (base_model): LoraModel(
    (model): GPT2ForSequenceClassification(
      (transformer): GPT2Model(
        (wte): Embedding(50257, 768)
        (wpe): Embedding(1024, 768)
        (drop): Dropout(p=0.1, inplace=False)
        (h): ModuleList(
          (0-11): 12 x GPT2Block(
            (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (attn): GPT2SdpaAttention(
              (c_attn): lora.Linear(
                (base_layer): Conv1D(nf=2304, nx=768)
                (lora_dropout): ModuleDict(
                  (default): Dropout(p=0.1, inplace=False)
                )
                (lora_A): ModuleDict(
                  (default): Linear(in_features=768, out_features=8, bias=False)
                )
```

```
# Test the best model on the entire validation dataset
model.eval()

# Initialize counters for accuracy
total_correct = 0
total_samples = 0

# Loop over the validation set
with torch.no_grad():
    for batch in val_dataloader:
        input_ids, attention_mask, labels = [item.to(device) for item in batch]

        # Forward pass
        outputs = model(input_ids, attention_mask=attention_mask)
        logits = outputs.logits
        predictions = torch.argmax(logits, dim=-1)

        # Calculate accuracy
        total_correct += (predictions == labels).sum().item()
        total_samples += labels.size(0)

# Calculate final accuracy
accuracy = total_correct / total_samples * 100
print(f"Accuracy on the entire validation dataset: {accuracy:.2f}%")
```

```
Accuracy on the entire validation dataset: 53.00%
```

```
import torch
from transformers import BertTokenizer, BertForSequenceClassification
from torch.utils.data import DataLoader, TensorDataset
from torch.optim import AdamW
from sklearn.model_selection import train_test_split
import pandas as pd
from peft import get_peft_model, LoraConfig

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Load the BERT tokenizer and model
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
model = BertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=2)
model.to(device)

# LoRA configuration for efficient fine-tuning with BERT
lora_config = LoraConfig(
    r=8,  # rank of low-rank matrices
    lora_alpha=32,  # scaling factor for LoRA layers
    target_modules=["attention.self.query", "attention.self.key", "attention.self.value", "attention.output.dense"],  # BERT compatible
    lora_dropout=0.1,  # dropout for LoRA layers
    bias="none",  # no bias term in LoRA layers
)

# Apply LoRA to the model
try:
    model = get_peft_model(model, lora_config)
    model.to(device)
except ValueError as e:
    print(f"Error: {e}")
    print("Please check the target modules. Make sure they match the BERT model structure.")
    raise

# Load your dataset (adjust the file path and column names)
df = data  # Replace with your dataset file path

# Extract binary strings and labels
binary_strings = df['Bitstream'].tolist()
labels = df['class'].tolist()

# Preprocess the binary strings by splitting them into halves and XOR'ing the halves
def xor_preprocess(binary_string):
    s1 = binary_string[:512]
    s2 = binary_string[512:]
    s1_int = int(s1, 2)
    s2_int = int(s2, 2)
    xor_result = s1_int ^ s2_int
    xor_binary_string = format(xor_result, '0512b')  # Adjusted to ensure 512 bits
    return xor_binary_string

# Apply preprocessing to all binary strings
processed_binary_strings = [xor_preprocess(s) for s in binary_strings]

# Tokenize the processed binary strings
```

```
inputs = tokenizer(processed_binary_strings, padding=True, truncation=True, max_length=512, return_tensors="pt")

# Convert to tensors and move to the appropriate device
input_ids = inputs['input_ids'].to(device)
attention_mask = inputs['attention_mask'].to(device)
labels = torch.tensor(labels).to(device)

# Create a DataLoader for batching
dataset = TensorDataset(input_ids, attention_mask, labels)
train_dataset, val_dataset = train_test_split(dataset, test_size=0.2)
train_dataloader = DataLoader(train_dataset, batch_size=2, shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size=2)

# Optimizer
optimizer = AdamW(model.parameters(), lr=5e-5)

# Training loop
epochs = 20  # Update epoch count to 20
best_accuracy = 0.0
best_model_path = "best_model.pth"

for epoch in range(epochs):
    model.train()
    total_loss = 0
    for batch in train_dataloader:
        input_ids, attention_mask, labels = [item.to(device) for item in batch]

        # Forward pass
        outputs = model(input_ids, attention_mask=attention_mask, labels=labels)
        loss = outputs.loss
        total_loss += loss.item()

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Epoch {epoch + 1}/{epochs}, Loss: {total_loss / len(train_dataloader)}")

    # Validation phase
    model.eval()
    total_correct = 0
    total_samples = 0
    with torch.no_grad():
        for batch in val_dataloader:
            input_ids, attention_mask, labels = [item.to(device) for item in batch]

            # Forward pass
            outputs = model(input_ids, attention_mask=attention_mask)
            logits = outputs.logits
            predictions = torch.argmax(logits, dim=-1)

            # Calculate accuracy
            total_correct += (predictions == labels).sum().item()
            total_samples += labels.size(0)

    accuracy = total_correct / total_samples
    print(f"Validation Accuracy: {accuracy * 100:.2f}%")

    # Save the best model based on validation accuracy
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        torch.save(model.state_dict(), best_model_path)
        print(f"Best model saved with accuracy: {accuracy * 100:.2f}%")

# Reload the best model
model.load_state_dict(torch.load(best_model_path))
model.to(device)

# Final evaluation on the full dataset can now proceed here.
```

```
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly i
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Epoch 1/20, Loss: 0.7050536767393351
Validation Accuracy: 41.50%
Best model saved with accuracy: 41.50%
Epoch 2/20, Loss: 0.6985265891999006
Validation Accuracy: 58.50%
Best model saved with accuracy: 58.50%
Epoch 3/20, Loss: 0.6949944455549121
Validation Accuracy: 41.50%
Epoch 4/20, Loss: 0.6966877183318139
```

```
Validation Accuracy: 41.50%
Epoch 5/20, Loss: 0.6979087771475315
Validation Accuracy: 41.50%
Epoch 6/20, Loss: 0.6975773316621781
Validation Accuracy: 58.50%
Epoch 7/20, Loss: 0.6938082890212536
Validation Accuracy: 41.50%
Epoch 8/20, Loss: 0.695544774979353
Validation Accuracy: 41.50%
Epoch 9/20, Loss: 0.6936459349095822
Validation Accuracy: 41.50%
Epoch 10/20, Loss: 0.6969192644208669
Validation Accuracy: 41.50%
Epoch 11/20, Loss: 0.6975535332411528
Validation Accuracy: 58.50%
Epoch 12/20, Loss: 0.6931470593810082
Validation Accuracy: 41.50%
Epoch 13/20, Loss: 0.6929664281755685
Validation Accuracy: 41.50%
Epoch 14/20, Loss: 0.6973433938622474
Validation Accuracy: 58.50%
Epoch 15/20, Loss: 0.6962995431572199
Validation Accuracy: 41.50%
Epoch 16/20, Loss: 0.6941995688527822
Validation Accuracy: 41.50%
Epoch 17/20, Loss: 0.6922929346561432
Validation Accuracy: 41.50%
Epoch 18/20, Loss: 0.6942176257818937
Validation Accuracy: 41.50%
Epoch 19/20, Loss: 0.6971428709477187
Validation Accuracy: 41.50%
Epoch 20/20, Loss: 0.6940503816306591
Validation Accuracy: 41.50%
<ipython-input-12-47456f2e6641>:122: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default val
  model.load_state_dict(torch.load(best_model_path))
PeftModel(
  (base_model): LoraModel(
    (model): BertForSequenceClassification(
      (bert): BertModel(
        (embeddings): BertEmbeddings(
          (word_embeddings): Embedding(30522, 768, padding_idx=0)
          (position_embeddings): Embedding(512, 768)
          (token_type_embeddings): Embedding(2, 768)
          (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (dropout): Dropout(p=0.1, inplace=False)
```

```
# Test the best model on the entire validation dataset
model.eval()

# Initialize counters for accuracy
total_correct = 0
total_samples = 0

# Loop over the validation set
with torch.no_grad():
    for batch in val_dataloader:
        input_ids, attention_mask, labels = [item.to(device) for item in batch]

        # Forward pass
        outputs = model(input_ids, attention_mask=attention_mask)
        logits = outputs.logits
        predictions = torch.argmax(logits, dim=-1)

        # Calculate accuracy
        total_correct += (predictions == labels).sum().item()
        total_samples += labels.size(0)

# Calculate final accuracy
accuracy = total_correct / total_samples * 100
print(f"Accuracy on the entire validation dataset: {accuracy:.2f}%")

# Optionally: Reverse logits and calculate accuracy again
# def reverse_logits(logits):
#     return logits * -1  # Reverse logits by multiplying by -1

# # Test the model with reversed logits on the entire validation dataset
# total_correct_reversed = 0
# with torch.no_grad():
#     for batch in val_dataloader:
#         input_ids, attention_mask, labels = [item.to(device) for item in batch]

#         # Forward pass
#         outputs = model(input_ids, attention_mask=attention_mask)
#         logits = outputs.logits
```

```
#         # Reverse the logits
#         reversed_logits = reverse_logits(logits)
#         reversed_predictions = torch.argmax(reversed_logits, dim=-1)

#         # Calculate accuracy with reversed logits
#         total_correct_reversed += (reversed_predictions == labels).sum().item()

# # Calculate final accuracy with reversed logits
# accuracy_reversed = total_correct_reversed / total_samples * 100
# print(f"Accuracy with reversed logits on the entire validation dataset: {accuracy_reversed:.2f}%")
```

```
Accuracy on the entire validation dataset: 58.50%
```