Assignment 1 — Data Collection and Preprocessing Report

Student: Yuqi Lai

Course: CSYE7374

1. Dataset Sources and Total Size

I constructed a diverse corpus totaling 1.20 GB of raw text across 1,252,305 documents. To ensure balanced linguistic coverage, I utilized a streaming pipeline to collect exactly 409.60 MB from each of the three complementary sources:

- **FineWeb** (138,582 docs): Captures modern conversational web content.
- **Wikipedia** (927,496 docs): Provides structured, fact-dense encyclopedic knowledge.
- **CC-News** (186,227 docs): Offers narrative journalistic writing and event descriptions.

```
--- Statistics for Report ---
Source          | Size (MB)  | Documents
-----------------------------------------
FineWeb         | 409.60     | 138582
Wikipedia       | 409.60     | 927496
CC_News         | 409.60     | 186227
```

(Figure 1: Statistics of Collected Raw Data)

2. Cleaning Strategies and Reasoning

To improve data quality and reduce noise, I implemented a three-stage cleaning pipeline:

2.1 Normalization

I removed HTML tags, URLs, Markdown symbols, and Wikipedia formatting. This ensures the tokenizer learns natural language structures rather than technical artifacts.

2.2 Length Filtering

Documents under **50 words** were removed as statistical outliers. Short texts often lack meaningful context and degrade sequence learning quality.

2.3 Deduplication (MD5)

I applied exact-match MD5 hashing to remove duplicate documents, preventing the model from memorizing repetitive data and improving generalization.

After cleaning, the final dataset contained **808,966** documents.

3. Tokenization Choices

I trained a custom **Byte-Level BPE tokenizer** with a vocabulary size of **32,000** and a context window of **1,024 tokens**.

Justification: A custom tokenizer provides better coverage for **post-2024 vocabulary** and domain-specific terms compared to the GPT-2 tokenizer. Choosing **32k** (vs 50k) reduces the embedding layer size, improving training efficiency.

Handling Variable Lengths: Tokenized documents were **concatenated and chunked** into fixed 1,024-token blocks. This "packed dataset" strategy eliminates padding, thereby maximizing GPU utilization.

4. Data Loader Implementation

I implemented a custom **PyTorch IterableDataset (GPTStreamingDataset)**, enabling memory-efficient streaming without loading the full 1.2GB into RAM.

Batching: The loader produces tensors of shape [4, 1024], confirming the batch size of 4 and sequence length of 1024.

Buffered Shuffling: To satisfy the shuffling requirement on a stream, I implemented a local shuffle buffer (size=10,000). This loads 10k samples into memory and shuffles them locally to ensure sufficient randomness in sample ordering for any downstream training workflow.

```
Captured batch 1: Input Shape torch.Size([4, 1024])
Captured batch 2: Input Shape torch.Size([4, 1024])
Captured batch 3: Input Shape torch.Size([4, 1024])
Captured batch 4: Input Shape torch.Size([4, 1024])
Captured batch 5: Input Shape torch.Size([4, 1024])
```

(Figure 2: Data Loader Output Example)

5. Challenges Encountered

5.1 Streaming Shuffle Issue

Standard **shuffle()** operations cannot be applied to streaming (Iterable) datasets because they do not support random indexing.

Streaming datasets are consumed sequentially and do not allow arbitrary access (e.g., dataset[100]), making global shuffling impossible.

Solution: To simulate randomness without full random access, I implemented a **manual shuffle buffer** inside the `__iter__()` method.

A fixed-size buffer (e.g., 10,000 samples) is filled incrementally, and each training sample is drawn randomly from this buffer while new samples stream in. This provides effective local shuffling without violating streaming constraints.

5.2 Memory Load

Deduplication, cleaning, and preprocessing large corpora can easily cause **OOM (Out-of-Memory)** errors if all data is loaded at once.

Similarly, naive tokenization with padding wastes significant memory because many sequences are shorter than the maximum block size.

Solution: I used **streaming + Python generators** to process documents incrementally instead of loading the entire dataset into memory.

During tokenization, I applied a **concatenate-and-chunk** strategy to construct fixed-size 1,024-token blocks. This approach avoids expensive padding and ensures high GPU utilization while keeping memory consumption low.

6. Reflections on Preprocessing Impact

Quality Improvement: Removing noisy, short, or duplicated content substantially increases the consistency and factual reliability of the training data.

Efficiency Gains: The smaller vocabulary (32k) and packed block strategy directly reduce model parameter count and minimize wasted computation on padding tokens.

Overall, the preprocessing pipeline significantly improves both **training stability** and **dataset usability**.