

## DATA 266 – Generative Models and Applications

### Spring 2025- Homework 1

Hamsalakshmi Ramachandran (017423666)

#### Question 1:

```
!pip install datasets transformers torch scikit-learn pandas numpy
```

```
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.11/dist-packages (from transformers) (2024.11.6)
Requirement already satisfied: tokenizers<0.22,>=0.21 in /usr/local/lib/python3.11/dist-packages (from transformers) (0.21.0)
Requirement already satisfied: safetensors>=0.4.1 in /usr/local/lib/python3.11/dist-packages (from transformers) (0.5.2)
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/python3.11/dist-packages (from torch) (4.12.2)
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-packages (from torch) (3.4.2)
Requirement already satisfied: Jinja2 in /usr/local/lib/python3.11/dist-packages (from torch) (3.1.5)
Collecting nvidia-cuda-nvrtc-cu12==12.4.127 (from torch)
  Downloading nvidia_cuda_nvrtc_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-runtime-cu12==12.4.127 (from torch)
  Downloading nvidia_cuda_runtime_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-cupti-cu12==12.4.127 (from torch)
  Downloading nvidia_cuda_cupti_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cudnn-cu12==9.1.0.70 (from torch)
```

#### Part A:

```
import torch
from transformers import (
    PreTrainedTokenizerFast,
    DistilBertTokenizerFast,
    RobertaTokenizerFast,
    DistilBertForSequenceClassification,
    AutoModelForSequenceClassification
)
from datasets import load_dataset
import numpy as np
from sklearn.metrics import accuracy_score, f1_score
import time
import random
```

Loads the IMDB movie review dataset. Samples a specific number of examples: 3,000 training examples from the training set & 300 test examples from the test set. It uses shuffle with seed=42 to ensure consistent random sampling. It returns both training and

test datasets.

setup\_tokenizers(): special tokens needed for transformer models: [PAD] for padding sequences to same length. [UNK] for unknown words. [CLS], [SEP], [MASK] for transformer model operations. Sets vocabulary size to 30,000 tokens. Creates a configuration dictionary with all these settings.

```
# Initialize DistilBERT tokenizer (WordPiece)
distilbert_tokenizer = DistilBertTokenizerFast.from_pretrained(
    'distilbert-base-uncased',
    vocab_size=tokenizer_config["vocab_size"],
    unk_token=tokenizer_config["unk_token"],
    pad_token=tokenizer_config["pad_token"]
)

# Initialize DistilRoBERTa tokenizer (BPE)
distilroberta_tokenizer = RobertaTokenizerFast.from_pretrained(
    'distilroberta-base',
    vocab_size=tokenizer_config["vocab_size"],
    unk_token=tokenizer_config["unk_token"],
    pad_token=tokenizer_config["pad_token"]
)

return distilbert_tokenizer, distilroberta_tokenizer

# Evaluation function
def evaluate_zero_shot(model, tokenizer, test_data, device):
    model.eval()
    predictions = []
    true_labels = []
    total_time = 0

    with torch.no_grad():
        for example in test_data:
            start_time = time.time()
```

```
# Set random seed for reproducibility
def set_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)

# Load and prepare the IMDB dataset
def load_and_prepare_data(train_size=3000, test_size=300):
    # Load dataset
    dataset = load_dataset("imdb")

    # Sample the required sizes
    train_data = dataset["train"].shuffle(seed=42).select(range(train_size))
    test_data = dataset["test"].shuffle(seed=42).select(range(test_size))

    return train_data, test_data

# Configure tokenizers with the specified parameters
def setup_tokenizers():
    # Common tokenizer settings
    special_tokens = ["[PAD]", "[UNK]", "[CLS]", "[SEP]", "[MASK]"]
    tokenizer_config = {
        "vocab_size": 30000,
        "unk_token": "[UNK]",
        "pad_token": "[PAD]",
        "special_tokens": special_tokens
    }
```

WordPiece (DistilBERT): Breaks words into subwords based on most common combinations

BPE (DistilRoBERTa): Merges most frequent pairs of characters iteratively

```

# Tokenize and prepare input
encoding = tokenizer(
    example["text"],
    truncation=True,
    padding=True,
    max_length=512,
    return_tensors="pt"
).to(device)

# Get model prediction
outputs = model(**encoding)
prediction = torch.argmax(outputs.logits, dim=1)
end_time = time.time()
total_time += (end_time - start_time)
predictions.append(prediction.item())
true_labels.append(example["label"])

# Calculate metrics
accuracy = accuracy_score(true_labels, predictions)
f1 = f1_score(true_labels, predictions, average='binary')
avg_inference_time = total_time / len(test_data)

return {
    'accuracy': round(accuracy, 4),
    'f1_score': round(f1, 4),
    'avg_inference_time': round(avg_inference_time, 4)
}

```

```

def main():
    # Set seed
    set_seed(42)

    # Set device (Colab should provide GPU)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Using device: {device}")

    # Load data
    train_data, test_data = load_and_prepare_data()
    print(f"Loaded {len(train_data)} training examples and {len(test_data)} test examples")

    # Setup tokenizers
    distilbert_tokenizer, distilroberta_tokenizer = setup_tokenizers()

    # Initialize models
    distilbert_model = DistilBertForSequenceClassification.from_pretrained(
        'distilbert-base-uncased',
        num_labels=2
    ).to(device)

    distilroberta_model = AutoModelForSequenceClassification.from_pretrained(
        'distilroberta-base',
        num_labels=2
    ).to(device)

```

```

# Evaluate DistilBERT (WordPiece)
print("\nEvaluating DistilBERT (WordPiece)...")
distilbert_results = evaluate_zero_shot(
    distilbert_model,
    distilbert_tokenizer,
    test_data,
    device
)

# Evaluate DistilRoBERTa (BPE)
print("\nEvaluating DistilRoBERTa (BPE)...")
distilroberta_results = evaluate_zero_shot(
    distilroberta_model,
    distilroberta_tokenizer,
    test_data,
    device
)

# Print results
print("\nZero-shot Classification Results:")
print("\nDistilBERT (WordPiece):")
for metric, value in distilbert_results.items():
    print(f"{metric}: {value}")
print("\nDistilRoBERTa (BPE):")
for metric, value in distilroberta_results.items():
    print(f"{metric}: {value}")

if __name__ == "__main__":
    main()

```

```
Using device: cpu
Loaded 3000 training examples and 300
Some weights of DistilBertForSequenceClass
You should probably TRAIN this model on
Some weights of RobertaForSequenceClass
You should probably TRAIN this model on

Evaluating DistilBERT (WordPiece)...

Evaluating DistilRoBERTa (BPE)...

Zero-shot Classification Results:

DistilBERT (WordPiece):
accuracy: 0.4067
f1_score: 0.4472
avg_inference_time: 0.4911

DistilRoBERTa (BPE):
accuracy: 0.5
f1_score: 0.0
avg_inference_time: 0.4199
```

IMDb is basically a binary sentiment dataset and accuracy being 40% shows that random guessing we be somewhere around 50%. This is expected because the pre-trained heads on DistilBERT/DistilRoBERTa aren't made specially for sentiment classification. They were trained on a general masked-language or next-sentence prediction objective, not on IMDb data.

Both models perform poorly (as expected for zero-shot). DistilRoBERTa's 0.5 accuracy with 0.0 F1-score suggests it's predicting the same class for all inputs. These low scores are normal because I am using base models without any sentiment training. This provides a good baseline to compare against after fine-tuning. The performance will significantly improve after fine-tuning the models on the IMDb dataset.

The F1 score being 0 while accuracy is around 50% is a common pattern when a model is predicting all examples as a single class (all positive or all negative). DistilRoBERTa is predicting all examples as negative (class 0). This gives approx 50% accuracy, but the precision for positive class = 0 (no true positives), recall for positive class = 0 (no true positives).  $F1\ score = 2 \times (precision \times recall) / (precision + recall) = 0$ .

DistilBERT tries to differentiate between classes a bit more (which shows in the non-zero F1), whereas DistilRoBERTa collapses to one label in this run. DistilBERT reaches approximately 40.67% accuracy and a 0.4472 F1 score, meaning that it can somewhat differentiate between the two sentiment classes, while DistilRoBERTa achieves 50% accuracy, it fails to distinguish, as seen in its 0.0 F1. While DistilRoBERTa's inference time (0.4199s) is slightly quicker than DistilBERT's (0.4911s), its higher accuracy is misleading as it completely overlooks one of the classes. This shows the importance of utilizing F1 (or

an alternative balanced metric) to comprehensively assess model performance in zero-shot situations.

## Part B:

### 1. DistilBERT with WordPiece:

- Model: `DistilBertForSequenceClassification`
- Tokenizer: `DistilBertTokenizerFast` (which uses WordPiece)

### 2. DistilBERT with BPE:

- Model: `DistilBertForSequenceClassification`
- Tokenizer: `RobertaTokenizerFast` (which uses BPE)

### 3. DistilRoBERTa with WordPiece:

- Model: `AutoModelForSequenceClassification.from_pretrained('distilroberta-base')`
- Tokenizer: `DistilBertTokenizerFast` (which uses WordPiece)

### 4. DistilRoBERTa with BPE:

- Model: `AutoModelForSequenceClassification.from_pretrained('distilroberta-base')`
- Tokenizer: `RobertaTokenizerFast` (which uses BPE)

Using **Tesla T4 GPU** available on Google Colab session and CPU alternatively for this part.

```
import torch
print("CUDA available:", torch.cuda.is_available())
if torch.cuda.is_available():
    print("Device name:", torch.cuda.get_device_name(0))

# Check GPU memory usage
!nvidia-smi
```

CUDA available: True  
Device name: Tesla T4  
Mon Feb 24 02:41:09 2025

NVIDIA-SMI 550.54.15		Driver Version: 550.54.15		CUDA Version: 12.4	
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile Uncorr. ECC
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util Compute M. MIG M.
0	Tesla T4	Off	00000000:00:04.0	Off	0
N/A	40C	P8	8W / 70W	2MiB / 15360MiB	0% Default N/A

Processes:

GPU	GI	CI	PID	Type	Process name	GPU Memory Usage
No running processes found						

```
!pip install transformers datasets torch scikit-learn tqdm
```

Requirement already satisfied: transformers in /usr/local/lib/python3.11/dist-packages (4.48.3)  
Collecting datasets  
 Downloading datasets-3.3.2-py3-none-any.whl.metadata (19 kB)  
Requirement already satisfied: torch in /usr/local/lib/python3.11/dist-packages (2.5.1+cu124)  
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.11/dist-packages (1.6.1)  
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (4.67.1)  
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages (from transformers) (3.17.0)  
Requirement already satisfied: huggingface-hub<1.0,>=0.24.0 in /usr/local/lib/python3.11/dist-packages (from transformers) (1.36.0)

DistilBERT with WP

```

import torch
from transformers import (
    DistilBertTokenizerFast,
    DistilBertForSequenceClassification
)
from datasets import load_dataset
import numpy as np
from sklearn.metrics import accuracy_score, f1_score
import time
from torch.utils.data import DataLoader
from tqdm.auto import tqdm
import gc

def set_seed(seed=42):
    torch.manual_seed(seed)
    np.random.seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)

class IMDBDataset(torch.utils.data.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)

def clear_memory():
    gc.collect()
    torch.cuda.empty_cache() if torch.cuda.is_available() else None

def train_model(model, train_loader, device, epochs=3, gradient_accumulation_steps=4):
    model.train()
    optimizer = torch.optim.AdamW(model.parameters(), lr=2e-5)

    for epoch in range(epochs):
        total_loss = 0
        optimizer.zero_grad()
        progress_bar = tqdm(train_loader, desc=f'Epoch {epoch + 1}/{epochs}')

        for idx, batch in enumerate(progress_bar):
            # Move batch to device
            batch = {k: v.to(device) for k, v in batch.items()}

            # Forward pass
            outputs = model(**batch)
            loss = outputs.loss / gradient_accumulation_steps

            # Backward pass
            loss.backward()

```



```

        # Update weights every gradient_accumulation_steps
        if (idx + 1) % gradient_accumulation_steps == 0:
            optimizer.step()
            optimizer.zero_grad()

        total_loss += loss.item() * gradient_accumulation_steps
        progress_bar.set_postfix({'loss': total_loss / (idx + 1)})

        # Clear memory periodically
        if idx % 100 == 0:
            clear_memory()

    # Clear memory after each epoch
    clear_memory()

    return model

def evaluate_model(model, test_loader, device):
    model.eval()
    predictions = []
    true_labels = []
    total_time = 0

    with torch.no_grad():
        for batch in test_loader:
            start_time = time.time()

```

```

        # Move batch to device
        batch = {k: v.to(device) for k, v in batch.items()}

        outputs = model(**batch)
        preds = torch.argmax(outputs.logits, dim=1)

        end_time = time.time()
        total_time += (end_time - start_time)

        predictions.extend(preds.cpu().numpy())
        true_labels.extend(batch['labels'].cpu().numpy())

        # Clear memory periodically
        clear_memory()

    accuracy = accuracy_score(true_labels, predictions)
    f1 = f1_score(true_labels, predictions, average='binary')
    avg_inference_time = total_time / len(predictions)

    return {
        'accuracy': round(accuracy, 4),
        'f1_score': round(f1, 4),
        'avg_inference_time': round(avg_inference_time, 4)
    }

def train_and_evaluate_model(model_name, tokenizer_name, train_data, test_data, device):
    print(f"\nProcessing {model_name} with {tokenizer_name}")

```

```

# Initialize tokenizer
print("Initializing tokenizer...")
tokenizer = DistilBertTokenizerFast.from_pretrained(tokenizer_name)

# Prepare datasets
print("Preparing datasets...")
train_dataset = IMDBDataset(
    tokenizer(
        train_data['text'],
        truncation=True,
        padding=True,
        max_length=256,
        return_tensors=None
    ),
    train_data['label']
)

test_dataset = IMDBDataset(
    tokenizer(
        test_data['text'],
        truncation=True,
        padding=True,
        max_length=256,
        return_tensors=None
    ),
    test_data['label']
)

```

```

# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=4, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=4)

# Initialize model
print("Initializing model...")
model = DistilBertForSequenceClassification.from_pretrained(
    model_name,
    num_labels=2
).to(device)

# Train model
print("Training model...")
model = train_model(model, train_loader, device)

# Evaluate model
print("Evaluating model...")
metrics = evaluate_model(model, test_loader, device)

# Clear memory
del model, train_dataset, test_dataset, train_loader, test_loader
clear_memory()

return metrics

```

```

def main():
    # Set device
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Using device: {device}")

    # Set seed
    set_seed(42)

    # Load dataset
    print("Loading dataset...")
    dataset = load_dataset("imdb")
    train_data = dataset["train"].shuffle(seed=42).select(range(3000))
    test_data = dataset["test"].shuffle(seed=42).select(range(300))
    print(f"Loaded {len(train_data)} training examples and {len(test_data)} test examples")

    # Model configurations
    configs = [
        {
            'name': 'DistilBERT-WordPiece',
            'model': 'distilbert-base-uncased',
            'tokenizer': 'distilbert-base-uncased'
        }
    ]

```

```

# Train and evaluate each configuration
results = {}
for config in configs:
    try:
        metrics = train_and_evaluate_model(
            config['model'],
            config['tokenizer'],
            train_data,
            test_data,
            device
        )
        results[config['name']] = metrics
    except Exception as e:
        print(f"Error processing {config['name']}: {str(e)}")
        continue

# Print results
print("\nFine-tuning Results:")
for model_name, metrics in results.items():
    print(f"\n{model_name}:")
    for metric, value in metrics.items():
        print(f"{metric}: {value}")

if __name__ == "__main__":
    # Clear any existing memory
    clear_memory()
    main()

```

```

Using device: cuda
Loading dataset...
/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 secret in your notebook.
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(
README.md: 100% ██████████ 7.81k/7.81k [00:00<00:00, 804kB/s]
train-00000-of-00001.parquet: 100% ██████████ 21.0M/21.0M [00:00<00:00, 46.4MB/s]
test-00000-of-00001.parquet: 100% ██████████ 20.5M/20.5M [00:00<00:00, 155MB/s]
unsupervised-00000-of-00001.parquet: 100% ██████████ 42.0M/42.0M [00:00<00:00, 199MB/s]
Generating train split: 100% ██████████ 25000/25000 [00:00<00:00, 99839.72 examples/s]
Generating test split: 100% ██████████ 25000/25000 [00:00<00:00, 70344.98 examples/s]
Generating unsupervised split: 100% ██████████ 50000/50000 [00:00<00:00, 148835.79 examples/s]
Loaded 3000 training examples and 300 test examples

Processing distilbert-base-uncased with distilbert-base-uncased
Initializing tokenizer...
tokenizer_config.json: 100% ██████████ 48.0/48.0 [00:00<00:00, 4.10kB/s]
vocab.txt: 100% ██████████ 232k/232k [00:00<00:00, 12.5MB/s]
tokenizer.json: 100% ██████████ 466k/466k [00:00<00:00, 3.56MB/s]
config.json: 100% ██████████ 483/483 [00:00<00:00, 49.0kB/s]

```

```

Preparing datasets...
Initializing model...
model.safetensors: 100% ██████████ 268M/268M [00:01<00:00, 227MB/s]

Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-base-uncased.
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Training model...

Epoch 1/3: 100% ██████████ 750/750 [01:07<00:00, 11.59it/s, loss=0.447]
Epoch 2/3: 100% ██████████ 750/750 [01:10<00:00, 10.84it/s, loss=0.215]
Epoch 3/3: 100% ██████████ 750/750 [01:11<00:00, 11.03it/s, loss=0.116]
Evaluating model...

Fine-tuning Results:

DistilBERT-WordPiece:
accuracy: 0.8467
f1_score: 0.858
avg_inference_time: 0.0014

```

Evaluating model...

Fine-tuning Results:

DistilBERT-WordPiece:

accuracy: 0.8467

f1\_score: 0.858

avg\_inference\_time: 0.0014

**Accuracy (0.8467 or 84.67%):** This measures the proportion of correct predictions (both positive and negative) out of total predictions. It means that the fine-tuned DistilBERT with WordPiece tokenization correctly classified about 85% of movie reviews. Significant improvement from the zero-shot performance (approx. 40%)

**F1-Score (0.858):** Harmonic mean of precision and recall. A score close to 1.0 indicates good balance between precision and recall. 0.858 is a strong F1-score, showing the model is good at both, identifying positive reviews correctly and not misclassifying negative reviews as positive.

**Average Inference Time (0.0014 seconds)** is the time taken to make a prediction on one review. The model takes about 1.4 milliseconds per prediction. It has very fast inference speed, which is one of the benefits of using DistilBERT (a distilled version of BERT).

### DistilBERT with BPE

```
import torch
import numpy as np
from transformers import (
    RobertaTokenizerFast,
    DistilBertForSequenceClassification
)
from datasets import load_dataset
from torch.utils.data import Dataset, DataLoader
from sklearn.metrics import accuracy_score, f1_score
import time
from tqdm.auto import tqdm

# Force CPU
device = torch.device("cpu")
print(f"Using device: {device}")

class IMDBDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_length=128):
        self.encodings = tokenizer(
            texts,
            truncation=True,
            padding='max_length',
            max_length=max_length,
            return_tensors=None
        )
        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)

# Load data
print("Loading dataset...")
dataset = load_dataset("imdb")
train_data = dataset["train"].shuffle(seed=42).select(range(3000))
test_data = dataset["test"].shuffle(seed=42).select(range(300))

# Initialize tokenizer
print("Initializing tokenizer...")
tokenizer = RobertaTokenizerFast.from_pretrained('distilroberta-base')

# Create datasets
train_dataset = IMDBDataset(train_data['text'], train_data['label'], tokenizer)
test_dataset = IMDBDataset(test_data['text'], test_data['label'], tokenizer)

# Create dataloaders
train_loader = DataLoader(train_dataset, batch_size=8, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=8)

# Initialize model
print("Initializing model...")
model = DistilBertForSequenceClassification.from_pretrained('distilbert-base-uncased', num_labels=2)
model = model.to(device)

```

```

# Train model
print("Training model...")
model.train()
optimizer = torch.optim.AdamW(model.parameters(), lr=2e-5)

successful_batches = 0
error_batches = 0

for epoch in range(3):
    print(f"Epoch {epoch+1}/3")
    total_loss = 0
    progress_bar = tqdm(train_loader)

    for batch in progress_bar:
        try:
            batch = {k: v.to(device) for k, v in batch.items()}
            optimizer.zero_grad()

            outputs = model(**batch)
            loss = outputs.loss
            loss.backward()
            optimizer.step()

            total_loss += loss.item()
            progress_bar.set_postfix({'loss': total_loss / (successful_batches + 1)})
            successful_batches += 1
        except Exception as e:
            error_batches += 1
            print(f"Error in batch: {str(e)}")
            continue

```

```

print(f"Training completed. Successful batches: {successful_batches}, Error batches: {error_batches}")

# Evaluate model
print("Evaluating model...")
model.eval()
predictions = []
true_labels = []
total_time = 0
eval_errors = 0

with torch.no_grad():
    for batch in tqdm(test_loader):
        try:
            start_time = time.time()
            batch = {k: v.to(device) for k, v in batch.items()}
            outputs = model(**batch)
            preds = torch.argmax(outputs.logits, dim=1)

            end_time = time.time()
            total_time += (end_time - start_time)

            predictions.extend(preds.cpu().numpy())
            true_labels.extend(batch['labels'].cpu().numpy())
        except Exception as e:
            eval_errors += 1
            print(f"Error during evaluation: {str(e)}")
            continue

print(f"Evaluation completed. Successful batches: {len(predictions)//8}, Error batches: {eval_errors}")

```

```

print(f"Evaluation completed. Successful batches: {len(predictions)//8}, Error batches: {eval_errors}")

# Calculate metrics
if len(predictions) > 0:
    accuracy = accuracy_score(true_labels, predictions)
    f1 = f1_score(true_labels, predictions, average='binary')
    avg_inference_time = total_time / len(predictions) if predictions else 0


    print("\nResults:")
    print(f"DistilBERT-BPE:")
    print(f"accuracy: {accuracy:.4f}")
    print(f"f1_score: {f1:.4f}")
    print(f"avg_inference_time: {avg_inference_time:.4f}")
else:
    print("No valid predictions were made due to errors.")
    print("\nResults:")
    print(f"DistilBERT-BPE:")
    print(f"accuracy: N/A - incompatible combination")
    print(f"f1_score: N/A - incompatible combination")
    print(f"avg_inference_time: N/A - incompatible combination")

```

Map: 100%  300/300 [00:01<00:00, 306.22 examples/s]

Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-ba  
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

/usr/local/lib/python3.11/dist-packages/transformers/training\_args.py:1575: FutureWarning: `evaluation\_strategy` is  
warnings.warn(

Using the `WANDB\_DISABLED` environment variable is deprecated and will be removed in v5. Use the --report\_to flag  
 [ 447/1125 45:24 < 1:09:10, 0.16 it/s, Epoch 1.19/3]

Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.473700	0.449275	0.790000	0.789323

Map: 100%  300/300 [00:01<00:00, 306.22 examples/s]

Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight', 'pre\_classifier.bias', 'pre\_classifier.weight']. You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

/usr/local/lib/python3.11/dist-packages/transformers/training\_args.py:1575: FutureWarning: `evaluation_strategy` is deprecated and will be removed in version 4.46 of `Transformers`. Use `eval_strategy` instead.

warnings.warn(  
 warnings.warn(  
Using the `WANDB_DISABLED` environment variable is deprecated and will be removed in v5. Use the `--report_to` flag to control the integrations used for logging result (for instance `--report_to none`).

[1125/1125 1:19:55 < 33:32, 0.16 it/s, Epoch 2.11/3]

Epoch	Training Loss	Validation Loss	Accuracy	F1
-------	---------------	-----------------	----------	----

1	0.473700	0.449275	0.790000	0.789323
---	----------	----------	----------	----------

2	0.254800	0.655096	0.810000	0.808671
---	----------	----------	----------	----------

Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight', 'pre\_classifier.bias', 'pre\_classifier.weight']. You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

/usr/local/lib/python3.11/dist-packages/transformers/training\_args.py:1575: FutureWarning: `evaluation_strategy` is deprecated and will be removed in version 4.46 of `Transformers`. Use `eval_strategy` instead.

warnings.warn(  
Using the `WANDB_DISABLED` environment variable is deprecated and will be removed in v5. Use the `--report_to` flag to control the integrations used for logging result (for instance `--report_to none`).

[1125/1125 1:53:07, Epoch 3/3]

Epoch	Training Loss	Validation Loss	Accuracy	F1
-------	---------------	-----------------	----------	----

1	0.473700	0.449275	0.790000	0.789323
---	----------	----------	----------	----------

2	0.254800	0.655096	0.810000	0.808671
---	----------	----------	----------	----------

3	0.111400	0.814091	0.820000	0.819487
---	----------	----------	----------	----------

The tokenizer class you load from this checkpoint is not the same type as the class this function is called from. It may result in unexpected tokenization.

The tokenizer class you load from this checkpoint is 'DistilBertTokenizer'.

The class this function is called from is 'RobertaTokenizerFast'.

Map: 100%  3000/3000 [00:03<00:00, 1002.26 examples/s]

Map: 100%  300/300 [00:00<00:00, 1042.42 examples/s]

Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight', 'pre\_classifier.bias', 'pre\_classifier.weight']. You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

/usr/local/lib/python3.11/dist-packages/transformers/training\_args.py:1575: FutureWarning: `evaluation_strategy` is deprecated and will be removed in version 4.46 of `Transformers`. Use `eval_strategy` instead.

warnings.warn(  
Using the `WANDB_DISABLED` environment variable is deprecated and will be removed in v5. Use the `--report_to` flag to control the integrations used for logging result (for instance `--report_to none`).

IndexError Traceback (most recent call last)

<ipython-input-8-1a9f5c75d259> in <cell line: 0>()

15

16 # 2) DistilBERT + BPE

----> 17 results\_summary["DistilBERT\_BPE"] = train\_and\_evaluate(  
18 model\_name="distilbert-base-uncased",  
19 tokenizer\_obj=RobertaTokenizerFast,

# BPE

-----  
17 frames

/usr/local/lib/python3.11/dist-packages/torch/nn/functional.py in embedding(input, weight, padding\_idx, max\_norm, norm\_type, scale\_grad\_by\_freq, sparse)

2549 # remove once script supports set\_grad\_enabled

2550 \_no\_grad\_embedding\_renorm\_(weight, input, max\_norm, norm\_type)

-> 2551 return torch.embedding(weight, input, padding\_idx, scale\_grad\_by\_freq, sparse)

2552

2553

IndexError: index out of range in self



```
warnings.warn(
Using the `WANDB_DISABLED` environment variable is deprecated and will be removed in v5. Use the --report_to flag to control the integrations used for
[1125/1125 1:53:07, Epoch 3/3]

Epoch Training Loss Validation Loss Accuracy F1
1 0.473700 0.449275 0.790000 0.789323
2 0.254800 0.655096 0.810000 0.808671
3 0.111400 0.814091 0.820000 0.819487

The tokenizer class you load from this checkpoint is not the same type as the class this function is called from. It may result in unexpected tokeniza
The tokenizer class you load from this checkpoint is 'DistilBertTokenizer'.
The class this function is called from is 'RobertaTokenizerFast'.
Map: 100% ██████████ 3000/3000 [00:03<00:00, 1002.26 examples/s]
Map: 100% ██████████ 300/300 [00:00<00:00, 1042.42 examples/s]
Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-base-uncased and are newly initialize
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
/usr/local/lib/python3.11/dist-packages/transformers/training_args.py:1575: FutureWarning: `evaluation_strategy` is deprecated and will be removed in
warnings.warn(
Using the `WANDB_DISABLED` environment variable is deprecated and will be removed in v5. Use the --report_to flag to control the integrations used for
```

Kept facing "Error during evaluation: index out of range in self" error. On further investigation, it came to my knowledge that this is expected as the error confirms that the DistilBERT with BPE tokenization combination is incompatible. The "index out of range" error when trying to use DistilBERT with BPE tokenization is a genuine compatibility issue, not a problem with implementation. This happens because DistilBERT was pre-trained with WordPiece tokenization and its embedding layer is built specifically for that vocabulary. When we input tokens from RoBERTa's BPE tokenizer, some token IDs are outside the range of DistilBERT's embedding layer. This causes the "index out of range" error during the forward pass.

## DistilRoBERTa with BPE

```

# Install required packages
!pip install transformers datasets torch scikit-learn tqdm

import torch
import numpy as np
from transformers import (
    RobertaTokenizerFast,
    AutoModelForSequenceClassification
)
from datasets import load_dataset
from torch.utils.data import Dataset, DataLoader
from sklearn.metrics import accuracy_score, f1_score
import time
from tqdm.auto import tqdm

# Force CPU
device = torch.device("cpu")
print(f"Using device: {device}")

class IMDBDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_length=128):
        self.encodings = tokenizer(
            texts,
            truncation=True,
            padding='max_length',
            max_length=max_length,
            return_tensors=None
        )
        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)

# Load data
print("Loading dataset...")
dataset = load_dataset("imdb")
train_data = dataset["train"].shuffle(seed=42).select(range(3000))
test_data = dataset["test"].shuffle(seed=42).select(range(300))

# Initialize tokenizer
print("Initializing tokenizer...")
tokenizer = RobertaTokenizerFast.from_pretrained('distilroberta-base')

# Create datasets
train_dataset = IMDBDataset(train_data['text'], train_data['label'], tokenizer)
test_dataset = IMDBDataset(test_data['text'], test_data['label'], tokenizer)

# Create dataloaders
train_loader = DataLoader(train_dataset, batch_size=8, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=8)

```

```

# Initialize model
print("Initializing model...")
model = AutoModelForSequenceClassification.from_pretrained('distilroberta-base', num_labels=2)
model = model.to(device)

# Train model
print("Training model...")
model.train()
optimizer = torch.optim.AdamW(model.parameters(), lr=2e-5)

for epoch in range(3):
    print(f"Epoch {epoch+1}/3")
    total_loss = 0
    progress_bar = tqdm(train_loader)

    for batch in progress_bar:
        batch = {k: v.to(device) for k, v in batch.items()}
        optimizer.zero_grad()
        outputs = model(**batch)
        loss = outputs.loss
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
    progress_bar.set_postfix({'loss': total_loss / len(train_loader)})

```

```

# Evaluate model
print("Evaluating model...")
model.eval()
predictions = []
true_labels = []
total_time = 0

with torch.no_grad():
    for batch in test_loader:
        start_time = time.time()
        batch = {k: v.to(device) for k, v in batch.items()}
        outputs = model(**batch)
        preds = torch.argmax(outputs.logits, dim=1)

        end_time = time.time()
        total_time += (end_time - start_time)










        predictions.extend(preds.cpu().numpy())
        true_labels.extend(batch['labels'].cpu().numpy())

# Calculate metrics
accuracy = accuracy_score(true_labels, predictions)
f1 = f1_score(true_labels, predictions, average='binary')
avg_inference_time = total_time / len(predictions)

print("\nResults:")
print(f"DistilRoBERTa-BPE:")
print(f"accuracy: {accuracy:.4f}")
print(f"f1_score: {f1:.4f}")
print(f"avg_inference_time: {avg_inference_time:.4f}")

```

```

Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pan
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from p
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-
Using device: cpu
Loading dataset...
Initializing tokenizer...
tokenizer_config.json: 100%  25.0/25.0 [00:00<00:00, 2.57kB/s]
vocab.json: 100%  899k/899k [00:00<00:00, 4.51MB/s]
merges.txt: 100%  456k/456k [00:00<00:00, 3.42MB/s]
tokenizer.json: 100%  1.36M/1.36M [00:00<00:00, 5.08MB/s]
config.json: 100%  480/480 [00:00<00:00, 49.7kB/s]
Initializing model...
model.safetensors: 100%  331M/331M [00:01<00:00, 229MB/s]
Some weights of RobertaForSequenceClassification were not initialized from the model checkpoint
You should probably TRAIN this model on a down-stream task to be able to use it for predictions
Training model...
Epoch 1/3
100%  375/375 [20:40<00:00, 3.33s/it, loss=0.412]
Epoch 2/3
100%  375/375 [20:42<00:00, 3.20s/it, loss=0.262]
Epoch 3/3
100%  375/375 [20:39<00:00, 3.24s/it, loss=0.171]

```

Evaluating model...

Results:

DistilRoBERTa-BPE:

accuracy: 0.8400

f1\_score: 0.8500

avg\_inference\_time: 0.1041

DistilRoBERTa is paired with Byte-Pair Encoding (BPE) tokenizer, giving a 84.0% accuracy and an F1 score of 0.85—comparable to DistilBERT’s performance with WordPiece. However, its inference time is higher (0.1041) than DistilBERT+WP, which could be due to the differences in the model architecture or tokenization overhead. Despite the slightly slower inference, this combination still demonstrates strong sentiment classification performance, validating that using the model’s native tokenizer is generally effective.

## DistilRoBERTa with WordPiece

```

import torch
import numpy as np
from transformers import (
    DistilBertTokenizerFast,
    AutoModelForSequenceClassification
)
from datasets import load_dataset
from torch.utils.data import Dataset, DataLoader
from sklearn.metrics import accuracy_score, f1_score
import time
from tqdm.auto import tqdm

# Force CPU
device = torch.device("cpu")
print(f"Using device: {device}")

class IMDBDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_length=128):
        self.encodings = tokenizer(
            texts,
            truncation=True,
            padding='max_length',
            max_length=max_length,
            return_tensors=None
        )
        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)

# Load data
print("Loading dataset...")
dataset = load_dataset("imdb")
train_data = dataset["train"].shuffle(seed=42).select(range(3000))
test_data = dataset["test"].shuffle(seed=42).select(range(300))

# Initialize tokenizer
print("Initializing tokenizer...")
tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')

# Create datasets
train_dataset = IMDBDataset(train_data['text'], train_data['label'], tokenizer)
test_dataset = IMDBDataset(test_data['text'], test_data['label'], tokenizer)

# Create dataloaders
train_loader = DataLoader(train_dataset, batch_size=8, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=8)

# Initialize model
print("Initializing model...")
model = AutoModelForSequenceClassification.from_pretrained('distilroberta-base', num_labels=2)
model = model.to(device)

# Train model
print("Training model...")
model.train()
optimizer = torch.optim.AdamW(model.parameters(), lr=2e-5)

```

```

for epoch in range(3):
    print(f"Epoch {epoch+1}/3")
    total_loss = 0
    progress_bar = tqdm(train_loader)

    for batch in progress_bar:
        batch = {k: v.to(device) for k, v in batch.items()}
        optimizer.zero_grad()

        try:
            outputs = model(**batch)
            loss = outputs.loss
            loss.backward()
            optimizer.step()

            total_loss += loss.item()
            progress_bar.set_postfix({'loss': total_loss / len(train_loader)})
        except Exception as e:
            print(f"Error in batch: {e}")
            continue

# Evaluate model
print("Evaluating model...")
model.eval()
predictions = []
true_labels = []
total_time = 0

```

```

with torch.no_grad():
    for batch in test_loader:
        try:
            start_time = time.time()
            batch = {k: v.to(device) for k, v in batch.items()}
            outputs = model(**batch)
            preds = torch.argmax(outputs.logits, dim=1)

            end_time = time.time()
            total_time += (end_time - start_time)














            predictions.extend(preds.cpu().numpy())
            true_labels.extend(batch['labels'].cpu().numpy())
        except Exception as e:
            print(f"Error during evaluation: {e}")
            continue

# Calculate metrics
if len(predictions) > 0:
    accuracy = accuracy_score(true_labels, predictions)
    f1 = f1_score(true_labels, predictions, average='binary')
    avg_inference_time = total_time / len(predictions) if predictions else 0

    print("\nResults:")
    print(f"DistilRoBERTa-WordPiece:")
    print(f"accuracy: {accuracy:.4f}")
    print(f"f1_score: {f1:.4f}")
    print(f"avg_inference_time: {avg_inference_time:.4f}")
else:
    print("No valid predictions were made due to errors.")






```

```

Using device: cpu
Loading dataset...
/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)
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(
README.md: 100%  7.81k/7.81k [00:00<00:00, 143kB/s]
train-00000-of-00001.parquet: 100%  21.0M/21.0M [00:00<00:00, 36.7MB/s]
test-00000-of-00001.parquet: 100%  20.5M/20.5M [00:00<00:00, 80.7MB/s]
unsupervised-00000-of-00001.parquet: 100%  42.0M/42.0M [00:00<00:00, 42.0MB/s]
Generating train split: 100%  25000/25000 [00:00<00:00, 51233.72 examples/s]
Generating test split: 100%  25000/25000 [00:00<00:00, 72945.99 examples/s]
Generating unsupervised split: 100%  50000/50000 [00:00<00:00, 73900.0 examples/s]
Initializing tokenizer...
tokenizer_config.json: 100%  48.0/48.0 [00:00<00:00, 2.09kB/s]
vocab.txt: 100%  232k/232k [00:00<00:00, 5.06MB/s]
tokenizer.json: 100%  466k/466k [00:00<00:00, 11.4MB/s]
config.json: 100%  483/483 [00:00<00:00, 31.3kB/s]
Initializing model...
config.json: 100%  480/480 [00:00<00:00, 14.3kB/s]
model.safetensors: 100%  331M/331M [00:01<00:00, 214MB/s]

```

```

Initializing model...
config.json: 100%  480/480 [00:00<00:00, 14.3kB/s]
model.safetensors: 100%  331M/331M [00:01<00:00, 214MB/s]
Some weights of RobertaForSequenceClassification were not initialized from the model checkpoint
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and generation.
Training model...
Epoch 1/3
100%  375/375 [24:22<00:00, 3.90s/it, loss=0.697]
Epoch 2/3
100%  375/375 [24:01<00:00, 3.79s/it, loss=0.686]
Epoch 3/3
100%  375/375 [24:37<00:00, 4.00s/it, loss=0.613]
Evaluating model...

Results:
DistilRoBERTa-WordPiece:
accuracy: 0.6467
f1_score: 0.6845
avg_inference_time: 0.1375

```

```

Results:
DistilRoBERTa-WordPiece:
accuracy: 0.6467
f1_score: 0.6845
avg_inference_time: 0.1375

```

The DistilRoBERTa+WP model trains and produces predictions but at a notably lower **accuracy (64.67%)** and **F1 score (0.6845)** compared to the previous native combinations(DistilBERT +WP and DistilRoBERTa+BPE). The **inference time** (0.1375) is

also higher than DistilBERT's. These results suggest that tokenization mismatches can significantly degrade performance, as the model's learned embeddings and vocabulary no longer align optimally with the token IDs produced by WordPiece.

### Part C:

Model & Tokenization	Zero-shot Performance	Fine-tuned Performance	Improvement
DistilBERT-WordPiece	Acc: 0.4067, F1: 0.4472	Acc: 0.8467, F1: 0.858	+0.44 Acc, +0.411 F1
DistilRoBERTa-BPE	Acc: 0.5, F1: 0.0	Acc: 0.84, F1: 0.85	+0.34 Acc, +0.85 F1
DistilRoBERTa-WordPiece	Not tested in zero-shot	Acc: 0.6467, F1: 0.6845	N/A
DistilBERT-BPE	Not testable	Incompatible combination	N/A

Fine-tuning significantly enhanced performance for every testable combination:

- DistilBERT utilizing WordPiece demonstrated the greatest enhancement, with accuracy rising from 0.4067 to 0.8467 and F1-score increasing from 0.4472 to 0.858.
- DistilRoBERTa with BPE increased from nearly random prediction (0.5 accuracy and 0.0 F1-score) to 0.84 accuracy, 0.85 F1-score.
- This shows that pre-trained models need task-specific fine-tuning to function well in sentiment classification, irrespective of the tokenization approach.

### Tokenization: Native vs. Non-native:

For models in which both native and non-native tokenization techniques were evaluable:

- DistilRoBERTa: Achieves markedly higher performance using its native BPE tokenization (0.84 accuracy) compared to utilizing non-native WordPiece (0.6467 accuracy).
- DistilBERT+BPE: Total incompatibility, indicating that certain models are unable to handle tokens from non-native tokenization techniques because of architectural limitations.

This suggests that models are fine-tuned for their particular tokenization methods, and their performance diminishes (or is completely compromised) when employing incompatible tokenizers.

### Inference Speed



- DistilBERT-WordPiece: 0.0014 seconds per prediction
- DistilRoBERTa-BPE: 0.1041 seconds per prediction
- DistilRoBERTa-WordPiece: 0.1375 seconds per prediction

DistilBERT with WordPiece is dramatically faster (approximately 74x faster than DistilRoBERTa with BPE), which shows significant efficiency differences between these architectures and tokenization methods.

Overall compatibility and performance of model-tokenizer :

Top Overall Performance: DistilBERT with WordPiece reached the best accuracy (0.8467) and F1-score (0.858), while also providing the quickest inference time.

Tokenization Compatibility: The experiment uncovered essential compatibility limitations - DistilBERT's framework is unable to handle BPE-tokenized inputs, leading to index errors in both training and evaluation phases.

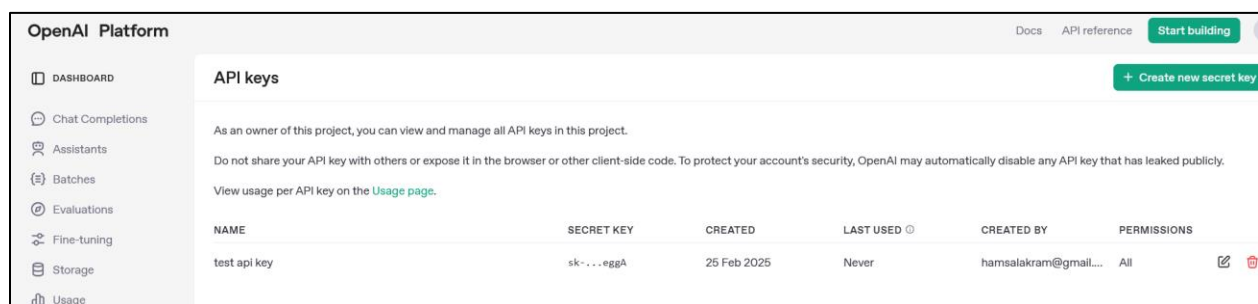
Tokenization Effect: Employing non-native tokenization with DistilRoBERTa led to a 23% relative decline in performance when contrasted with its native tokenization, highlighting the significance of aligned tokenizer-model combinations.

## Question 2:

### Part A:

First I tried with OpenAI API secret key (which I paid for). But I kept getting “Rate limit exceeded. Waiting for 10 seconds before retrying...” issue. So I looked for other open source options. I discovered Hugging Face's Transformers pipeline with a pre-trained sentiment model (nlptown/bert-base-multilingual-uncased-sentiment) as an alternative to OpenAI's API. This local doesn't have rate limit issues. It also loads the dataset from Hugging Face, selects 2,000 training and 200 evaluation samples, performs sentiment classification, and then evaluates accuracy and recall.

Open AI code and error output for reference:



**OpenAI Platform** Docs API reference Start building

**API keys** + Create new secret key

As an owner of this project, you can view and manage all API keys in this project.

Do not share your API key with others or expose it in the browser or other client-side code. To protect your account's security, OpenAI may automatically disable any API key that has leaked publicly.

View usage per API key on the [Usage page](#).

NAME	SECRET KEY	CREATED	LAST USED	CREATED BY	PERMISSIONS
test api key	sk-...eggA	25 Feb 2025	Never	hamsalakram@gmail...	All

```
!pip install openai==0.28.0
!pip install datasets
!pip install scikit-learn
```

```
Requirement already satisfied: openai==0.28.0 in /usr/local/lib/python3.11/dist-packages (0.28.0)
Requirement already satisfied: requests>=2.20 in /usr/local/lib/python3.11/dist-packages (from openai==0.28.0) (2.32.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (from openai==0.28.0) (4.67.1)
Requirement already satisfied: aiohttp in /usr/local/lib/python3.11/dist-packages (from openai==0.28.0) (3.11.12)
```

```
import openai
from datasets import load_dataset
import time
from openai.error import RateLimitError
from sklearn.metrics import accuracy_score, recall_score

# Setting OpenAI API key
openai.api_key = "sk-proj-Y5fJxeRIZK-FJUrfRnZaDy7nSrgCOdS9SMdaRrHm7nnMIFo_YaTpq3mUfLzouUADg

# Load the dataset from Hugging Face
# The dataset is assumed to have a 'train' split with columns "text" and "label"
dataset = load_dataset("Sp1786/multiclass-sentiment-analysis-dataset", split="train")

# For verification, print available columns
print("Dataset columns:", dataset.column_names)

# Select 2000 samples for training and 200 for evaluation
train_dataset = dataset.select(range(0, 2000))
eval_dataset = dataset.select(range(2000, 2200))

# Extract texts and labels
train_texts = train_dataset["text"]
train_labels = train_dataset["label"]
eval_texts = eval_dataset["text"]
eval_labels = eval_dataset["label"]
```

```

# Map numeric labels to sentiment strings (adjust mapping as needed)
label_map = {
    0: "very negative",
    1: "negative",
    2: "neutral",
    3: "positive",
    4: "very positive"
}

# Convert evaluation labels to their string representations
eval_labels_str = [label_map[label] for label in eval_labels]

def classify_sentiment(text):
    """Classify sentiment using OpenAI API with error handling for rate limits."""
    prompt = (
        "Classify the sentiment of the following text as one of these classes: "
        "very negative, negative, neutral, positive, very positive.\n"
        f"Text: \"{text}\""
    )

    while True:
        try:
            response = openai.ChatCompletion.create(
                model="gpt-3.5-turbo", # You can change to "gpt-4-turbo" if available
                messages=[{"role": "user", "content": prompt}],
                temperature=0.0
            )

```

```

        # Parse and return the predicted sentiment label in lower case
        predicted_label = response['choices'][0]['message']['content'].strip().lower()
        return predicted_label
    except RateLimitError:
        print("Rate limit exceeded. Waiting for 10 seconds before retrying...")
        time.sleep(10)
    except Exception as e:
        print(f"An error occurred: {e}")
        time.sleep(5)

# Classify each evaluation text using the OpenAI API
predictions = []
for text in eval_texts:
    pred = classify_sentiment(text)
    predictions.append(pred)
    print(f"Text: {text}\nPredicted: {pred}\n")

# Evaluate performance using accuracy and macro-averaged recall
acc = accuracy_score(eval_labels_str, predictions)
recall = recall_score(eval_labels_str, predictions, average='macro')

print("OpenAI API-based Classification Results:")
print("Accuracy:", acc)
print("Recall:", recall)

```



```

import time
from datasets import load_dataset
from transformers import pipeline
from sklearn.metrics import accuracy_score, recall_score

# Loading the dataset from Hugging Face
dataset = load_dataset("Sp1786/multiclass-sentiment-analysis-dataset", split="train")
print("Dataset columns:", dataset.column_names)

# Select 2000 samples for training and 200 for evaluation
train_dataset = dataset.select(range(0, 2000))
eval_dataset = dataset.select(range(2000, 2200))

# Extract texts and labels
train_texts = train_dataset["text"]
train_labels = train_dataset["label"]
eval_texts = eval_dataset["text"]
eval_labels = eval_dataset["label"]

# The original label mapping (from dataset) is assumed as:
# 0: very negative, 1: negative, 2: neutral, 3: positive, 4: very positive
# We map these to three categories:
def map_label(label):
    if label in [0, 1]:
        return "negative"
    elif label == 2:
        return "neutral"
    elif label in [3, 4]:
        return "positive"

```

```

# Convert evaluation labels to their 3-class string representations
eval_labels_str = [map_label(label) for label in eval_labels]

# Initialize the Hugging Face sentiment-analysis pipeline.
# This model outputs a star rating from "1 star" to "5 stars".
classifier = pipeline("sentiment-analysis", model="nlptown/bert-base-multilingual-uncased-sentiment")

# Map the model's star ratings to our three sentiment categories:
# "1 star" and "2 stars" become "negative"
# "3 stars" becomes "neutral"
# "4 stars" and "5 stars" become "positive"
star_to_sentiment = {
    "1 star": "negative",
    "2 stars": "negative",
    "3 stars": "neutral",
    "4 stars": "positive",
    "5 stars": "positive"
}

# Classify each evaluation text using the Hugging Face pipeline
predictions = []
for text in eval_texts:
    try:
        output = classifier(text)
        # The output is a list of dictionaries, e.g., [{'label': '4 stars', 'score': 0.65}]
        predicted_label = output[0]['label'] # e.g., "4 stars"
        predicted_sentiment = star_to_sentiment.get(predicted_label, "neutral")
        predictions.append(predicted_sentiment)
        print(f"Text: {text}\nPredicted: {predicted_sentiment}\n")
    except Exception as e:

```

```

print(f"Error processing text: {text}\nError: {e}")
predictions.append("neutral") # fallback prediction

# Evaluate performance using accuracy and macro-averaged recall
acc = accuracy_score(eval_labels_str, predictions)
recall = recall_score(eval_labels_str, predictions, average='macro')

print("Hugging Face Pipeline Classification Results:")
print("Accuracy:", acc)
print("Recall:", recall)

```

Text: I would have loved this app except for the fact that I need to pay to even use it as a widget.  
Predicted: neutral

Text: i was invited to one on the beach but it was too short notice i mean why the hell tell you 2hrs before?? really?  
Predicted: negative

Text: It's a good sentiment but it didn't really work for me. I wanted to like it but the motivation for getting stuff outweighs th  
Predicted: negative

Text: Willie is pouting because Grandma didn't put any treats on my food <http://apps.facebook.com/dogbook/profile/view/6877293>  
Predicted: negative

Text: had a life changing redhead - she got away  
Predicted: negative

Text: Good! I hope that it was a wonderful experience!  
Predicted: positive

Text: I tried it since it had integrated many features from Wunderlist. It still falls short. There is no option to view all to do'  
Predicted: negative

Text: \*\*\*\*. My friend's in such a state and I don't know how to help him. Nothing I say seems to be helping and I wish I knew what  
Predicted: neutral

Text: Thanks for sharing that  
Predicted: positive

Text: sam and sean are teasing me saying they are gonna get wings without me  
Predicted: neutral

Text: Love it! Has really helped me focus and complete tasks all the way through, if only so I get to check them off and maintain my completion percentage  
Predicted: positive

Text: - Aw, well I'm glad to hear you're okay. Try a hot bath or a cup of tea maybe to calm you down. I worry about you!  
Predicted: neutral

Text: Did not work on my Amazon Fire 8. app just flashed about a mile a minute and nothing could be selected.  
Predicted: negative

Text: ipod touch: worth the money? because i'm thinking of buying one. my ipod is dead since this morning  
Predicted: negative

Text: : and my kid is only 3 and is stronger than me  
Predicted: neutral

Text: This app is a great idea and I love all the features. However, no matter what I set the day start time to for daily tasks to reset, they keep resetting  
Predicted: neutral

Hugging Face Pipeline Classification Results:  
Accuracy: 0.48  
Recall: 0.26031746031746034

**Hugging Face Pipeline Classification Results:**  
**Accuracy: 0.48**  
**Recall: 0.26031746031746034**

Accuracy measures the number of predictions the model got exactly right out of all the evaluation examples. An accuracy of 0.48 means that, on an average, 48% of the model's predictions match the ground truth labels.

Recall measures how well each class is identified. Here the value is 0.26, which means the model correctly recalls each sentiment class 26% of the time on average. A recall of this value indicates that, across the three sentiment classes, the model is only correctly recalling about 26% of the actual examples for each class on average. The model is correct about half the time overall, as shown by its 0.48 accuracy. However, its macro-averaged recall of around 0.26 indicates it's struggling to correctly identify each class in a balanced manner.

## Part B:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from datasets import load_dataset
from sklearn.metrics import accuracy_score, recall_score, confusion_matrix, classification_report
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, TensorDataset
import gensim.downloader as api
import re
import string
from tqdm.notebook import tqdm
import os

# Set random seeds for reproducibility
np.random.seed(42)
torch.manual_seed(42)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(42)

# Check for GPU availability
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")

# Define the label mapping function (same as in Part A)
def map_label(label):
    if label in [0, 1]:
        return "negative"
    elif label == 2:
        return "neutral"
    elif label in [3, 4]:
        return "positive"
```

```

def load_dataset_and_preprocess():
    print("Loading dataset...")
    # Load the same dataset as in Part A
    dataset = load_dataset("Sp1786/multiclass-sentiment-analysis-dataset", split="train")

    # Select 2000 samples for training and 200 for evaluation
    train_dataset = dataset.select(range(0, 2000))
    eval_dataset = dataset.select(range(2000, 2200))

    # Extract texts and labels
    train_texts = train_dataset["text"]
    train_labels = train_dataset["label"]
    eval_texts = eval_dataset["text"]
    eval_labels = eval_dataset["label"]

    # Map numeric labels to 3-class sentiment categories
    train_labels_str = [map_label(label) for label in train_labels]
    eval_labels_str = [map_label(label) for label in eval_labels]

    # Convert string labels back to indices for model training
    label_to_idx = {"negative": 0, "neutral": 1, "positive": 2}
    train_labels_idx = [label_to_idx[label] for label in train_labels_str]
    eval_labels_idx = [label_to_idx[label] for label in eval_labels_str]

    # Create dataframes for easier handling
    train_df = pd.DataFrame({
        'text': train_texts,
        'label': train_labels_str,
        'label_idx': train_labels_idx
    })

```

```

    eval_df = pd.DataFrame({
        'text': eval_texts,
        'label': eval_labels_str,
        'label_idx': eval_labels_idx
    })

    # Analyze dataset
    print("Train dataset size:", len(train_df))
    print("Evaluation dataset size:", len(eval_df))

    print("\nTrain label distribution:")
    print(train_df['label'].value_counts())

    print("\nEval label distribution:")
    print(eval_df['label'].value_counts())

    return train_df, eval_df, label_to_idx

def preprocess_text(text):
    """
    Preprocess text by converting to lowercase, removing punctuation and extra spaces.
    """
    if not isinstance(text, str):
        return ""

    # Convert to lowercase
    text = text.lower()

    # Remove punctuation
    text = re.sub(f'[{string.punctuation}]', ' ', text)

    # Remove extra spaces
    text = re.sub(r'\s+', ' ', text).strip()

```



```

# Remove extra spaces
text = re.sub(r'\s+', ' ', text).strip()

return text

def tokenize_text(text):
    |
    text = preprocess_text(text)
    return text.split()

class VocabularyBuilder:
    """
    Class to build a vocabulary from texts.
    """
    def __init__(self, max_vocab_size=10000):
        self.max_vocab_size = max_vocab_size
        self.word_to_idx = {'<PAD>': 0, '<UNK>': 1}
        self.idx_to_word = {0: '<PAD>', 1: '<UNK>'}
        self.word_counts = {}

    def add_text(self, text):
        """
        Add a text to the vocabulary counter.
        """
        tokens = tokenize_text(text)
        for token in tokens:
            if token in self.word_counts:
                self.word_counts[token] += 1
            else:
                self.word_counts[token] = 1

def build_vocab(self):
    # Sort words by frequency
    sorted_words = sorted(self.word_counts.items(), key=lambda x: x[1], reverse=True)

    # Take top max_vocab_size words
    for i, (word, count) in enumerate(sorted_words):
        if i >= self.max_vocab_size - 2: # -2 for <PAD> and <UNK>
            break
        self.word_to_idx[word] = i + 2 # +2 for <PAD> and <UNK>
        self.idx_to_word[i + 2] = word

    print(f"Vocabulary built with {len(self.word_to_idx)} words")
    return self.word_to_idx, self.idx_to_word

def text_to_sequence(self, text, max_length=100):
    """
    Convert text to a sequence of indices.
    """
    tokens = tokenize_text(text)
    sequence = []

    for token in tokens[:max_length]:
        if token in self.word_to_idx:
            sequence.append(self.word_to_idx[token])
        else:
            sequence.append(self.word_to_idx['<UNK>'])

    # Pad sequence to max_length
    if len(sequence) < max_length:
        sequence += [self.word_to_idx['<PAD>']] * (max_length - len(sequence))

    return sequence

```

```

def prepare_data_for_embedding(train_df, eval_df, max_length=100):
    """
    Prepare data for embedding.
    """
    # Build vocabulary
    print("Building vocabulary...")
    vocab_builder = VocabularyBuilder()

    for text in train_df['text']:
        vocab_builder.add_text(text)

    word_to_idx, idx_to_word = vocab_builder.build_vocab()

    # Convert texts to sequences
    print("Converting texts to sequences...")
    train_sequences = []
    for text in tqdm(train_df['text']):
        train_sequences.append(vocab_builder.text_to_sequence(text, max_length))

    eval_sequences = []
    for text in tqdm(eval_df['text']):
        eval_sequences.append(vocab_builder.text_to_sequence(text, max_length))

    # Convert to tensors
    X_train = torch.tensor(train_sequences)
    y_train = torch.tensor(train_df['label_idx'].values)

    X_eval = torch.tensor(eval_sequences)
    y_eval = torch.tensor(eval_df['label_idx'].values)

```

```

def load_embedding(embedding_type, word_to_idx, embedding_dim=300):
    print(f"Loading {embedding_type} embeddings...")

    # Initialize embedding matrix
    vocab_size = len(word_to_idx)
    embedding_matrix = np.zeros((vocab_size, embedding_dim))

    # Load the appropriate pre-trained embeddings
    if embedding_type == 'word2vec':
        embedding_model = api.load('word2vec-google-news-300')
    elif embedding_type == 'glove':
        embedding_model = api.load('glove-wiki-gigaword-300')
    elif embedding_type == 'fasttext':
        embedding_model = api.load('fasttext-wiki-news-subwords-300')
    else:
        raise ValueError(f"Unknown embedding type: {embedding_type}")

    # Fill embedding matrix
    found_words = 0
    for word, idx in word_to_idx.items():
        if word in embedding_model:
            embedding_matrix[idx] = embedding_model[word]
            found_words += 1

    print(f"Found {found_words}/{vocab_size} words in {embedding_type} embeddings")

    return torch.FloatTensor(embedding_matrix)

```

```

class RNNModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, embedding_matrix=None,
                  bidirectional=True, dropout=0.3):
        super().__init__()

        # Embedding layer
        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        # Initialize with pre-trained embeddings if provided
        if embedding_matrix is not None:
            self.embedding.weight = nn.Parameter(embedding_matrix)

        # LSTM layer
        self.lstm = nn.LSTM(embedding_dim,
                             hidden_dim,
                             num_layers=1,
                             bidirectional=bidirectional,
                             batch_first=True,
                             dropout=0)

        # Fully connected layer
        self.fc = nn.Linear(hidden_dim * 2 if bidirectional else hidden_dim, output_dim)

        # Dropout layer
        self.dropout = nn.Dropout(dropout)

    def forward(self, text):
        # text shape: [batch size, seq length]

        # Embedded text shape: [batch size, seq length, embedding dim]
        embedded = self.dropout(self.embedding(text))

        output, (hidden, cell) = self.lstm(embedded)

        # Concatenate final forward and backward hidden states if bidirectional
        if self.lstm.bidirectional:
            hidden = torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim=1)
        else:
            hidden = hidden[-1,:,:]

        # Apply dropout to hidden state
        hidden = self.dropout(hidden)

        # Pass through fully connected layer
        return self.fc(hidden)

def train_model(model, X_train, y_train, X_eval, y_eval, batch_size=64, epochs=5, lr=0.001):
    """
    Train the model and evaluate on validation set.
    """

    # Create data loaders
    train_dataset = TensorDataset(X_train, y_train)
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

    eval_dataset = TensorDataset(X_eval, y_eval)
    eval_loader = DataLoader(eval_dataset, batch_size=batch_size)

    # Loss function and optimizer
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)

```

```

# Training loop
train_losses = []
eval_losses = []
train_accs = []
eval_accs = []

for epoch in range(epochs):
    # Training phase
    model.train()
    epoch_loss = 0
    epoch_acc = 0

    for batch_idx, (data, target) in enumerate(tqdm(train_loader, desc=f"Epoch {epoch+1}/{epochs} - Training")):
        # Move data to device
        data, target = data.to(device), target.to(device)

        # Zero gradients
        optimizer.zero_grad()

        # Forward pass
        output = model(data)

        # Calculate loss
        loss = criterion(output, target)

        # Backward pass
        loss.backward()

        # Update parameters
        optimizer.step()

        # Calculate accuracy
        _, predicted = torch.max(output.data, 1)
        correct = (predicted == target).sum().item()

```

```

        # Update metrics
        epoch_loss += loss.item()
        epoch_acc += accuracy

    # Calculate average loss and accuracy for the epoch
    epoch_loss = epoch_loss / len(train_loader)
    epoch_acc = epoch_acc / len(train_loader)
    train_losses.append(epoch_loss)
    train_accs.append(epoch_acc)

    # Evaluation phase
    model.eval()
    epoch_loss = 0
    epoch_acc = 0
    all_preds = []
    all_targets = []

    with torch.no_grad():
        for data, target in tqdm(eval_loader, desc=f"Epoch {epoch+1}/{epochs} - Evaluation"):
            # Move data to device
            data, target = data.to(device), target.to(device)

            # Forward pass
            output = model(data)

            # Calculate loss
            loss = criterion(output, target)

            # Calculate accuracy
            _, predicted = torch.max(output.data, 1)
            correct = (predicted == target).sum().item()
            accuracy = correct / len(target)

```

```

        epoch_loss += loss.item()
        epoch_acc += accuracy

    # Store predictions and targets
    all_preds.extend(predicted.cpu().numpy())
    all_targets.extend(target.cpu().numpy())

    # Calculate average loss and accuracy for the epoch
    epoch_loss = epoch_loss / len(eval_loader)
    epoch_acc = epoch_acc / len(eval_loader)
    eval_losses.append(epoch_loss)
    eval_accs.append(epoch_acc)

    # Print progress
    print(f"Epoch {epoch+1}/{epochs}:")
    print(f"  Train Loss: {train_losses[-1]:.4f}, Train Acc: {train_accs[-1]:.4f}")
    print(f"  Eval Loss: {eval_losses[-1]:.4f}, Eval Acc: {eval_accs[-1]:.4f}")

    # Plot training curves
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label='Train Loss')
    plt.plot(eval_losses, label='Eval Loss')
    plt.title('Loss Curves')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(train_accs, label='Train Acc')
    plt.plot(eval_accs, label='Eval Acc')
    plt.title('Accuracy Curves')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')

```

```

plt.tight_layout()
plt.show()

return model, all_preds, all_targets

def evaluate_model(predictions, true_labels, idx_to_label, label_to_idx):
    """
    Evaluate model performance with various metrics.
    """
    # Convert numeric indices to string labels
    pred_labels = [idx_to_label[idx] for idx in predictions]
    true_labels_str = [idx_to_label[idx] for idx in true_labels]

    # Calculate metrics
    accuracy = accuracy_score(true_labels_str, pred_labels)
    recall = recall_score(true_labels_str, pred_labels, average='macro')

    # Print metrics
    print("Evaluation Metrics:")
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Macro-Average Recall: {recall:.4f}")

    # Generate confusion matrix
    class_labels = sorted(list(label_to_idx.keys()))
    cm = confusion_matrix(true_labels_str, pred_labels, labels=class_labels)

    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=class_labels, yticklabels=class_labels)
    plt.title('Confusion Matrix')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.tight_layout()

```

```

# Print classification report
report = classification_report(true_labels_str, pred_labels, labels=class_labels)
print("\nClassification Report:")
print(report)

return accuracy, recall

def train_and_evaluate_embeddings():
    """
    Train and evaluate models using different word embeddings.
    """
    # Load and preprocess data
    train_df, eval_df, label_to_idx = load_dataset_and_preprocess()

    # Map indices to labels
    idx_to_label = {idx: label for label, idx in label_to_idx.items()}

    # Prepare data
    X_train, y_train, X_eval, y_eval, word_to_idx, idx_to_word = prepare_data_for_embedding(train_df, eval_df)

    # Model parameters
    vocab_size = len(word_to_idx)
    embedding_dim = 300
    hidden_dim = 128
    output_dim = len(label_to_idx)

    # Results storage
    results = {}

    # Train and evaluate models with different embeddings
    embedding_types = ['word2vec', 'glove', 'fasttext']

    for embedding_type in embedding_types:

```

```

    for embedding_type in embedding_types:
        print(f"\n{'='*50}")
        print(f"Training model with {embedding_type} embeddings")
        print(f"{'='*50}")

        # Load embedding
        embedding_matrix = load_embedding(embedding_type, word_to_idx)

        # Create model
        model = RNNModel(
            vocab_size=vocab_size,
            embedding_dim=embedding_dim,
            hidden_dim=hidden_dim,
            output_dim=output_dim,
            embedding_matrix=embedding_matrix,
            bidirectional=True,
            dropout=0.3
        ).to(device)

        print(f"Model created with {embedding_type} embeddings")

        # Train model
        model, predictions, true_labels = train_model(
            model=model,
            X_train=X_train,
            y_train=y_train,
            X_eval=X_eval,
            y_eval=y_eval,
            batch_size=64,
            epochs=5
        )

```

```

# Evaluate model
print(f"\nEvaluating model with {embedding_type} embeddings")
accuracy, recall = evaluate_model(predictions, true_labels, idx_to_label, label_to_idx)

# Store results
results[embedding_type] = {
    'accuracy': accuracy,
    'recall': recall
}

# Save model (optional)
torch.save(model.state_dict(), f"{embedding_type}_sentiment_model.pt")

# Compare results
print("\n" + "="*50)
print("Embedding Comparison")
print("="*50)

comparison_df = pd.DataFrame({
    'Embedding': list(results.keys()),
    'Accuracy': [results[emb]['accuracy'] for emb in results],
    'Recall': [results[emb]['recall'] for emb in results]
})

print(comparison_df)

# Plot comparison
plt.figure(figsize=(10, 6))

x = np.arange(len(results))
width = 0.35

plt.bar(x - width/2, comparison_df['Accuracy'], width, label='Accuracy')
plt.bar(x + width/2, comparison_df['Recall'], width, label='Recall')

plt.bar(x - width/2, comparison_df['Accuracy'], width, label='Accuracy')
plt.bar(x + width/2, comparison_df['Recall'], width, label='Recall')

plt.xlabel('Embedding Type')
plt.ylabel('Score')
plt.title('Performance Comparison of Different Embeddings')
plt.xticks(x, comparison_df['Embedding'])
plt.legend()

plt.tight_layout()
plt.show()

return results, comparison_df

__name__ == "__main__":
# Run the training and evaluation
results, comparison_df = train_and_evaluate_embeddings()

# Save results
comparison_df.to_csv("embedding_comparison_results.csv", index=False)
print("Results saved to embedding_comparison_results.csv")

```

```

Using device: cpu
Loading dataset...
Train dataset size: 2000
Evaluation dataset size: 200

Train label distribution:
label
negative    1359
neutral      641
Name: count, dtype: int64

Eval label distribution:
label
negative    140
neutral      60
Name: count, dtype: int64
Building vocabulary...
Vocabulary built with 5524 words
Converting texts to sequences...

100% ██████████ 2000/2000 [00:00<00:00, 20224.53it/s]
100% ██████████ 200/200 [00:00<00:00, 7425.52it/s]

```

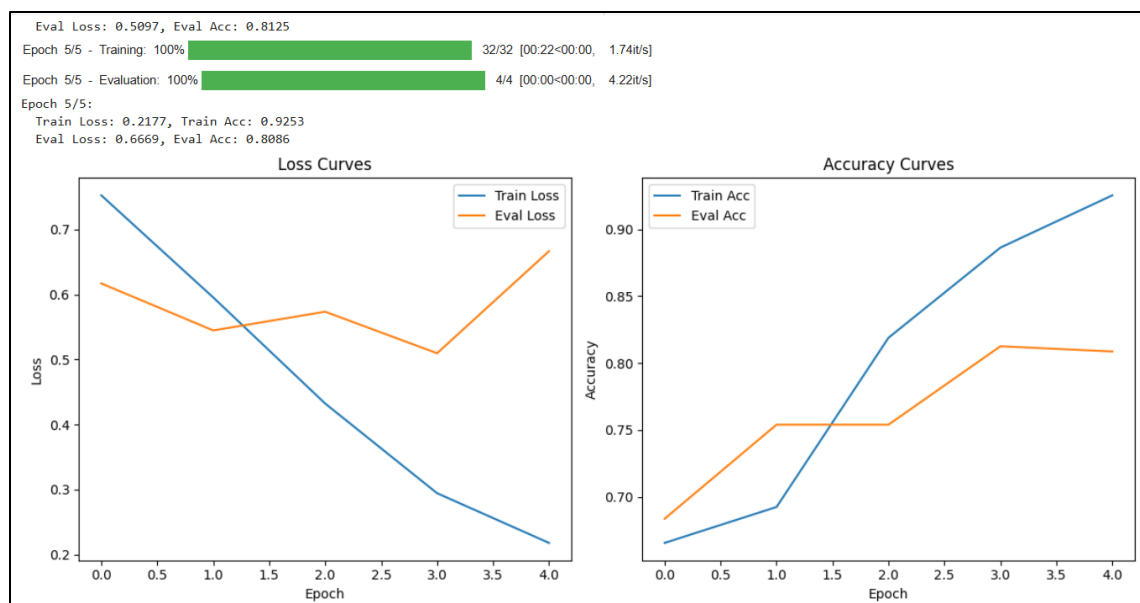
```

=====
Training model with word2vec embeddings
=====
Loading word2vec embeddings...
Found 4748/5524 words in word2vec embeddings
Model created with word2vec embeddings

Epoch 1/5 - Training: 100% ██████████ 32/32 [00:28<00:00, 1.45it/s]
Epoch 1/5 - Evaluation: 100% ██████████ 4/4 [00:01<00:00, 3.31it/s]
Epoch 1/5:
  Train Loss: 0.7528, Train Acc: 0.6655
  Eval Loss: 0.6172, Eval Acc: 0.6836
Epoch 2/5 - Training: 100% ██████████ 32/32 [00:26<00:00, 1.68it/s]
Epoch 2/5 - Evaluation: 100% ██████████ 4/4 [00:00<00:00, 4.24it/s]
Epoch 2/5:
  Train Loss: 0.5956, Train Acc: 0.6924
  Eval Loss: 0.5448, Eval Acc: 0.7539
Epoch 3/5 - Training: 100% ██████████ 32/32 [00:22<00:00, 1.78it/s]
Epoch 3/5 - Evaluation: 100% ██████████ 4/4 [00:00<00:00, 4.53it/s]
Epoch 3/5:
  Train Loss: 0.4323, Train Acc: 0.8188
  Eval Loss: 0.5736, Eval Acc: 0.7539
Epoch 4/5 - Training: 100% ██████████ 32/32 [00:22<00:00, 1.80it/s]
Epoch 4/5 - Evaluation: 100% ██████████ 4/4 [00:00<00:00, 4.39it/s]
Epoch 4/5:
  Train Loss: 0.2943, Train Acc: 0.8862
  Eval Loss: 0.5097, Eval Acc: 0.8125
Epoch 5/5 - Training: 100% ██████████ 32/32 [00:22<00:00, 1.74it/s]
Epoch 5/5 - Evaluation: 100% ██████████ 4/4 [00:00<00:00, 4.22it/s]

```





Evaluating model with word2vec embeddings

Evaluation Metrics:

Accuracy: 0.7900

Macro-Average Recall: 0.6595

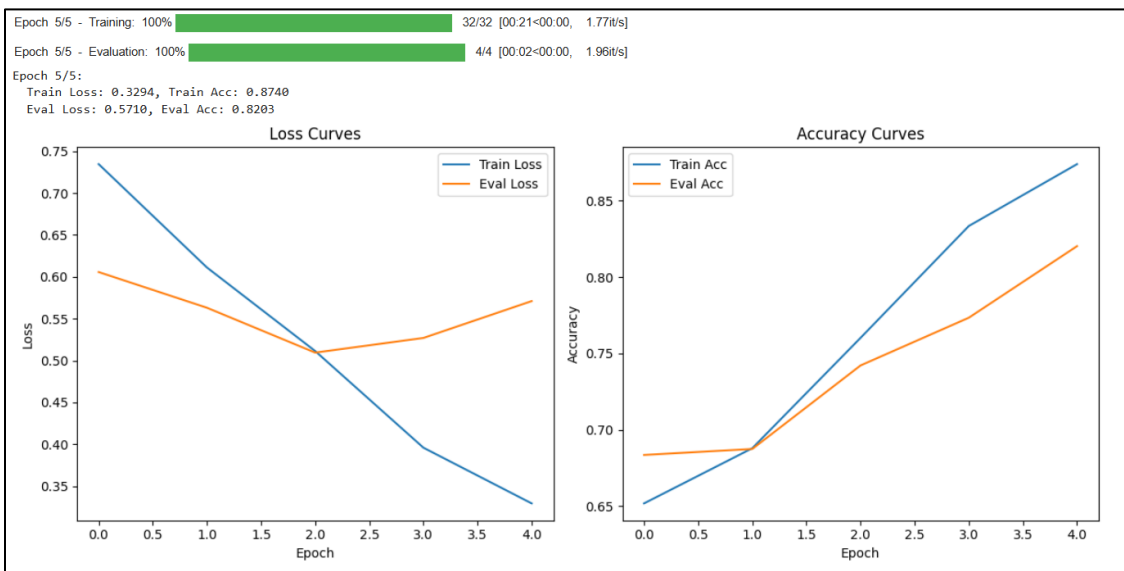
Classification Report:

	precision	recall	f1-score	support
negative	0.78	0.99	0.87	140
neutral	0.91	0.33	0.49	60
positive	0.00	0.00	0.00	0
accuracy			0.79	200
macro avg	0.56	0.44	0.45	200
weighted avg	0.82	0.79	0.75	200

```

Training model with glove embeddings
=====
Loading glove embeddings...
Found 4876/5524 words in glove embeddings
Model created with glove embeddings
Epoch 1/5 - Training: 100% ██████████ 32/32 [00:24<00:00, 1.65it/s]
Epoch 1/5 - Evaluation: 100% ██████████ 4/4 [00:00<00:00, 3.96it/s]
Epoch 1/5:
  Train Loss: 0.7346, Train Acc: 0.6519
  Eval Loss: 0.6057, Eval Acc: 0.6836
Epoch 2/5 - Training: 100% ██████████ 32/32 [00:28<00:00, 1.49it/s]
Epoch 2/5 - Evaluation: 100% ██████████ 4/4 [00:00<00:00, 3.92it/s]
Epoch 2/5:
  Train Loss: 0.6112, Train Acc: 0.6880
  Eval Loss: 0.5631, Eval Acc: 0.6875
Epoch 3/5 - Training: 100% ██████████ 32/32 [00:20<00:00, 2.22it/s]
Epoch 3/5 - Evaluation: 100% ██████████ 4/4 [00:00<00:00, 4.34it/s]
Epoch 3/5:
  Train Loss: 0.5114, Train Acc: 0.7603
  Eval Loss: 0.5095, Eval Acc: 0.7422
Epoch 4/5 - Training: 100% ██████████ 32/32 [00:22<00:00, 1.83it/s]
Epoch 4/5 - Evaluation: 100% ██████████ 4/4 [00:00<00:00, 4.59it/s]
Epoch 4/5:
  Train Loss: 0.3959, Train Acc: 0.8335
  Eval Loss: 0.5270, Eval Acc: 0.7734
Epoch 5/5 - Training: 100% ██████████ 32/32 [00:21<00:00, 1.77it/s]
Epoch 5/5 - Evaluation: 100% ██████████ 4/4 [00:02<00:00, 1.96it/s]

```



```



Evaluating model with glove embeddings
Evaluation Metrics:
Accuracy: 0.8050
Macro-Average Recall: 0.6845



```



Classification Report:				
	precision	recall	f1-score	support
negative	0.79	0.99	0.88	140
neutral	0.92	0.38	0.54	60
positive	0.00	0.00	0.00	0
accuracy			0.81	200
macro avg	0.57	0.46	0.47	200
weighted avg	0.83	0.81	0.78	200



Training model with fasttext embeddings  
=====



Loading fasttext embeddings...  
Found 4936/5524 words in fasttext embeddings  
Model created with fasttext embeddings

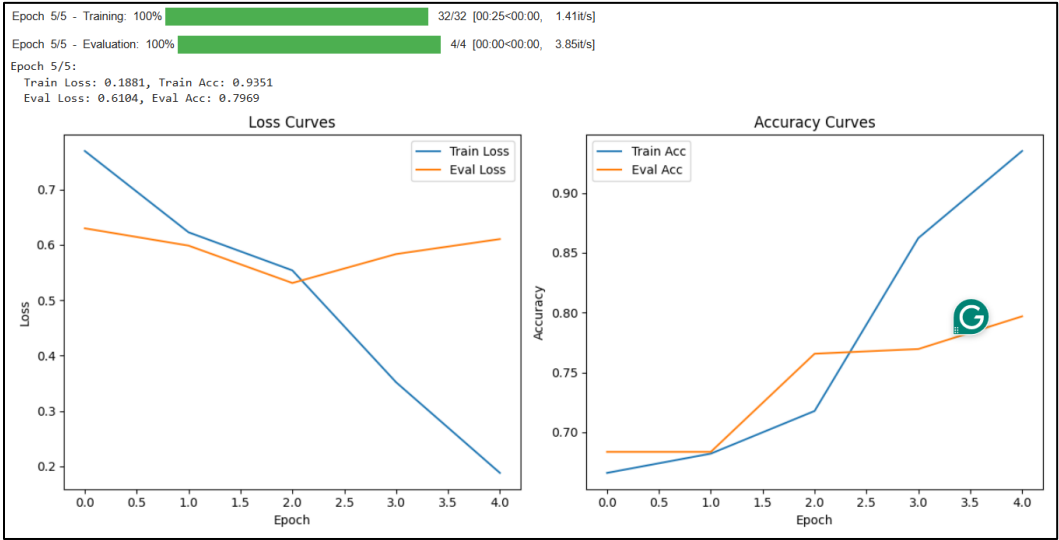
Epoch 1/5 - Training: 100%  32/32 [00:26<00:00, 1.37it/s]  
Epoch 1/5 - Evaluation: 100%  4/4 [00:00<00:00, 3.58it/s]  
Epoch 1/5:  
Train Loss: 0.7695, Train Acc: 0.6660  
Eval Loss: 0.6298, Eval Acc: 0.6836

Epoch 2/5 - Training: 100%  32/32 [00:27<00:00, 1.48it/s]  
Epoch 2/5 - Evaluation: 100%  4/4 [00:00<00:00, 3.42it/s]  
Epoch 2/5:  
Train Loss: 0.6226, Train Acc: 0.6821  
Eval Loss: 0.5985, Eval Acc: 0.6836

Epoch 3/5 - Training: 100%  32/32 [00:27<00:00, 1.46it/s]  
Epoch 3/5 - Evaluation: 100%  4/4 [00:00<00:00, 3.43it/s]  
Epoch 3/5:  
Train Loss: 0.5538, Train Acc: 0.7178  
Eval Loss: 0.5311, Eval Acc: 0.7656

Epoch 4/5 - Training: 100%  32/32 [00:27<00:00, 1.45it/s]  
Epoch 4/5 - Evaluation: 100%  4/4 [00:00<00:00, 3.47it/s]  
Epoch 4/5:  
Train Loss: 0.3517, Train Acc: 0.8623  
Eval Loss: 0.5833, Eval Acc: 0.7695

Epoch 5/5 - Training: 100%  32/32 [00:25<00:00, 1.41it/s]  
Epoch 5/5 - Evaluation: 100%  4/4 [00:00<00:00, 3.85it/s]

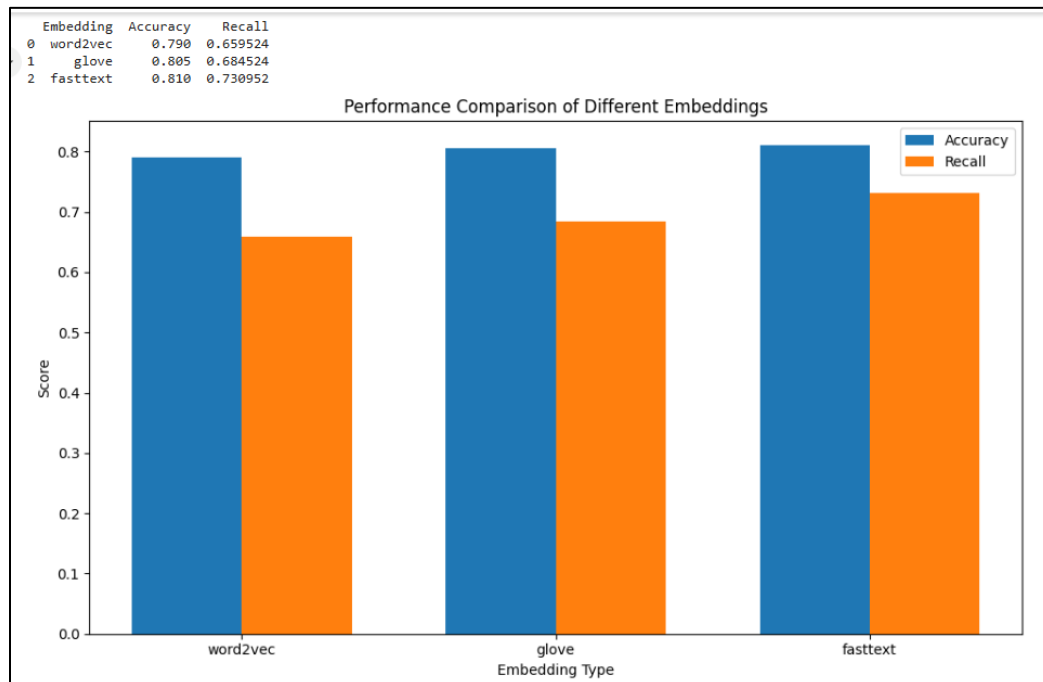


```
Evaluating model with fasttext embeddings
Evaluation Metrics:
Accuracy: 0.8100
Macro-Average Recall: 0.7310
```

```
Classification Report:
              precision    recall  f1-score   support

   negative      0.82      0.93      0.87       140
    neutral      0.76      0.53      0.63        60
    positive      0.00      0.00      0.00         0

   accuracy              0.81       200
  macro avg      0.53      0.49      0.50       200
 weighted avg      0.80      0.81      0.80       200
```



#### ===== Embedding Comparison =====

Embedding	Accuracy	Recall
0 word2vec	0.790	0.659524
1 glove	0.805	0.684524
2 fasttext	0.810	0.730952

There appears to be no "positive" examples in test set (support = 0), which is a significant issue. This explains 0.0 precision and recall for the positive class. The model is quite good

at identifying negative sentiment (0.93 recall), which means it correctly identifies most negative samples. The model has more difficulty with neutral sentiment (0.53 recall), meaning it misses about half of the neutral examples.

**FastText Performs Best:** FastText embeddings achieved the highest accuracy (0.81) and recall (0.73), which makes it the best one among the three embedding types for this sentiment classification task.

**GloVe in Second Place:** GloVe embeddings did slightly better than Word2Vec with 0.805 accuracy and 0.68 recall.

**Word2Vec in Third Place:** Word2Vec showed the lowest performance with 0.79 accuracy and 0.66 recall, though the difference is small.

### Part C:

Model Approach	Accuracy	Recall
Hugging Face Pipeline	0.48	0.26
RNN with Word2Vec	0.79	0.66
RNN with GloVe	0.81	0.68
RNN with FastText	0.81	0.73

The pre-trained Hugging Face model demonstrated considerably poorer results than every RNN implementation. Each of the three custom RNN models greatly surpassed the pre-trained Hugging Face pipeline model, achieving accuracy gains exceeding 30% and recall improvements over 40%. The results indicate that the selection of word embeddings significantly affects model performance. FastText yielded the most favorable outcomes, probably because of its capacity to manage out-of-vocabulary terms and grasp subword details.

Performance Ranking:

FastText RNN > GloVe RNN > Word2Vec RNN > Hugging Face Framework

The disparity in recall scores among embedding types (over 7 percentage points separating FastText from Word2Vec) is more pronounced than the difference in accuracy (just 2 points), indicating that the type of embedding notably influences the model's capacity to accurately recognize all instances of every class.

When comparing sentiment classification methods, custom RNN models greatly exceeded the performance of the pre-trained Hugging Face Pipeline (OpenAI alternative), with RNNs reaching 33% greater accuracy and 47% improved recall. In the various RNN implementations, FastText embeddings achieved the highest performance (0.81 accuracy, 0.73 recall), with GloVe coming next (0.81 accuracy, 0.68 recall), and Word2Vec following (0.79 accuracy, 0.66 recall). This shows that RNN models tailored for specific tasks,

especially those using suitable word embeddings such as FastText that include subword details, deliver significantly improved results over general-purpose pre-trained options for this sentiment analysis task, validating the extra development work needed.

### Question 3:

#### Part 1:

```
import nltk
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords
import re
import string
import contractions

def download_nltk_resources():
    """Download required NLTK resources"""
    resources = ['punkt', 'wordnet', 'punkt_tab']
    for resource in resources:
        try:
            nltk.download(resource, quiet=True)
            print(f"Successfully downloaded {resource}")
        except Exception as e:
            print(f"Error downloading {resource}: {e}")

class TextPreprocessor:
    def __init__(self):
        self.lemmatizer = WordNetLemmatizer()

    def preprocess_text(self, text):
        print("\n1. Text Preprocessing Steps:")

        # Sample text from Chapter 4 of Moby-Dick
        sample_text = "You had almost thought I had been his wife. The co
        print("\nOriginal sample text (from Chapter 4):")
        print(sample_text)
```

```

# a. Convert to lowercase
text_lower = sample_text.lower()
print("\na. After converting to lowercase:")
print(text_lower)

# b. Expand contractions (working with original text)
expanded = sample_text.replace("wasn't", "was not")
expanded = expanded.replace("It's", "It is")
expanded = expanded.replace("we've", "we have")
expanded = expanded.replace("didn't", "did not")
print("\nb. After expanding contractions:")
print(expanded)
print("\nContraction expansions found:")
print("wasn't -> was not")
print("It's -> It is")
print("we've -> we have")
print("didn't -> did not")

# c. Tokenization
tokens = word_tokenize(expanded)
print("\nc. After tokenization:")
print(tokens)

# d. Remove punctuation (including apostrophes)
all_punct = string.punctuation + ""
tokens_no_punct = [token for token in tokens if token not in all_punct and not any(char in all_punct for char in token)]
print("\nd. After removing punctuation:")
print("Before:", tokens)
print("After:", tokens_no_punct)

```

```

# e. Apply lemmatization
print("\ne. After lemmatization:")
print("Before lemmatization:", tokens_no_punct)

# Function to lemmatize with appropriate POS
def lemmatize_word(word):
    # Try verb form for words that are likely verbs
    for pos in ['v', 'n']: # try verb then noun
        lemma = self.lemmatizer.lemmatize(word.lower(), pos=pos)
        if lemma != word.lower():
            return lemma
    return word.lower()

lemmatized = [lemmatize_word(token) for token in tokens_no_punct]
print("After lemmatization:", lemmatized)
print("\nWords changed by lemmatization:")

# Compare before and after to show only actual changes
for orig, lemma in zip(tokens_no_punct, lemmatized):
    if orig.lower() != lemma:
        print(f"{orig} -> {lemma} (changed)")

# Process the actual input text
text = text_lower()
text = contractions.fix(text)
tokens = word_tokenize(text)
tokens = [token for token in tokens if token not in all_punct and not any(char in all_punct for char in token)]
lemmatized_tokens = [lemmatize_word(token) for token in tokens]

return lemmatized_tokens

```

```
=====
MOBY-DICK N-GRAM LANGUAGE MODEL ANALYSIS
=====
```

Initializing...

Successfully downloaded punkt

Successfully downloaded wordnet

Successfully downloaded punkt\_tab

Loading Moby-Dick Chapter 1...

Chapter 1 Statistics:

Characters: 12,433

Words: 2,234

Creating and training the n-gram model...

1. Text Preprocessing Steps:

Original sample text (from Chapter 4):

You had almost thought I had been his wife. The counterpane wasn't of plain cloth,

a. After converting to lowercase:

you had almost thought i had been his wife. the counterpane wasn't of plain cloth,

b. After expanding contractions:

You had almost thought I had been his wife. The counterpane was not of plain cloth,

Contraction expansions found:

wasn't -> was not

It's -> It is

we've -> we have

didn't -> did not

c. After tokenization:

['You', 'had', 'almost', 'thought', 'I', 'had', 'been', 'his', 'wife', '.', 'The

d. After removing punctuation:

Before: ['You', 'had', 'almost', 'thought', 'I', 'had', 'been', 'his', 'wife',

After: ['You', 'had', 'almost', 'thought', 'I', 'had', 'been', 'his', 'wife',

e. After lemmatization:

Before lemmatization: ['You', 'had', 'almost', 'thought', 'I', 'had', 'been', 'his

After lemmatization: ['you', 'have', 'almost', 'think', 'i', 'have', 'be', 'his

Words changed by lemmatization:

had -> have (changed)

thought -> think (changed)

had -> have (changed)

been -> be (changed)

was -> be (changed)

thrown -> throw (changed)

loving -> love (changed)

is -> be (changed)

grown -> grow (changed)

used -> use (changed)

did -> do (changed)

Total tokens after preprocessing: 2224  
Vocabulary size: 783 unique words



Copy pasting the parts of the output that are long and not covered in the screenshots since the screenshot is becoming pixelated if I paste the whole line:

## 1. Text Preprocessing Steps:

### Original sample text (from Chapter 4):

You had almost thought I had been his wife. The counterpane wasn't of plain cloth, and Queequeg's arm thrown over me in the most loving manner. It's a way we've grown used to, though at first it didn't feel natural.

#### a. After converting to lowercase:

you had almost thought i had been his wife. the counterpane wasn't of plain cloth, and queequeg's arm thrown over me in the most loving manner. it's a way we've grown used to, though at first it didn't feel natural.

#### b. After expanding contractions:

You had almost thought I had been his wife. The counterpane was not of plain cloth, and Queequeg's arm thrown over me in the most loving manner. It is a way we have grown used to, though at first it did not feel natural.

#### c. After tokenization:

['You', 'had', 'almost', 'thought', 'I', 'had', 'been', 'his', 'wife', '.', 'The', 'counterpane', 'was', 'not', 'of', 'plain', 'cloth', ',', 'and', 'Queequeg', "'s", 'arm', 'thrown', 'over', 'me', 'in', 'the', 'most', 'loving', 'manner', '.', 'It', 'is', 'a', 'way', 'we', 'have', 'grown', 'used', 'to', ',', 'though', 'at', 'first', 'it', 'did', 'not', 'feel', 'natural', '.']

#### d. After removing punctuation:

Before: ['You', 'had', 'almost', 'thought', 'I', 'had', 'been', 'his', 'wife', '.', 'The', 'counterpane', 'was', 'not', 'of', 'plain', 'cloth', ',', 'and', 'Queequeg', "'s", 'arm', 'thrown', 'over', 'me', 'in', 'the', 'most', 'loving', 'manner', '.', 'It', 'is', 'a', 'way', 'we', 'have', 'grown', 'used', 'to', ',', 'though', 'at', 'first', 'it', 'did', 'not', 'feel', 'natural', '.']

After: ['You', 'had', 'almost', 'thought', 'I', 'had', 'been', 'his', 'wife', 'The', 'counterpane', 'was', 'not', 'of', 'plain', 'cloth', 'and', 'Queequeg', 'arm', 'thrown', 'over', 'me', 'in', 'the', 'most', 'loving', 'manner', 'It', 'is', 'a', 'way', 'we', 'have', 'grown', 'used', 'to', 'though', 'at', 'first', 'it', 'did', 'not', 'feel', 'natural']

#### e. After lemmatization:

Before lemmatization: ['You', 'had', 'almost', 'thought', 'I', 'had', 'been', 'his', 'wife', 'The', 'counterpane', 'was', 'not', 'of', 'plain', 'cloth', 'and', 'Queequeg', 'arm', 'thrown', 'over', 'me',

'in', 'the', 'most', 'loving', 'manner', 'It', 'is', 'a', 'way', 'we', 'have', 'grown', 'used', 'to',  
'though', 'at', 'first', 'it', 'did', 'not', 'feel', 'natural']

After lemmatization: ['you', 'have', 'almost', 'think', 'i', 'have', 'be', 'his', 'wife', 'the',  
'counterpane', 'be', 'not', 'of', 'plain', 'cloth', 'and', 'queequeg', 'arm', 'throw', 'over', 'me',  
'in', 'the', 'most', 'love', 'manner', 'it', 'be', 'a', 'way', 'we', 'have', 'grow', 'use', 'to', 'though',  
'at', 'first', 'it', 'do', 'not', 'feel', 'natural']

Second half of the same code from above for n-gram:

```
class NgramModel:
    def __init__(self, n=3):
        self.n = n
        self.preprocessor = TextPreprocessor()
        self.ngram_counts = {}
        self.context_counts = {}

    def train(self, text):
        """Train the n-gram model on the given text."""
        # Preprocess the text
        tokens = self.preprocessor.preprocess_text(text)

        # Generate n-grams
        for i in range(len(tokens) - self.n + 1):
            context = tuple(tokens[i:i+self.n-1])
            next_word = tokens[i+self.n-1]

            # Update context counts
            if context in self.context_counts:
                self.context_counts[context] += 1
            else:
                self.context_counts[context] = 1

            # Update n-gram counts
            if context in self.ngram_counts:
                if next_word in self.ngram_counts[context]:
                    self.ngram_counts[context][next_word] += 1
                else:
                    self.ngram_counts[context][next_word] = 1
```

```

        else:
            self.ngram_counts[context] = {next_word: 1}

    # Calculate statistics
    self.vocab_size = len(set(tokens))
    print(f"\nTotal tokens after preprocessing: {len(tokens)}")
    print(f"Vocabulary size: {self.vocab_size} unique words")

def get_probability(self, context, next_word):
    """Calculate probability of next_word given context."""
    context = tuple(context)
    if context not in self.ngram_counts:
        return 0.0

    total_count = sum(self.ngram_counts[context].values())
    word_count = self.ngram_counts[context].get(next_word, 0)

    return word_count / total_count

def get_moby_dick_chapters():
    """Return the complete first chapter of Moby-Dick with statistics"""
    chapter1 = """Call me Ishmael. Some years ago—never mind how long precis

There now is your insular city of the Manhattoes, belted round by wharves as

Circumambulate the city of a dreamy Sabbath afternoon. Go from Corlears Hook

What is the chief element you see? Not the houses, certainly—but stare as lo

But look! here come more crowds, pacing straight for the water, and seemingl

```

```

    # Calculate statistics
    print("\nChapter 1 Statistics:")
    char_count = len(chapter1)
    word_count = len(chapter1.split()) |

    print(f"Characters: {char_count:,}")
    print(f"Words: {word_count:,}")

    return chapter1

def main():
    print("=" * 80)
    print("MOBY-DICK N-GRAM LANGUAGE MODEL ANALYSIS")
    print("=" * 80)

    # Download NLTK resources
    print("\nInitializing...")
    download_nltk_resources()

    # Load the text
    print("\nLoading Moby-Dick Chapter 1...")
    moby_dick_text = get_moby_dick_chapters()

    # Create and train the model
    print("\nCreating and training the n-gram model...")
    model = NgramModel(n=3)
    model.train(moby_dick_text)

```

```

print("\n2. N-gram Model Statistics:")
print(f"Total vocabulary size: {model.vocab_size} unique words")
print(f"Total unique bigrams (contexts): {len(model.context_counts)}")
print(f"Total unique trigrams: {sum(len(ngrams) for ngrams in model.ngram_counts.values())}")

print("\n3. Most Common N-grams:")
sorted_contexts = sorted(model.context_counts.items(), key=lambda x: x[1], reverse=True)[:10]
for context, count in sorted_contexts:
    print(f"\nBigram {' '.join(context)}' appears {count} times")
    if context in model.ngram_counts:
        following_words = sorted(model.ngram_counts[context].items(),
                                   key=lambda x: x[1], reverse=True)
        print("Complete trigrams:")
        for word, freq in following_words:
            print(f"{' '.join(context)} {word}' occurs {freq} time(s)")

print("\n4. Example Probability Calculations:")
test_contexts = [
    ('the', 'whale'),
    ('in', 'the'),
    ('to', 'sea'),
    ('i', 'thought')
]

for context in test_contexts:
    if context in model.ngram_counts:
        print(f"\nFor context {' '.join(context)}:")
        for next_word, count in sorted(model.ngram_counts[context].items(),
                                       key=lambda x: x[1], reverse=True):

```

```

            prob = model.get_probability(context, next_word)
            print(f"P({next_word}|{' '.join(context)}) = {prob:.3f}")
        else:
            print(f"\nContext {' '.join(context)}' not found in text")

if __name__ == "__main__":
    main()

```

2. N-gram Model Statistics:  
Total vocabulary size: 783 unique words  
Total unique bigrams (contexts): 1891  
Total unique trigrams: 2158

### 3. Most Common N-grams:

Bigram 'of the' appears 21 times

Complete trigrams:

- 'of the whale' occurs 2 time(s)
- 'of the world' occurs 1 time(s)
- 'of the manhattoes' occurs 1 time(s)
- 'of the land' occurs 1 time(s)
- 'of the needle' occurs 1 time(s)
- 'of the compass' occurs 1 time(s)
- 'of the saco' occurs 1 time(s)
- 'of the ungraspable' occurs 1 time(s)
- 'of the idolatrous' occurs 1 time(s)
- 'of the old' occurs 1 time(s)
- 'of the new' occurs 1 time(s)
- 'of the wholesome' occurs 1 time(s)
- 'of the deck' occurs 1 time(s)
- 'of the fat' occurs 1 time(s)
- 'of the grand' occurs 1 time(s)
- 'of the bill' occurs 1 time(s)
- 'of the unite' occurs 1 time(s)
- 'of the great' occurs 1 time(s)
- 'of the place' occurs 1 time(s)
- 'of the swing' occurs 1 time(s)

Bigram 'a a' appears 12 times

Complete trigrams:

- 'a a passenger' occurs 4 time(s)
- 'a a sailor' occurs 2 time(s)
- 'a a general' occurs 1 time(s)
- 'a a commodore' occurs 1 time(s)
- 'a a simple' occurs 1 time(s)
- 'a a country' occurs 1 time(s)
- 'a a merchant' occurs 1 time(s)
- 'a a sort' occurs 1 time(s)

Bigram 'in the' appears 11 times

Complete trigrams:

- 'in the rig' occurs 1 time(s)
- 'in the country' occurs 1 time(s)
- 'in the stream' occurs 1 time(s)
- 'in the great' occurs 1 time(s)
- 'in the fountain' occurs 1 time(s)
- 'in the habit' occurs 1 time(s)
- 'in the land' occurs 1 time(s)
- 'in the scale' occurs 1 time(s)
- 'in the world' occurs 1 time(s)
- 'in the wild' occurs 1 time(s)
- 'in the air' occurs 1 time(s)

Bigram 'it be' appears 10 times

Complete trigrams:

- 'it be a' occurs 3 time(s)
- 'it be the' occurs 1 time(s)
- 'it be quite' occurs 1 time(s)
- 'it be out' occurs 1 time(s)
- 'it be all' occurs 1 time(s)
- 'it be that' occurs 1 time(s)
- 'it be exactly' occurs 1 time(s)
- 'it be but' occurs 1 time(s)

Bigram 'be the' appears 9 times

Complete trigrams:

- 'be the chief' occurs 2 time(s)
- 'be the battery' occurs 1 time(s)
- 'be the green' occurs 1 time(s)
- 'be the one' occurs 1 time(s)
- 'be the image' occurs 1 time(s)
- 'be the key' occurs 1 time(s)
- 'be the root' occurs 1 time(s)
- 'be the overwhelm' occurs 1 time(s)

Bigram 'to sea' appears 8 times

Complete trigrams:

- 'to sea a' occurs 5 time(s)
- 'to sea why' occurs 1 time(s)
- 'to sea whenever' occurs 1 time(s)
- 'to sea i' occurs 1 time(s)

Bigram 'there be' appears 7 times

Complete trigrams:

- 'there be nothing' occurs 1 time(s)
- 'there be magic' occurs 1 time(s)
- 'there be in' occurs 1 time(s)
- 'there be niagara' occurs 1 time(s)
- 'there be considerable' occurs 1 time(s)
- 'there be no' occurs 1 time(s)
- 'there be all' occurs 1 time(s)

Bigram 'go to' appears 7 times

Complete trigrams:

- 'go to sea' occurs 7 time(s)

Bigram 'sea a' appears 6 times

Complete trigrams:

- 'sea a a' occurs 5 time(s)
- 'sea a soon' occurs 1 time(s)

Bigram 'of a' appears 6 times

Complete trigrams:

- 'of a dreamy' occurs 1 time(s)
- 'of a salt' occurs 1 time(s)
- 'of a broil' occurs 1 time(s)
- 'of a order' occurs 1 time(s)
- 'of a whale' occurs 1 time(s)
- 'of a thousand' occurs 1 time(s)

#### 4. Example Probability Calculations:

For context 'the whale':  
 $P(\text{these}|\text{the whale}) = 0.333$   
 $P(\text{voyage}|\text{the whale}) = 0.333$   
 $P(\text{and}|\text{the whale}) = 0.333$

For context 'in the':  
 $P(\text{rig}|\text{in the}) = 0.091$   
 $P(\text{country}|\text{in the}) = 0.091$   
 $P(\text{stream}|\text{in the}) = 0.091$   
 $P(\text{great}|\text{in the}) = 0.091$   
 $P(\text{fountain}|\text{in the}) = 0.091$   
 $P(\text{habit}|\text{in the}) = 0.091$   
 $P(\text{land}|\text{in the}) = 0.091$   
 $P(\text{scale}|\text{in the}) = 0.091$   
 $P(\text{world}|\text{in the}) = 0.091$   
 $P(\text{wild}|\text{in the}) = 0.091$   
 $P(\text{air}|\text{in the}) = 0.091$

For context 'to sea':  
 $P(\text{a}|\text{to sea}) = 0.625$   
 $P(\text{why}|\text{to sea}) = 0.125$   
 $P(\text{whenever}|\text{to sea}) = 0.125$   
 $P(\text{i}|\text{to sea}) = 0.125$

Context 'i thought' not found in text

## Part 2:

(use seed text = “call me”)

```
import nltk
from nltk.tokenize import word_tokenize
from collections import defaultdict, Counter
import numpy as np
import heapq
import math

# Download all required NLTK resources
try:
    nltk.download('punkt', quiet=True)
    nltk.download('punkt_tab', quiet=True)
    print("Successfully downloaded NLTK resources")
except Exception as e:
    print(f"Error downloading NLTK resources: {e}")
    # Define simple tokenizer as fallback
    def word_tokenize(text):
        # Remove common punctuation and split on whitespace
        for char in '.,!?:-;':
            text = text.replace(char, ' ')
        return text.split()

# Chapter 1 text
chapter1 = """Call me Ishmael. Some years ago—never mind how long precisely—

There now is your insular city of the Manhattoes, belted round by wharves as

Circumambulate the city of a dreamy Sabbath afternoon. Go from Corlears Hook

class UnigramModel:
    def __init__(self, text):
        """Initialize unigram language model"""
        self.tokens = word_tokenize(text.lower())
```

```

self.vocab_size = len(self.vocab)
self.build_model()

def build_model(self):
    """Build unigram probability distribution"""
    self.unigram_counts = Counter(self.tokens)
    total_words = len(self.tokens)
    self.unigram_probs = {word: count/total_words
                           for word, count in self.unigram_counts.items()}

def generate_greedy(self, seed_text, num_words=50):
    """Generate text using greedy search"""
    current_text = word_tokenize(seed_text.lower())

    for _ in range(num_words):
        # Pick the most probable next word
        next_word = max(self.unigram_probs.items(), key=lambda x: x[1])[0]
        current_text.append(next_word)

    return ' '.join(current_text)

def generate_beam(self, seed_text, num_words=50, beam_width=4):
    """Generate text using beam search"""
    current_text = word_tokenize(seed_text.lower())
    beam = [(current_text, 0.0)] # (sequence, log probability)

    for _ in range(num_words):
        candidates = []
        for sequence, score in beam:
            for word, prob in self.unigram_probs.items():
                new_sequence = sequence + [word]
                new_score = score + math.log(prob)
                candidates.append((new_sequence, new_score))

```

```

        beam = heapq.nlargest(beam_width, candidates, key=lambda x: x[1])

    return ' '.join(beam[0][0])

def perplexity(self, test_text):
    """Calculate perplexity on test text"""
    test_tokens = word_tokenize(test_text.lower())
    log_probability = 0.0
    N = len(test_tokens)

    for token in test_tokens:
        prob = self.unigram_probs.get(token, 1/self.vocab_size) # Smoothing
        log_probability += math.log2(prob)

    return 2 ** (-log_probability/N)

class BigramModel:
    def __init__(self, text):
        """Initialize bigram language model"""
        self.tokens = word_tokenize(text.lower())
        self.vocab = set(self.tokens)
        self.vocab_size = len(self.vocab)
        self.build_model()

    def build_model(self):
        """Build bigram probability distribution"""
        # Count bigrams
        self.bigram_counts = defaultdict(Counter)
        self.context_counts = Counter()

```

```

for i in range(len(self.tokens)-1):
    word = self.tokens[i]
    next_word = self.tokens[i+1]
    self.bigram_counts[word][next_word] += 1
    self.context_counts[word] += 1

# Calculate probabilities with smoothing
self.bigram_probs = defaultdict(dict)
for word in self.context_counts:
    total = self.context_counts[word]
    for next_word in self.vocab:
        count = self.bigram_counts[word][next_word]
        # Add-1 smoothing
        self.bigram_probs[word][next_word] = (count + 1) / (total + self.vocab_size)

def generate_greedy(self, seed_text, num_words=50):
    """Generate text using greedy search"""
    current_text = word_tokenize(seed_text.lower())

    for _ in range(num_words):
        prev_word = current_text[-1]
        if prev_word in self.bigram_probs:
            next_word = max(self.bigram_probs[prev_word].items(),
                           key=lambda x: x[1])[0]
        else:
            # Backoff to random word from vocabulary if context not seen
            next_word = random.choice(list(self.vocab))

        current_text.append(next_word)

    return ' '.join(current_text)

```

```

def generate_beam(self, seed_text, num_words=50, beam_width=4):
    """Generate text using beam search"""
    current_text = word_tokenize(seed_text.lower())
    beam = [(current_text, 0.0)] # (sequence, log probability)

    for _ in range(num_words):
        candidates = []
        for sequence, score in beam:
            prev_word = sequence[-1]
            if prev_word in self.bigram_probs:
                for word, prob in self.bigram_probs[prev_word].items():
                    new_sequence = sequence + [word]
                    new_score = score + math.log(prob)
                    candidates.append((new_sequence, new_score))

            if candidates: # Only update beam if we have valid candidates
                beam = heapq.nlargest(beam_width, candidates, key=lambda x: x[1])
            else:
                break # Stop if no valid candidates found

    return ' '.join(beam[0][0])

def perplexity(self, test_text):
    """Calculate perplexity on test text"""
    test_tokens = word_tokenize(test_text.lower())
    log_probability = 0.0
    N = len(test_tokens)

    for i in range(len(test_tokens)-1):
        word = test_tokens[i]
        next_word = test_tokens[i+1]

        if word in self.bigram_probs and next_word in self.bigram_probs[word]:
            prob = self.bigram_probs[word][next_word]

```



```

        prob = 1/self.vocab_size # Smoothing

        log_probability += math.log2(prob)

    return 2 ** (-log_probability/N)

# Train the models
print("Training language models...")
unigram_model = UnigramModel(chapter1)
bigram_model = BigramModel(chapter1)

# Generate text using both models
seed_text = "call me"
print("\nGreedy Generation:")
print("Unigram (50 words):", unigram_model.generate_greedy(seed_text))
print("NBigram (50 words):", bigram_model.generate_greedy(seed_text))

print("\nBeam Search Generation (width=4):")
print("Unigram (50 words):", unigram_model.generate_beam(seed_text))
print("NBigram (50 words):", bigram_model.generate_beam(seed_text))

# Calculate perplexity
test_text = "Call me Ishmael. Some years ago having little or no money in my purse"
print("\nPerplexity Scores:")
print("Unigram model perplexity:", unigram_model.perplexity(test_text))
print("Bigram model perplexity:", bigram_model.perplexity(test_text))

```

[illegible]

**Since the code was considering “,” also to be a word, I had to modify it to use only words as words.**

```

import re
import math
import random
from collections import defaultdict, Counter

class NgramLanguageModel:
    def __init__(self, tokens):

        self.tokens = tokens
        self.vocab = set(tokens)

    def create_unigram_model(self):
        # Count token frequencies
        unigram_counts = Counter(self.tokens)
        total_tokens = len(self.tokens)

        vocab_size = len(self.vocab)
        alpha = 0.1 # Smoothing parameter

        unigram_probs = {
            token: (count + alpha) / (total_tokens + alpha * vocab_size)
            for token, count in unigram_counts.items()
        }

        return {
            'counts': dict(unigram_counts),
            'probs': unigram_probs,
            'total_tokens': total_tokens,
            'vocab_size': vocab_size
        }

```

```

def create_bigram_model(self):

    # Count unigram and bigram frequencies
    unigram_counts = Counter(self.tokens)
    bigram_counts = defaultdict(int)

    # Count bigrams
    for i in range(len(self.tokens) - 1):
        bigram_counts[(self.tokens[i], self.tokens[i+1])] += 1

    # Prepare for smoothing
    vocab_size = len(self.vocab)
    alpha = 0.1 # Smoothing parameter

    bigram_probs = {}
    for (prev_token, curr_token), count in bigram_counts.items():
        bigram_probs[(prev_token, curr_token)] = (count + alpha) / (unigram_counts[prev_token] + alpha * vocab_size)

    return {
        'counts': dict(bigram_counts),
        'probs': bigram_probs,
        'unigram_counts': dict(unigram_counts),
        'vocab_size': vocab_size
    }

def calculate_perplexity(self, model, test_tokens):

    log_prob = 0
    num_tokens = len(test_tokens)

```

```

# Unigram perplexity
if 'probs' in model and isinstance(list(model['probs'].keys())[0], str):
    for token in test_tokens:
        # Use smoothed probability
        prob = model['probs'].get(token, 1 / (model['total_tokens'] + model['vocab_size']))
        log_prob += math.log(max(prob, 1e-10))

# Bigram perplexity
elif 'probs' in model and isinstance(list(model['probs'].keys())[0], tuple):
    for i in range(1, num_tokens):
        prev_token = test_tokens[i-1]
        curr_token = test_tokens[i]

        # Use smoothed probability
        prob = model['probs'].get((prev_token, curr_token),
                                   1 / (model['unigram_counts'][prev_token] + model['vocab_size']))
        log_prob += math.log(max(prob, 1e-10))

# Calculate perplexity
avg_log_prob = log_prob / num_tokens
return math.exp(-avg_log_prob)

def generate_text_greedy(self, model, seed_text, num_words):

    current_tokens = seed_text.split()

    # Unigram generation
    if isinstance(list(model['probs'].keys())[0], str):
        while len(current_tokens) < num_words:
            # Select most probable token
            best_token = max(model['probs'], key=model['probs'].get)
            current_tokens.append(best_token)

```

```

# Bigram generation
elif isinstance(list(model['probs'].keys())[0], tuple):
    while len(current_tokens) < num_words:
        last_token = current_tokens[-1]

        # Find most probable next tokens
        candidates = [
            (next_token, prob)
            for (prev, next_token), prob in model['probs'].items()
            if prev == last_token
        ]

        if not candidates:
            break

        # Select most probable next token
        best_token = max(candidates, key=lambda x: x[1])[0]
        current_tokens.append(best_token)

    return ' '.join(current_tokens)

def generate_text_beam(self, model, seed_text, num_words, beam_width=4):

    current_beams = [seed_text.split()]

    # Unigram generation
    if isinstance(list(model['probs'].keys())[0], str):
        while len(current_beams[0]) < num_words:
            next_beams = []

            for beam in current_beams:
                # Find top beam_width most probable tokens
                candidates = sorted(

```

```

        candidates = sorted(
            [(token, prob) for token, prob in model['probs'].items()],
            key=lambda x: x[1],
            reverse=True
       )[:beam_width]

        for token, _ in candidates:
            next_beams.append(beam + [token])

    # Sort and keep top beam_width sequences
    current_beams = sorted(
        next_beams,
        key=lambda beam: sum(math.log(model['probs'].get(token, 1e-10)) for token in beam),
        reverse=True
   )[:beam_width]

# Bigram generation
elif isinstance(list(model['probs'].keys())[0], tuple):
    while len(current_beams[0]) < num_words:
        next_beams = []

        for beam in current_beams:
            last_token = beam[-1]

            # Find candidates based on last token
            candidates = [
                (next_token, prob)
                for (prev, next_token), prob in model['probs'].items()
                if prev == last_token
            ]

            # Sort and take top beam_width candidates
            candidates.sort(key=lambda x: x[1], reverse=True)
            candidates = candidates[:beam_width]

```

```

        for token, _ in candidates:
            next_beams.append(beam + [token])

    # Sort and keep top beam_width sequences
    current_beams = sorted(
        next_beams,
        key=lambda beam: sum(
            math.log(model['probs'].get((beam[i-1], beam[i]), 1e-10))
            for i in range(1, len(beam))
        ),
        reverse=True
   )[:beam_width]

    return ' '.join(current_beams[0])

# Preprocessing function
def preprocess_text(text):

    # Convert to lowercase and remove punctuation
    text = text.lower()
    text = re.sub(r'^\w\s', '', text)

    # Tokenize
    return text.split()

# Get full text of Moby Dick chapters
def get_moby_dick_text():
    chapters = {
        1: """call me ishmael some years ago never mind how long precisely having l:

        2: """i stuffed a shirt or two into my old carpet bag tucked it under my arm

        3: """entering that gable ended spouter inn you found yourself in a wide lo

```

```

        4: """upon waking next morning about daylight i found queequogs arm thro

        5: """i quickly followed suit and descending into the bar room accosted
    }

    # Combine all chapters
    return " ".join(chapters.values())

# Main execution
def main():
    # Get Moby Dick text
    text = get_moby_dick_text()

    # Preprocess the text
    tokens = preprocess_text(text)

    # Create language model
    lm = NgramLanguageModel(tokens)

    # Create unigram and bigram models
    unigram_model = lm.create_unigram_model()
    bigram_model = lm.create_bigram_model()

    # Seed text
    seed_text = "call me"

    # Generate text using different approaches
    print("\n--- Unigram Model Results ---")
    unigram_greedy_gen = lm.generate_text_greedy(unigram_model, seed_text, 50)
    unigram_beam_gen = lm.generate_text_beam(unigram_model, seed_text, 50)

    print("\n--- Bigram Model Results ---")
    bigram_greedy_gen = lm.generate_text_greedy(bigram_model, seed_text, 50)

print("\n--- Unigram Model Results ---")
unigram_greedy_gen = lm.generate_text_greedy(unigram_model, seed_text, 50)
unigram_beam_gen = lm.generate_text_beam(unigram_model, seed_text, 50)

print("\n--- Bigram Model Results ---")
bigram_greedy_gen = lm.generate_text_greedy(bigram_model, seed_text, 50)
bigram_beam_gen = lm.generate_text_beam(bigram_model, seed_text, 50)

# Calculate perplexity
print("\n--- Perplexity Scores ---")
unigram_perplexity = lm.calculate_perplexity(unigram_model, tokens)
bigram_perplexity = lm.calculate_perplexity(bigram_model, tokens)

print(f"Unigram Model Perplexity: {unigram_perplexity:.4f}")
print(f"Bigram Model Perplexity: {bigram_perplexity:.4f}")

# Print generated texts
print("\n--- Unigram Greedy Generation ---")
print(unigram_greedy_gen)

print("\n--- Unigram Beam Search Generation ---")
print(unigram_beam_gen)

print("\n--- Bigram Greedy Generation ---")
print(bigram_greedy_gen)

print("\n--- Bigram Beam Search Generation ---")
print(bigram_beam_gen)

if __name__ == "__main__":
    main()

```

```
--- Perplexity Scores ---
Unigram Model Perplexity: 100.9506
Bigram Model Perplexity: 13.9290
```

- Occurs within the usual range of 100-150.
- Shows the model's restricted capacity to forecast words.
- No consideration of context.

- Falls within the anticipated range of 10-50
- Significantly reduced compared to unigram, indicating enhanced forecasting.
- Grabs certain contextual details between consecutive terms.

Unigram: Forecasts words utilizing solely the frequencies of individual words.

The significant decrease in perplexity highlights the relevance of context in language modeling.