# 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/?utm_source=chatgpt.com

The code of the pipeline is structured using the following 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

For the purpose of inspectibility, steps that drop records track which records are dropped and can are output to files by the pipeline.

The pipeline performs the following data cleaning steps in their listed order.

**Basic pre-processing**

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

**Relevance filtering**

5. Quality filtering
6. Language filtering

**Deduplication**

7. Exact deduplication
8. Fuzzy deduplication

**Safety**

9. PII masking
10. Toxicity removal

11. Paragraph segmentation
12. Case normalisation

And then finally a tokeniser step takes place on the clean data.

## 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 (reference?), 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, this is still useful training data for an LLM to support coding abilities and reasoning. We want to include only code and language that is natural and cohesive, this means that code without much documentation will be excluded.

#### 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 and the upper was 15,000.

Insert word count distribution and reasoning for lower limit.

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. (Need to insert a distribution for this)

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 (justify this). Minhash is used in shards to identify paragraphs with jaccard similarity above 0.8 (justify this). 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

(justify the use of this)

This method should be noted as likely to produce a lot of false positives, however given we are dealing with toxic content it is better to play it safe and remove them regardless.

## 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 (verify this). Notably lots of the filtering in this implementation relies on very basic code (e.g regex statements vs llm classifiers).