```
import random
import csv

# Function to generate the dataset with label noise
def generate_large_dataset(label_noise_prob=0.15):  # Reduced from 0.2 to 0.15
    subjects = [
        "I", "You", "He", "She", "We", "They", "The day", "My friend", "The movie",
        "The weather", "Life", "Work", "School", "The team", "The food", "My phone",
        "The game", "The trip", "This place", "That idea", "The book", "The party",
        "The project", "The city", "The experience"
    ]
    verbs = [
        "is", "feels", "seems", "looks", "was", "went", "sounds", "appears",
        "has been", "will be", "turned out", "became", "remains", "gets"
    ]
    positives = [
        "great", "awesome", "wonderful", "good", "fantastic", "amazing", "nice",
        "perfect", "lovely", "excellent", "brilliant", "super", "fabulous", "cool",
        "incredible", "enjoyable", "beautiful", "exciting", "happy", "pleasant",
        "satisfying", "delightful", "impressive", "charming", "splendid"
    ]
    negatives = [
        "bad", "terrible", "awful", "horrible", "poor", "lousy", "dreadful", "sad",
        "miserable", "disappointing", "rotten", "pathetic", "annoying", "boring",
        "frustrating", "ugly", "depressing", "lame", "unpleasant", "irritating",
        "disastrous", "grim", "hopeless", "dull", "bleak"
    ]
    neutrals = [
        "okay", "fine", "average", "so-so", "normal", "decent", "alright", "typical",
        "nothing special", "fair", "passable", "mediocre", "standard", "plain",
        "usual", "middling", "adequate", "not great", "not bad", "tolerable",
        "acceptable", "unremarkable", "neutral", "moderate", "sufficient"
    ]
    adverbs = [
        "really", "very", "quite", "pretty", "so", "totally", "somewhat", "kind of",
        "a bit", "slightly", "", ""  # Empty for natural variation
    ]
    connectors = [
        "and", "but", "because", "though", "since", "while", "yet", ""
    ]

    simple_templates = [
        "{subject} {verb} {adverb} {sentiment}.",
        "{subject} {verb} {sentiment} today.",
        "{subject} {verb} {adverb} {sentiment} lately.",
        "{subject} {verb} {sentiment} this week."
    ]
    question_templates = [
        "Is {subject} {adverb} {sentiment}?",
        "Why {verb} {subject} {adverb} {sentiment}?",
        "Does {subject} {verb} {sentiment}?",
        "How {adverb} {sentiment} {verb} {subject}?"
    ]
    exclamation_templates = [
        "Wow, {subject} {verb} {adverb} {sentiment}!",
        "What a {sentiment} time {subject} {verb}!",
        "So {sentiment} that {subject} {verb}!",
        "How {sentiment} {subject} {verb}!"
    ]
    compound_templates = [
        "{subject} {verb} {adverb} {sentiment} {connector} it’s fine.",
        "I think {subject} {verb} {sentiment} {connector} that’s true.",
        "{subject} {verb} {sentiment} {connector} I don’t mind.",
        "{subject} {verb} {adverb} {sentiment} {connector} it could be worse."
    ]

    train_size, test_size, val_size = 5000, 1000, 1000
    total_size = train_size + test_size + val_size  # 7000
    train_per_class = 1667
    test_val_per_class = 333
    total_per_class = train_per_class + test_val_per_class + test_val_per_class

    texts = []
    labels = []

    for sentiment_list, label in [(negatives, 0), (positives, 1), (neutrals, 2)]:
        count = 0
        while count < total_per_class:
            template_type = random.random()
            if template_type < 0.3:
                template = random.choice(simple_templates)
```

```
                    sentence = template.format(
                        subject=random.choice(subjects),
                        verb=random.choice(verbs),
                        adverb=random.choice(adverbs),
                        sentiment=random.choice(sentiment_list)
                    )
                elif template_type < 0.6:
                    template = random.choice(question_templates)
                    sentence = template.format(
                        subject=random.choice(subjects),
                        verb=random.choice(verbs),
                        adverb=random.choice(adverbs),
                        sentiment=random.choice(sentiment_list)
                    )
                elif template_type < 0.8:
                    template = random.choice(exclamation_templates)
                    sentence = template.format(
                        subject=random.choice(subjects),
                        verb=random.choice(verbs),
                        adverb=random.choice(adverbs),
                        sentiment=random.choice(sentiment_list)
                    )
                else:
                    template = random.choice(compound_templates)
                    sentence = template.format(
                        subject=random.choice(subjects),
                        verb=random.choice(verbs),
                        adverb=random.choice(adverbs),
                        sentiment=random.choice(sentiment_list),
                        connector=random.choice(connectors)
                    )
                if sentence not in texts and not (label == 1 and "nothing" in sentence.lower() and "special" not in sentence.lower()):
                    texts.append(sentence)
                    # Add label noise
                    if random.random() < label_noise_prob:
                        noisy_label = random.choice([0, 1, 2])
                        while noisy_label == label:  # Ensure the noisy label is different
                            noisy_label = random.choice([0, 1, 2])
                        labels.append(noisy_label)
                    else:
                        labels.append(label)
                    count += 1

        combined = list(zip(texts, labels))
        random.shuffle(combined)
        texts, labels = zip(*combined)
        texts, labels = list(texts[:total_size]), list(labels[:total_size])

        train_texts, train_labels = [], []
        test_texts, test_labels = [], []
        val_texts, val_labels = [], []

        class_counts = {0: 0, 1: 0, 2: 0}
        for text, label in zip(texts, labels):
            if class_counts[label] < train_per_class:
                train_texts.append(text)
                train_labels.append(label)
                class_counts[label] += 1
            elif class_counts[label] < train_per_class + test_val_per_class:
                test_texts.append(text)
                test_labels.append(label)
                class_counts[label] += 1
            elif class_counts[label] < train_per_class + test_val_per_class + test_val_per_class:
                val_texts.append(text)
                val_labels.append(label)
                class_counts[label] += 1

        train_texts, train_labels = train_texts[:5000], train_labels[:5000]
        test_texts, test_labels = test_texts[:1000], test_labels[:1000]
        val_texts, val_labels = val_texts[:1000], val_labels[:1000]

        return (train_texts, train_labels), (test_texts, test_labels), (val_texts, val_labels)

    # Save dataset to CSV files
    def save_dataset_to_csv(train_data, test_data, val_data):
        def write_csv(filename, texts, labels):
            with open(filename, 'w', newline='', encoding='utf-8') as f:
                writer = csv.writer(f)
                writer.writerow(['text', 'label'])
                for text, label in zip(texts, labels):
                    writer.writerow([text, label])

        write_csv('train_dataset.csv', train_data[0], train_data[1])
```

```
    write_csv('test_dataset.csv', test_data[0], test_data[1])
    write_csv('val_dataset.csv', val_data[0], val_data[1])
    print("Datasets saved as CSV: train_dataset.csv, test_dataset.csv, val_dataset.csv")

# Generate and save the dataset
if __name__ == "__main__":
    (train_texts, train_labels), (test_texts, test_labels), (val_texts, val_labels) = generate_large_dataset(label_noise_prob=0.15)

    print(f"Training set: {len(train_texts)} samples")
    print(f"Testing set: {len(test_texts)} samples")
    print(f"Validation set: {len(val_texts)} samples")

    for name, labels in [("Train", train_labels), ("Test", test_labels), ("Val", val_labels)]:
        neg = sum(1 for l in labels if l == 0)
        pos = sum(1 for l in labels if l == 1)
        neu = sum(1 for l in labels if l == 2)
        print(f"{name} - Negative: {neg}, Positive: {pos}, Neutral: {neu}")

    save_dataset_to_csv(
        (train_texts, train_labels),
        (test_texts, test_labels),
        (val_texts, val_labels)
    )
```

```
Training set: 5000 samples
Testing set: 999 samples
Validation set: 988 samples
Train - Negative: 1667, Positive: 1667, Neutral: 1666
Test - Negative: 333, Positive: 333, Neutral: 333
Val - Negative: 333, Positive: 322, Neutral: 333
Datasets saved as CSV: train_dataset.csv, test_dataset.csv, val_dataset.csv
```

```
pip install torch transformers datasets pandas numpy matplotlib seaborn gradio==4.44.0 scikit-learn openai
```

Show hidden output

```
import torch
from torch.utils.data import Dataset, DataLoader
import gradio as gr
from transformers import AutoTokenizer, DistilBertForSequenceClassification, DistilBertConfig
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
from tqdm import tqdm
import os
import openai
from openai import OpenAIError

# Set up the OpenAI API key (retrieve from environment variable or set directly)
try:
    openai.api_key = os.getenv("OPENAI_API_KEY")
    if not openai.api_key:
        raise ValueError("OpenAI API key not found in environment variables.")
    client = openai.Client(api_key=openai.api_key)
    USE_GPT4O = True
    print("OpenAI API key found. GPT-4o will be used for sentiment analysis if available.")
except (ValueError, OpenAIError) as e:
    print(f"Failed to initialize OpenAI API: {e}. Falling back to DistilBERT.")
    USE_GPT4O = False

# 1. Data Preparation
class SentimentDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_length=128):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_length = max_length

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

    def __getitem__(self, idx):
        text = str(self.texts[idx])
        label = self.labels[idx]

        if not isinstance(label, (int, np.integer)) or label < 0 or label > 2:
            raise ValueError(f"Invalid label at index {idx}: {label}. Must be an integer in [0, 2].")

        encoding = self.tokenizer(
```

```
                text,
                add_special_tokens=True,
                max_length=self.max_length,
                padding='max_length',
                truncation=True,
                return_tensors='pt'
            )

            input_ids = encoding['input_ids'].flatten()
            attention_mask = encoding['attention_mask'].flatten()
            if input_ids.shape != (self.max_length,) or attention_mask.shape != (self.max_length,):
                raise ValueError(f"Invalid encoding shape at index {idx}: input_ids {input_ids.shape}, attention_mask {attention_mask.shape}

            return {
                'input_ids': input_ids,
                'attention_mask': attention_mask,
                'labels': torch.tensor(label, dtype=torch.long)
            }

    # 2. Model Setup with Dropout and Freezing Layers (Using DistilBERT)
    def initialize_model():
        tokenizer = AutoTokenizer.from_pretrained('distilbert-base-uncased')
        config = DistilBertConfig.from_pretrained('distilbert-base-uncased', num_labels=3, dropout=0.3)
        model = DistilBertForSequenceClassification.from_pretrained('distilbert-base-uncased', config=config)

        for param in model.distilbert.transformer.layer[:1].parameters():
            param.requires_grad = False

        return model, tokenizer

    # 3. Trainer with Human Feedback, Validation, and GPT-4o Integration
    class SentimentTrainer:
        def __init__(self, model, tokenizer):
            self.model = model
            self.tokenizer = tokenizer
            self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
            self.model.to(self.device)
            self.training_losses = []
            self.validation_losses = []
            self.feedback_data = []
            self.feedback_predictions = []
            self.all_feedback_predictions = []
            self.accuracy_history = []
            self.feedback_threshold = 5
            self.loss_scale_factor = 12.0
            self.min_loss_threshold = 3.0
            self.val_loss_increase_tolerance = 0.05
            self.min_epochs_before_stopping = 2

        def train(self, train_dataset, val_dataset, epochs=4, batch_size=16, drop_last=True, lr=5e-5):
            train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, drop_last=drop_last, num_workers=2)
            val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False, num_workers=2)
            optimizer = torch.optim.AdamW(self.model.parameters(), lr=lr, weight_decay=0.01)

            self.model.train()
            total_batches = len(train_dataloader)
            print(f"Total batches per epoch: {total_batches}")

            if total_batches == 0:
                print("Warning: No batches to train on. Check dataset size and batch_size.")
                return self.training_losses, self.validation_losses

            for epoch in range(epochs):
                epoch_train_loss = 0
                self.model.train()
                for batch in tqdm(train_dataloader, total=total_batches, desc=f"Epoch {epoch + 1}/{epochs}"):
                    optimizer.zero_grad()
                    input_ids = batch['input_ids'].to(self.device)
                    attention_mask = batch['attention_mask'].to(self.device)
                    labels = batch['labels'].to(self.device)

                    outputs = self.model(input_ids=input_ids, attention_mask=attention_mask, labels=labels)
                    loss = outputs.loss
                    scaled_loss = loss * self.loss_scale_factor
                    scaled_loss = torch.max(scaled_loss, torch.tensor(self.min_loss_threshold, device=self.device))
                    scaled_loss.backward()
                    torch.nn.utils.clip_grad_norm_(self.model.parameters(), max_norm=1.0)
                    optimizer.step()

                    epoch_train_loss += scaled_loss.item()

                avg_train_loss = epoch_train_loss / total_batches
```

```
            self.training_losses.append(avg_train_loss)
            print(f"Epoch {epoch + 1}/{epochs} completed, Avg Train Loss: {avg_train_loss:.4f}")

            self.model.eval()
            epoch_val_loss = 0
            with torch.no_grad():
                for batch in val_dataloader:
                    input_ids = batch['input_ids'].to(self.device)
                    attention_mask = batch['attention_mask'].to(self.device)
                    labels = batch['labels'].to(self.device)
                    outputs = self.model(input_ids=input_ids, attention_mask=attention_mask, labels=labels)
                    loss = outputs.loss
                    scaled_loss = loss * self.loss_scale_factor
                    scaled_loss = torch.max(scaled_loss, torch.tensor(self.min_loss_threshold, device=self.device))
                    epoch_val_loss += scaled_loss.item()

            avg_val_loss = epoch_val_loss / len(val_dataloader)
            self.validation_losses.append(avg_val_loss)
            print(f"Validation Loss: {avg_val_loss:.4f}")

            if avg_val_loss < self.min_loss_threshold:
                print(f"Validation loss {avg_val_loss:.4f} below minimum threshold {self.min_loss_threshold}, stopping early.")
                break
            if epoch > self.min_epochs_before_stopping:
                prev_val_loss = self.validation_losses[epoch-1]
                if avg_val_loss > prev_val_loss * (1 + self.val_loss_increase_tolerance):
                    print(f"Validation loss increased significantly (from {prev_val_loss:.4f} to {avg_val_loss:.4f}), stopping early.")
                    break

        return self.training_losses, self.validation_losses

    def retrain_with_feedback(self, batch_size=4):
        print(f"Checking feedback entries: {len(self.feedback_data)}")
        if len(self.feedback_data) < self.feedback_threshold:
            print(f"Feedback entries: {len(self.feedback_data)}. Need {self.feedback_threshold} to retrain.")
            return False

        print("Retraining with human feedback...")
        feedback_texts = [item[0] for item in self.feedback_data]
        feedback_labels = [item[1] for item in self.feedback_data]
        feedback_dataset = SentimentDataset(feedback_texts, feedback_labels, self.tokenizer)
        self.train(feedback_dataset, feedback_dataset, epochs=1, batch_size=batch_size, drop_last=False, lr=1e-5)
        self.feedback_data = []
        self.feedback_predictions = []
        print("Feedback data and predictions cleared after retraining.")
        return True

    def predict(self, text):
        if not text.strip():
            return 2, 0.5

        if USE_GPT4O:
            try:
                prompt = f"Classify the sentiment of the following text as 'positive', 'negative', or 'neutral':\n\n{text}\n\nReturn on
                response = client.chat.completions.create(
                    model="gpt-4o",
                    messages=[
                        {"role": "system", "content": "You are a sentiment analysis expert."},
                        {"role": "user", "content": prompt}
                    ],
                    max_tokens=10,
                    temperature=0.0
                )
                gpt_sentiment = response.choices[0].message.content.strip().lower()
                print(f"GPT-4o predicted sentiment: {gpt_sentiment}")

                sentiment_map = {"negative": 0, "positive": 1, "neutral": 2}
                if gpt_sentiment not in sentiment_map:
                    raise ValueError(f"Invalid sentiment returned by GPT-4o: {gpt_sentiment}")

                sentiment = sentiment_map[gpt_sentiment]
                score = 0.9
                print(f"Input: {text}, Predicted (via GPT-4o): {sentiment}, Score: {score}")
                return sentiment, score

            except (OpenAIError, ValueError, Exception) as e:
                print(f"Error using GPT-4o: {e}. Falling back to DistilBERT.")

        self.model.eval()
        inputs = self.tokenizer(text, return_tensors='pt', padding=True, truncation=True, max_length=128)
        inputs = {k: v.to(self.device) for k, v in inputs.items()}
```

```
            with torch.no_grad():
                outputs = self.model(**inputs)
                logits = outputs.logits
                probs = torch.softmax(logits, dim=-1)
                sentiment = torch.argmax(probs, dim=-1).item()
                score = probs[0][sentiment].item()
                print(f"Input: {text}, Probabilities: {probs.tolist()[0]}, Predicted (via DistilBERT): {sentiment}, Score: {score}")
                return sentiment, score

        def add_feedback(self, text, human_label):
            label_map = {"Negative": 0, "Positive": 1, "Neutral": 2}
            print(f"Received feedback for text: {text}, human_label: {human_label}")
            if human_label not in label_map:
                print(f"Invalid feedback label: {human_label}. Expected one of {list(label_map.keys())}")
                return False

            pred_sentiment, pred_score = self.predict(text)
            self.feedback_data.append((text, label_map[human_label]))
            self.feedback_predictions.append((text, pred_sentiment, human_label))
            self.all_feedback_predictions.append((text, pred_sentiment, human_label))
            accuracy = self.calculate_accuracy()
            self.accuracy_history.append(accuracy)
            print(f"Feedback added. Current feedback_data: {self.feedback_data}")
            print(f"Current feedback_predictions: {self.feedback_predictions}")
            print(f"All feedback predictions: {self.all_feedback_predictions}")
            print(f"Accuracy history: {self.accuracy_history}")
            retrained = self.retrain_with_feedback()
            return retrained

        def calculate_accuracy(self):
            if not self.all_feedback_predictions:
                return 0
            correct = 0
            total = 0
            for _, pred, feedback in self.all_feedback_predictions:
                if pred == {"Negative": 0, "Positive": 1, "Neutral": 2}[feedback]:
                    correct += 1
                total += 1
            return correct / total if total > 0 else 0

    # 4. Visualization with Validation Loss and Accuracy Line Graph (in Percentage)
    def create_visualizations(trainer):
        plt.figure(figsize=(18, 4))

        plt.subplot(1, 3, 1)
        if trainer.training_losses:
            plt.plot(trainer.training_losses, 'b-o', label='Train Loss')
            plt.plot(trainer.validation_losses, 'r-o', label='Val Loss')
            plt.legend()
        plt.title('Training and Validation Loss')
        plt.xlabel('Epoch')
        plt.ylabel('Loss')

        plt.subplot(1, 3, 2)
        if trainer.all_feedback_predictions:
            scores = [trainer.predict(text)[1] for text, _, _ in trainer.all_feedback_predictions]
            sns.histplot(scores, bins=20)
        plt.title('Sentiment Predictions (Feedback)')
        plt.xlabel('Score')

        plt.subplot(1, 3, 3)
        if trainer.accuracy_history:
            accuracy_percent = [acc * 100 for acc in trainer.accuracy_history]
            plt.plot(accuracy_percent, 'g-o', label='Accuracy (%)')
            plt.legend()
            plt.ylim(0, 110)  # Adjusted upper limit to 110 to make 100% visible
            plt.ylabel('Accuracy (%)')
        else:
            plt.text(0.5, 0.5, 'No feedback yet', ha='center')
        plt.title('Accuracy Over Feedback Cycles')
        plt.xlabel('Feedback Entry')

        plt.tight_layout()
        return plt

    # 5. Gradio Interface (Updated Title and Centered Alignment)
    def create_interface(trainer):
        # Custom CSS to reduce padding, margins, and center the title
        custom_css = """
        .gr-column {
            padding: 5px !important;
            margin: 0 !important;
```

```
            display: flex !important;
            flex-direction: column !important;
        }
        .gr-textbox, .gr-radio, .gr-button {
            margin-bottom: 5px !important;
        }
        .update-visualizations-container {
            display: flex !important;
            justify-content: center !important;
            margin-top: 10px !important;
        }
        .title {
            text-align: center !important;
        }
        """

        def predict_sentiment(text):
            print("Predict Sentiment called with text:", text)
            sentiment, score = trainer.predict(text)
            sentiment_map = {0: "Negative", 1: "Positive", 2: "Neutral"}
            return f"Sentiment: {sentiment_map[sentiment]} (Score: {score:.3f})"

        def provide_feedback(text, feedback):
            print("Provide Feedback called with text:", text, "feedback:", feedback)
            if feedback is None:
                return "Please select a feedback option", None, ""
            print(f"Gradio feedback received: text={text}, feedback={feedback}")
            retrained = trainer.add_feedback(text, feedback)
            updated_plot = create_visualizations(trainer)
            retrain_message = "Retraining triggered after 5th feedback!" if retrained else ""
            return f"Feedback recorded: {feedback}", updated_plot, retrain_message

        with gr.Blocks(css=custom_css, analytics_enabled=False) as demo:  # Disable analytics
            print("Creating Gradio interface...")
            gr.Markdown(
                "# Fine-Tuning Language Model for Sentiment Analysis with Feedback Loop",
                elem_classes=["title"]
            )

            # Main layout: Row with two columns, redistributed components
            with gr.Row():
                with gr.Column(variant="compact", scale=1):
                    text_input = gr.Textbox(label="Enter text", lines=1)
                    predict_btn = gr.Button("Analyze")
                    sentiment_output = gr.Textbox(
                        label="Prediction",
                        interactive=False,
                        lines=1
                    )
                    feedback_input = gr.Radio(
                        ["Positive", "Negative", "Neutral"],
                        label="Your Feedback",
                        min_width=200
                    )
                    print("Created text_input, predict_btn, sentiment_output, and feedback_input")
                with gr.Column(variant="compact", scale=1):
                    feedback_btn = gr.Button("Submit Feedback")
                    feedback_output = gr.Textbox(
                        label="Feedback Status",
                        interactive=False,
                        min_width=200,
                        lines=2
                    )
                    retrain_status = gr.Textbox(
                        label="Retraining Status",
                        interactive=False,
                        min_width=200,
                        lines=5
                    )
                    print("Created feedback_btn, feedback_output, and retrain_status")

            # Visualizations section (below the row)
            plot_output = gr.Plot(label="Visualizations")
            # Center the Update Visualizations button
            with gr.Group(elem_classes=["update-visualizations-container"]):
                update_plot_btn = gr.Button("Update Visualizations")
            print("Created plot_output and update_plot_btn")

            # Set up event handlers
            predict_btn.click(
                fn=predict_sentiment,
                inputs=text_input,
```

```
            outputs=sentiment_output
        )
        print("Set up predict_btn event handler")

        feedback_btn.click(
            fn=provide_feedback,
            inputs=[text_input, feedback_input],
            outputs=[feedback_output, plot_output, retrain_status]
        )
        print("Set up feedback_btn event handler")

        update_plot_btn.click(
            fn=lambda: create_visualizations(trainer),
            outputs=plot_output
        )
        print("Set up update_plot_btn event handler")

        # Initialize the plot on launch
        demo.load(
            fn=lambda: create_visualizations(trainer),
            outputs=plot_output
        )
        print("Set up demo.load for initial plot")

    print("Gradio interface created successfully")
    return demo

# 6. Main Function (Updated to Remove Model Saving/Loading)
def main():
    # Load datasets
    for file in ['train_dataset.csv', 'test_dataset.csv', 'val_dataset.csv']:
        if not os.path.exists(file):
            raise FileNotFoundError(f"Dataset file {file} not found. Please run the dataset generation script first.")

    train_df = pd.read_csv('train_dataset.csv')
    val_df = pd.read_csv('val_dataset.csv')

    train_texts, train_labels = train_df['text'].tolist(), train_df['label'].tolist()
    val_texts, val_labels = val_df['text'].tolist(), val_df['label'].tolist()

    # Initialize a new model (no loading)
    print("Training a new model...")
    model, tokenizer = initialize_model()

    # Initialize the trainer
    train_dataset = SentimentDataset(train_texts, train_labels, tokenizer)
    val_dataset = SentimentDataset(val_texts, val_labels, tokenizer)
    trainer = SentimentTrainer(model, tokenizer)

    # Train the model (no saving/loading of trainer state)
    trainer.train(train_dataset, val_dataset, epochs=4, batch_size=16)

    # Launch the Gradio interface
    interface = create_interface(trainer)
    interface.launch()

if __name__ == "__main__":
    main()
```

```
Failed to initialize OpenAI API: OpenAI API key not found in environment variables.. Falling back to DistilBERT.
Training a new model...
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as s
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-base-uncased and ar
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Total batches per epoch: 312
Epoch 1/4: 100%|██████████| 312/312 [51:57<00:00,  9.99s/it]Epoch 1/4 completed, Avg Train Loss: 7.5311

Validation Loss: 6.6800
Epoch 2/4:   8%|▊         | 24/312 [04:11<48:26, 10.09s/it]
```