**Mainpipe Data Engineering Assessment**

Michael Albany

**Overview**

I relied on a number of sources to determine LLM pre-processing pipelines best practices:

<https://latitude-blog.ghost.io/blog/ultimate-guide-to-preprocessing-pipelines-for-llms/>

<https://latitude-blog.ghost.io/blog/how-to-clean-noisy-text-data-for-llms/>

<https://aws.amazon.com/blogs/machine-learning/an-introduction-to-preparing-your-own-dataset-for-llm-training/>

<https://arxiv.org/html/2506.05209v1>

<https://arxiv.org/html/2406.11794v4>

<https://arxiv.org/html/2411.12372v1>

This project implements a scalable data pre-processing pipeline for LLM training datasets. Its structured around three core classes.

* PipelineStep – Performs transformations and filtering on the data
* Pipeline – Orchestrates the pipeline steps in succession
* Validator – Checks data from transformations PipelineSteps are applied correctly

Steps removing records log what data is removed to ensure inspectability.

The pipeline performs the following data cleaning steps in order.

**Basic pre-processing**

1. Null cleaning
2. Non-utf8 character cleaning
3. Html element cleaning
4. Special character cleaning

**Relevance filtering**

1. Quality filtering
2. Language filtering

**Deduplication**

1. Exact deduplication
2. Fuzzy deduplication

**Safety**

1. PII masking
2. Toxicity removal

Finally a tokeniser step takes place on the cleaned data.

**Data exploration**

Many of the design choices are based on information gathered in my exploratory analysis. See 1.1 – EDA Writeup.pdf

**Design choices**

**Basic pre-processing**

Empty text, strange characters (non-utf8), special characters and html elements all produce noise that interferes with LLM’s. The initial pipeline steps clean the noise. For html cleaning the trafilatura library was used as it provides various functions for dealing with HTML, including extracting metadata (which can be useful for filtering purposes). Interestingly the reliparse library is also good at dealing with HTML elements and works a lot faster than trafiltura, so this could be considered for scale-up.

Notably the non-utf8 character cleaning step wasn’t perfect in removing noise and special characters had to be cleaned further through regex.

**Relevance filtering**

**Aim**

Based on the EDA we could tell approximately 26% of the dataset is code, Only code and natural language that is cohesive and well documented is retained.

**The quality filtering step**

A number of actions occur in this step so I wanted to go through them below. It loosely follows the dataset quality filtering used for the common pile dataset: <https://arxiv.org/html/2506.05209v1>

Wordcount outliers that are either too long or too short are removed. The lower threshold for this was 30 words, based on some of the data preparation writeups I found and the upper was 15,000, which was based on the log scaled document lengths distribution (below).

A graph of a graph

AI-generated content may be incorrect.Repetitive text (e.g uninformative headings or duplicate sentences) is removed using a ‘repetitiveness score’. In this case I calculated how many times tri-grams were repeated in each text and divided by the length of the test.

Text coherency was also measured by the amount of stopwords present. Text without at least one stopword was filtered out.

**Language filtering**

Non-english texts were filtered out using the langdetect library. This was one of the longest running processes in the pipeline, which is why it occurs after the quality filtering step (and many low-quality docs are removed).

**Deduplication**

**Exact deduplication**

Each document is hashed using MD5 to generate a hashing. Documents are assigned to shards allowing for parallel processing. Within each shard duplicates are dropped keeping the first instance.

**Fuzzy deduplication**

Documents are first split at a paragraph level. Minhash is used in shards to identify paragraphs with jaccard similarity above 0.8. Duplicates after the first are removed and the paragraphs are rolled back into documents.

The duplication process in this case is limited by only comparing documents within shards. Given scalability I would implement a deduplication outside of shards post the initial deduplication.

**Safety**

Phone, emails and TFN’s were masked using regex.

Documents containing toxic content were identified through regex based on ‘List of Dirty Naughty and Obscene and Otherwise Bad Words’: <https://github.com/LDNOOBW/List-of-Dirty-Naughty-Obscene-and-Otherwise-Bad-Words>

This method should be noted as likely to produce a lot of false positives, however the fact that we are dealing with toxic content justifies conservative filtering.

**Tokenisation**

The tokenisation step converts the cleaned text to tokenised exports. It uses Hugging face AutoTokenizer for GPT-2. This model was chosen for its fast processing.

**Scalability**

The pipeline was run on my cpu at home, however the structure was designed with scalability in mind. This is achieved through chunked processing, modular pipeline steps, deduplication sharding, configuratble parameters (e.g minhash permutations) and runtime tracking at each step.

JSON lines output is appendable which reduces the risk of generating many small files. Notably lots of the filtering in this implementation relies on very basic code (e.g regex statements vs llm classifiers).

**Challenges**

Given more time I would implement more extensive data validation. The Validation class is set up in a modular way to implement different validations across each pipeline step. For example I would verify duplication. As well as this I would reconcile dropped records with the total and output record counts. I would also generate more reports showing distribution of n-grams and repetition throughout the steps.

Lots of the steps and pipeline code was coded from scratch, there is room to leverage existing libraries for data tagging and pipeline use.