Mainpipe Data Preparation Pipeline

Mary Kolyaei
maryamkolyaie@gmail.com

Abstract

This document presents the design, implementation, and rationale behind the Mainpipe Data Preparation Pipeline task, an end-to-end workflow that converts raw multi-domain text into a cleaned, English-only, deduplicated, PII-masked, tokenised, and sharded dataset ready for LLM pre-training.

1 Overview

The goal of the project was to build an industrial-style data pipeline capable of filtering, normalising, deduplicating, and tokenising raw text data, producing a training-ready dataset.

The pipeline:

- Loads raw JSONL data.
- Performs multi-stage cleaning and quality filtering.
- Detects and masks PII.
- Deduplicates exact and near-duplicate documents.
- Computes quality scores and assigns mixture subsets.
- Tokenises text and exports training-ready JSONL.
- Shards the dataset and generates a manifest.
- Produces metrics, plots, and inspectability outputs.

2 Delivered

1. GitHub repository containing the full data pipeline and a README explaining how to run it. This includes a `requirements.txt` file and a `Dockerfile`. An example end-to-end run is provided in the notebook `Maincode project.ipynb`. Plots and reports are generated in the `plots/` and `reports/` folders. Each step of the pipeline is implemented in a separate `.py` file, and these scripts are independent, meaning they can be run individually once the required input data are available.

2. ***Due to university system restrictions (no Git LFS installation permitted), the large cleaned datasets could not be uploaded to GitHub. These files are available upon request.***

3 Running the Pipeline

- Local Environment:

```
pip install -r requirements.txt
python run_pipeline.py
```

- Docker Execution: A Dockerfile is provided for full containerisation.

4 Pipeline Architecture & Stages

The pipeline is implemented as a sequence of standalone Python scripts. Each stage logs detailed metadata and drop reasons to maximise inspectability.

4.1 Stage 1: Ingestion (ingest.py)

- Input: `mainpipe_data_v1.jsonl`
- Output: `mainpipe_ingested_v1.parquet`
- Generates stable document IDs via SHA-1 hashing.
- Adds timestamps and source metadata.
- Saves a structured Parquet dataset.

4.2 Stage 2: Text Cleaning & Early Filtering (text_clean_and_filter.py)

- Removes empty, null-like, numeric-only, or short texts.
- Normalises Unicode and whitespace.
- Computes character/word stats.
- Performs English-only filtering via language detection.
- Outputs cleaned, dropped, and JSONL versions.

4.3 Stage 3: Deep Cleaning & PII Masking (deep_clean_and_pii.py)

- Removes HTML tags & boilerplate (cookie notices, footers).
- Normalises repeated characters.
- Computes token-level statistics (unique-token ratio, stopword ratio).
- Detects spam-like repetition.
- Masks PII patterns:
  - `<EMAIL>`, `<PHONE>`, `<CREDIT_CARD>`, `<IBAN>`

4.4 Stage 4: Deduplication (duplication.py)

- Exact deduplication using SHA256 over canonicalised text.
- Near-duplicate detection using truncated canonical text (first 500 chars).
- Assigns drop reasons: `exact_duplicate`, `near_duplicate`.

4.5 Stage 5: Quality Scoring & Mixture Assignment (scoring_and_mixture.py)

Each document receives a final **quality_score** in $[0, 1]$ computed from:

- language confidence,
- token count,
- unique-token ratio,
- PII penalty.

Documents are assigned to:

- `high_quality`
- `rest`

4.6 Stage 6: Tokenisation (Tokenisation_JSONL_export.py)

- Uses HuggingFace GPT-2 tokenizer.
- Produces `input_ids`, `attention_mask`, `n_tokens`.
- Filters docs outside 10–2048 tokens.
- Exports training-ready JSONL.

4.7 Stage 7: Sharding (sharding.py)

- Splits tokenised dataset into shard files.
- Generates a `manifest.json` with:
  - shard counts,
  - total documents,
  - total tokens,
  - tokenizer name,
  - file paths.

5 Metrics, Diagnostics & Inspectability

The pipeline logs:

- document counts per stage,
- drop reasons,
- PII statistics,
- language-score distribution,
- quality-score histograms,
- token length distributions,

Running:

```
python generate_metrics_and_plots.py
```

produces plots in `plots/` and machine-readable metrics in `reports/`.

6 Example insights from plots

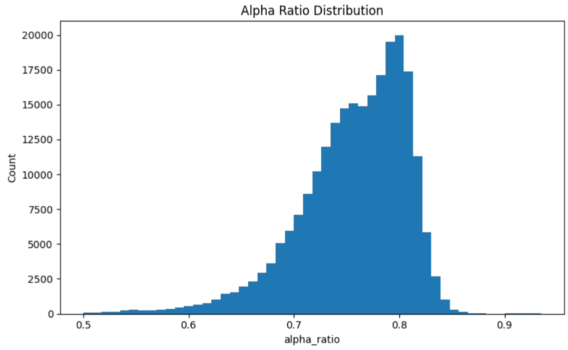

Figure 1: Alpha Ratio Distribution

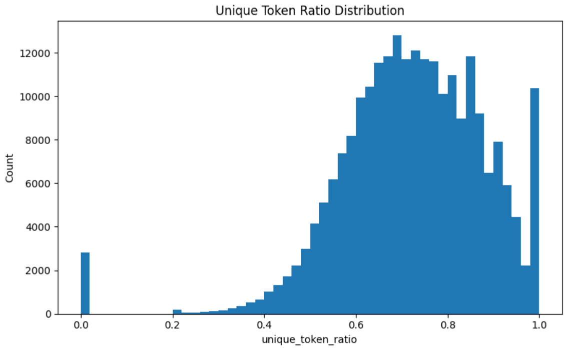

Figure 2: Unique Token Ratio Distribution

Plots were generated to assess the quality and structure of the processed dataset, including distributions for the alpha ratio, character length, token length, unique token ratio, repetition ratio, and stopword ratio. two examples are provided as follow:
Alpha Ratio Distribution: The alpha ratio (fraction of alphabetic characters in a document) shows a strong concentration between 0.70 and 0.80, with a smooth bell-shaped distribution. This indicates that most documents contain predominantly alphabetic characters rather than code, markup, or noisy logs. Very few documents exhibit extremely low alpha ratios, which suggests that earlier cleaning steps (HTML stripping, removal of numeric-only text) were effective. However, the small presence of low-alpha outliers also suggests that further optimisation may be possible
Unique Token Ratio Distribution: Most documents cluster between 0.55 and 0.80 for unique-token ratio, with a small number near 1.0, representing short documents where almost every token is unique (e.g., titles or single sentences). Extremely low ratios are rare, suggesting that repetitive or boilerplate-heavy documents were successfully filtered. Nonetheless, some remaining low-diversity cases may still benefit from additional heuristics or ML-based content classification

7 Distributed Scaling Plan and Future Improvements

The current implementation is lightweight for a take-home task and was executed on a research computing portal with 32 vCPUs and 128 GB RAM using pure Python v 3.10.10. At a large scale, however, LLM data pipelines must operate across clusters, handling billions of documents. The entire workflow changes fundamentally:

Distributed Ingestion

At real scale, ingestion would no longer happen on a single machine. Instead, the entire process would be handled by PySpark running on a distributed compute cluster. Spark workers are able to read thousands of files in parallel directly from cloud storage. All early transformations—things like Unicode normalisation, whitespace cleaning, and initial filtering—run as distributed map operations across many executors. Spark automatically manages batching, memory pressure, and spill-to-disk behaviour, ensuring that even extremely large datasets can be processed efficiently without manual intervention.

Scalable Deduplication

Deduplication becomes a fundamentally different problem once the dataset reaches the scale. Instead of using Pandas grouping, each document would have a SHA256 hash computed using a Spark UDF, and the dataset would then be repartitioned by a prefix of that hash. This ensures that all potential duplicates end up on the same worker, avoiding costly global shuffles. Deduplication then happens locally within each partition. Near-duplicate detection would also be upgraded, moving from simple text truncation to approaches such as MinHash, SimHash, or Locality-Sensitive Hashing (LSH), etc, which are designed for clustering similar documents in large text corpora.

Efficient Language Detection

While Lingua provides accuracy for language identification, it is **too heavy** to use at web scale because it loads multiple statistical models into memory. In a distributed environment, a lighter and more efficient approach would be necessary. A small supervised model—such as a fastText classifier or a distilled transformer—can be deployed to each executor. Inference can then be performed in batches, and the model can be broadcast once per job to minimise overhead. This makes language detection highly efficient even when processing billions of documents.

High-Performance Tokenisation

Tokenisation is usually one of the slowest stages, and Python-based tokenisers cannot keep up at scale. Tokenisers can run inside Spark executors through vectorised Pandas UDFs or via Ray Data pipelines. Once tokenised, the data can be written directly to object storage in parallel, producing balanced shards that feed downstream training jobs without becoming a bottleneck.

Additional Large-Scale Enhancements

- machine-learning–based quality classification to prioritise, cluster, or exclude documents at scale,
- perplexity-based semantic filtering,
- hashed partitions for incremental deduplication,
- GPU-accelerated toxicity/PII detection,
- multi-version manifests for dataset lineage.

8 References

- https://aws.amazon.com/blogs/machine-learning/an-introduction-to-preparing-your-own-dataset-for-llm-training/
- https://huggingface.co/docs/datasets/en/quickstart