Ml PROJECT Report

Topic: LLM

Name: Pranshul Thakur

Registration No: 12219336

Roll No: 3, K22UR

1. Problem Understanding & Definition

1.1 Clarity of Problem Statement (4 Marks)

This project focuses on building a **large language model (LLM)** capable of **text generation, sequence modelling, and contextual understanding**. The goal is to train a transformer-based model that can **generate coherent text, learn from structured data, and adapt to various NLP tasks**. The dataset consists of **literary text** from data.txt, which provides structured prose and poetry, making it suitable for training an LLM.

1.2 Justification for Solving the Problem (3 Marks)

LLMs are revolutionizing **AI-driven applications**, including **chatbots, code generation, automated summarization, and AI-assisted writing**. Training an LLM on structured literary data enhances its **ability to understand natural language, generate human-like responses, and improve contextual learning**. **By preprocessing and tokenizing the text effectively, the model can be optimized for high-quality language generation.**

1.3 Defined Objectives & Hypotheses (3 Marks)

Objectives:

- Build and train a **transformer-based LLM** capable of generating meaningful text.
- Preprocess and tokenize the dataset to improve training efficiency.
- Implement **batch processing and optimized data handling** to scale the model.
- Evaluate the LLM's **performance in text generation and contextual accuracy**.

Hypothesis:

- Proper text preprocessing (e.g., stopword removal, lowercasing) improves model accuracy.
- Training on structured text allows better generalization in **text generation and reasoning tasks**.
- Larger context windows lead to **more coherent and context-aware responses**.

2. Dataset Selection & Preprocessing

2.1 Dataset Relevance and Quality (3 Marks)

2.1.1 Dataset Selection

The dataset (data.txt) consists of **literary text**, making it a good candidate for **training an LLM**. It is designed to help the model learn **contextual relationships, text structure, and coherence** in language generation.

Dataset Overview:

- **Source:** Provided text file (data.txt)
- **Format:** Plain text
- **Feature Type:** Unstructured text
- **Length:** Multiple paragraphs of structured literary text

2.2 Handling Missing Values, Outliers, and Data Normalization (3 Marks)

2.2.1 Handling Missing Values

The dataset was **checked for missing values** using strip() and empty line checks. The preprocessing step:

- **Removes blank lines** before encoding the text.
- **Ignores encoding errors** by using errors='ignore'.

2.2.2 Handling Outliers

Outliers (unwanted noise in the text) were handled by:

- **Removing stopwords** to improve model focus on essential words.
- **Normalizing text** (e.g., converting to lowercase) for consistency.

2.2.3 Data Normalization & Standardization

- **Lowercased all text** to maintain uniformity.
- **Stopword removal** applied using the nltk.stopwords library.
- **Tokenization applied** using AutoTokenizer from bert-base-uncased.

2.3 Feature Selection & Engineering (4 Marks)

A. Feature Selection

- The dataset was tokenized using **BERT tokenizer (AutoTokenizer)**.
- Stopwords were removed using **NLTK stopwords** to reduce unnecessary words.
- **Non-alphabetic characters were kept** to retain meaning.

B. Feature Engineering

- **Encoding function created** (encode = lambda s: tokenizer.encode(s, add_special_tokens=True)) to convert text into token IDs.
- **Custom preprocessing function added (preprocess_text)** for stopword removal and text normalization.
- **Sequence chunking applied** to structure input data for transformer models.

C. Code Snippets

```
You, 18 hours ago | 1 author (You) | Run Cell | Run Below | Debug Cell
#%%
import torch
import torch.nn as nn
from torch.nn import functional as F
import mmap
import random
import pickle
import argparse
from transformers import AutoTokenizer
import time
import nltk
from nltk.corpus import stopwords
nltk.download('stopwords')
# from flash_attn import flash_attn_qkvpacked_func, flash_attn_func



Run Cell | Run Above | Debug Cell
#%%
device = 'cpu'  # for now
# if torch.cuda.is_available() else 'cpu'



Run Cell | Run Above | Debug Cell
#%%
parser = argparse.ArgumentParser(description='This is a demonstration program')
```

```
Run Cell | Run Above | Debug Cell
#%%
block_size = 8 # input tokens
batch_size = 4 # training samples
max_iterations = 1000 # training steps
learning_rate = 3e-4 # step size updation at rate of 0.0003
model_evaluation_iterations = 250
embedded_dim = 256
parallel_head = 4
no_layer = 4 #
dropout = 0.2 # to prevent overfitting



Run Cell | Run Above | Debug Cell
#%%
stop_words = set(stopwords.words('english')) #



Run Cell | Run Above | Debug Cell
#%%
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased") # loading uncased bert tokenizer
vocab_size = len(tokenizer)



Run Cell | Run Above | Debug Cell
#%%
encode = lambda s: tokenizer.encode(s, add_special_tokens=True) # encoding and decoding func
decode = lambda l: tokenizer.decode(l, skip_special_tokens=True)
```

```
Run Cell | Run Above | Debug Cell
#%%
Codeium: Refactor | Explain | Generate Docstring | X
def preprocess_text(text):
    text = text.lower().strip()  # Convert text to lowercase and remove extra spaces
    text = ' '.join([word for word in text.split() if word not in stop_words])  # Remove stopwords
    return text



Run Cell | Run Above | Debug Cell
#%%
Codeium: Refactor | Explain | Generate Docstring | X
def load_half_dataset_into_memory(filename):
    with open(filename, 'r', encoding='utf-8') as f:
        f.seek(0, 2) # moving file pointer to end to find the end location
        half_point = f.tell() // 2 # determining half point
        f.seek(0) # moving file pointer back to start
        data = f.read(half_point)
    return preprocess_text(data)  # Apply preprocessing before returning data
```

```
Run Cell | Run Above | Debug Cell
#%%
Codeium: Refactor | Explain | Generate Docstring | X
def get_random_chunk(split):
    filename = "train_split.txt" if split == 'train' else "val_split.txt"
    with open(filename, 'rb') as f:
        with mmap.mmap(f.fileno(), 0, access=mmap.ACCESS_READ) as mm:
            file_size = len(mm)
            start_pos = random.randint(0, file_size - block_size * batch_size)
            if start_pos > 0:
                mm.seek(start_pos - 1)
                while mm.read(1) != b"\n" and mm.tell() < file_size:
                    pass
                start_pos = mm.tell()
            end_pos = start_pos + block_size * batch_size
            if end_pos > file_size:
                start_pos = max(0, file_size - block_size * batch_size)
                mm.seek(start_pos)
            block = mm.read(block_size * batch_size)
            decoded_block = block.decode('utf-8', errors='ignore').replace('\r', '').strip() # data normalization a
            if not decoded_block:
                print("Warning: Encountered empty chunk, retrying...")
                return get_random_chunk(split)
            processed_block = preprocess_text(decoded_block)
            data = torch.tensor(encode(processed_block), dtype=torch.long)
    return data
```