Defence, Safety & Security
www.tno.nl
+31 88 866 10 00
info@tno.nl

GPTNL-DEL-4001 – 6 November 2025

System Architecture Document

Data Curation Pipeline

Author(s)	Julio A. de Oliveira Filho [Editor] Andrei Roncea [Editor] Louis Weyland Karim El Assal Leon Helwerda Thomas van Osch
Classification report	TNO Public
Title	TNO Public
Report text	TNO Public
Number of pages	95
Number of appendices	0
Project name	GPT-NL

Contents

1 Introduction 4
1.1 Architectural overview of the GPT-NL Data Curation Pipeline 6
1.2 Scope of the System Architecture Work and Relation to Other Work 8
1.3 How to further read this document 9

2 Architecture and Design of the Data Extraction Phase 10
2.1 Extraction process 11
2.2 Datasource Inbox 14
2.3 Data Source Extractors 16
2.4 Curation Input Format 22
2.5 Code Organization and Data Folders 27

3 Architecture and Design of the Data Curation Phase 32
3.1 Data Curation Process 32
3.2 Comparison of Data Preprocessing Frameworks 37
3.3 Pipeline and Modules Repositories 39
3.4 Curation Stages in Datatrove 44
3.5 Executing a Pipeline 64
3.6 Code Organization and Data Folders 71

4 Appendices 78
4.1 Assessing and Monitoring Energy at the Curation and Training Pipeline 78
4.2 GPT-NL data curation at SURF's HPC *Snellius* 80
4.3 SW Stacks and Framework for GPT-NL 82
4.4 Croissant Format During Curation 86

1 Introduction

The GPT-NL project aims to develop a Dutch-English large language model (LLM) from the ground up to promote technological sovereignty and strengthen the Dutch and broader European LLM ecosystem. Achieving this objective requires a structured systems engineering approach encompassing requirements elicitation, design, implementation, and validation. Beyond the creation of the model itself, sovereignty and community growth depend on transparent dissemination of knowledge about how such a system is built. This document therefore presents the architectural blueprint—both in code and documentation—for the first part of this development phase: the System Architecture of the **Data Curation Pipeline**.

The documentation and systematic management of this technological blueprint are intended to stimulate new research directions and enable future improvements. The **GPT-NL System Architecture** effort serves as the foundation for these goals by providing a coherent, well-documented engineering framework for large-scale model development.

From a general point of view, the system architecture activities provide a structured conceptual model defining the organization, behaviour, and interactions of system components. It offers a high-level view of how hardware, software, data, and processes collaborate to achieve the intended system goals. Through clear specification of components, interfaces, and design principles, the architecture ensures that key system attributes—such as performance, scalability, security, and maintainability—are addressed systematically and in alignment with stakeholder requirements and operational constraints.

Within the GPT-NL team, system architecture plays a coordinating role by providing a shared technical framework that guides design, implementation, and verification across teams. This work, conducted under **Work Package 13 (WP13)**, facilitates communication among engineers, researchers, and developers by defining clear interfaces and dependencies. The architectural team ensures design consistency, manages technical risks, and balances trade-offs among quality attributes. As a result, this document and the associated work contribute to the alignment of strategic objectives and technical execution, promoting system coherence, continuity, and effective integration throughout the development lifecycle.

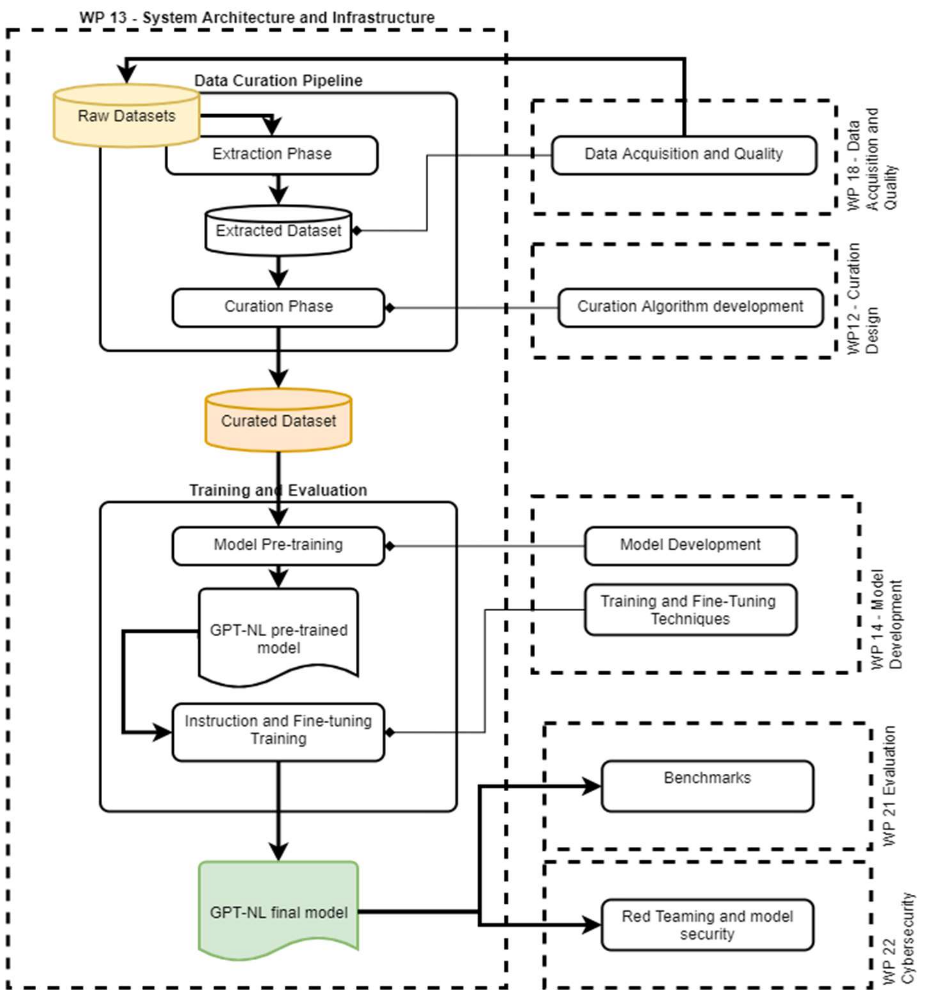

WP 13 - System Architecture and Infrastructure
Data Curation Pipeline
Raw Datasets
Extraction Phase
Extracted Dataset
Curation Phase
Curated Dataset
Training and Evaluation
Model Pre-training
GPT-NL pre-trained model
Instruction and Fine-tuning Training
GPT-NL final model
Data Acquisition and Quality
WP 18 - Data Acquisition and Quality
Curation Algorithm development
WP12 - Curation Design
Model Development
Training and Fine-Tuning Techniques
WP 14 - Model Development
Benchmarks
WP 21 Evaluation
Red Teaming and model security
WP 22 Cybersecurity

Figure 1: Overview main GPT-NL Processes and Working Packages

At a high level, LLM development can be divided into two main components, as depicted in Figure 1: **data curation** and **model training and validation**. These components differ significantly in their technical focus and data processing requirements.

- The **data curation pipeline** encompasses all processes from data acquisition to the creation of a uniform dataset ready for model training. This includes systematic reasoning and documentation of inclusion and exclusion criteria, as well as the production of standardized datasets for both training and public release. The data curation pipeline is sub-divided in two phases: the **data extraction phase** and the **data curation phase**. The whole curation pipeline and its phases are detailed in the next sections. Architectural artifacts from this pipeline include:
    - Software developed for data acquisition, extraction, curation, and dataset deployment.
    - Documentation of third-party software and hardware stacks—such as Data-Trove, PrivateAI, and SURF's Snellius—including configuration details, versioning, and integration procedures.
    - CI/CD frameworks for testing, logging, and evaluating both the platform and resulting datasets.
    - Records of architectural decisions, design rationales, and supporting technical documentation.

    - Security, privacy, and energy monitoring mechanisms for development and operational phases.
    - Final technical reports and communication materials, including this document and supporting white papers.
- The **model training and validation phase** includes data preparation, tokenization, model pre-training, instruction tuning, fine-tuning, and performance evaluation. It results in a standardized and reproducible model package for internal use and community release. Artifacts from this phase include:
    - Software for data mixing, tokenization, model training, fine-tuning, and deployment.
    - Documentation of third-party stacks such as OlMO and the Snellius HPC infrastructure, detailing configurations and integration.
    - CI/CD support for testing and performance tracking.
    - Documentation of design decisions, system rationale, and supporting non-functional design considerations.
    - Security, privacy, and energy monitoring tools.
    - Final deliverables, including technical documentation and dissemination materials.

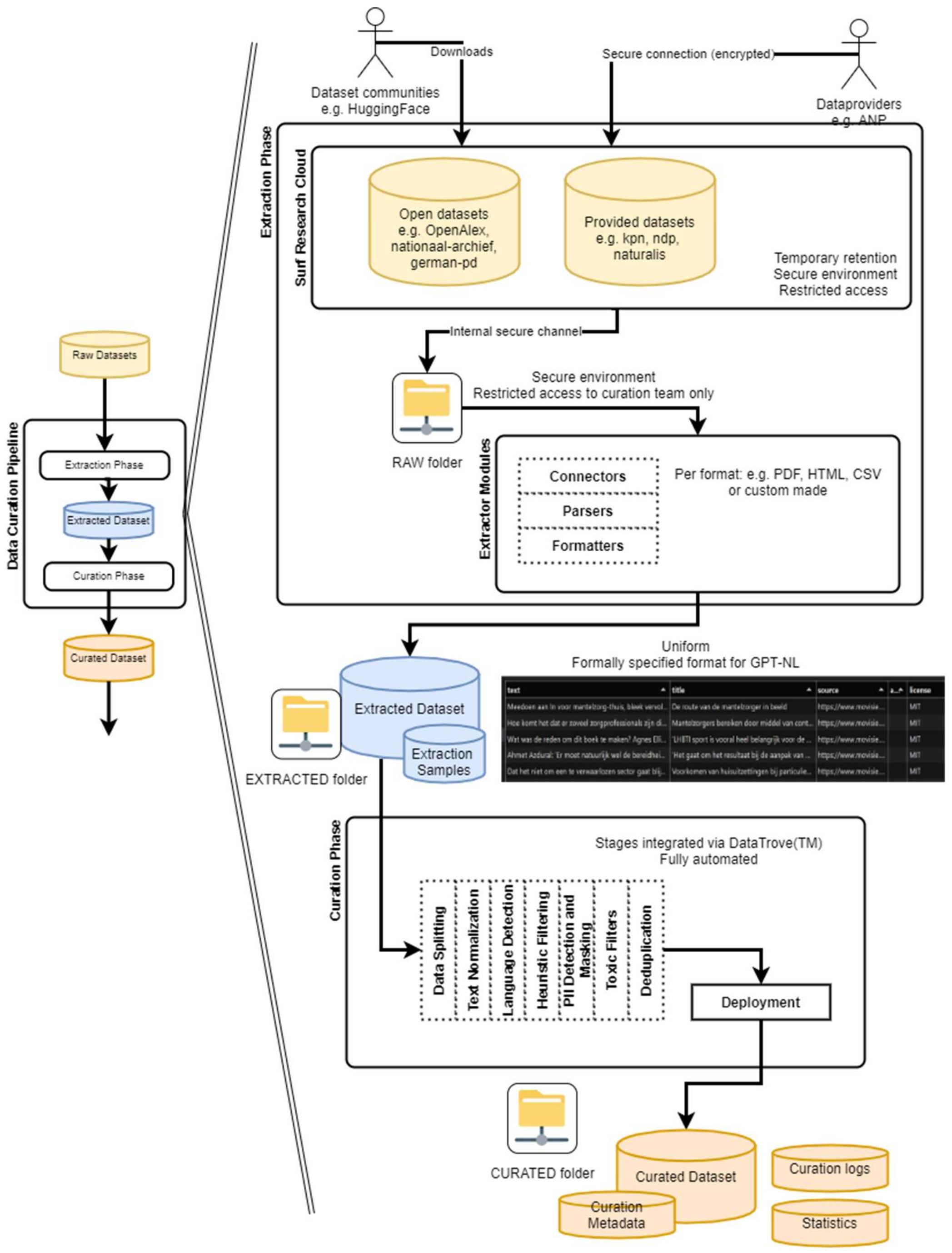
This document, **System Architecture Document – Data Curation Pipeline**, covers the data curation process. Details on model training and validation are presented in the related document: **System Architecture Document – Training And Evaluation Pipeline**[1]**.** As introduction, we present in the following the architectural overview of the Data Curation Pipeline.

1.1 Architectural overview of the GPT-NL Data Curation Pipeline

The data curation pipeline is a structured sequence of processes and software modules responsible for transforming raw data from external sources into a standardized, high-quality dataset ready for model training. A diagram of the processes and components involved are detailed in Figure 2. This transformation occurs in two main stages: (1) receiving and managing incoming raw data and extracting textual content to create an *extracted* dataset, and (2) preprocessing this content—through filtering, normalization, and privacy protection—to produce the curated dataset. The curated dataset forms the input for the second major component of LLM development: model training and validation.

In the GPT-NL project, data is collected from diverse external sources and providers. Each dataset and provider is carefully reviewed to ensure that all materials used for model training are either publicly available and openly licensed or obtained with explicit consent for this purpose. The processes of selecting and approving data sources fall outside the architectural scope and are managed within the working package *WP18 - Data Quality*.

[1] TNO, GPT-NL Project, Report; *GPTNL-DEL-4002-1.0-System Architecture Document – Training Pipeline*, December 2025.



text	title	source	a...	license
Meedoen aan In voor mantelzorg-thuis, bleek vervol...	De route van de mantelzorger in beeld	https://www.movisie...		MIT
Hoe komt het dat er zoveel zorgprofessionals zijn di...	Mantelzorgers bereiken door middel van cont...	https://www.movisie...		MIT
Wat was de reden om dit boek te maken? Agnes Elli...	'LHBTI sport is vooral heel belangrijk voor de ...	https://www.movisie...		MIT
Ahmet Azdural: 'Er moet natuurlijk wel de bereidhei...	'Het gaat om het resultaat bij de aanpak van ...	https://www.movisie...		MIT
Dat het niet om een te verwaarlozen sector gaat bij...	Voorkomen van huisuitzettingen bij particulie...	https://www.movisie...		MIT

Figure 2: Data Curation Pipeline and its main phases and components

Once datasets are approved and agreements with the data provider are in place, the curation pipeline begins by physically collecting the data and integrating it into the GPT-NL system. Each dataset passes through two distinct processing phases: the **data extraction phase** and the **data curation phase**. The pipeline concludes when the data is transformed from its raw form into a standardized, normalized, and annotated dataset — referred to as the *curated* dataset.

The **data extraction phase** provides a secure environment for ingesting data from external providers. It accommodates the consumption of multiple data formats—such as PDF documents, CSV tables, or database exports—and extracts relevant textual content and metadata. Metadata is used to characterize and assess the data (e.g., authorship, timestamps, language, and licensing), informing later filtering and quality assurance processes. The outcome of this phase is a uniform, standardized dataset stored in .parquet format, referred to as the *extracted* dataset. Each raw dataset generates its own extracted counterpart which is submitted to the curation phase in separate. We often use the term “collection” to identify the extracted files provenient from a distinct *raw* dataset. Details of this phase will be discussed in the Data extraction phase section.

The **data curation phase** applies automated processing steps to refine the extracted data. These steps include normalization, filtering, de-biasing, privacy protection, and annotation. Algorithms and techniques developed under the working package *WP12 – Data Curation* are implemented and integrated by the architecture team to ensure scalability and consistency. In some cases, external tools such as PrivateAI are incorporated into the GPT-NL stack. The output of this phase is a *curated* dataset — still stored as .parquet files — enhanced with new metadata, including quality measures, inclusion and exclusion criteria, aggregated statistics, and indicators such as the presence of harmful or sensitive content. Personally identifiable information (PII) belonging to non-public individuals is anonymized or removed. Details of this phase will be discussed in the Data curation phase section.

Throughout both phases, detailed process logs and metadata are collected to ensure transparency, traceability, and processual (KPI) statistics. All software components — whether developed internally or integrated from external sources — are versioned and tracked to guarantee reproducibility. Cybersecurity measures are applied at every stage, including restricted access to raw data, authentication and authorization controls, and secure management of repositories and CI/CD pipelines. For clarity and brevity, many of these supporting processes and architectural mechanisms are referenced in the main sections, but detailed in the **Architectural Support** section.

The next two sections provide a deep dive in the data extraction and the data curation phases.

1.2 Scope of the System Architecture Work and Relation to Other Work

The overview of the processes, tasks, and artefacts related to the architectural work is depicted in Figure 1. The system architecture team collaborates with all other working packages, but closest with **WP12 (Data Curation)**, **WP14 (Model Development)**, and **WP18 (Data Acquisition and Quality)**. While WP12 and WP14 lead algorithmic development—such as the selection of filters, models, and training techniques—WP13 focuses on translating these designs into structured, maintainable, and scalable code. This includes defining clean interfaces between modules, ensuring continuous data processing flows suitable for HPC environments, and addressing non-functional aspects such as security, documentation, and energy efficiency. WP18 is responsible for the processes of contacting data providers and acquiring/creating datasets. They are strongly involved with the architecture team assessing the quality of the data during and after the curation phase.

Besides the close work developed with **WP12, WP14, and WP18**, Figure 1 also depicts cybersecurity and evaluation tasks(**WP21** and **WP22**). Their activities are out of the scope of the architecture team, but their insights and outcomes influence and are influenced by the GPT-NL architecture work. For example, **WP21 (Post-development) Evaluation** and the **Cybersecurity work package (WP22)** operate independently to ensure objective assessment and verification. WP21 evaluates the trained model's performance on key tasks, while the cybersecurity and red-teaming teams assess its resilience and safety. Although separate, these teams collaborate closely with WP13 by consuming its architectural artifacts, interfaces, and documentation, and by providing feedback that informs subsequent development cycles. This interaction ensures a robust, secure, and transparent development process for the GPT-NL system.

1.2.1 Architecture Team

The GPT-NL architecture team has a multidisciplinary composition with SW architects and engineers, open-source specialists, high performance computers architects, ML engineers, and data scientists. Members of TNO and SURF form the team. Acknowledgements for the support of SURF in all the management and proper usage of the Snellius supercomputer.

1.3 How to further read this document

The architecture documentation is organized into two main reports: the **Architecture of the GPT-NL Data Curation Pipeline** and the **Architecture of the GPT-NL Training Pipeline**. The present document covers the data curation part. Readers should start with the introductory section of the data curation pipeline, which provides a general overview of the system's architecture and objectives. The subsequent sections describe the **Data Extraction phase**, including its input format (**GPT Curation Dataset format**), extractor architecture (**Datasource extractors**), and associated (**data folder structure and source code organization**). This part outlines how data is ingested and prepared for curation. The **Data Curation phase** then details the data preparation processing pipeline, introducing the **DataTrove framework**, modularization principles, execution methodology, and **curation stages**. The section concludes with information on **code organization**, deployment at the **Snellius A100 Cluster**, and procedures for **assessing and monitoring the curation pipeline**. Supplementary analyses are provided for additional information on the used SW stacks in **Appendix A** and formal specifications (in croissant format) for the projects' datasets **Appendix B**.

2 Architecture and Design of the Data Extraction Phase

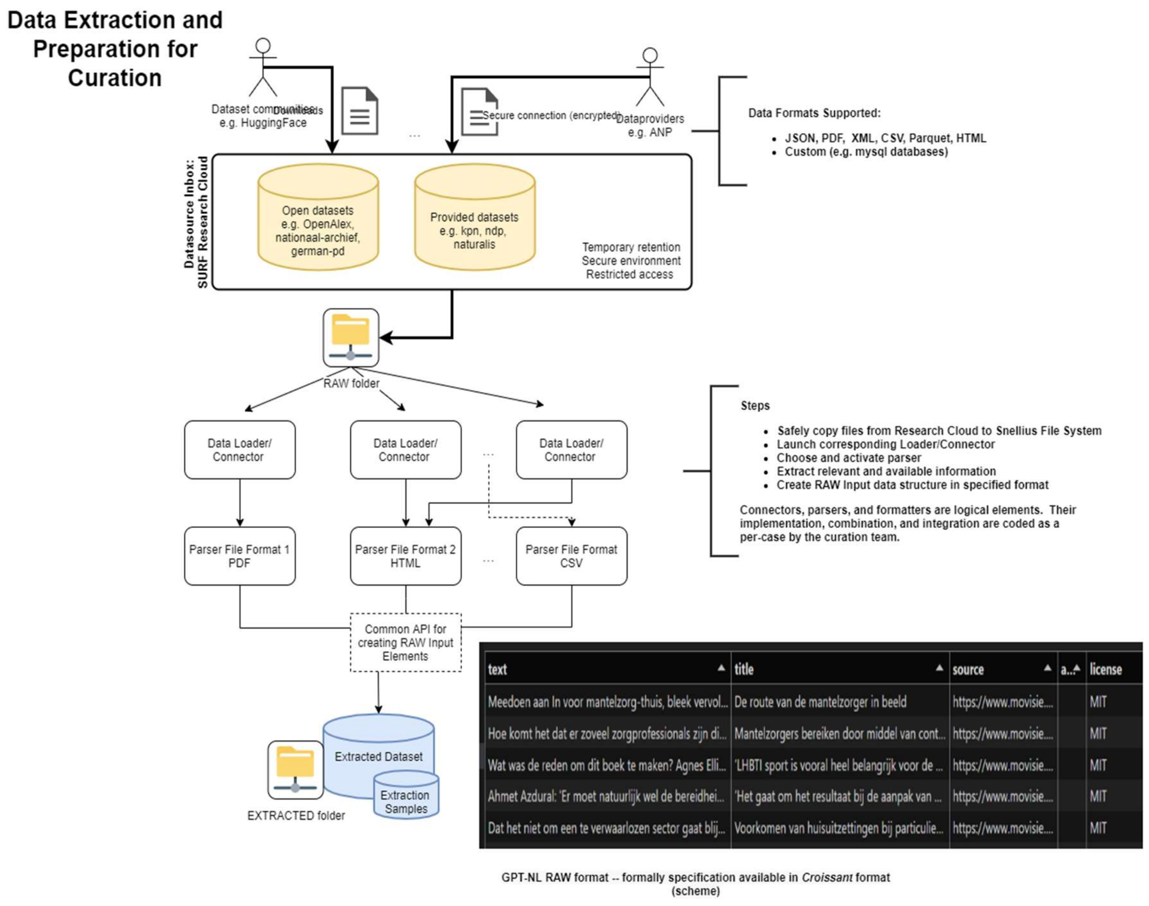
The GPT-NL project collects data from a wide range of sources. Some datasets come from publicly available repositories in the machine learning communities such as HuggingFace—while others are provided by private data providers with explicit permission for use in GPT-NL training. This diversity introduces a key challenge: raw data comes in diverse data structures, varying labeling conventions and file formats. In their original state, these datasets are not suitable for automated processing during the curation or training phases. Within the GPT-NL project, this unprocessed form is referred to as the **RAW** format.

In many cases—particularly with private data providers—raw data is subject to copyright restrictions and must be protected against unauthorized access or leakage during curation. Additionally, it is in the project's interest to collect metadata and statistics about the raw data, such as authorship (for attribution), data sources, timestamps, version information, and preliminary size estimates (e.g., character or token counts).

The goal of the ***data extraction phase*** *is to securely receive and convert raw data from external sources into a structured, homogeneous format—the* ***EXTRACTED dataset****.*

Before the curation phase can begin, all incoming data must first be transformed into this **EXTRACTED** format. The extraction process ensures that the curation pipeline starts with a standardized input, preventing downstream components from having to handle multiple formats or inconsistent structures. It also helps decouple the original data from subsequent processing, ensuring that only the relevant textual content needed for curation and training is retained. This selective extraction further safeguards the rights and confidentiality of data providers.

The **extraction phase** is represented in the first part of the data curation diagram. Its inputs consist of external data sources—typically a collection of files, databases, or online repositories. These files are securely transferred to an internal protected environment known as the *Datasource Inbox*, which is described in greater detail in the Datasources section.


Data Extraction and Preparation for Curation
Dataset communities e.g. HuggingFace
Downloads
Secure connection (encrypted)
Dataproviders e.g. ANP
Data Formats Supported:
JSON, PDF, XML, CSV, Parquet, HTML
Custom (e.g. mysql databases)
Datasource Inbox: SURF Research Cloud
Open datasets e.g. OpenAlex, nationaal-archief, german-pd
Provided datasets e.g. kpn, ndp, naturalis
Temporary retention
Secure environment
Restricted access
RAW folder
Data Loader/ Connector
Data Loader/ Connector
Data Loader/ Connector
Steps
Safely copy files from Research Cloud to Snellius File System
Launch corresponding Loader/Connector
Choose and activate parser
Extract relevant and available information
Create RAW Input data structure in specified format
Connectors, parsers, and formatters are logical elements. Their implementation, combination, and integration are coded as a per-case by the curation team.
Parser File Format 1 PDF
Parser File Format 2 HTML
Parser File Format CSV
Common API for creating RAW Input Elements
Extracted Dataset
Extraction Samples
EXTRACTED folder
GPT-NL RAW format -- formally specification available in *Croissant* format (scheme)

text	title	source	a...	license
Meedoen aan In voor mantelzorg-thuis, bleek vervol...	De route van de mantelzorger in beeld	https://www.movisie....		MIT
Hoe komt het dat er zoveel zorgprofessionals zijn di...	Mantelzorgers bereiken door middel van cont...	https://www.movisie....		MIT
Wat was de reden om dit boek te maken? Agnes Elli...	'LHBTI sport is vooral heel belangrijk voor de ...	https://www.movisie....		MIT
Ahmet Azdural: 'Er moet natuurlijk wel de bereidhei...	'Het gaat om het resultaat bij de aanpak van ...	https://www.movisie....		MIT
Dat het niet om een te verwaarlozen sector gaat blij...	Voorkomen van huisuitzettingen bij particulie...	https://www.movisie....		MIT

Figure 3: Overview Data Extraction phase

2.1 Extraction process

The data extraction process consists of several key steps that work together to transform raw data into structured Parquet files. The first two steps occur in the Datasource Inbox and were discussed in detail in the previous section. The extraction activity itself is performed in the last two steps shown in the diagram:

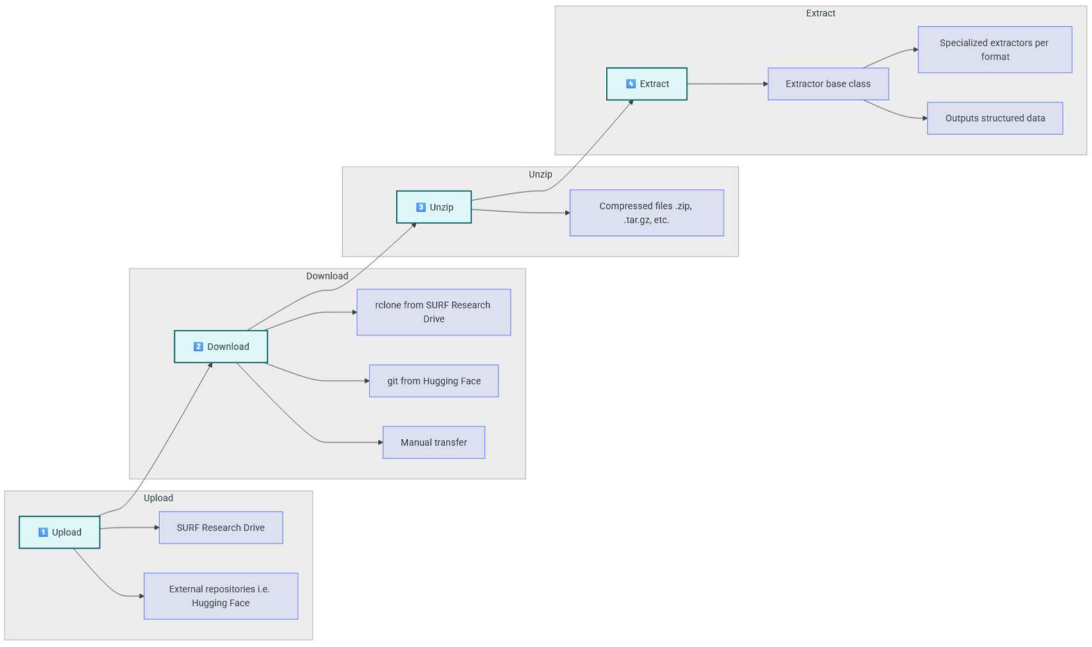


Figure 4: Tasks within the extraction process

1. **Upload** Most datasets are provided by data contributors through the SURF Research Drive, while others are already available to download from ML community websites, such as Hugging Face. The procedures and policies governing data upload — including safe transfer, access control, and privacy compliance, and supported data formats — are discussed in the Datasource Inbox section.

2. **Download** Datasets are retrieved using one of three methods, depending on their source:

   - From the SURF Research Drive to the Snellius cluster using the rclone utility, which enables efficient, parallelized, and encrypted file transfers. This is performed using automated HPC jobs whenever possible.
   - From Hugging Face using git, allowing version-controlled synchronization of datasets.
   - In a few cases, datasets are transferred manually from the SURF Research Drive to Snellius (e.g., when access restrictions or network issues prevent automated transfers).

   The target directory structure and conventions used for storing downloaded data are detailed in the Code Organization and Data Folders section. These steps include data checksum validation and hash verification to ensure data integrity after transfer.

3. **Unzip** Most datasets are provided in compressed formats such as .zip, .tar.gz, or .xz. These are decompressed using standard Linux utilities (e.g., unzip, tar, or gzip). In some cases, decompression is integrated into the extraction process itself to streamline workflow execution.

4. **Extract** The core component of the GPT-NL data pipeline is the **extraction framework**, built around a modular architecture. It comprises a base Extractor class and multiple specialized extractor implementations for different data formats and schemas.

   Each extractor:

   - Connects and Reads the RAW input files – Connector module.

   - Parses them according to the dataset-specific structure – Parser module.
   - Outputs a normalized and structured representation of the data – Formatter module.

  The unified data format produced by these extractors is described in the Curation Input Format Section, and the storage location of EXTRACTED datasets is documented in the Code Organization and Data Folders. The Extractor class provides a common interface that integrates **connectors**, **parsers**, and **formatters**, ensuring consistent handling of diverse data sources.

Once data is available in the *Datasource Inbox*, it is processed using specialized software modules known as *extractors*. Extractors consist of a sequence of **connectors**, **parsers**, and **formatters**, which together transform heterogeneous raw data into a consistent structure suitable for the next processing phase. The GPT-NL project implements a flexible software stack that supports the design, configuration, and composition of these modules, allowing them to handle the wide variety of raw data formats encountered across different sources. Extractor modules were implemented for each one of the datasets and are available in the data extraction code repository.

- **Connectors** manage access to and retrieval of data from their original source. Their implementation depends on the source type and access method. For example, when dealing with web-based content, a connector functions as a *scraper*, capable of using network protocols to retrieve HTML data. In GPT-NL, most connectors operate within the secure Snellius environment and provide an abstraction layer for accessing files, databases, or API endpoints. The output of a connector is an interface exposing the source content—typically in textual or tabular form—to the next processing stage. There is no single, universal interface; connectors may consist of custom-built modules or widely used open-source libraries. For instance, the pypdf package is commonly used to process .pdf documents.

- **Parsers** interpret and extract meaningful information from the content provided by connectors. Their key role is to understand the data format and isolate relevant elements, primarily raw text, and associated metadata. This metadata can include attributes such as authorship, section titles, and publication dates. If such information is unavailable, parsers generate default values to maintain consistency. In some cases, parsers may apply light filtering to exclude irrelevant content—though this is outside the formal scope of GPT-NL's curation process. Parsers interact with connectors through tailored interfaces that allow them to process content at the level of files or file sections. Their output is a structured data representation ready to be processed by the formatter modules.

- **Formatters** take the structured output from the parsers and organize it into a standardized dataset format designed for the data curation pipeline. This step ensures that all extracted data—regardless of its origin or initial structure—follows a uniform schema. By enforcing this consistency, the curation pipeline can operate efficiently and reliably, without needing to manage the complexities or idiosyncrasies of individual datasource formats.

The internal operation of extractors is described in detail in the Datasource Extractors section. The result of this process is a collection of structured data records that conform to the **EXTRACTED** format specification and are stored as .parquet files—sometimes referred to as the *GPT-NL input format*. The structure and content of this format are formally defined in the Curation Input Format section.

In fact, at the end of the extraction a sampling process takes place using the extracted set as input. This sampling process generates a random subset— smaller than the original dataset. This sample is submitted to curation cycles and the effects of the curation pipeline (discussed in the next section) can be evaluated by the data quality team. The data quality decides on eventual modifications on the curation parameters, preserving most of the quality data. These new parameter settings are put in place and documented by the curation team, which is now ready to move on to the curation phase. This sampling process is discussed in the Preparation for the Curation phase.

In the following sections we dive into the architectural details, requirements, and processes for each of the extraction stages and components.

2.2 Datasource Inbox

The GPT-NL project receives data from a variety of partners and external data sources. Many datasets are subject to intellectual property (IP) rights and copyright restrictions, and their use is authorized solely for project purposes. To comply with these legal and contractual obligations—established under **Work Package 18 (Data Acquisition and Quality)**—the project must ensure a secure environment for receiving, storing, and managing partner-contributed data. This safeguards both the proprietary value of the data and the privacy of contributing entities.

This section describes the environment used in GPT-NL to securely receive data from partners, referred to as the **Datasource Inbox**.

The Datasource Inbox is hosted at **SURF** and provides a direct, secure connection to the Snellius research cluster, where subsequent data processing and curation take place. SURF and Snellius were selected as the primary data handling platforms due to their robust security controls, reliable data transfer mechanisms, large-scale storage capacity, and researcher-oriented usability. These characteristics ensure that sensitive datasets can be ingested and processed in full compliance with project and partner requirements.

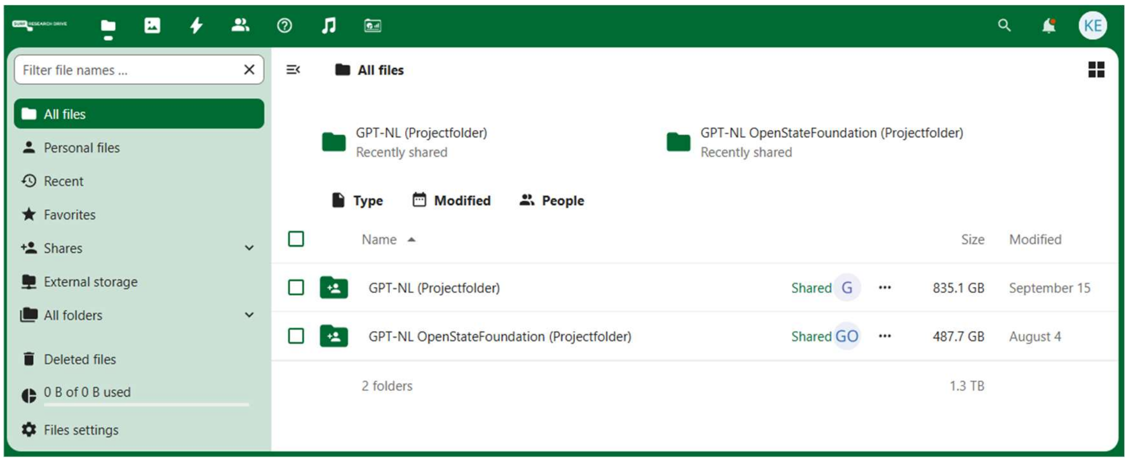
Other project datasources are public ML communities – such as HuggingFace – from which we download publicly available datasets like Common Corpus into the Datasource Inbox. We discuss briefly these two options in the following.

2.2.1 SURF Research Drive

The SURF Research Drive is a secure, cloud-based storage service designed specifically for researchers, students and information professionals, to store, share, and collaborate on data. It offers scalable storage, making it ideal for managing large datasets commonly used in research. The platform ensures that sensitive and valuable research data is safeguarded through advanced security protocols. Researchers can collaborate seamlessly across institutions, making data sharing more efficient while maintaining control over data access. Additionally, it integrates well with other research tools, enhancing workflow efficiency and ensuring compliance with data management regulations.

GPT-NL uses the research drive as the designated storage location for data providers to upload their datasets. Using rclone, the data was easily transferred to Snellius. rclone was set up by following SURF instructions, which uses a WebDAV interface for the Research Drive. Data was transferred from the Research Drive to Snellius using a command like the one below, executed in a Snellius job to allow long-running transfers.

```
rclone copy \
  "RD:/GPT-NL (Projectfolder)/Instituut voor NL Taal/CorpusGysseling_1.0.zip" \
  /projects/0/prjs0986/wp12/raw/instituut-voor-nl-taal
```



Name			Size	Modified
GPT-NL (Projectfolder)	Shared G	···	835.1 GB	September 15
GPT-NL OpenStateFoundation (Projectfolder)	Shared GO	···	487.7 GB	August 4
2 folders			1.3 TB	

Figure 5: Screenshot of the SURF Research Drive

2.2.2 Hugging Face

Hugging Face is a platform that provides a wide range of tools and resources for natural language processing (NLP) and machine learning. It offers an extensive collection of pre-trained models, datasets, and libraries that facilitate the development and deployment of NLP applications. Hugging Face is known for its user-friendly interface and active community, making it easier for researchers and developers to collaborate and share their work.

GPT-NL uses Hugging Face as a source for several public datasets. The datasets were downloaded using git-lfs. Example commands to download a dataset from Hugging Face look like this:

```
git lfs install
git clone https://huggingface.co/datasets/coastalcph/multi_eurlex /projects/0/prjs0986/wp12/raw/multi-eu
rlex
```

2.2.3 What happens to the data in the Datasource Inbox?

In accordance with the folder structure defined in the data folder structure documentation, data received in the **Datasource Inbox** are automatically and securely transferred to the following directory within the GPT-NL project space on Snellius:

/projects/0/prjs0986/wp12/raw/

This automated transfer ensures that all ingested datasets are consistently stored in the designated **raw data repository**, from which the extraction framework can directly access and process them.

Access to this directory is strictly controlled and limited to authorized members of relevant work packages. Furthermore, the retention of partner-contributed data within this folder is **time-bound**, in compliance with contractual and data management requirements.

For details on the file formats used by different data sources and the corresponding extractor implementations, refer to the Data Source Extractors Section.

2.3 Data Source Extractors

Once a dataset is received in the inbox and transferred to the appropriate folder within the local Snellius curation directory, the GPT-NL data extraction system initiates a modular processing framework. This framework transforms the data sources into a unified tabular structure and stores them in Parquet format. The resulting dataset serves as the foundational input for subsequent curation operations.

The architecture of the data source extractors employs a plugin-based design pattern, allowing new data sources to be integrated easily while ensuring consistency and reliability across the system.

Modules used at each extractor are versioned (in the repository) and automatically evaluated before application using a CI/CD pipeline. That is to ensure reproducibility of the data extraction in the future. Logging is produced to identify errors and warning messages during the extraction and help in the debug of modules. However, if an extraction process is successful and no errors occurred, logged files are discarded, as they do not contain information about the process itself. Example, there is no performance measurements at this stage. The reason for that is that the extraction phase is likely to be performed very few times and is not particularly computing intensive.

Extractors also determine the granularity or size of the text chunks extracted. For some datasources, a chunk of extracted text corresponds to a paragraph. In other sources, a chunk of extracted text is the full content of a document, e.g., all chapters of a book in one long textual stream. That creates data entries in the extracted data that may vary a lot on their sizes. This is kept like that by design and eventually adjusted later in the process only when necessary. In other words, the extraction process does not automatically chunk the extracted information.

In certain cases, the *unzip* and *extract* stages are combined into a single streaming process. This approach is used for very large datasets containing numerous files, as the Snellius cluster imposes limits on the number of files (inodes) per user. The streaming workflow allows processing data on the fly, thereby avoiding storage exhaustion and improving efficiency.

For most datasets, the *download* and *unzip* operations are automated and defined in .job scripts. When these steps are not present, data were downloaded or decompressed manually, or the dataset size allowed direct command-line handling. The *extract* operation is consistently implemented in extract[-*].job files.

The following section provides a detailed analysis of the extraction framework, its architectural components, and implementation details.

2.3.1 Extraction framework - Core Components

2.3.1.1 Base Extractor Framework (extractor.py)

The heart of the system is the abstract Extractor class that defines the common interface and workflow for all data extraction operations. This has a few features:

- Set the input and output directories.
- Set the UID: the unique identifier for the dataset extraction run. If not set manually, a random one is generated. Manual setting is useful for parallel processing.

Custom derivations of the Extractor class defines behaviour. The process will be explained later in this section. Extractors return a DataFrame structure with the obligatory fields (e.g., text, id) per row, and eventually other fields with optional information. That is dependent on the availability of this information in the current dataset. When ingesting tabular format (e.g., from open datasets in HuggingFace), it is sufficient to indicate a relationship between the fields of the dataframe and the corresponding fields in the source table (e.g., ‘info’ -> ‘text’). For data extractors for which it is not easy to establish this relationship, a DataFrame should override the get_data_docs() method, which returns an iterable of dictionaries, each containing a text field (obligatory) and optional metadata fields.

2.3.1.2 Extractor Implementations

The system includes specialized extractors for various data formats and sources, each inheriting from the base Extractor class. The format-based ones are listed here:

Class	Data format	Datasets extracted
CodebookExtractor	Codebook	CenterData (txt part)
CSVExtractor	CSV	Movisie, BNR and Woogle
JsonlExtractor	JSON Lines	Common Crawl, DPC and KPN
JsonlZipstreamExtractor	Compressed JSON Lines	ANP
OrtExtractor	Orthographic transcription (*.ort)	JASMIN 1.0 and CGNAnn 2.0.3
ParquetExtractor	Parquet	Belgian Journal, Common Corpus, Danish PD, English PD, French PD Books, Germn PD Newspapers, German PD, LoC PD Books, Spanish PD Books, Spanish PD Newspapers, Swedish PD, TEDEUTenders, and YouTube-Commons
PdfExtractor	PDF	Tweede Kamer, Auditdienst Rijk, HBO and CenterData (pdf part)
SqliteExtractor	SQLite databases	NDP print and NDP web
XmlExtractor	Generic XML	DAESO 1.0 and CorpusMiddelnederlands 1.0
XmlaltoExtractor	ALTO XML	Noord-Hollands Archief and Zeeuws Archief
XmlpageExtractor	PAGE XML	Utrechts Archief

These are the specialized extractors for specific datasets:

Class	Datasets extracted
AmericanStoriesExtractor	American Stories
CorpuscoderingExtractor	CorpusGysseling 1.0
CulturaxExtractor	Cultura X (NL part)

DANSExtractor	DANS
DNBExtractor	DNB
EuropeanParliamentExtractor	European Parliament
FryskeAkademyExtractor	Fryske Akademy
ICTRechtExtractor	ICTRecht
KBExtractor	Koninklijke Bibliotheek
KBOpenKrantenExtractor	Koninklijke Bibliotheek Open Kranten
LassyLargeExtractor	LASSY Large 7.0
MultiEurlexExtractor	MultiEURLEX
NationaalArchiefExtractor	Nationaal Archief
NaturalisExtractor	Naturalis
NTVGExtractor	Nederlands Tijdschrift voor Geneeskunde
OfficieleBekendmakingenExtractor	Officiële Bekendmakingen
OpenraadsinformatieExtractor	Open Raadsinformatie
PblExtractor	Planbureau voor de Leefomgeving
RechtspraakExtractor	Rechtspraak
WaarbenjijnuExtractor	Waarbenjij.nu
WikidataExtractor	Wikidata
WikiwijsExtractor	Wikiwijs

2.3.1.2.1 Adding New Extractors

Adding or customizing an extractor is simple. We show the main steps involved with a simplified example of the PDF extractor below. Code that outputs the progress is omitted. We provide additional comments in this example code explaining intermediate steps.

First, we inherit from the base Extractor class and implement the get_data_docs() method or override get_df() for DataFrame-based processing.

```
# file: gptnl_data_extraction/pdf_extractor/pdf_extractor.py
import gc
from collections.abc import Generator
from pathlib import Path
from gptnl_data_extraction.extractor import DataDoc, Extractor
from gptnl_data_extraction.utils import read_pdf

class PdfExtractor(Extractor):
  # Guarantees we are going to connect with .pdf files only.
  def filter_input_files(self, files: list[Path]) -> list[Path]:
    return [path for path in files if path.match("**/*.pdf")]

  def get_data_docs(self, files: list[Path]) -> Generator[list[DataDoc], None, None]:
    datadocs = []
    for file in files:
      # read_pdf is a wrapper for the pypdf python package in this case,
      # already extracting only the textual information of a pdf.
      # This can obviously only be used for typical, well-formed pdfs.
      text = read_pdf(file)
      if len(text) == 0:
          continue

      # In this case, we decide to make the title equal to its filename.
      # Details on this implementation are set in `main.py`.
      title = self.filename_to_title_mapper(file)

      # Format into DataDoc so it can be processed.
      # For this extractor, only text and title are extracted from the sources.
      datadocs.append(
          DataDoc(
              title=title,
              text=text,
          )
      )

      # Sometimes, we control the size of extracted elements
      # Avoiding saves at each entry make it more efficient
      # But memory overhauls must be avoided.
      if len(datadocs) >= 2000: # Not too many, to avoid memory issues.
          yield datadocs
          # Clear memory.
          del datadocs[:]
          gc.collect()
          datadocs = []
    yield datadocs
```

Note that this approach still uses the concepts of connecting, parsing, and formatting as explained in the more generic architectural view. However, the composition of these activities can be freely designed on case-by-case within this method. This procedure gives more freedom to the extraction team to deal with the diversity of formats as they come.

Second, we write unit tests for the new extractor to confirm it works. Unit tests are important in this context to automate the validation of the extraction phase and future reproducibility. A couple of documents randomly sampled from the datasource are used to validate the operation.

```
# file: gptnl_data_extraction/pdf_extractor/tests/test_pdf_extractor.py
from pathlib import Path
import pandas as pd
from gptnl_data_extraction.extractor import DatasetMetadata
from gptnl_data_extraction.pdf_extractor import PdfExtractor

this_dir = Path(__file__).resolve().parent

def test_pdf_extractor():
    input_dir = this_dir
    output_path = this_dir / "test_output.parquet"
    dataset_name = "Example"
    dataset_url = "https://example.com"
    license = "MIT"
    title0 = "0d54f080-dca2-4c89-a68d-705889424127.pdf"
    title1 = "34af8277-28b4-41c9-a998-8ff0ba335820.pdf"
    content_prelude0 = "Tweede Kamer der Stat"
    content_prelude1 = "16% Hernieuwbare ener"

    job = PdfExtractor(
        "extraction-test",
        DatasetMetadata(dataset_name, dataset_url, license),
    ).set_io(input_dir, output_path)
```

```
job.process()

df = pd.read_parquet(output_path)

assert df["dataset_name"][0] == df["dataset_name"][1] == dataset_name
assert df["dataset_url"][0] == df["dataset_url"][1] == dataset_url
assert df["dataset_license"][0] == df["dataset_license"][1] == license
assert df["license"][0] == df["license"][1] == license
assert df["title"][0] == title0
assert df["title"][1] == title1
assert df["text"][0][: len(content_prelude0)] == content_prelude0
assert df["text"][1][: len(content_prelude1)] == content_prelude1
```

Third, we configure dataset metadata and any format-specific parameters and add the job configuration to main.py registry. The registry ensures the whole process can be launched from a central place in using a single mechanism. Once an extractor is registered, the team can launch it by its name, e.g., below *tweedekamer*.

```
 # file: gptnl_data_extraction/main.py
...
PdfExtractor(
        "tweedekamer",
        DatasetMetadata(
            name="Tweede Kamer",
            url="https://opendata.tweedekamer.nl/",
            license="public-domain",
        ),
    ).set_filename_to_title_mapper(lambda file: file.stem) # Just take the filename without extension.
...
```

Fourth, we create corresponding SLURM .job files for cluster execution. Parameters are set depending on dataset sizes, expected computing availability, etc. Note that the mechanism on the jobs is always the same: running the extract module. The module receives as a parameter the specific extractor (e.g., tweedekamer) to be used.

```
#!/bin/bash
#SBATCH --job-name extr-tweedekamer
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --time=2-00:00:00
#SBATCH --output=output/slurm-%j-extract-tweedekamer.out
#SBATCH --mem=5GB

./init_snellius.sh
poetry run extract tweedekamer \
  --input        /projects/0/prjs0986/wp12/raw/tweedekamer \
  --output-path  /projects/0/prjs0986/wp12/extracted/tweedekamer/tweedekamer.parquet
```

2.3.1.2.2 *Running an Extractor*

You can now run an extractor from the command line using the extract script. The general usage is:

```
> poetry run extract -i <input> -o <output_path> <job_name>
```

The job_name corresponds to the name of the dataset as defined in main.py. The -i and -o flags specify the input and output directories, respectively.

2.3.1.3 Jobs

Jobs ensure the steps described above are run on the Snellius cluster. Every extractor has one or more job files: if it is a file type extractor, there will be job files for each dataset. If it is a dataset extractor, there will be a single job file for the dataset (per step).

Per dataset, you can run the entire extraction process by starting these jobs in order (examples from the CenterData dataset (pdf part)):

5. download[-*].job (if needed)

```
#!/bin/bash
#SBATCH --job-name dl-centerdata
#SBATCH --time=1:00:00
#SBATCH --output=output/slurm-%j-download-centerdata.out
#SBATCH --cpus-per-task=1
#SBATCH --ntasks=1
#SBATCH --mem=5GB

rclone copy \
  "RD:GPT-NL (Projectfolder)/CenterData" \
  /projects/0/prjs0986/wp12/raw/centerdata
```

6. unzip[-*].job (if needed)

```
#!/bin/bash
#SBATCH --job-name unzip-centerdata
#SBATCH --time=00:05:00
#SBATCH --output=output/slurm-%j-unzip-centerdata.out
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --mem=0.5GB
#SBATCH --partition=rome

DIR=/projects/0/prjs0986/wp12/raw/centerdata
FILES=(
  "${DIR}/Centerdata_Codebook_1.tar"
  "${DIR}/Centerdata_Codebook_2.tar"
  "${DIR}/Centerdata_Publications.tar"
)

for FILE in "${FILES[@]}"; do
  tar -xvf "$FILE" -C "$DIR/"
done
```

7. extract[-*].job

```
#!/bin/bash
#SBATCH --job-name extr-centerdata
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --time=01:30:00
#SBATCH --output=output/slurm-%j-extract-centerdata.out
#SBATCH --mem=5GB
#SBATCH --cpus-per-task=1

./init_snellius.sh
poetry run extract centerdata-pdf \
  --input        /projects/0/prjs0986/wp12/raw/centerdata \
  --output-path  /projects/0/prjs0986/wp12/extracted/centerdata/centerdata-pdf.parquet
```

Some extraction jobs are designed to be run in parallel. These are array jobs and are clearly described in the job file itself. Sometimes individual, parallel extraction jobs failed due to memory limits, so run_parallel_job.sh is provided to help re-run individual array jobs that failed. This retains the UID that is associated with one extraction run.

2.3.1.4 Data Processing Utilities

Several utility functions support data processing tasks, including:

- Previewing a parquet file.

```
poetry run preview <input_file> [<num_entries>]
```

- Combining multiple parquet files into one. If you use * in file_mask, surround file_mask with double quotes to avoid shell expansion.

```
poetry run combine <file_mask> <output_file>
```

- Reducing a parquet file to only a few entries.

```
poetry run reduce <input_file> <output_file> <num_entries>
```

2.3.2 Additional information

Details of the extraction repository organization and its operation can be found in the Data folder structure and Code Organization section in this document.

2.4 Curation Input Format

The output of the extraction phase is a dataset with a standardized structure and storage format. This section specifies the formal definition and role of this dataset within the GPT-NL data processing workflow.

Definition

> ***EXTRACTED dataset (GPT-NL Input Dataset):*** *A harmonized data product generated by the extraction framework. It consolidates heterogeneous external datasets into a uniform tabular structure and standardized storage format suitable for subsequent curation and analysis.*

2.4.1 Role in the System

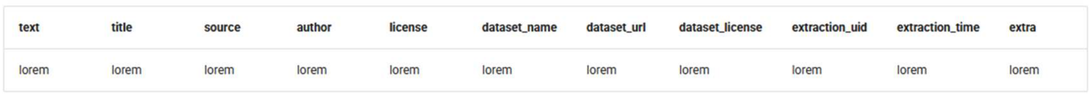
- **End Point of the Extraction Phase:** The EXTRACTED dataset constitutes the final output of the data extraction phase. It represents the fully processed and normalized version of all input datasets ingested from external sources.

- **Starting Point of the Curation Phase:** The same dataset serves as the primary input to the data curation phase, providing a consistent basis for further quality assurance, enrichment, and transformation steps.

The EXTRACTED dataset defines the standard input format for all downstream components in the GPT-NL pipeline. This structure is also employed for the sample datasets described in the preceding section, ensuring alignment and interoperability across the extraction and curation phases. It is also worth to mention that the final curation format (after the curation phase) is just an extension of the data structure defined here – this extension includes further annotation aggregated during the curation process.

2.4.2 Extracted Data Structure and Storage Format

The data extracted from source datasets during the data extraction phase are organized into a standardized tabular structure, as shown below:



text	title	source	author	license	dataset_name	dataset_url	dataset_license	extraction_uid	extraction_time	extra
lorem	lorem	lorem	lorem	lorem	lorem	lorem	lorem	lorem	lorem	lorem

Each table adhering to this schema is stored in the GPT-NL **extracted** database in the .parquet format. All extracted datasets share this internal structure, ensuring interoperability and consistency across data sources.

2.4.2.1 Rationale for Using the Parquet Format

The .parquet format was selected due to its efficiency, flexibility, and compatibility with large-scale data processing systems:

- **Columnar Storage Efficiency:** Parquet stores data by columns rather than by rows, enabling more effective compression and faster access to specific attributes. This is particularly advantageous for analytical and distributed workloads, such as those executed during the data curation phase.

- **Compression and Encoding:** Parquet supports several built-in compression and encoding algorithms (e.g., *Snappy*, *GZIP*), which reduce storage footprint while improving read and write performance.

- **Self-Describing Schema:** Each Parquet file includes an embedded schema definition, facilitating data validation, discovery, and schema evolution. This feature simplifies versioning and maintenance when dataset structures evolve over time.

- **Optimized for Distributed Processing:** Parquet integrates natively with distributed data processing frameworks such as **Apache Spark**, **Hive**, and **Presto**, supporting scalable, parallelized data transformations and queries across large datasets.

- **Cross-Platform and Language Interoperability:** The format is widely supported across programming environments, including **Python**, **Java**, and **R**, ensuring compatibility with diverse analytical and machine learning workflows.

Overall, the use of Parquet provides a robust and scalable foundation for managing extracted datasets in GPT-NL, balancing storage efficiency, accessibility, and long-term maintainability.

2.4.3 Explanation of data structure fields

In the following, we describe the format fields in detail:

- text[sc:String] : Content from the document parsed (text). That is a textual information relevant for the work. No limitations on the character set used nor the size of this field. It should be simply a textual extraction one datasource document. The curation pipeline and its steps will, when necessary, normalize this text to a well defined format. This is discussed in the curation pipeline phase, not here. This field is obligatory and must be non-empty.

- title[sc:String] : The title of the document extracted. This field is obligatory, but it can be left as an empty string if the title is not known or attributed. In many cases, the title is an urn that uniquely identifies the document where the text was extracted.
- source[sc:String] : an identifier for the source where the document is found in string format. In some cases, this source is recorded as a url. Sources may contain several documents. Field is obligatory, and its value is always non-empty.
- author[sc:String] : The author(s) of the document. Field may be an empty string, if author not know.
- license[sc:String] : The license attributed to the document. Document licenses can be different from the source license.
- dataset_name[sc:String] : The name or title of the external dataset (or source) containing the document. This field identifies the dataset within the GPT-NL raw set (original sources). It is never empty.
- dataset-url[sc:String] : The source's URL from which the dataset (not the document) was obtained. This helps trace the origin of the data. It can be empty if the content is not known.
- dataset_license[sc:String] : The license under which the dataset is distributed (e.g., CC-BY, MIT, proprietary). This field indicates usage rights and restrictions. The referred license is about the source, not the document. If the document license differs, it is captured in the field license. It can be empty if the content is not known.
- extraction_uid[sc:String] : *U*niversal *U*nique *ID*entifier (ID). In GPT-NL **we adopt the ULID specification.** ULID is the acronym for *Universally Unique Lexicographically Sortable Identifier*. The specification is an alternative to the UUID standard. We propose the use of the ULID formats for some reasons:
    - It is 128-bit compatible with UUID (so, it can be used where UUID is).
    - Canonically encoded as a 26-character string, as opposed to the 36-character UUID
    - It is case-insensitive.
    - Avoids some of the data fragmentation problems created by UUIDs in very large data (such as the ones we have).
    - It is lexicographically sortable – not so important for our work, though, but handy for data operations.
    - An example of a ULID is 01ARZ3NDEKTSV4RRFFQ69G5FAV

  Additionally, we append a suffix to the UID fields: _gpt_nl. This suffix can be easily eliminated but allow us to distinctly identify UID's generated and used in GPT-NL's project and datasets.

  This is an extraction identifier, such that all the entries (rows) are marked with the same UID generated during the extraction phase. It allows us to identify accurately which extraction run produced the entries in this dataset.
- extraction_time[sc:string] : Timestamp (datetime) for the data extraction expressed in UTC format. This is a used, universal, date-time format supported by almost any library.

   - An example of a timestamp in UTC format looks like 2010-11-12T13:14:15Z
   - We use a UTF-8 string format to represent the UTC timestamp.
- extra[sc:struct] : This field is a container for other tables. The structure format within this field is not defined. It can be custom according to the needs of the extraction phase. The curation process is not encouraged to use these fields; they are allowed in here to register additional metadata information collected during the extraction phase.

An example of such data record in a JSON frame would be:

```
[
  {
    "text" : "Historie van mejuffrouw Sara Burgerhart Betje Wolff en Aagje Deken GEBRUIKT EXEMPLA
AR exemplaar universiteitsbibliotheek Leiden, signatuur: 1282 D 11 en D 12 ALGEMENE OPMERKING
EN Dit bestand is, met een aantal hierna te noemen aanpassingen, een diplomatische weergave van Hi
storie van mejuffrouw Sara Burgerhart uit",
    "title" : "Historie van mejuffrouw Sara Burgerhart",
    "source" : "https://dbnl.org/tekst/wolf016hist01_01",
    "author" : "Aagje Deken",
    "license" : "CC-BY",
    "dataset_name": "OpenDutchNews",
    "dataset_url": "https://data.opendutchnews.nl/archive/2025",
    "dataset_license": "CC-BY 4.0",
    "extraction_uid": "01JV89ZPKHECDV65A1891EJ0W0_gptnl",
    "extraction_time": "2025-10-07T09:15:23Z",
    "extra": {
      "language": "nl",
      "source_type": "news articles",
      "notes": "Includes regional news from 2023–2025"
    }
  }
]
```

2.4.3.1 General policy for missing data

The fields described above are all obligatory in the final format – however, some of them are allowed to be empty. When the proper content of a field is not known, or for some reason cannot be recorded, its value should be an empty string (not a NULL element).

2.4.4 Formal Format Description

In the data science and machine learning communities, it is standard practice to describe dataset structures using a formal schema specification. For this purpose, GPT-NL adopts the *Croissant schema*, developed and maintained by the MLCommons and Hugging Face communities. The metadata used in this project follows the conventions and structure defined by this schema.

The Croissant specification provides a standardized, machine-readable framework for describing datasets. It defines key dataset components such as:

- **Entities:** Logical groupings of related data elements or records.
- **Features:** Individual attributes or variables describing each entity.
- **Relationships:** Links between entities, enabling representation of structured or hierarchical data.
- **Metadata:** Descriptive information about the dataset's provenance, licensing, authorship, and usage constraints.

This scheme facilitates interoperability between dataset repositories and tools by offering a consistent method for expressing dataset content and structure. Its growing adoption in the large language model (LLM) community further supports long-term compatibility, discoverability, and integration with emerging data curation and benchmarking frameworks.

The *Croissant* schema for the **EXTRACTED** dataset is:

```
{
   "@type": "sc:Dataset",
   "name": "gpt_nl_uncurated_dataset",
   "description": "The uncurated dataset contains data after the extraction stage.  No stage of the curatio
n pipeline is applied yet. This dataset is the starting     point of the curation pipeline.",
   "license": "All rights reserved.  License to be defined.",
   "url": "https://example.com/dataset <TO BE DEFINED>",
   "distribution": [
      {
         "@type": "cr:FileObject",
         "@id": "unique_id of the dataset",
         "name": "name.pdf",
         "contentUrl": "data/name.pdf",
         "encodingFormat": "text/csv"
      }
   ],
   "recordSet": [
      {
         "@type": "cr:RecordSet",
         "name": "gpt_nl_uncurated_recordset",
         "description": "Minimal record of the uncurated dataset.",
         "field": [
            {
               "@type": "cr:Field",
               "name": "uid",
               "description": "The unique_id of the data record.",
               "dataType": "sc:String",
               "references": {
                  "fileObject": {
                     "@id": "unique_id of the data record "
                  },
                  "extract": {
                     "column": "uid"
                  }
               }
            },
            {
               "@type": "cr:Field",
               "name": "text",
               "description": "The third column contains the data record",
               "dataType": "sc:String",
               "references": {
                  "fileObject": {
                     "@id": "name.pdf"
                  },
                  "extract": {
                     "column": "text"
                  }
               }
            },
            {
               "@type": "cr:Field",
               "name": "meta",
               "description": "Metadata associated with each record.",
               "dataType": "sc:struct",
               "field" : [
                  {
```

```
                            "@type": "cr:Field",
                            "name": "source",
                            "description": "A human readable identifier.",
                            "dataType": "sc:String",
                            "references": {
                                "fileObject": {
                                    "@id": "name.pdf"
                                },
                                "extract": {
                                    "column": "source"
                                }
                            }
                        },
                        {
                            "@type": "cr:Field",
                            "name": "source_url",
                            "description": "A human readable identifier.",
                            "dataType": "sc:URL",
                            "references": {
                                "fileObject": {
                                    "@id": "name.pdf"
                                },
                                "extract": {
                                    "column": "source_url"
                                }
                            }
                        },
                        {
                            "@type": "cr:Field",
                            "name": "timestamp",
                            "description": "Timestamp of the datasource.",
                            "dataType": "sc:Timestamp",
                            "references": {
                                "fileObject": {
                                    "@id": "name.pdf"
                                },
                                "extract": {
                                    "column": "timestamp"
                                }
                            }
                        }
                    ],
                    "references": {
                        "fileObject": {
                            "@id": "name.pdf"
                        },
                        "extract": {
                            "column": "meta"
                        }
                    }
                }
            ]
        }
    ]
}
```

2.5 Code Organization and Data Folders

The primary codebase supporting the extraction phase is maintained in the Data Extraction Repository. This section describes the organization of the repository and the related efforts to make it publicly accessible as an open-source resource.

In addition, two main data directories are central to managing the datasets and operational outputs of this phase:

- /<project-root>/wp12/raw — containing the raw input data; and
- /<project-root>/wp12/extracted — containing the datasets produced through the extraction process.

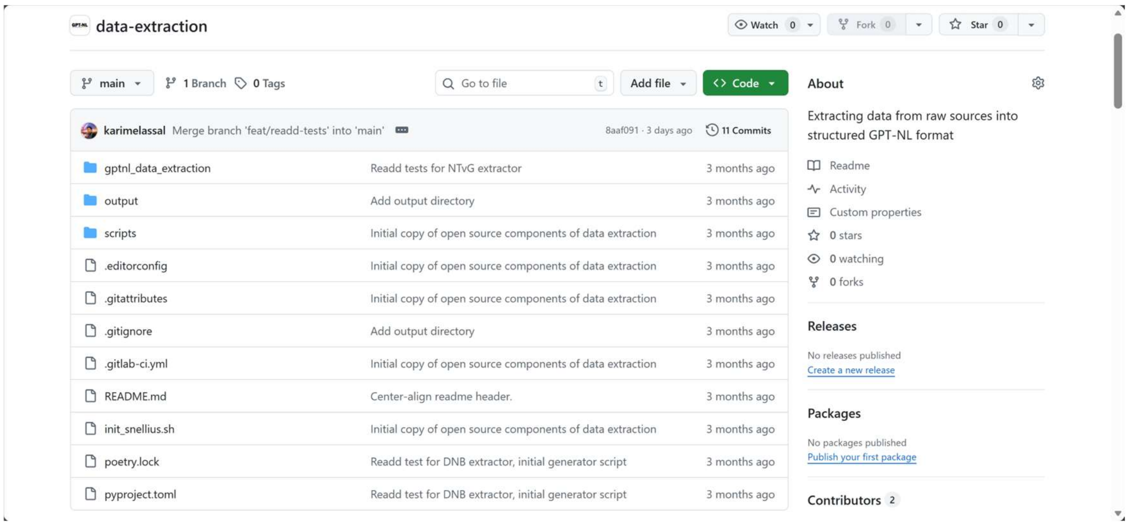
The overall structure and organization of these directories are described in detail in the following subsections.

2.5.1 Data Extraction Repository

The Data Extraction Repository contains data extraction modules for the GPT-NL project. These modules convert raw data files into various formats and from various sources, provided by partners as well as data sets available under open licenses, into a structured format used by the entirety of the data curation pipeline.

The function of the extraction repository is to have a centralized location for data ingestion jobs and extractor implementations. Extraction involves opening data sets, often stored in large (numbers of) files, in an efficient manner, parsing relevant material from them, and writing chunks of data sets while adhering to a specification of the data. This avoids challenges surrounding unstructured data in later stages of the pipeline.

The extraction repository provides uncurated data sets in a columnar format, which may then be inspected using helper scripts (some of which are also in this repository), passed through some stages of a pipeline for initial curation and eventually used as the source of truth for further curation practices. Through data extraction, the data set contains not only rows of text that belongs in one document, section, or paragraph, but also the associated title, source, author, license, and source identifiers belonging to the document and data set. This makes it possible to track and manage this additional metadata smoothly during later stages.



karimelassal Merge branch 'feat/readd-tests' into 'main' 8aaf091 · 3 days ago 11 Commits

gptnl_data_extraction	Readd tests for NTvG extractor	3 months ago
output	Add output directory	3 months ago
scripts	Initial copy of open source components of data extraction	3 months ago
.editorconfig	Initial copy of open source components of data extraction	3 months ago
.gitattributes	Initial copy of open source components of data extraction	3 months ago
.gitignore	Add output directory	3 months ago
.gitlab-ci.yml	Initial copy of open source components of data extraction	3 months ago
README.md	Center-align readme header.	3 months ago
init_snellius.sh	Initial copy of open source components of data extraction	3 months ago
poetry.lock	Readd test for DNB extractor, initial generator script	3 months ago
pyproject.toml	Readd test for DNB extractor, initial generator script	3 months ago

About

Extracting data from raw sources into structured GPT-NL format

Readme
Activity
Custom properties
0 stars
0 watching
0 forks

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

Contributors 2

Figure 6: Extraction repository on GitHub

The data extraction repository is located on TNO’s internal GitLab as well as mirrored to GitHub.

2.5.1.1 Structure

The following folders and files are essential during development on the extraction repository:

- gptnl_data_extraction/: Contains the extractors, base interfaces and utilities. Each extractor consists of an implementation in Python to convert raw data from files into chunks of spec-compliant Parquet files, jobs to download the files from external sources, start the extraction, and finally tests to demonstrate the working of the extractor.
- scripts/: Shell scripts to start jobs for multiple extractors.
- poetry.toml: Contains metadata about the repository and the scripts that can be run using poetry run.

2.5.1.2 Key Responsibilities

- Obtaining raw data sets.
- Extraction of data from raw data files.
- Writing chunks that conform to a project-wide, accessible standard.

2.5.1.3 Open-Sourcing Notes

The data extraction repository works closely with raw data, which means that we have some considerations when making this repository open source. We have removed some personally identifying information of contributors, such as email addresses in job scripts, but we keep a notice of points of contact and authors clearly available in the repository.

The tests demonstrate whether an extractor is correctly able to write rows of data from its respective raw format. Previously, these tests used a small sample of a data set for each of the extractors. In the open-source version of the repository, samples obtained from private/proprietary data sets have been replaced with synthetic data sets, using randomly generated text, fields and metadata like timestamps and URLs. This ensures continued testing capabilities while assuring that we do not violate licenses of the data sets which were made available to the project.

2.5.2 Data Folders

The data management framework relies on a structured directory hierarchy to ensure traceability, reproducibility, and efficient data handling throughout the extraction and curation pipeline. The primary folders are as follows:

- /project-root/data/raw: Contains the raw data received directly from external data suppliers. These files may vary in format, size, and type. Decompression may be required for compressed data sources.
- /project-root/data/extracted: Contains the datasets that have been processed and transformed from the raw data into the standardized format required for the curation pipeline (as specified in curation-input-format.md).

```
/project-root
└── /data
    ├── /README.md  # Description of the folder structure and references to this documentation.
    ├── /raw
    │
```

```
│   ├── /source_001
│   │   ├── received_data_file.zip
│   │   ├── ...
│   │   └── README.md  # Source and data format description, processing notes.
│   ├── /source_002
│   ├── ...
│   └── /source_100
├── /extracted
│   ├── /source_001
│   │   ├── data00001.parquet
│   │   ├── ...
│   │   └── data99999.parquet
│   ├── /source_002
│   ├── ...
│   └── /source_100
```

2.5.2.1 Raw Data Folder

The *raw data* directory contains the original data as received from the various data suppliers. The content may include files in diverse formats and sizes, depending on the source. When necessary, a decompression step should be performed to prepare the files for further processing.

Best practices for managing raw data include:

- Retaining the original files in their received form. Retention is time boxed.
- Including a README.md file in each data source folder to document the source, format, and any preprocessing steps applied.
- Setting dataset-specific access permissions to ensure compliance with privacy and security requirements.
- Restricting write access for non-extraction team members to preserve data integrity.

```
/project-root
└── /data
    └── /raw
        ├── /source_001
        │   ├── received_data_file.zip
        │   ├── ...
        │   └── README.md  # Source and data format description, processing notes.
        ├── /source_002
        ├── ...
        └── /source_100
```

2.5.2.2 Extracted Data

The *extracted data* directory contains datasets generated from the raw sources through the extraction process. These datasets serve as standardized inputs for subsequent curation tasks.

Key requirements and recommendations for managing extracted data include:

- Storing data in **Parquet** format as specified in curation-input-format.md.
- Applying dataset-level access permissions to protect sensitive content and ensure data security.
- Restricting write access for non-extraction personnel to maintain dataset consistency.
- Structuring data into multiple smaller Parquet files (typically between 128 MB and 1 GB each) to facilitate parallel processing. For smaller datasets (a few gigabytes), fewer, larger files may be preferred; for larger datasets (hundreds of gigabytes or more), a greater number of smaller files improves processing efficiency.

```
/project-root
└── /data
    └── /extracted
        ├── /source_001
        │   ├── data00001.parquet
        │   ├── ...
        │   ├── data99999.parquet
        │   └── /source_001_samples
        ├── /source_002
        ├── ...
        └── /source_100
```

3 Architecture and Design of the Data Curation Phase

3.1 Data Curation Process

The GPT-NL data curation phase is organized as a sequence of modular processing stages, each dedicated to a specific aspect of data quality. Instead of a single monolithic cleaning pass, the pipeline divides curation into discrete **stages**, including normalization, language filtering, heuristic quality filtering, PII masking, toxicity removal, and deduplication. By progressing through these focused phases, the pipeline addresses each quality dimension while maintaining clear separation of concerns. This design makes it easy to understand how and why the data is transformed at each step, and it allows each stage to be configured, inspected, and rerun independently without disrupting the overall flow.

To realize this design, the GPT-NL Curation Pipeline Phase was guided by a set of core architectural requirements. The system needed to balance transparency, scalability, and robustness while supporting large volumes of data processing. These requirements shape the structure of the following sections: the first two chapters focus on the *logical architecture*, how modularity, metadata logging, and extensibility are achieved, while the third focuses on the *execution architecture*, on how the system scales, and on HPC infrastructure.

In practice, this meant developing a framework that could:

- **Support modular, stage-based execution**, reflecting the independent and auditable phases of the curation design. Each processing step should run as a self-contained module with clearly defined inputs and outputs, enabling transparent inspection, version control, and easy reruns.
- **Enable fine-grained metadata tracking throughout all transformations**. Every change to the dataset, whether filtering, masking, or enrichment, must be traceable through structured metadata, ensuring that data lineage and provenance can be reconstructed at any point.
- **Allow custom extension of the curation modules**. The architecture must remain open and extensible, allowing researchers to integrate new filters or models that address language- or domain-specific needs without disrupting the overall flow.
- **Handle massive parallelism on HPC infrastructure** such as SURF’s Snellius SLURM cluster. The system must distribute workloads efficiently across numerous compute nodes to process terabyte-scale datasets within reasonable timeframes.
- **Provide robust fault tolerance and resumable execution**, critical for long-running jobs. Failures or node interruptions should not invalidate an entire curation run. Instead, stages should be reproducible from well-defined checkpoints.

3.1.1 Independent Processing Stages

A foundational design requirement for the GPT-NL curation phase was that each processing stage should operate independently and produce a clearly defined new version of the dataset. Rather than modifying data in place or chaining transformations in a black box, the system must create an explicit boundary between stages: each phase reads from a prior output and emits its own result. This requirement ensures that the pipeline is not just functional, but transparent. The curators can clearly inspect how the data evolves across the pipeline.

By persisting the full output of each phase, the system must enable:

- *Reproducibility*, by ensuring that every version of the dataset is traceable and not over-written by later processing.
- *Auditability*, by allowing teams to examine the exact inputs, outputs, and decisions made at every step.
- *Stage execution*, where individual stages can be re-executed in isolation, e.g., to test a different threshold or fix an issue, without requiring upstream stages to be repeated unnecessarily.

Crucially, the architecture should enforce logging at each phase, not just of what is retained, but also of what is removed. For every discarded item, the pipeline should record why it was filtered (e.g., “low average word length” or “toxic language”) along with sufficient context to evaluate that decision later. These logs must be structured in a way that allows the evaluation team to verify whether removals were justified, and whether any filtered data should be reconsidered.

This design also reflects a core commitment to iterative refinement. Filters and thresholds are often imperfect at first. Therefore, the architecture must preserve not only the clean outputs but also the removed data from each stage in a recoverable form. This allows us to later reprocess just the discarded subset with updated parameters. This significantly reduces compute costs and accelerates experimentation.

In sum, treating curation as a series of isolated, reproducible phases, each of which generates a complete and versioned output, is not an implementation detail, it is a core architectural decision. It underpins our ability to scale, experiment safely, collaborate across teams.

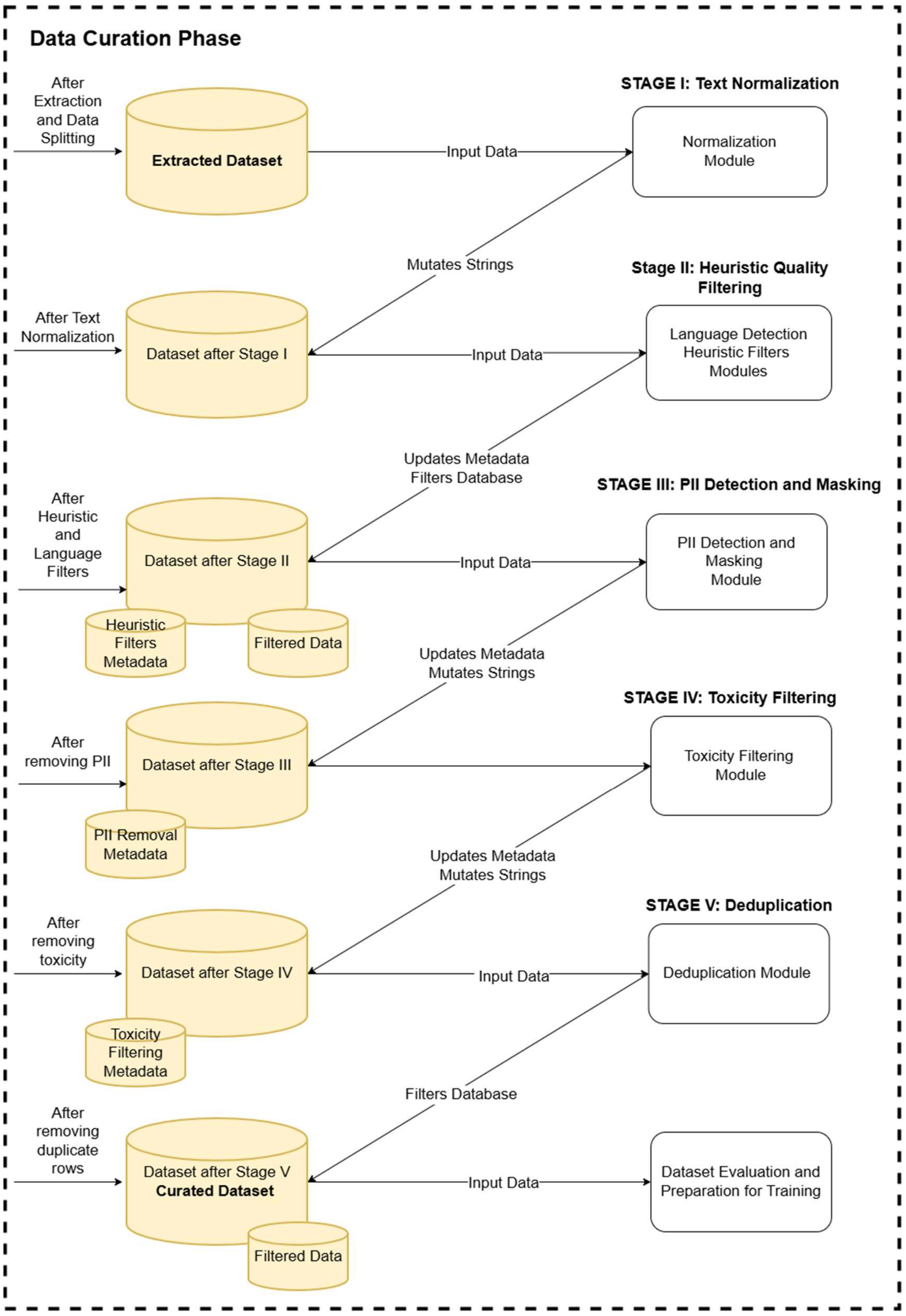

Data Curation Phase
After Extraction and Data Splitting
Extracted Dataset
Input Data
STAGE I: Text Normalization
Normalization Module
Mutates Strings
Stage II: Heuristic Quality Filtering
After Text Normalization
Dataset after Stage I
Input Data
Language Detection Heuristic Filters Modules
Updates Metadata Filters Database
STAGE III: PII Detection and Masking
After Heuristic and Language Filters
Dataset after Stage II
Input Data
PII Detection and Masking Module
Heuristic Filters Metadata
Filtered Data
Updates Metadata Mutates Strings
STAGE IV: Toxicity Filtering
After removing PII
Dataset after Stage III
Toxicity Filtering Module
PII Removal Metadata
Updates Metadata Mutates Strings
STAGE V: Deduplication
After removing toxicity
Dataset after Stage IV
Input Data
Deduplication Module
Toxicity Filtering Metadata
Filters Database
After removing duplicate rows
Dataset after Stage V Curated Dataset
Input Data
Dataset Evaluation and Preparation for Training
Filtered Data

Figure 7: Overview Data Curation pipeline

3.1.2 Stages for Different Quality Dimensions

Having defined the independent stage structure of the GPT-NL curation pipeline, this section details the specific quality dimensions that each stage addresses. Each phase focuses on a single transformation goal so that every aspect of dataset quality can be evaluated, improved, and documented in isolation.

Together, these stages ensure that the curated corpus is linguistically clean, representative of the intended languages, free of personal data, and safe for downstream model training. For each quality dimension, different modules or models can be evaluated and swapped in, allowing experimentation and optimization without altering the overall pipeline structure.

Each phase is scoped to one transformation or evaluation goal:

- **Data Splitting**: Splits large .parquet files into a set of files with more typical sizes. Useful for controlling data flow and SW worker distribution during the curation. Also, large files tend to create problems on the OS side (writing problems, cluttering disk access, memory overload). This stage is not a data processing step but rather an adjustment of file sizes. It is included though in almost all the datasets, unless the dataset itself is already small.

- **Text Normalization** (Normalization-Pipeline.md): Standardizes formatting across the dataset by unifying Unicode characters, normalizing punctuation, and cleaning up whitespace. It consists of three sequential components: FTFYFormatter, PunctuationFormatter, and WhitespaceFormatter. This stage does not modify metadata and purely transforms text for consistent downstream processing.

- **Language Detection** (Language-Detector.md): Tags each document with a predicted language and confidence score, and removes any texts not in the desired set (e.g., non-Dutch or non-English). It enriches each entry's metadata with language and language_score, following the Croissant format.

- **Heuristic Quality Filtering** (Heuristic-Filters.md): Applies rule-based filters to assess document quality, removing entries with excessive repetition, malformed formatting, or other low-quality traits. This stage appends heuristic statistics to both retained and removed data, aiding downstream evaluation and analysis.

- **PII Detection and Masking** (PII.md): Identifies and masks personally identifiable information (e.g., names, emails, organizations) using the PrivateAI module. It preserves text structure and records metadata about detected entities and replacements.

- **Toxicity Filtering** (Harmful-language-Toxicity.md, Harmful-language-Negating-Bias.md): Detects and removes harmful, toxic, or biased content using ML-based classifiers. Filtered or modified entries are annotated with toxicity scores, labels, and span-level metadata.

- **Deduplication** (Deduplication.md): Removes duplicate or near-duplicate documents using a multi-step min-hash–based procedure: signature generation, similarity bucketing, cluster creation, and final filtering. It reduces redundancy across datasets without altering text or metadata and is applied per dataset rather than globally.

Each phase enriches the dataset or improves its quality in an isolated, inspectable way. Metadata generated during earlier phases is preserved and extended by later ones—for example, a document might accumulate a language label, quality metrics, masked entities, and a toxicity score. This cumulative metadata trail provides full visibility into how and why data was curated at each step.

The following table summarizes the impact of each stage on metadata, string mutation, and filtering.

Stages	Updates Metadata	Mutates Strings	Filters Database
Unicode Normalization Filters	✗	✓	✗
Language Detectors	✓	✗	✓
Heuristic Filters	✓	✗	✓
PII Detection	✓	✓	✗
Harmful Language Detector	✓	✓	✗
Deduplication	✗	✗	✓

3.1.3 Design for Scalable Execution

From the very beginning, the GPT-NL curation pipeline was designed with scalability at its core. Processing massive amounts of data requires efficient distribution of work across high-performance computing (HPC) resources. To achieve this, each stage of the pipeline is built to be parallelizable, ensuring that datasets can be split into hundreds of shards—each processed by a separate task within an SLURM job array. This approach allows the system to handle massive throughput, isolate faults (so a single failed shard can be retried independently), and balance load dynamically, as tasks begin execution as soon as compute nodes become available.

Each stage in the pipeline is also designed with independent resource configurations tailored to its specific requirements. Lightweight stages, such as normalization, can run efficiently on standard CPU nodes, while computationally heavier stages like PII masking may require GPU nodes and longer runtimes. These settings—number of tasks, memory, time limits, and hardware specifications—are defined in YAML configuration files that serve as the single source of truth for how a curation run is executed.

Versioning these configurations is critical to ensuring the pipeline's robustness and transparency. It allows every curated dataset to be traced back to the exact configuration that produced it, provides clear documentation for auditing and decision-making, and facilitates comparison between pipeline versions—highlighting how changes such as adjusted thresholds or newly added stages affect results.

In essence, the combination of shard-level parallelism and declarative, versioned configuration makes the GPT-NL pipeline both highly scalable and fully reproducible. This foundation supports not only efficient execution but also transparent governance and continuous improvement at scale.

3.1.4 Conclusion

With these architectural requirements defined, the next step is to select a framework that supports this scalable, configurable, and auditable design. The following section compares **Datatrove** and **Data-Juicer**, two candidate frameworks for implementing the curation pipeline, and explains the rationale behind our choice.

3.2 Comparison of Data Preprocessing Frameworks

Following the architectural requirements outlined in the Data Curation Phase, the next step was to identify a framework that could translate these design principles into a practical implementation. The GPT-NL pipeline required a data processing system that could support **modular, stage-based execution**, ensuring that each phase of curation remained independent, auditable, and transparent. It also needed to enable **fine-grained metadata tracking** across all transformations to allow full reconstruction of data lineage and provenance. To maintain flexibility, the framework had to allow **custom extensions** for different quality dimensions, accommodating both general and Dutch-specific modules. In terms of execution, it had to handle **massive parallelism** on HPC infrastructure such as SURF's SLURM cluster, while providing **robust fault tolerance** and **resumable execution** to safeguard long-running jobs. Finally, because the pipeline operates on sensitive national-level data, the chosen framework needed to ensure **security compliance** and data governance throughout the workflow.

Two frameworks emerged as the most viable candidates: **Datatrove** (developed by Hugging Face) and **Data-Juicer** (developed by Alibaba). Both offer modular pipelines and distributed execution, but they differ in complexity, integration flexibility, and metadata handling. The following table summarizes the comparative analysis.

The following table summarizes the comparison between Datatrove and Data-Juicer across key criteria.

Criteria	Datatrove	Data-Juicer
Pipeline Structure	Provides a flexible pipeline capable of sequentially reading data files, applying multiple operations (read, write, filter), and saving intermediate outputs. Each segment produces a new dataset version, automatically appends metadata, and stores removed data separately for auditability. 🟩 Well-structured and transparent design.	Follows a similar stage-based architecture to Datatrove, with modular reading, filtering, and writing steps. 🟧 Slightly less emphasis on metadata management and traceability.
Data Reader Support	Supports multiple input formats out-of-the-box: - Hugging Face datasets - IPC - CSV - JSONL - Parquet - WARC 🟩 Broad native compatibility ensures seamless integration with large-scale corpora and web archives.	Supports: - Hugging Face (limited) - CSV - JSON - Parquet - Text 🟥 Automatically selects loaders based on the most common file extension, which can be problematic when handling mixed-format datasets.
Heuristic Filtering	Includes a rich set of built-in heuristic modules: - Gopher rules - C4 rules - Regex rules - CodeParrot rules (custom implementation) 🟩 Extensive modular filtering capabilities with strong alignment to academic and HF standards.	Provides Gopher rules and common text cleaning utilities (e.g., remove HTML tags, IPs, links, emails). 🟧 Offers good coverage but less modular and less customizable than Datatrove.
Deduplication	Offers several deduplication methods: - MinHash filter - Bloom filter - Exact string - Exact substring 🟩 Comprehensive options for precision and auditability. 🟥 Currently slower on very large datasets (hours), though deterministic and reproducible.	Implements: - MD5 hash - MinHashLSH - SimHash 🟩 Very fast deduplication (seconds)

Language Detection	Uses **FastText** with project-specific extensions to detect Dutch and English. 🟩 Integrated filtering ensures language consistency and full metadata traceability.	Uses **FastText** for English/Chinese detection (configurable). 🟧 Functional but less configurable for multilingual European corpora.
Scalability	🟩 Excellent SLURM integration and proven performance on HPC clusters. 🟥 Local parallel execution requires Linux (not a target use case).	🟩 Integrates with SLURM and 🟩 supports Ray for distributed processing. 🟧 Less tested at scale in public HPC research environments.
Ease of Use	🟩 Smooth user experience, widely adopted in the Hugging Face ecosystem. 🟩 Strong logging and resume functionality for long-running processes. 🟥 PyPI version still maturing; GitHub version recommended.	🟧 More complex installation due to numerous dependencies. 🟥 PyPI version also incomplete; best used via GitHub.
Extensibility (Custom Filters)	🟩 Highly flexible — custom modules (e.g., PII detection, harmful-language filters) can be integrated with minimal boilerplate. Designed for research extensibility.	🟧 Allows custom filters, though implementation requires more complex configuration. Documentation provides examples but limited modularity.
Documentation and Developer Support	🟧 Documentation could be expanded, but codebase is clean and intuitive to extend. 🟩 Clear logging and structured metadata simplify debugging and development.	🟩 Comprehensive documentation with detailed developer guide. 🟧 Framework complexity slightly higher and codebase harder to navigate.
Security and Data Governance	🟩 Developed and maintained by the **Hugging Face** community, ensuring transparent open-source governance and full local execution. No external API calls are required, making it suitable for handling sensitive or confidential data.	🟥 Several components make API calls to **Alibaba Cloud** servers by default, which raises potential compliance and privacy concerns in restricted or national-level research environments.
Additional Notes	🟩 Provides detailed logs summarizing filtering statistics (e.g., number of records removed, time per step). 🟩 Transparent and auditable workflow aligned with GPT-NL design principles. 🟥 Does not offer easy configuration for filters via YAML by default.	🟩 Provides configuration files for parameter tuning. 🟩 Includes tools to evaluate document quality and token counts. 🟧 Interface for visualizing filtering impact is less integrated.

Legend: ▨ Strong feature, ▨ Moderate feature, ▩ Weak feature. |

3.2.1 Decision

After a thorough comparison of Datatrove and Data-Juicer, **Datatrove** showed it to best meet the needs and requirements of the GPT-NL project. This decision follows directly from the design rationale described in the *Data Curation Phase*. The modular, auditable, and stage-based structure envisioned in the architecture aligns seamlessly with Datatrove's execution model. Each GPT-NL curation stage can be mapped directly to a Datatrove pipeline stage, allowing independent execution, versioning, and metadata tracking. The latest Datatrove release includes components such as the *FineWebQuality Filter* and *C4Badword Filter*, which address many of GPT-NL's processing needs. Its Document structure-storing text, id, and metadata- matches GPT-NL's requirement for accumulating contextual metadata across stages, ensuring full traceability of transformations. The framework's native integration with SLURM provides large-scale parallelism and fault-tolerant execution on HPC environments like SURF's Snellius cluster. In addition, Datatrove's lightweight, Pythonic interface enables rapid customization and the development of new filters without modifying the core codebase. Finally, due to stringent data governance and cybersecurity requirements, Datatrove's open-source and fully auditable nature ensures compliance with TNO and SURF's infrastructure security standards, making it the most reliable and transparent choice for implementing the GPT-NL curation pipeline.

In contrast, while **Data-Juicer** provides a wide range of built-in features and solid documentation, its design introduces several drawbacks for the GPT-NL context. The framework's heavier dependency footprint and partial reliance on cloud-oriented components, including API calls to Alibaba servers, raise security and compliance concerns for use in restricted research environments. Moreover, its higher complexity and less transparent metadata handling make it less aligned with the modular, auditable, and HPC-based design required by GPT-NL.

Therefore, Datatrove was chosen as the processing backbone for the GPT-NL data curation pipeline. It provides the optimal balance between flexibility, reproducibility, performance, and maintainability—allowing the team to implement the architectural principles of the GPT-NL curation framework effectively and securely.

3.3 Pipeline and Modules Repositories

The GPT-NL data curation framework is built around a modular architecture that separates responsibilities across multiple repositories. Instead of keeping all logic in a single monolithic codebase, the system is split into two coordinated parts, the *Modules Repository* and the *Pipeline Repository*, that communicate via a Private PyPI package registry. This separation ensures maintainability, scalability, and clear version control, while also enabling collaborative development across teams.

This approach follows the TNO coding principles, which emphasize modularity, testability, reusability, and transparent workflows:

Principle	How this applies to us
Readability	Each module focuses on a single responsibility, making it easier to understand and maintain.
Documentation	Documentation stays close to the code it describes, improving clarity and long-term usability.
Reproducibility	Versioning modules independently allows pipeline runs to be reproduced precisely, even years later.
Reusability	Self-contained modules can be reused across pipelines and projects without modification.
Testability	Modules can be tested in isolation, enabling safe iteration without breaking the full pipeline.
Shareability	Isolated modules make open-sourcing straightforward once the project is ready.
Reviewability	Code reviews become simpler when changes are limited to specific modules rather than a monolithic codebase.

Although this split introduces some development overhead, the long-term benefits outweigh the cost. The architecture remains clear, versioned, and easy to evolve, especially as GPT-NL grows, and new components are added. The repositories involved are:

Repository	Description
Pipeline	Defines the orchestration logic: the order of stages, how they run, how data flows between them, and how tasks are executed on HPC infrastructure.
Modules	Contains isolated, self-contained data processing modules, each implementing a specific task such as normalization or PII detection.
Private PyPI index	Acts as the bridge between the two repositories by hosting all module versions used by the pipeline.

Historically, both the pipeline and modules lived in a single monolithic repository (e.g., the original Normalization repository), but that structure made it too easy to break reproducibility. The current architecture solves this by enforcing strict separation between development (modules) and execution (pipeline).

3.3.1 The Modules Repository: Focused, Reusable Components

The Modules Repository contains all individual building blocks used throughout the GPT-NL data curation pipeline. Each module is designed as a small, focused, and independently versioned component that implements a specific transformation, normalization, filtering rule, or analysis step. By keeping modules isolated and strictly separated from the pipeline itself, we ensure that development remains agile, reproducible, and scalable. When a module improves, whether through refined heuristics, expanded PII detection, or new filtering logic, no change is required in the pipeline code. The pipeline simply locks a specific version from the private PyPI index, ensuring that past experiments can always be reproduced exactly.

This architectural choice avoids the pitfalls that come from mixing pipeline and module development in a single repository. If they lived together, it would be tempting to reference modules locally, bypassing proper versioning and breaking portability. Instead, each module is self-contained, lives in its own folder, and follows the interface defined by HuggingFace Datatrove's PipelineStep. The pipeline, located in a separate repository, acts only as an orchestrator that chains together whichever module versions are required for a particular curation configuration.

- **C4 Filters**: The C4 Filters module applies the heuristics described in the Colossal Cleaned Common Crawl (C4) paper. It removes low quality or boilerplate web content by enforcing linguistic signals such as terminal punctuation, minimum word counts, and minimum sentence counts. It also excludes pages containing templated text such as "lorem ipsum", curly braces that suggest code snippets, cookie notifications, or "Javascript" boilerplate. Additional checks remove lines with excessively long words. Together these heuristics help eliminate the most common forms of noisy web text. See the original paper: https://jmlr.org/papers/volume21/20-074/20-074.pdf.

- **FineWeb**: The FineWeb filters build upon the heuristics used in HuggingFace's FineWeb dataset. The rules overlap with C4 but are tuned more aggressively toward high quality web corpora. Lines without punctuation, lines with fewer than three words, pages with fewer than five sentences, and pages containing structural artifacts are removed. More information: https://huggingface.co/spaces/HuggingFaceFW/blogpost-fineweb-v1.

- **FTFY**: The FTFY module applies Unicode normalization and encoding repair based on the FTFY library. Its purpose is to clean up messy Unicode text that often arises from mis-encoded web pages, script conversions, or scraping artifacts. Documentation: https://ftfy.readthedocs.io/en/latest/.

- **Gopher Modules**: The Gopher filters draw inspiration from the data quality rules described in DeepMind's Gopher work. They apply structural and statistical checks to detect unnatural, non-linguistic, or synthetic documents. These checks include symbol density, average word length, word count, the presence of stop words, and frequency of structural tokens such as bullet characters.

- **LLM Processing**: The LLM Processing module provides an interface for sending text through a selected language model during the curation pipeline. Users can specify a model and prompt and receive the LLM generated output.

- **Machine Translation**: This module wraps machine translation utilities using HuggingFace models, currently centred on the large google/madlad400-10b-mt model. The module is mainly used for converting multilingual input into Dutch.

- **NordicPile**: The NordicPile filters enforce minimum thresholds for factors such as document length, average line length, and average word length, and they limit the proportion of digit only content. These heuristics catch types of low quality text that often slip through other filters. Based on: https://arxiv.org/pdf/2303.17183.

- **PII**: The PII module identifies and formats Personally Identifiable Information. It detects structured numeric identifiers including phone numbers, IBANs, passport numbers, and account numbers, as well as names and organizations. The module integrates with PrivateAI for more advanced detection and supports masking to replace detected PII with normalized placeholders.

- **Punctuation Formatter**: This module normalizes a broad range of Unicode punctuation into standard ASCII style punctuation. It unifies quotation marks, dashes, ellipses, and other variations that would otherwise fragment token distributions.

- **Quality Analysis**: The Quality Analysis module evaluates text quality using perplexity scores derived from KenLM language models. High perplexity may indicate unnatural, repetitive, or incoherent text.

- **Regex Formatter**: The Regex Formatter applies pattern-based substitution, cleanup of boilerplate artifacts, format normalization, and other transformations through regular expressions.

- **TNO Filters**: The TNO Filters module consists of custom quality heuristics adapted to the needs of the GPT-NL project. These filters focus online level and paragraph level structure such as enforcing minimum and maximum lengths and evaluating average line statistics. Inspired by the DataJuicer framework: https://github.com/modelscope/data-juicer.

- **Toxic Language Detection**: This module filters out toxic or harmful content using models such as IMSyPP/hate_speech_nl and tomh/toxigen_hatebert. It detects hate speech, harassment, and other forms of harmful language.

- **Whitespace Formatter**: The Whitespace Formatter normalizes all whitespace characters to the standard Unicode space (U+0020). It replaces tabs, irregular spacing, unusual separators, and non-breaking spaces. Every module in this repository is published to a private PyPI index with semantic versioning.

When the pipeline is executed, it installs the versions specified in its configuration. Older versions remain fully accessible so that historical pipelines and experiments never break. By separating modules from the pipeline, the GPT-NL project achieves a balance between flexibility and stability. Module development can proceed rapidly and independently, while the pipeline remains a clean and deterministic definition of how those modules are assembled.

3.3.2 The Pipeline Repository: Orchestrating the Workflow

The Pipeline Repository provides an orchestration layer that connects all components of the GPT-NL data curation process. While the Modules Repository manages the detailed **processing logic**, the pipeline defines **how data flows** from raw input to curated output, **which module versions** are used, how stages are **configured**, and how these stages are executed across HPC systems.

A key idea behind this design is the strict separation between **orchestration** and **processing**. The pipeline itself performs no filtering or transformation; each stage delegates its core logic to a versioned module imported from the private PyPI registry. This keeps the pipeline stable and ensures that improvements in individual modules do not require changes in pipeline code.

Pipeline runs are fully defined through **YAML configuration files**, which act as declarative blueprints for a workflow. These files describe the sequence of stages, the exact module versions to use, stage-specific parameters, and the structure of input and output folders. Because configuration files are versioned, curated datasets can always be linked back to the exact settings that produced them.

The pipeline is designed to scale naturally to **high-performance computing (HPC)** clusters. For large datasets, it expands each stage into parallel tasks, configures SLURM resource requirements, and generates job scripts. This includes CPU/GPU allocation, memory settings, batching strategies, and job dependencies. Logs and intermediate data are stored predictably, supporting monitoring and debugging of long-running workloads.

Beyond execution, the pipeline integrates with tools for analysing energy usage, supports parquet validation, and enables dataset quality comparisons. A structured changelog records major adjustments such as module version updates or the introduction of new stages, ensuring a clear history of how the workflow evolves.

Together, these features make the Pipeline Repository the operational backbone of GPT-NL's data curation framework: responsible for workflow design, reproducibility, scalability, and structured execution.

3.3.3 Understanding the Architecture

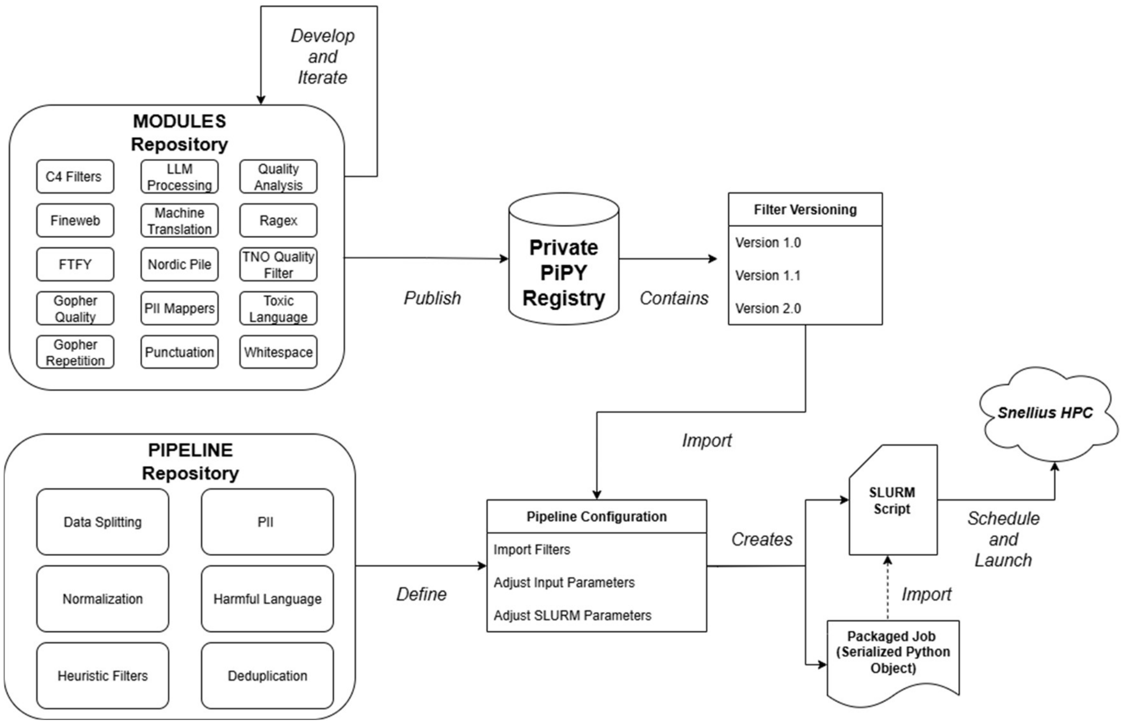


Figure 8: Overview Pipeline and Modules repository

The diagram above illustrates how the distinct parts of the GPT-NL curation framework connect. It shows the full life cycle of a module: how it is developed, published, integrated into the pipeline, and executed at scale on Snellius. Each part has a clear role, and the arrows represent the flow of both code and data.

The process begins in the **Modules Repository**, where components such as filters, formatters, translators, or quality evaluators are developed and evaluated. Once a module reaches a stable state, it is published to the **Private PyPI Registry** with a version number, which becomes the authoritative source for the pipeline.

The **Pipeline Repository** imports these modules by name and version. The YAML configuration file specifies the sequence of stages, the module versions to use, and the resource requirements for running the workflow. The pipeline acts as the blueprint, tying all modules and configurations together.

From this configuration, the pipeline generates a **packaged job** with the selected modules and settings. It also produces the corresponding **SLURM script**, which defines how the job should run on Snellius, including parallelization strategy, memory limits, CPUs, GPUs, and dependencies. These scripts are submitted to the HPC system, which schedules and launches tasks at scale.

Finally, the data flows through the system **stage by stage**, with each stage applying transformations defined by its module. Because module versions and pipeline configurations are pinned, the resulting dataset is fully traceable. Anyone can reconstruct the exact environment and processing logic used in any past run.

3.3.4 Conclusion

The separation of the Modules Repository and the Pipeline Repository is a deliberate architectural choice. The modules define *what* is done — the individual transformations, filters, and analyzes. The pipeline defines *how* these components are executed — their order, configuration, and scaling across HPC infrastructure.

Together, they form a future-proof ecosystem that supports rapid development, transparent versioning, and large-scale execution. As GPT-NL evolves, this structure ensures that new processing steps or improvements can be integrated cleanly without disrupting the entire system.

3.4 Curation Stages in Datatrove

This page explains **how GPT-NL data curation stages are implemented using Datatrove**, building on the modular repository structure described in Pipeline Modularization and leading toward the execution logic explained in Executing a Pipeline.

It focuses on how each curation stage maps to Datatrove components, how configurations are defined through YAML files, and what parameters can be customized per stage.

The YAML configuration, in conjunction with the Python package jsonargparse, plays a significant role in defining the structure of the pipeline stages. This design ensures maximum flexibility and ease of use. The objective is to provide users with full control over pipeline execution through the YAML file, including the ability to specify which stages to run, which modules to activate within each stage, and the corresponding input parameters and threshold values. Below you can find an overview of the example .yaml file, showing how the distinct stages are invoked, and how the input arguments of each stage are accessed through the file:

```yaml
stages:
  - stage: data_splitting
    input_folder: test-data/0. raw
    hpc_n_tasks: "1"
    hpc_time: "00:20:00" # should take max 4.86h
    hpc_partition: "genoa"
  - stage: string_normalization
    ParquetReader:
      # For more information about the reader https://github.com/huggingface/datatrove/blob/v0.3.0/src/da
tatrove/pipeline/readers/parquet.py
      paths_file: "null" # If define specific parquet file is used instead of a whole folder
      limit: "-1" # Defines the number of documents to run the pipeline on
      skip: "0" # Skip the first n documents
      recursive: "false" # if recursive is set to true glob_patterns needs to be set to null (internal bug...)
      glob_pattern: "*parquet"
      shuffle_files: "false"
    hpc_n_tasks: "4"
    hpc_time: "00:20:00" # should take max 4.86h
    hpc_partition: "genoa"
    FTFYFormatter:
      normalization: "NFC"
  - stage: heuristic_filtering
    hpc_n_tasks: "4"
    hpc_time: "00:20:00" # should take max 4.86h
    hpc_partition: "genoa"
    LanguageFilter:
      languages: [
        "en", # English
        "nl", # Dutch
        "da", # Danish
```

```
          "sv", # Swedish
          "af", # Afrikaans
          "fy", # Frisian
          "de", # German
        ]
      language_threshold: "0.65"
      backend: "ft176"
    NordicPileQualityFilter:
      max_digit_fraction: "0.2"
      min_n_char: "50"
      min_mean_med_char: "9"
      min_mean_med_word: "2.1"
    GopherQualityFilter:
      min_doc_words: "null"
      max_doc_words: "null"
      min_avg_word_length: "null"
      max_avg_word_length: "null"
      max_symbol_word_ratio: "0.1"
      max_bullet_lines_ratio: "0.9"
      max_ellipsis_lines_ratio: "0.3"
      max_non_alpha_words_ratio: "0.8"
      min_stop_words: "2"
    GopherRepetitionFilter:
      dup_line_frac: "0.35"
      dup_para_frac: "0.35"
      dup_line_char_frac: "0.2"
      dup_para_char_frac: "0.2"
      top_n_grams: [[2, 0.25], [3, 0.23], [4, 0.21]]
      dup_n_grams:
        [[5, 0.20], [6, 0.19], [7, 0.18], [8, 0.17], [9, 0.16], [10, 0.15]]
  - stage: pii_masking
    hpc_time: "04:00:00"
    hpc_partition: gpu_a100
    # hpc_reservation: gpt-nl
    hpc_n_tasks: "4" # Number of data trove tasks with split up files
    hpc_gpus: "1"
    hpc_cpus_per_task: "16" # Need 16 cores (per private AI GPU instance - actually needs 64 but CPU
affinity warnings can be ignored with GPU instance)
    #hpc_mem_per_cpu_gb: "1"  # 120/128 = 0.9375
    hpc_mem_per_cpu_gb: "4" # Need 64GB ram per private AI instance, 64/16 = 4
    # Start multiple containers with different ports and wait healthy containers
    PII_PrivateAI_TNO:
      chunk_pool_workers: 32 # Number of workers for chunks
      doc_pool_workers: 16 # Number of workers for documents
      request_batch_size: 64 # Chunks of the same document to be sent in the same request to PAI
      batch_size: 16 # Documents to handle in the same batch
      api_endpoint: "http://localhost:808{CUDA_VISIBLE_DEVICES}/" # Template for endpoint per GPU i
nstance (uses task array index and comma-separated indexes from CUDA_VISIBLE_DEVICES)
      replacement_type: "MARKER" # GPU instance does not support SYNTHETIC
      synthetic_replacement_chance: 1.00 # Replace 100% of markers with own synthetic data
      synthetic_replacement_locale: "nl-NL" # Depending on the dominant language use : English en-GB,
Dutch nl-NL
  - stage: toxic_language_detection
    hpc_partition: gpu_h100
    hpc_reservation: gpt-nl
    hpc_gpu: "1"
    hpc_time: "00:20:00" # should take max 4.86h
    hpc_cpus_per_task: "4" # h100: 1/4 node = 16 cores + 1 GPU + 180 GiB
    hpc_mem_per_cpu_gb: "15"
    hpc_n_tasks: "4"
  - stage: deduplication
    hpc_n_tasks: "4"
    hpc_partition: "genoa"
```

Types of arguments:

- **hpc_***: Arguments prefixed with hpc* correspond to HPC configuration parameters that are essential for integration with the SLURM workload management framework. For more information on the hpc_* arguments, please read Executing a Pipeline.
- **rest**: All parameters that do not begin with the hpc_ prefix are specific to the modules within each stage.

By using the YAML configuration file together with the jsonargparse library, it is possible not only to adjust individual parameters but also to enable or disable specific modules or module-specific filters. This can be achieved by assigning a value of NULL to the corresponding parameter or by setting the entire module to NULL.

The provided YAML file illustrates that each stage defines both the execution of the curation process and the set of modules to be included. The following sections provide a detailed explanation of the construction and design principles of the Stage module.

3.4.1 Integration of DataTrove in the Stage class

The Datatrove framework forms the core of the curation pipeline. It defines the implementation and execution of filters and formatters — referred to here as modules — and governs how these modules are orchestrated within a pipeline, whether locally or on an HPC cluster. Consequently, when defining stages, it is essential to align with the Datatrove structure to ensure optimal integration and interoperability within the pipeline.

3.4.1.1 Structure of the Stage class

Each GPT-NL curation stage is implemented as an independent **Datatrove stage**, wrapped in a base Stage class. These stages correspond directly to the conceptual architecture outlined in the Data Curation Phase — such as normalization, language filtering, PII masking, and deduplication, but are now instantiated and configurable within the Datatrove framework.

Each stage:

- Is implemented in a separate Python class (in gptnl_data_curation_pipeline/stages/).
- Each stage regroups all the corresponding modules. For example, the heuristic filtering stage consolidates multiple filtering mechanisms, including the LanguageFilter, NordicPileFilter, and GopherQualityFilter, among others. Similarly, the normalization stage comprises several formatting modules such as FTFYFormatter, PunctuationFormatter, and WhitespaceFormatter.
- Defines which Datatrove components (readers, filters, writers) are executed.
- Supports both **local** and **HPC** execution modes through the Stage base class.
- The system is fully configurable through YAML parameters. Leveraging the Python library jsonargparse, both the input of submodules within a stage and the parameters of the stage itself are readily accessible and modifiable. This approach ensures a high degree of flexibility in YAML-based configuration, allowing precise control over stage-level and module-level settings.
- The Stage class encapsulates the modules, their input parameters, and the HPC-specific parameters to construct a Datatrove Pipeline executor (local or hpc), as illustrated in the mock code below. In this example, stage_spec contains all modules to be

executed along with their initialization arguments. Additional input arguments for the PipelineExecutor—such as those for LocalPipelineExecutor or SlurmPipelineExecutor—are parameters specific to the pipeline configuration. For the SLURM-based executor, invoking executor.run() generates an SLURM script that includes all necessary parameters required by the SLURM workload management framework. Furthermore, all modules are serialized into a pickle file, which can subsequently be accessed and loaded independently on multiple nodes. This enables parallel data curation across the cluster while ensuring that all nodes utilize identical module configurations. Each stage generates its own Slurm script. Within a .yaml file, the stages run in separate scripts, each with a dependency that ensures it waits for the previous stage's script to complete successfully.

```python
def run_stage(
    self,
    job_name: str | None = None,
    pipeline_config_path: str | None = None,
    depending_slurm_jobs: SlurmPipelineExecutor | None = None,
  ) -> None | SlurmPipelineExecutor:
    stage_spec = self.get_stage_spec()
    if not os.path.exists(self.logs_folder):
      os.makedirs(self.logs_folder)
    if pipeline_config_path:
      copyfile(
        src=pipeline_config_path, dst=self.logs_folder / "pipeline_config.yaml"
      )
    with open(self.logs_folder / "stage_params.yaml", "w") as f:
      yaml.dump(vars(self), f)

    if self.processing_type == ProcessingType.local:

      executor = LocalPipelineExecutor(
        pipeline=stage_spec,
        logging_dir=str(self.logs_folder),
        workers=-1,
        tasks=1,
      )
      executor.run()
    else:
      venv_path = Path(__file__).parents[1] / "venv" / "bin" / "activate"

      if job_name is None:
        job_name = self.__class__.__name__

      sbatch_args = self.get_sbatch_args()

      executor = SlurmPipelineExecutor(
        job_name=job_name,
        pipeline=stage_spec,
        tasks=self.hpc_n_tasks,
        cpus_per_task=self.hpc_cpus_per_task,
        mem_per_cpu_gb=self.hpc_mem_per_cpu_gb,
        time=self.hpc_time,
        workers=-1,
        logging_dir=str(self.logs_folder),
        slurm_logs_folder=str(self.slurm_logs_folder),
        sbatch_args=sbatch_args,
        randomize_start_duration=3,
        partition=self.hpc_partition,
        mail_type=self.mail_type,
        mail_user=self.mail_user,
        depends=depending_slurm_jobs,
        tasks_per_job=1,
        max_array_size=30001,
      )

      # This correction is necessary because the initialization of qos in the
      # constructor does not work (bug in DataTrove).
      executor.qos = None
      return executor
```

Furthermore, each stage_spec list is wrapped by the Datatrove ParquetReader and ParquetWriter modules to handle the reading and creation of Parquet databases. For each module in the yaml file, the reader and writer module can be explicitly defined with overwriting input arguments.

```
def set_up_modules(self, args: Namespace):
    """Construct the modules"""

    args = self.set_up_base_module(args)

    # First, add the Parquet reader
    if "ParquetReader" in self._base_module:
        parquet_reader_module = self._base_module["ParquetReader"]
        self.active_modules.append(
            parquet_reader_module["cls"](
                **vars(args.get("ParquetReader", Namespace()))
            )
        )

    # Then, add the modules from _set_up_modules
    self._set_up_modules(args)

    # Finally, add the Parquet writer
    if "ParquetWriter" in self._base_module:
        parquet_writer_module = self._base_module["ParquetWriter"]
        args["ParquetWriter"].output_folder = str(self.output_folder)
        self.active_modules.append(
            parquet_writer_module["cls"](
                **vars(args.get("ParquetWriter", Namespace()))
            )
        )
```

where the ParrquetReadr and ParquetWriter can take the following arguments:

```
class ParquetReader(BaseDiskReader):
  """Read data from Parquet files.
    Will read each batch as a separate document.

  Args:
    data_folder: a str, tuple or DataFolder object representing a path/filesystem
    paths_file: optionally provide a file with one path per line (without the `data_folder` prefix) to read.
    limit: limit the number of documents to read. Useful for debugging
    skip: skip the first n rows
    batch_size: the batch size to use (default: 1000)
    read_metadata: if True, will read the metadata (default: True)
    file_progress: show progress bar for files
    doc_progress: show progress bar for documents
    adapter: function to adapt the data dict from the source to a Document.
      Takes as input: (self, data: dict, path: str, id_in_file: int | str)
        self allows access to self.text_key and self.id_key
```

```
            Returns: a dict with at least a "text" and "id" keys
        text_key: the key containing the text data (default: "text").
        id_key: the key containing the id for each sample (default: "id").
        default_metadata: a dictionary with any data that should be added to all samples' metadata
        recursive: whether to search files recursively. Ignored if paths_file is provided
        glob_pattern: pattern that all files must match exactly to be included (relative to data_folder). Ignor
ed if paths_file is provided
        shuffle_files: shuffle the files within the returned shard. Mostly used for data viz. purposes, do not u
se with dedup blocks
    """
```

```
class ParquetWriter(DiskWriter):
    """Write data to Parquet files.

    Args:
        output_folder: a str, tuple, or DataFolder object representing the output path/filesystem.
        output_filename: optional custom filename for the output file. Defaults to "${rank}.parquet".
        compression: compression algorithm to use. Options: "snappy", "gzip", "brotli", "lz4", "zstd". Default
: "snappy".
        adapter: function to adapt the data dict from a Document to the Parquet format.
            Takes as input: (self, data: dict)
            Returns: a dict suitable for writing to Parquet.
        batch_size: number of rows per batch when writing. Default: 1000.
        expand_metadata: if True, expands metadata fields into separate columns. Default: False.
        max_file_size: maximum size of each output file in bytes. Default: 5GB.
        schema: optional schema definition for the Parquet file. If None, inferred from data.
    """
```

Additional wrappers for filters or mappers are also defined. We discuss them in the Section DataTrove wrappers for Filters and Formatters.

3.4.1.2 Standardized HPC-parameters for the Stage class

The parameters of the Stage class define how the curation step is executed by including the following arguments. For detailed information on HPC-related parameters, refer to Executing a Pipeline.

```
  processing_type: ProcessingType
    """Type of processing: local or on the hpc."""

    input_folder: Path
    """Path to the data input directory."""

    output_folder: Path
    """Path to the data output directory."""

    logs_folder: Path
    """Path to the logs directory."""

    slurm_logs_folder: Path
    """Path to the slurm logs directory."""

    hpc_n_tasks: int
    """Total number of tasks to run on HPC (comply with execution time limit)"""

    hpc_time: str
    """Time limit for job"""

    hpc_partition: str
```

```
"""HPC Partition to use"""

hpc_cpus_per_task: int
"""How many CPUs for each task"""

hpc_mem_per_cpu_gb: int
"""How much memory for each CPU"""

hpc_gpus: int
"""How many GPUs"""

hpc_reservation: str | None
"""Name of the slurm eservation"""

hpc_exclude: str | None
"""Define which partition to exclude"""

hpc_nice: int | None
"""Define adjusted scheduling priority"""

hpc_ear: bool
"""To activate the Energy Aware Runtime (EAR) on HPC."""

mail_type: str
"""HPC: Type of email to receive (NONE, BEGIN, END, FAIL, REQUEUE, or ALL)"""

mail_user: str | None
"""HPC: Email address to send notifications to"""

env_commands: str
"""HPC: Additional env commands that are executed at the start of slurm job"""
```

3.4.2 Stage Overview and Subcomponents

Below, each GPT-NL curation stage is listed with:

- **Purpose** — what the stage does.
- **Datatrove/External/Created components used** — readers, filters, models, writers.
- **Configurable parameters** — expected YAML fields.

3.4.2.1 Data Splitting

Description: DataSplittingStage is a processing stage that takes a directory of Parquet files and splits them into smaller Parquet files based on configurable constraints. Its primary purpose is to manage file sizes and optionally break down very large rows into smaller chunks for easier handling.

File Splitting by Size: You can specify a target maximum file size (max_file_size). The stage uses this value to decide when to start writing to a new file. Note that this is not an exact limit because the size check occurs before writing a batch of rows. If files exceed the target size, reducing batch_size helps improve alignment with the limit.

Row Splitting: If rows are very large, you can split them into smaller chunks using line_chunk_size. This performs simple string chunking (which may truncate words at boundaries) to reduce row length and improve file size control.

Batch Control: The batch_size parameter determines how frequently data is written to disk. Smaller batch sizes lead to more frequent size checks and better adherence to the target file size.

File Size Margin: Because size checks happen before writing a batch, actual file sizes can exceed the target by a margin proportional to line_length * batch_size. For example, with very long rows and large batch sizes, the overshoot can be significant.

Use Cases:

Preparing large datasets for distributed processing by splitting them into manageable chunks. Reducing memory and storage overhead when dealing with extremely large rows. Controlling output file sizes for downstream systems that have size constraints.

Components: Datatrove ParquetReader, Datatrove ParquetWriter

Config Parameters:

```
- `max_file_size`: int
  """Maximum size per split file."""

- `line_chunk_size`: int | None
  """Number of characters to split the input rows into shorter rows. If None, no splitting of rows is done.""
"

- `batch_size`: int
  """How frequently to write to disk (and check file size, affecting file size margin)."""
```

code: In this case, the ParquetReader and ParquetWriter are overridden because their input arguments need to be specified each time in the YAML file.

```
def get_stage_spec(self):
    reader = ParquetReader(
        data_folder=str(self.input_folder),
        file_progress=True,
        doc_progress=True,
        # Logs may be present in  a subdirectory of self.input_folder.
        # As all input files are in the root of self.input_folder,
        # we prevent log files being read by ParquetReader by setting recursive to False
        recursive=False,
        glob_pattern="*.parquet",
        text_key="text",
        id_key="extraction_uid",
    )
    writer = ParquetWriter(
        output_folder=str(self.output_folder),
        max_file_size=self.max_file_size,
        batch_size=self.batch_size,
        expand_metadata=True,
        adapter=gptnl_parquet_writer_adapter,
    )

    if self.line_chunk_size is not None:
        return [
            reader,
            TextChunkerStep(chunk_size=self.line_chunk_size),
            writer,
        ]
    else:
        return [
            reader,
            writer,
        ]
```

3.4.2.2 Text Normalization

Description: StringNormalizationStage is a processing stage designed to normalize and clean text data by applying a sequence of formatting modules. It ensures that text is standardized for downstream tasks such as NLP or data analysis. The stage uses an ordered set of formatters, each responsible for a specific aspect of normalization:

Components:

- FTFYFormatter
- PunctuationFormatter
- WhitespaceFormatter

Config Parameters:

```
FTFYFormatter:
 normalization: "NFC" # Unicode normalization form. Options: ['NFC', 'NFKC', 'NFD', 'NFKD']

PunctuationFormatter:
  punctionation_unicode: dict[old_string, new_string] # Mapping of Unicode punctuation to normalized A
SCII equivalents. # Default mapping includes common replacements: # Chinese/Japanese punctuation
→ Western equivalents # Curly quotes → straight quotes # Special symbols → standard characters
  default: {
  "，": ",", # Chinese comma → comma
  "。": ".", # Chinese period → period
  "、": ",", # Ideographic comma → comma
  "„": "", # Double low quote → double quote
  "”": "", # Right double quote → double quote
  "“": "", # Left double quote → double quote
  "«": "", # Left angle quote → double quote
  "»": "", # Right angle quote → double quote
  "１": "", # Full-width quote → double quote
  "」": "", # Closing quote → double quote
  "「": "", # Opening quote → double quote
  "《": "", # Opening angle quote → double quote
  "》": "", # Closing angle quote → double quote
  "´": "'", # Acute accent → apostrophe
  "∶": ":", # Ratio sign → colon
  "：": ":", # Full-width colon → colon
  "？": "?", # Full-width question mark → question mark
  "！": "!", # Full-width exclamation → exclamation
  "（": "(", # Full-width left parenthesis → (
  "）": ")", # Full-width right parenthesis → )
  "；": ";", # Full-width semicolon → semicolon
  "–": "-", # En dash → hyphen
  "—": " - ", # Em dash → spaced hyphen
  "．": ". ", # Full-width period → period + space
  "～": "~", # Full-width tilde → tilde
  "’": "'", # Right single quote → apostrophe
  "…": "...", # Ellipsis → three dots
  "━": "-", # Heavy dash → hyphen
  "〈": "<", # Opening angle bracket → <
  "〉": ">", # Closing angle bracket → >
  "【": "[", # Opening bracket → [
  "】": "]", # Closing bracket → ]
  "％": "%", # Full-width percent → percent
  "▶": "-", # Bullet arrow → hyphen
}
```

```
WhitespaceFormatter:
  Various_whiteSPACE: dict[str] # Set of whitespace characters to normalize. If not provided, defaults to
:
  default: {  " ", "\u200b", "", "", "", "￼", "   "   }
```

Code:

```
class StringNormalizationStage(Stage):

    def __init__(self):
        super().__init__()
        self._modules = OrderedDict(
            {
                "FTFYFormatter": {
                    "cls": FTFYFormatter,
                    "default": {"normalization": "NFC"},
                },
                "PunctuationFormatter": {"cls": PunctuationFormatter, "default": {}},
                "WhitespaceFormatter": {
                    "cls": WhitespaceFormatter,
                    "default": {},
                },
            }
        )
```

3.4.2.3 Heuristic Filtering Stage

Description: In this case, the heuristic filtering stage does include all the heuristic filters such as the Gopher rules, Nordic Pile but also the LanguageFilter. The aim of this stage is to remove any data points that do not fulfill the criteria set by the heuristic filters/language filter.

Components:

- LanguageFilter
- GopherQualityFilter
- GopherRepetitionFilter
- NordicPileQualityFilter

Config Parameters:

```
LanguageFilter:
  languages: # Languages to Keep
    [
      "en", # English
      "nl", # Dutch
      "da", # Danish
      "sv", # Swedish
      "af", # Afrikaans
      "fy", # Frisian
      "de", # German
    ]
  language_threshold: "0.65" # Minimum confidence score for language detection
  backend: "ft176" # Language detection model to use (FastText 176 languages)

NordicPileQualityFilter:
  max_digit_fraction: "0.2" # Maximum fraction of digits allowed in the text
  min_n_char: "50" # Minimum number of characters required in a document
  min_mean_med_char: "9" # Minimum average median character length per line
  min_mean_med_word: "2.1" # Minimum average median word length per line
```

```
GopherQualityFilter:
  min_doc_words: "null" # Minimum number of words in a document (null = no limit)
  max_doc_words: "null" # Maximum number of words in a document (null = no limit)
  min_avg_word_length: "null" # Minimum average word length (null = no limit)
  max_avg_word_length: "null" # Maximum average word length (null = no limit)
  max_symbol_word_ratio: "0.1" # Maximum ratio of symbol-only words
  max_bullet_lines_ratio: "0.9" # Maximum ratio of bullet-point lines
  max_ellipsis_lines_ratio: "0.3" # Maximum ratio of lines containing ellipses (...)
  max_non_alpha_words_ratio: "0.8" # Maximum ratio of words without alphabetic characters
  min_stop_words: "2" # Minimum number of stop words required in the document

GopherRepetitionFilter:
  dup_line_frac: "0.35" # Maximum fraction of duplicate lines allowed
  dup_para_frac: "0.35" # Maximum fraction of duplicate paragraphs allowed
  dup_line_char_frac: "0.2" # Maximum fraction of duplicate characters in lines
  dup_para_char_frac: "0.2" # Maximum fraction of duplicate characters in paragraphs
  top_n_grams: [[2, 0.25], [3, 0.23], [4, 0.21]] # Thresholds for top repeated n-grams (n, max ratio)
  dup_n_grams: [
      [5, 0.20],
      [6, 0.19],
      [7, 0.18],
      [8, 0.17],
      [9, 0.16],
      [10, 0.15],
    ] # Thresholds for duplicate n-grams (n, max ratio)
```

Code:

```python
class HeuristicFilteringStage(Stage):

    def __init__(self):
        super().__init__()
        self._modules = OrderedDict(
            {
                "LanguageFilter": {
                    "cls": LanguageFilter,
                    "default": {
                        "languages": [
                            "en",  # English
                            "nl",  # Dutch
                            "da",  # Danish
                            "sv",  # Swedish
                            "af",  # Afrikaans
                            "fy",  # Frisian
                            "de",  # German
                        ]
                    },
                },
                "NordicPileQualityFilter": {
                    "cls": NordicPileQualityFilter,
                    "default": {},
                },
                "GopherQualityFilter": {
                    "cls": GopherQualityFilter,
                    "default": {
                        "min_doc_words": None,
                        "max_doc_words": None,
                        "min_avg_word_length": None,
                        "max_avg_word_length": None,
                        "max_symbol_word_ratio": 0.1,
                        "max_bullet_lines_ratio": 0.9,
                        "max_ellipsis_lines_ratio": 0.3,
                        "max_non_alpha_words_ratio": 0.8,
                        "min_stop_words": 2,
                    },
                },
                "GopherRepetitionFilter": {
                    "cls": GopherRepetitionFilter,
                    "default": {
                        "top_n_grams": [[2, 0.25], [3, 0.23], [4, 0.21]],
                        "dup_n_grams": [
                            [5, 0.20],
                            [6, 0.19],
                            [7, 0.18],
                            [8, 0.17],
                            [9, 0.16],
                            [10, 0.15],
                        ],
                    },
                },
            }
        )
```

3.4.2.4 PII Masking

Description: The PII (Personally Identifiable Information) module is designed to detect and handle sensitive information in text data. Its primary goal is to identify entities such as names, addresses, phone numbers, email addresses, and other personal identifiers, and then apply configurable strategies to protect privacy.

Components:

PII_PrivateAI_TNO

Config Parameters:

```
PrivateAIFormatter:
  category: "" # Category for the formatter (skipped in defaults)
  api_endpoint: "http://localhost:8080/" # API endpoint for PrivateAI service
  replacement_type: "SYNTHETIC" # Replacement strategy (synthetic data generation)
  entity_grouping_window: 4500 # Window size for grouping detected entities
  entity_types: [] # Entity types to detect (skipped in defaults)
  check_public_figure: true # Whether to check for public figures
  record_processed_entities: true # Whether to record processed entities
  request_batch_size: 1 # Number of requests per batch
  chunk_pool_workers: 32 # Number of workers for chunk processing
  doc_pool_workers: 32 # Number of workers for document processing
  api_endpoint_attempt_delay: 10 # Delay between API retry attempts (seconds)
  max_api_endpoint_attempts: 5 # Maximum number of API retry attempts
  verbose: false # Enable verbose logging
  validator: null # Optional validator function
  public_figure_csv_files: [] # List of CSV files for public figures
  synthetic_replacement_strategies: {} # Strategies for synthetic replacements
  synthetic_replacement_locale: "nl-NL" # Locale for synthetic replacements
  synthetic_replacement_chance: 1.0 # Probability of applying synthetic replacement
```

Code:

```
class PIIMaskingStage(Stage):

    def __init__(self):
        super().__init__()
        self._modules = OrderedDict(
            {
                "PII_PrivateAI_TNO": {
```

```
                "cls": PII_PrivateAI_TNO,
                "default": {
                    "api_endpoint": "http://localhost:8080/",
                    "replacement_type": "SYNTHETIC",
                    "entity_grouping_window": 4500,
                    "check_public_figure": True,
                    "record_processed_entities": True,
                    "request_batch_size": 1,
                },
                "skip": {"entity_types", "category"},
            },
        }
    )
```

Output: PII replaced with synthetic placeholders; metadata stores entity type and position.

3.4.2.5 Harmful and Toxic Content Filtering

Description: The ToxicLanguageDetection module is designed to identify and flag toxic or offensive language in text data. It leverages pre-trained models specialized for detecting hate speech and offensive content in both Dutch and English.

Components:

ToxicLanguageDetection

Config Parameters:

```
ToxicLanguageDetection:
  threshold: 0.995 # The toxicity score above which a chunk is toxic
  max_chunk_length: 256 # Maximum length of a text chunk to process at once
  device: null # Device to run the model on (e.g., "cpu" or "cuda")
  supported_languages:
    - "nl" # Dutch
    - "en" # English
  nltk_language_map:
    nl: "dutch" # Mapping for NLTK language processing
    en: "english"
  model_label_map:
    nl:
      LABEL_0: "Acceptable"
      LABEL_1: "Inappropriate"
      LABEL_2: "Offensive"
      LABEL_3: "Violent"
    en:
      LABEL_0: "Acceptable"
      LABEL_1: "Toxic"
```

Code:

```
class ToxicLanguageDetectionStage(Stage):

    def __init__(self):
        super().__init__()
        self._modules = OrderedDict(
            {
                "ToxicLanguageDetection": {
                    "cls": ToxicLanguageDetection,
                    "default": {
                        "threshold": 0.995,
                        "max_chunk_length": 256,
                    },
                },
            }
        )
```

Output Metadata: Adds toxicity_score and category labels.

3.4.2.6 Deduplication

Description: The Deduplication Stage is responsible for identifying and removing duplicate or near-duplicate items from the dataset. This process ensures data quality and reduces redundancy before downstream tasks such as indexing or analysis. Because deduplication involves computing hashes, comparing n-grams, and processing large volumes of data, it is typically the longest-running stage in the pipeline. To improve efficiency and fault tolerance, this stage is split into multiple sub-stages/SLURM scripts, each handling a portion of the workload.

Stage 1: Signature Generation

- Component: ParquetReader → MinhashDedupSignature
- Function: Reads input files and computes MinHash signatures for each document.
- Parallelization: Runs as multiple tasks (self.hpc_n_tasks), each processing a subset of files.
- Output: Stores signatures in intermediate/signatures.

Stage 2: Bucket Assignment

- Component: MinhashDedupBuckets
- Function: Groups signatures into hash buckets for efficient duplicate detection.
- Parallelization: Number of tasks equals num_buckets from minhash_config.
- Output: Buckets saved in intermediate/buckets.
- Dependency: Starts after Stage 1 finishes.

Stage 3: Clustering

- Component: MinhashDedupCluster
- Function: Combines bucket results to identify clusters of duplicates.
- Parallelization: Single task (global clustering).
- Output: IDs of duplicates stored in intermediate/remove_ids.
- Dependency: Starts after Stage 2.

Stage 4: Filtering

- Component: ParquetReader → MinhashDedupFilter → ParquetWriter
- Function: Reads original data and removes duplicates, keeping one representative per cluster.
- Parallelization: Same number of tasks as Stage 1 for consistency.
- Output: Deduplicated dataset + exclusion list in intermediate/removed.
- Dependency: Starts after Stage 3.

Components (Datatrove):

- MinhashDedupSignature
- MinhashDedupBuckets
- MinhashDedupCluster
- MinhashDedupFilter

Config Parameters:

```
DeduplicationStage:
  # Number of hash buckets used to group similar items.
  # More buckets reduce collisions but increase memory usage.
  num_buckets: 14

  # Number of hash values stored per bucket.
  # Higher values improve granularity but use more memory.
  hashes_per_bucket: 8

  # Size of n-grams used for hashing.
  # Larger n-grams capture more context but may miss small changes.
  n_grams: 5

  # Whether to use 64-bit hashes instead of 32-bit.
  # Improves uniqueness and reduces collisions at the cost of memory.
  use_64bit_hashes: true
```

Code:

```
class DeduplicationStage(Stage):

    bit_precision: int
    """Whether to use 64-bit hashes for the Minhash config. Better precision means fewer false positives (collisions)."""

    def __init__(self):
        """Initialize the pipeline stage.

        Args:
            default_intermediate_folder (str, optional): default folder to   use for intermediate results. Defaults to "".
        """
        super().__init__()

        self._modules = OrderedDict(
            {
                "MinhashConfig": {
                    "cls": MinhashConfig,
                    "default": {
                        "num_buckets": 14,
                        "hashes_per_bucket": 8,
                        "n_grams": 5,
                    },
                }
            }
        )
```

Execution: Multi-step process (signature → buckets → cluster → filter) using sequential HPC jobs.

Output:

- deduplicated_output/ — final dataset.
- removed/ — duplicates excluded.
- logs/ — Datatrove and Slurm logs.

3.4.2.7 LLMProcessingStage

Description: This module defines a pipeline stage called LLMProcessingStage, which is designed to process text using a Large Language Model (LLM) as part of a data curation workflow. The stage begins with a RowSplitter that splits long text rows into smaller segments for easier processing. Next, RegexFilter removes unwanted patterns such as numeric sequences, file extensions, or certain keywords that indicate irrelevant content. After filtering, a RowCombiner can merge previously split rows into coherent units if needed. The core of the stage is the LLMProcessingStep, which uses a specified LLM model (in this case, microsoft/phi4) and a predefined prompt to rewrite or combine sentences into a fluent and grammatically correct paragraph in Dutch without adding new information. Finally, a LanguageFilter ensures that only texts in the desired languages (Dutch and English) pass through, based on a confidence threshold.

Components:

- LanguageFilter (Datatrove)
- RegexFilter (Datarove)
- LLMProcessingStep
- RowSplitterOrCombiner

Config Parameters:

```
RegexFilter:
  regex_exp: "your-regex-here" # Regex expression used to filter rows
  exclusion_writer: null # Optional writer for excluded documents

LanguageFilter:
  languages: ["nl", "en"] # List of languages to keep; None for all
  language_threshold: 0.65 # Minimum confidence score to accept a document
  exclusion_writer: null # Optional writer for excluded documents
  backend: "ft176" # Language detection backend; options: ft176 or glotlid
  label_only: false # If true, only adds language label without filtering
  keep_top_pairs_threshold: -1 # Keep pairs with score above this; -1 disables

LLMProcessingStep:
  model_name: "microsoft/phi4" # Name of the LLM model to use (e.g., microsoft/phi4, gpt-4)
  prompt: "Your prompt text here" # Prompt text provided to the LLM for processing
  max_tokens: null # Maximum tokens for output; defaults len(input_text)/3 if null
  temperature: 0.1 # Sampling temperature; lower values make more deterministic
  batch_size: 10 # Number of items processed per batch
  use_chat_template: true # Whether to apply chat template formatting
  debug_mode: false # Enable debug mode for verbose logging

RowSplitterOrCombiner:
  split: true # Whether to split rows (true) or recombine them (false)
  separator: "\n" # Delimiter used for splitting or combining rows
  identifier_metadata_field: "row_splitter_id" # Metadata field name for tracking splits; removed during re
combination
```

Code:

```
class LLMProcessingStage(Stage):
    """Use LLM to process text"""

    def __init__(self):
        super().__init__()
        self._modules = OrderedDict(
            {
                "RowSplitter": {
                    "cls": RowSplitterOrCombiner,
                    "default": {
                        "split": True,
                    },
                },
                "RegexFilter": {
                    "cls": RegexFilter,
                    "default": {
                        "regex_exp": r"([\d°]{3,})|(\.(\w{2,4}))|(\/\w+\/\w+)|(WorldCat)|(Wikikids)",
                    },
                },
                "RowCombiner": {
                    "cls": RowSplitterOrCombiner,
                    "default": {
                        "split": False,
                    },
                },
                "LLMProcessingStep": {
                    "cls": LLMProcessingStep,
                    "default": {
                        "model_name": "microsoft/phi4",
                        "prompt": "Given the following sentences, produce a fluent and coherent paragraph that contains all the information in the sentences. Do not generate any information that is not in the sentences. Ensure that the paragraph is grammatically and syntactically correct in Dutch. Do not produce any additional text, only the paragraph.\nSentences:\n{text}\n\nParagraph:\n",
                    },
                },
                "LanguageFilter": {
                    "cls": LanguageFilter,
                    "default": {
                        "languages": ["nl", "en"],
                        "language_threshold": 0.65,
                    },
                },
            }
        )
```

3.4.2.8 MachineTranslation

Description: The MachineTranslation module is typically designed to translate text from one language to another within a data processing pipeline. Its main purpose is to ensure multilingual datasets are normalized into a target language for downstream tasks such as analysis, training, or curation.

Components:

- RegexFormatter
- ParquetWriter
- LanguageFilter
- ExtraFieldFilterStep
- RegexFormattedRowsParquetWriter
- TranslatorStep
- SplittedRowsCombiner

Config Parameters:

```
RegexFormatter:
  pattern: ["your-pattern-here"] # List of regex patterns to match
  repl: [""] # Replacement strings for each pattern; defaults to empty string
  flags: [re.DOTALL] # Regex flags applied during matching; defaults to DOTALL

LanguageFilter:
  languages: ["nl", "en"] # List of languages to keep; None for all
  language_threshold: 0.65 # Minimum confidence score to accept a document
  exclusion_writer: null # Optional writer for excluded documents
  backend: "ft176" # Language detection backend; options: ft176 or glotlid
  label_only: false # If true, only adds language label without filtering
  keep_top_pairs_threshold: -1 # Keep language pairs with score above; -1 disables

ExtraFieldFilterStep:
  extra_field_name: "your_field_name" # Name of the extra metadata field to check
  allowed_values: ["value1", "value2"] # List of allowed values for the field
  raise_when_field_missing: true # Raise an error if the field is missing
  keep_when_field_missing: false # Keep the document the field is missing
  exclusion_writer: null # Optional writer for excluded documents

TranslatorStep:
  model_name: "ModelSpace/GemmaX2-28-9B-v0.1" # Name of the translation model to use
  use_vllm: true # Whether to use vLLM for inference
  chunk_mode: "characters" # options: characters or tokens
  chunk_size: 1024 # Size of each chunk for processing
  batch_size: 128 # Number of chunks processed per batch
  stop_at: null # Optional limit on items to process; null for no limit
  batch_stop_at: null # Optional limit on batches; null for no limit
  dry_run: false # If true, runs without performing actual translation
  source_language: null # Source language; null for auto-detection
  destination_language: "Dutch" # Target language for translation

SplittedRowsCombiner: # No inpit parameters

### class MachineTranslation(Stage):
  """Translates into Dutch, splitting text into smaller chunks."""

  def __init__(self):
    super().__init__()
    self._modules = OrderedDict(
      {
        "RegexFormatter": {

          "cls": RegexFormatter,
          "default": {
            "pattern": [
              r"\b(\w+)\s+\1\s+\1(\s+\1)*\b",
              r"\s?\[[a-zA-Z&;_à-üÀ-Ü\s]{0,15}\]",
            ],
            "repl": [r"\1", ""],
            "flags": [re.DOTALL, re.S],
          },
        },
        "RegexFormattedRowsParquetWriter": {
          "cls": ParquetWriter,
          "default": {"expand_metadata": True, "output_folder": ""},
        },

        "ExtraFieldFilter": {
          "cls": ExtraFieldFilterStep,
          "default": {
            "extra_field_name": ["original_language"],
            "allowed_values": ["en"],
                "raise_when_field_missing": True,
                "keep_when_field_missing": False,
            },
        },
        "TranslatorStep": {
            "cls": TranslatorStep,
            "default": {
                "chunk_size": 1024,
                "batch_size": 32,
                "stop_at": None,
                "batch_stop_at": None,
                "dry_run": False,
            },
        },
        "PostLanguageFilter": {
            "cls": LanguageFilter,
            "default": {
                "languages": ["nl"],
                "language_threshold": 0.65,
            },
        },

        "SplittedRowsCombiner": {"cls": SplittedRowsCombiner, "default": {}},
    }
)
```

3.4.2.9 Extending with New Stages

To implement a new stage:

1. Create a new class in gptnl_data_curation_pipeline/stages/.
2. Inherit from Stage.
3. Define Datatrove operations in get_stage_spec() or override run_stage() if multi-step.
4. Add the stage to stages in run_pipeline.py.
5. Update documentation and YAML schema accordingly.

3.4.3 DataTrove Wrappers for Filters and Formatters

Besides the wrappers and custom classes discussed above, we also defined special wrappers for filters and formatters and targeting specific cases. In the following we explain how that was implemented. If you want to add new operations such as heuristic filters and PII mappers, you must follow a specific structure to integrate your operations into the Datatrove pipeline. Fortunately, it is straightforward. You just need to adhere to the following templates.

3.4.3.1 Filters

The most important aspect is that your Filter operation class needs to have a method called **filter**, which takes a data point, also known as a “Document”. From this Document, you can access all the information attached to that data point (ID, metadata, text). The filter function should return False if it fails a certain filter or True if it passes all the filters.

⚠️ **Very important:** Do not define a method called **run**. BaseFilter already has a method **run** defined, which internally handles how the data points are accessed. Only overwrite the method if you know exactly what you are doing! ⚠️

```python
# Put in src/operators/heuristic_filters/

from datatrove.data import Document
from datatrove.pipeline.filters.base_filter import BaseFilter
from datatrove.pipeline.writers.disk_base import DiskWriter


class TemplateFilter(BaseFilter):
    name = "⚙ Template quality filter"
    _requires_dependencies = ["python package"]

    def __init__(
        self,
        ...,
        exclusion_writer: DiskWriter = None,
    ):
        """
        Filter to apply tempalte's quality heuristic rules.
        Reference: https://example.pdf

        Args:
            template_arg_1:
            template_arg_2:
            ...
            exclusion_writer:
        """
        super().__init__(exclusion_writer)
        self.template_arg_1 = template_arg_1
        self.template_arg_2 = template_arg_2


    def filter(self, doc: Document) -> bool | tuple[bool, str]:
        """
        You must have a filter function defined!!!

        Args:
            doc: Applies the heuristics rules to decide if a document should be REMOVED


        Returns: False if sample.text does not pass any of the the heuristic tests

        """
        from your_python_package import word_tokenize

        text = doc.text
        words = word_tokenize(text)
        n_words = len(words)

        # words < min_doc_words or words > max_doc_words
        if n_words < self.template_arg_1:
            return False, "failed_filter_1"
        if wrods not in self.template_arg_2:
            return False, "failed_filter_2"

        ...
```

3.4.3.2 PII Mappers

In comparison to the filters, the PII operations class needs to have a method called **format** which takes at least a text string. The most important aspect is that the function returns the modified text.

⚠️ **Very important:** Do not define a method called **run**. BaseFilter already has a method **run** defined, which internally handles how the data points are accessed. Only overwrite the method if you know exactly what you are doing! ⚠️

```
# Put in src/operators/pii_mappers/

from datatrove.pipeline.formatters.base import BaseFormatter
from datatrove.pipeline.fromatters.pii import PIIReplacer

class TemplatePIIFormatter(BaseFormatter):
    """
    Replaces a certain string in the document text.
    Args:
        remove_emails: Replace email addresses
        remove_ips: Replace IP addresses only_remove_public_ips: by default we only replace public (an
d thus PII) IPs

    """

    name = "🔧 Template PII"

    def __init__(
        self,
        remove_emails: bool = True,
        email_replacement: tuple[str, ...] | str = ("email@example.com", "firstname.lastname@example.org
"),
    ):
        super().__init__()
        self.remove_emails = remove_emails

        self.emails_replacer = PIIReplacer(
            r"\b[A-Za-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[A-Za-z0-9!#$%&'*+/=?^_`{|}~-]+)*@(?:(?:[A-Za-z0-9](?:[
"
            r"A-Za-z0-9-]*[A-Za-z0-9])?\.)+[A-Za-z0-9](?:[A-Za-z0-9-]*[A-Za-z0-9])?|\[(?:(?:25[0-5]|2[0-4][0-9]
|["
            r"01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[A-Za-z0-9-]*[A-Za-z0-9]:)])",
            email_replacement,
        )

    def format(self, text: str) -> str:
        if self.remove_emails:
            text = self.emails_replacer.replace(text)
        if self.remove_ips:
            text = self.ip_replacer.replace(text)
        return text
```

3.5 Executing a Pipeline

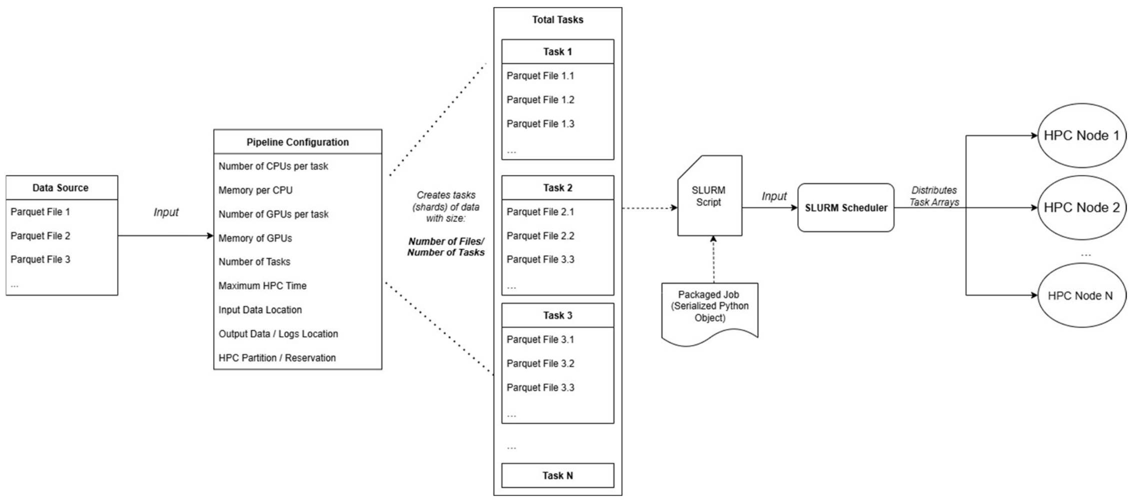


Figure 9: Overview execution and parallelization

This Section explains how a GPT-NL curation stage is executed from the moment data enters the system until all outputs and logs are produced on the HPC cluster. Rather than listing technical parameters in isolation, the goal here is to give a coherent narrative that ties together the design choices introduced in the previous documents: modular processing stages, YAML-driven configuration, and the separation between the pipeline repository and the modules' repository. Execution is where these ideas become concrete—where configuration turns into action and data begins its journey through curated transformation steps to produce the GPT-NL training corpus.

The execution architecture is intentionally built around transparency. Every operation—how files are grouped, how tasks are created, how SLURM schedules work, and how failures are isolated—follows the same design principles that shape the earlier phases of the system. The pipeline is meant to be observable: you should be able to understand what happened, why it happened, and how it can be reproduced.

3.5.1 From Input Files to Executable Units

Each stage begins in a physical place: a folder containing Parquet files. These files represent the current version of the dataset, already processed by the previous stage. When the pipeline reads this folder, Datatrove transforms the set of files into parallelizable units called shards. A shard is nothing more than a subset of the input files. The reason for this design is simple: Datatrove reads and processes whole Parquet files; it does not split them internally. This means the granularity of parallelism is determined by the number and size of the files you provide. As an example:

- If you have 512 files and configure 128 tasks, the system will create ~128 shards of about 4 files each.
- Each shard is then passed to a dedicated task that executes the chosen stage's logic (e.g., normalization).
- These tasks are designed to be parallel; thus they do not depend on each other and can run simultaneously.

This approach has key advantages:

- Scalability: In our example, processing 128 shards in parallel is dramatically faster than processing them sequentially.
- Fault Isolation: If one task fails because of a corrupted file or out-of-memory error, it does not impact the others, only that task (alongside its input shard) must be rerun.
- Efficient Resource Utilization: Tasks can be scheduled and executed as soon as resources are available, instead of waiting for a single long process to finish.

A crucial consideration is shard uniformity. All shards should contain roughly the same amount of data; otherwise, one large shard (a “straggler”) may finish far later than the others, delaying the entire stage. For this reason, it is generally recommended to store data within many smaller Parquet files rather than a few very large ones. The pipeline itself does not split individual files across shards, so file granularity directly affects parallelism and runtime balance. It is, however, possible to split parquet files into smaller ones using a special stage called *data splitting*.

3.5.2 The YAML Configuration as the Execution Contract

In earlier documents, the YAML configuration was introduced as the blueprint that defines the pipeline logically. During execution, this YAML file becomes a contract: every resource request, every module parameter, every input and output path originates from this file. The pipeline does not contain hard-coded decisions; it reads, respects, and enforces whatever the YAML declares.

The YAML then specifies how the stage should behave on the HPC system: which partition to use, how much memory each task needs, whether GPUs are required, and how long the stage is allowed to run. These parameters map directly to SLURM’s scheduling interface. The pipeline converts YAML keys into sbatch arguments, so that the user can tune HPC behavior entirely through configuration rather than code.

The same YAML also specifies module parameters. These values—thresholds, language lists, model names, syntactic rules—shape the actual processing performed by the modules. In other words, the YAML simultaneously defines the logical transformation and the physical execution environment. This tight coupling between description and execution enables full reproducibility.

Below is a simplified example of a GPT-NL pipeline configuration:

```yaml
stages:
  - stage: data_splitting
    input_folder: test-data/0. raw
    hpc_n_tasks: "1"
    hpc_time: "00:20:00"
    hpc_partition: "genoa"
  - stage: string_normalization
    ParquetReader:
      # For more information about the reader https://github.com/huggingface/datatrove/blob/v0.3.0/src/da
tatrove/pipeline/readers/parquet.py
      paths_file: "null" # If define specific parquet file is used instead of a whole folder
      limit: "-1" # Defines the number of documents to run the pipeline on
      skip: "0" # Skip the first n documents
      recursive: "false" # if recursive is set to true glob_patterns needs to be set to null (internal bug...)
      glob_pattern: "*parquet"
      shuffle_files: "false"
    hpc_n_tasks: "4"
    hpc_time: "00:20:00" # should take max 4.86h
    hpc_partition: "genoa"
    FTFYFormatter:
      normalization: "NFC"
  - stage: heuristic_filtering
```

```
    hpc_n_tasks: "4"
    hpc_time: "00:20:00" # should take max 4.86h
    hpc_partition: "genoa"
    LanguageFilter:
      languages: [
          "en", # English
          "nl", # Dutch
          "da", # Danish
          "sv", # Swedish
          "af", # Afrikaans
          "fy", # Frisian
          "de", # German
        ]
      language_threshold: "0.65"
      backend: "ft176"
    NordicPileQualityFilter:
      max_digit_fraction: "0.2"
      min_n_char: "50"
      min_mean_med_char: "9"
      min_mean_med_word: "2.1"
    GopherQualityFilter:
      min_doc_words: "null"
      max_doc_words: "null"
      min_avg_word_length: "null"
      max_avg_word_length: "null"
      max_symbol_word_ratio: "0.1"
      max_bullet_lines_ratio: "0.9"
      max_ellipsis_lines_ratio: "0.3"
      max_non_alpha_words_ratio: "0.8"
      min_stop_words: "2"
    GopherRepetitionFilter:
      dup_line_frac: "0.35"
      dup_para_frac: "0.35"
      dup_line_char_frac: "0.2"
      dup_para_char_frac: "0.2"
      top_n_grams: [[2, 0.25], [3, 0.23], [4, 0.21]]
      dup_n_grams:
        [[5, 0.20], [6, 0.19], [7, 0.18], [8, 0.17], [9, 0.16], [10, 0.15]]
  - stage: pii_masking
    hpc_time: "04:00:00"
    hpc_partition: gpu_a100
    # hpc_reservation: gpt-nl
    hpc_n_tasks: "4" # Number of data trove tasks with split up files
    hpc_gpus: "1"
    hpc_cpus_per_task: "16" # Need 16 cores (per private AI GPU instance - actually needs 64 but CPU
affinity warnings can be ignored with GPU instance)
    #hpc_mem_per_cpu_gb: "1"  # 120/128 = 0.9375
    hpc_mem_per_cpu_gb: "4" # Need 64GB ram per private AI instance, 64/16 = 4
    # Start multiple containers with different ports and wait for healthy containers
    env_commands: "for gpu in ${{CUDA_VISIBLE_DEVICES//,/ }}; do CUDA_VISIBLE_DEVICES=$gpu
apptainer run --nv --contain --pwd /app --env PAI_PORT=$((gpu+SLURM_ARRAY_TASK_ID+8080)) --
env PAI_TRITON_HTTP_PORT=$((gpu+SLURM_ARRAY_TASK_ID+SLURM_ARRAY_TASK_MAX+8
089)) /projects/0/prjs0986/wp13/private-ai/private_ai_gpu.sif & done; sleep 40"
    PII_PrivateAI_TNO:
      chunk_pool_workers: 32 # Number of workers for chunks
      doc_pool_workers: 16 # Number of workers for documents
      request_batch_size: 64 # Chunks of the same document to be sent in the same request to PAI
      batch_size: 16 # Documents to handle in the same batch
      api_endpoint: "http://localhost:808{CUDA_VISIBLE_DEVICES}/" # Template for endpoint per GPU i
nstance (uses task array index and comma-separated indexes from CUDA_VISIBLE_DEVICES)
      replacement_type: "MARKER" # GPU instance does not support SYNTHETIC
      synthetic_replacement_chance: 1.00 # Replace 100% of markers with own synthetic data
      synthetic_replacement_locale: "nl-NL" # Depending on the dominant language use : English en-GB,
Dutch nl-NL
  - stage: toxic_language_detection
    hpc_partition: gpu_h100
    hpc_reservation: gpt-nl
    hpc_gpu: "1"
    hpc_time: "00:20:00" # should take max 4.86h
    hpc_cpus_per_task: "4" # h100: 1/4 node = 16 cores + 1 GPU + 180 GiB
    hpc_mem_per_cpu_gb: "15"
    hpc_n_tasks: "4"
  - stage: deduplication
    hpc_n_tasks: "4"
    hpc_partition: "genoa"
```

The first part of the YAML file (not included in example) defines global execution rules that apply to the entire pipeline:

- processing_type – Determines where the pipeline runs: local (on a single machine) or HPC (distributed across the cluster).
- input_from_previous_output – Ensures that each stage automatically uses the output of the previous stage as its input, chaining them into a continuous workflow.
- output_folder_template – Defines a structured naming pattern for storing results, using variables such as {stage_idx} (stage index) and {stage_name} (stage name). This makes outputs traceable and reproducible.
- logs_folder and slurm_logs_folder – Specify where runtime logs and SLURM scheduler logs are saved. These logs are essential for debugging, performance analysis, and auditing.
- hpc_exclude – Allows you to exclude specific compute nodes (e.g., those reserved for other jobs) from being used.

Together, these settings define the execution environment and output structure for the entire run.

The *stages* section is the core of the YAML file. It defines which steps the pipeline will execute, in which order, and with which parameters. Each stage corresponds to a processing module (e.g., normalization, filtering, PII removal) and typically includes three categories of configuration:

1. Execution Parameters – These determine how the stage runs on the HPC:
   - hpc_n_tasks: Number of tasks (shards) into which the data will be split.
   - hpc_time: Maximum runtime for the stage.
   - hpc_partition: Which partition (or queue) on the cluster to use.
   - hpc_gpus, hpc_cpus_per_task, hpc_mem_per_cpu_gb: Hardware requirements for GPU or CPU tasks.
2. I/O and Input Settings - These specify where the data comes from and where results should be written. For example, input_folder points to the raw dataset for the first stage.
3. Module-Specific Parameters - These configure how the processing logic behaves. For instance:
   - FTFYFormatter.normalization sets the Unicode normalization mode.

  - LanguageFilter.languages defines which languages to keep and the minimum detection score.
  - PII_PrivateAI_TNO.synthetic_replacement_locale specifies the locale for synthetic PII generation.

By combining these three layers, the YAML precisely defines not only what happens at each stage but also how it should happen.

In many ways, the YAML file is the single source of truth for a pipeline run. It encapsulates:

- The logical sequence of curation stages.
- The physical execution environment (resources, partitions, runtime limits).
- The parallelization strategy (number of tasks, shards, workers).
- The operational metadata (paths, logs, monitoring, and notifications).

Because of this, the YAML file is archived with the output dataset itself, ensuring that anyone reviewing the results in the future can reproduce the exact same run, down to the number of tasks and normalization parameters used. This traceability is crucial for research, quality assurance, and future iterations of the GPT-NL project.

3.5.3 Task Orchestration with SLURM

Once shards and tasks are defined, execution is delegated to SLURM, the workload manager responsible for distributing jobs across the HPC cluster. SLURM ensures that each task runs on an appropriate node with the requested resources and that tasks are efficiently queued and scheduled.

The pipeline does not submit each task separately. Instead, it creates an SLURM job array, a single job with many elements, each representing one task. SLURM then assigns these tasks to available nodes and CPUs, launching new tasks as resources become free.

Here’s how this orchestration works in practice:

1. Job Array Creation: The pipeline packages the code, configuration, and shard information for each task and bundles them into a job array.
2. Scheduling: SLURM reads the resource requests from the YAML and places tasks in a queue, scheduling them as soon as CPUs, memory, and GPUs become available.
3. Execution: Each task runs independently on its assigned shard. If 128 workers are allowed, 128 tasks run in parallel. When one finishes, the next queued task begins.
4. Completion and Monitoring: Once all tasks complete, SLURM marks the job array as finished. Logs and metrics from each task are collected for analysis.

The distinction between tasks and workers is key. “Tasks” refers to the total number of work units, while “workers” is the number of tasks that can run concurrently. It is common to configure more tasks than workers, especially on shared HPC infrastructure, where the total number of available cores is limited. SLURM will automatically queue and execute tasks in waves, ensuring efficient resource utilization.

3.5.4 Fault Tolerance, Logging, and Reproducibility

One of the defining properties of this execution architecture is that **each shard is processed independently**. If a single task encounters an out-of-memory error or a corrupted file, only that task fails; the rest continue normally. Datatrove writes comprehensive logs for each shard, capturing statistics such as how many documents were filtered, which modules triggered removals, and how the data evolved. SLURM writes scheduler logs for each task, containing stdout, stderr, error traces, and resource usage information.

When a stage is rerun, Datatrove automatically skips shards that completed successfully and retries only the ones that failed. This avoids the inefficiency of restarting an entire stage from scratch. The combination of precise logging and idempotent tasks makes it straightforward to diagnose failures, reproduce results, and maintain a clear history of execution.

These logs become part of the dataset's provenance. Anyone reviewing a curated dataset can trace exactly how it was produced, which stage transformed it, which shard processed it, and what parameters were applied. This level of transparency is essential for auditing and for maintaining trust in a national-scale language model pipeline.

3.5.5 Why This Architecture Scales

The SLURM-based execution model integrates principles that allow the GPT-NL pipeline to scale without architectural changes:

- **Parallelism by sharding** enables hundreds of tasks to run simultaneously, limited only by available hardware.
- **Job arrays** compact thousands of potential submissions into a single manageable entity, reducing scheduler overhead and simplifying monitoring.
- **Declarative configuration via YAML** ensures complete reproducibility and traceability of every run.
- **Idempotent shard processing** makes failures cheap to recover from, improving robustness on very large datasets.
- **Flexible resource management** allows each stage to request precisely the CPUs, memory, and GPUs it needs, adapting to datasets of different sizes and different computational characteristics.

This structure has already proven capable of processing hundreds of gigabytes of text efficiently. Scaling to larger corpora typically requires only adjusting the number of shards or increasing the allowed runtime—never changing the architecture itself.

3.5.6 Conclusion

The execution layer brings together all core principles of the GPT-NL curation framework. The modular stages defined in the pipeline repository, the versioned modules in the modules' repository, and the YAML configuration converge into a unified workload that SLURM can distribute across the HPC cluster. The result is a system that remains fully transparent, reproducible, fault-tolerant, and scalable.

Execution is not a black box but a carefully orchestrated, inspectable process where every input, task, log, and output have a clear place and purpose. This guarantees that curated datasets are not only high quality, but also trustworthy—each one backed by a clear and auditable record of how it was produced.

3.6 Code Organization and Data Folders

The primary codebases supporting the curation phases are maintained in the Pipeline Repository and the Modules Repository. This section describes the repository organization and the efforts to make it publicly accessible as an open-source resource.

In addition, one main data directory is central to managing the datasets and operational outputs of this phase:

- /<project-root>/wp12/curated — containing the raw input data; and

The structure of these directories is described in the following subsection.

3.6.1 Data Folders

The data management framework relies on a structured directory hierarchy to ensure traceability, reproducibility, and efficient data handling throughout the extraction and curation pipeline. The primary folders are as follows:

- /<project-root>/data/curated: Contains the curated datasets produced after the different stages of the curation pipeline.
- ~/data-curation-pipeline/pipeline-configs: Contains the YAML configuration files used to define and execute curation pipelines. This folder corresponds to the pipeline-configs directory in the Curation Pipeline Repository.

```
/project-root
└── /data
    ├── /README.md  # Description of the folder structure and references to this documentation.
    └── /curated
        ├── /pipeline{run_idx}_{yaml_filename}_{previous_commit_shorthash}
        │   ├── stage_{stage_idx}_{stage_name}
        │   │   ├── logs/
        │   │   ├── slurm_logs/
        │   │   ├── data00001.parquet
        │   │   ├── ...
        │   │   └── data99999.parquet
        │   ├── ...
        │   └── stage_6_toxic_language_detection
        ├── /pipeline00023_test-run_2b69608
        ├── ...
        └── /pipeline99999_full-dataset_9dff500

~/data-curation-pipeline/
└── /pipeline-configs
    ├── /test-pl.yaml
    ├── ...
    └── /full-final-final-final.yaml
```

3.6.2 15.1.1 Curation Data Folders and Run Logs

The *curated data* directory contains the outputs produced by executing the curation pipelines. This directory also includes logs and metadata associated with each pipeline execution.

Each curation run is stored in a versioned folder that captures both the pipeline configuration and the corresponding commit hash, ensuring traceability and reproducibility of pipeline runs. A single folder contains all stages for a given pipeline run, a design choice that prioritizes version control over per-stage access management.

```
/project-root
└── /data
    └── /curated
        ├── /pipeline{run_idx}_{yaml_filename}_{previous_commit_shorthash}
        │   ├── stage_{stage_idx}_{stage_name}
        │   │   ├── logs/
        │   │   ├── slurm_logs/
        │   │   ├── data00001.parquet
        │   │   ├── ...
        │   │   └── data99999.parquet
        │   ├── ...
        │   └── stage_6_toxic_language_detection
        ├── /pipeline00023_test-run_2b69608
        ├── ...
        └── /pipeline99999_full-dataset_9dff500
```

Where:

- **{run_idx}** is incremented for each pipeline execution.
- **{yaml_filename}** denotes the configuration file used for the run.
- **{previous_commit_shorthash}** refers to the abbreviated commit hash of the configuration version in the version control system.
- All modifications must be committed prior to execution to ensure reproducibility and traceability.

3.6.3 15.1.2 Pipeline Configurations

The *pipeline configuration* directory contains the YAML files used as inputs for the curation pipelines. These configurations define the sequence, parameters, and operational settings for each curation run.

Best practices include:

- Maintaining all configuration files under version control, preferably within the Curation Pipeline Repository.
- Using descriptive filenames, as these are embedded in the output directory names of corresponding pipeline runs.
- Including timestamps or unique identifiers in filenames when multiple configuration variants exist, to facilitate differentiation and traceability.

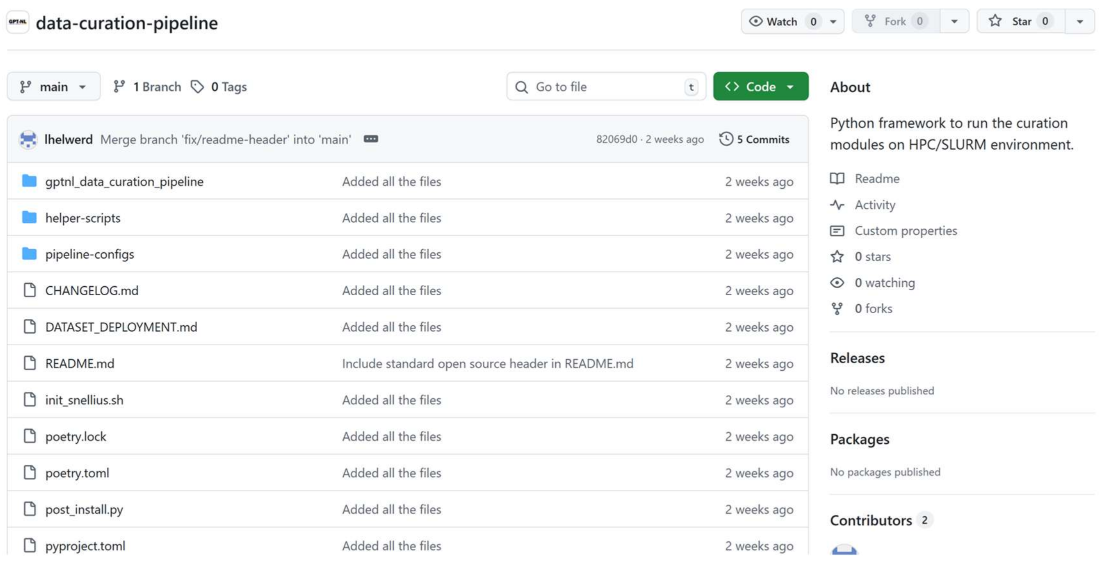
```
~/data-curation-pipeline/
└── /pipeline-configs
    ├── /test-pl.yaml
    ├── ...
    └── /full-final-final-final.yaml
```

3.6.4 Pipeline Repository

The pipeline repository is responsible for providing high-quality text data as a curated training set of a large language model. This involves the tasks of orchestrating a number of pipeline steps or stages which apply filters and modify data in order to remove bad quality, harmful language or personally identifying information. The pipeline also contains the execution logic to transform a raw data set into this quality data, including configuration for each filter and operational parameters such as time limits and resource reservations.

Development in the pipeline repository focuses on building these pipeline configurations using modules from the [modulesrepository](#modules-repository). The reason for the split between the pipeline and modules repository is that we want to keep the data curation modules isolated, small and easy to use. In the pipeline, we will want to specify which version of the module we want to use, by tagging the dependency with the known working version. By keeping the two separate, we avoid promotion of local referencing, which goes against the goals of versioning and modular reusability.

The pipelines developed in the repository are meant to be run on an SLURM cluster, particularly focused on the architecture of Snellius with its differentiated partitions for short jobs and specific, resource-heavy jobs.



lhelwerd Merge branch 'fix/readme-header' into 'main' 82069d0 · 2 weeks ago 5 Commits

gptnl_data_curation_pipeline	Added all the files	2 weeks ago
helper-scripts	Added all the files	2 weeks ago
pipeline-configs	Added all the files	2 weeks ago
CHANGELOG.md	Added all the files	2 weeks ago
DATASET_DEPLOYMENT.md	Added all the files	2 weeks ago
README.md	Include standard open source header in README.md	2 weeks ago
init_snellius.sh	Added all the files	2 weeks ago
poetry.lock	Added all the files	2 weeks ago
poetry.toml	Added all the files	2 weeks ago
post_install.py	Added all the files	2 weeks ago
pyproject.toml	Added all the files	2 weeks ago

About

Python framework to run the curation modules on HPC/SLURM environment.

Readme
Activity
Custom properties
0 stars
0 watching
0 forks

Releases

No releases published

Packages

No packages published

Contributors 2

Pipeline repository on GitHub.

The pipeline repository is located on TNO's internal GitLab as well as mirrored to GitHub.

3.6.4.1 Structure

The following folders and files are essential during development on the pipeline repository:

- gptnl_data_curation_pipeline/: Contains the scripts that allow starting a DataTrove based pipeline.
- helper-scripts/: Contains additional scripts to inspect and validate output formats of the pipeline and perform delivery of a completed data set.
- pipeline-configs/: Contains YAML files which configure the pipelines to run in an SLURM cluster using DataTrove-based stages.
- pyproject.toml: Contains metadata about the repository, the module dependencies with its tagged versions and the scripts that can be run using poetry run ...

3.6.4.2 Key Responsibilities

- Orchestrating stages by preparing input and output data folders, determining which stages are needed and splitting up stages if necessary.
- Starting SLURM jobs for all the relevant stages.
- Tracking configurations for data sets to manage resources and allowing reusability while bookkeeping a record of pipelines.

3.6.4.3 Open-Sourcing Notes

In order to open-source this repository, we have cleaned up the configuration files to not contain personal information of the contributors (such as email addresses). Instead, authors will be added in a notice as part of the repository alongside the README and license files.

We open-source the pipeline configuration as an indicator of how to perform dataset curation on an HPC system similar to Snellius. We considered not providing the configuration because it is of limited use on other types of clusters. While local running is supported, it is not optimized for stage runs. Additionally, certain information may become clear about the (partially proprietary) input data sets and the quality of them. However, this contribution does not contain the data sets themselves. Moreover, other channels of the GPT-NL project will include separate details of the data sets provided by partners. As such, pipeline configurations are a standalone contribution of the curation and cluster development team for pipeline construction, highlighting our work in streamlining this effort.

The documentation has been standardized and includes detailed information on filters and configuration examples, to allow reuse in setting up pipelines for other data sets similarly.

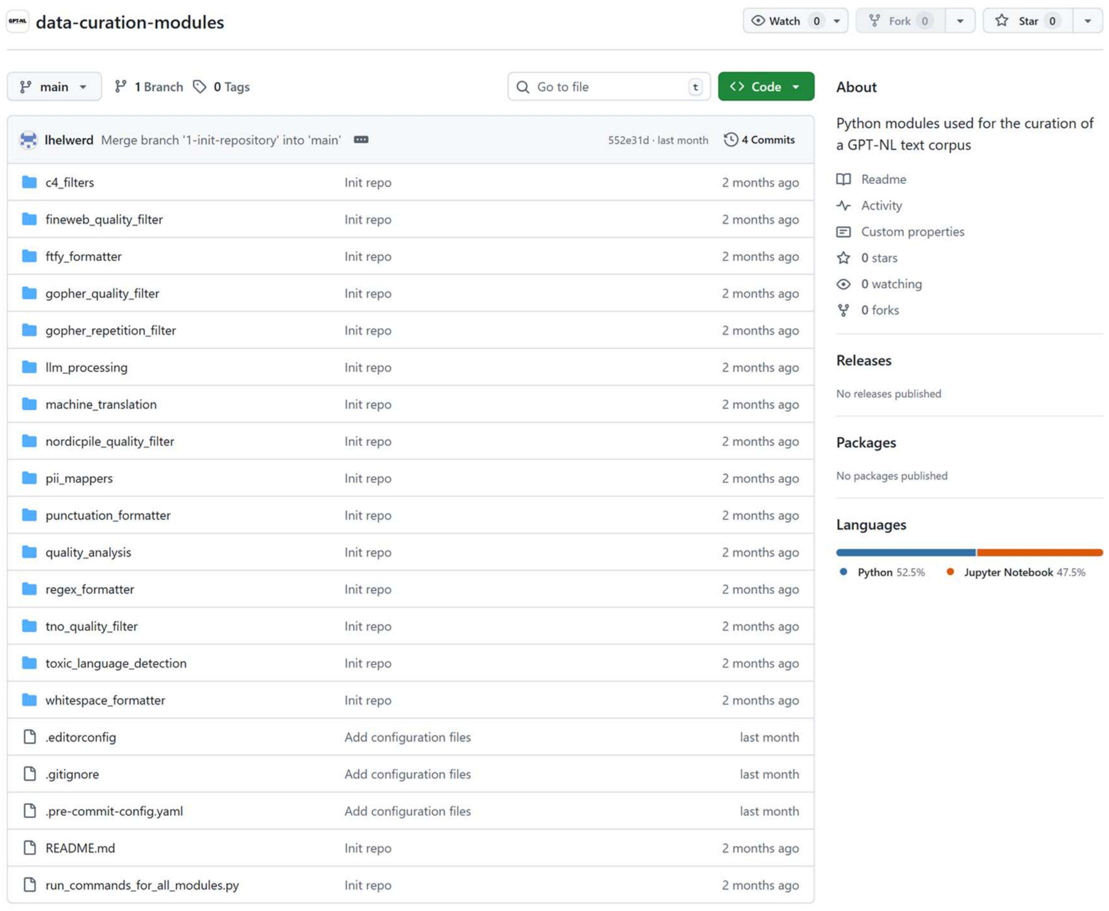
The repository makes use of the modules obtained from a PyPI registry. We hosted this registry at TNO services, but for an open-source variant we also have the modules available in the public PyPI index. This means that we remove tokens used to obtain the packages.

3.6.5 Modules Repository

The data curation modules repository contains modules for the curation pipeline, which filter, transform, reformat or otherwise process rows of text (usually referred to as documents) from data sets. The modules are meant to be reusable and focused on a specific modification task as part of a pipeline stage.

Development in the module repository focuses on building these small, isolated modules to be reused from within the pipeline repository. The reasons for keeping the two portions separate are also mentioned in that section. In the modules, we keep track of versions of the modules that become available, thus allowing running pipelines with a known working, tested version. By keeping the two separate, we avoid promotion of local referencing, which goes against the goals of versioning and modular reusability.

Keeping the modules together in one repository helps with focusing development resources, code quality and publication effort. Initially, we use a private PyPI registry to store tagged versions of each module, and this infrastructure becomes usable for local runs as well as SLURM installations, in our case the Snellius HPC. During our open-source efforts, the private registry is replaced with a public PyPI package index, reflecting the approach used during development and first milestones of our curation efforts.


data-curation-modules

Watch 0 | Fork 0 | Star 0

main | 1 Branch | 0 Tags | Go to file | Code

lhelwerd Merge branch '1-init-repository' into 'main' 552e31d · last month 4 Commits

c4_filters	Init repo	2 months ago
fineweb_quality_filter	Init repo	2 months ago
ftfy_formatter	Init repo	2 months ago
gopher_quality_filter	Init repo	2 months ago
gopher_repetition_filter	Init repo	2 months ago
llm_processing	Init repo	2 months ago
machine_translation	Init repo	2 months ago
nordicpile_quality_filter	Init repo	2 months ago
pii_mappers	Init repo	2 months ago
punctuation_formatter	Init repo	2 months ago
quality_analysis	Init repo	2 months ago
regex_formatter	Init repo	2 months ago
tno_quality_filter	Init repo	2 months ago
toxic_language_detection	Init repo	2 months ago
whitespace_formatter	Init repo	2 months ago
.editorconfig	Add configuration files	last month
.gitignore	Add configuration files	last month
.pre-commit-config.yaml	Add configuration files	last month
README.md	Init repo	2 months ago
run_commands_for_all_modules.py	Init repo	2 months ago

About

Python modules used for the curation of a GPT-NL text corpus

Readme
Activity
Custom properties
0 stars
0 watching
0 forks

Releases

No releases published

Packages

No packages published

Languages

● Python 52.5% ● Jupyter Notebook 47.5%

Modules repository on GitHub

The curation modules repository is located on TNO's internal GitLab as well as mirrored to GitHub.

3.6.5.1 Structure

Each module is stored in its own folder in the curation modules repository. They are integrated as plugins of the curation pipeline, built using HuggingFace's Datatrove module. Simply put, they implement and conform to the same schema as a PipelineStep defined by Datatrove.

Typically, a module consists of the following:

- A folder with the actual source code of the module, which can be further split out into public interfaces (with normal folders and file names) as well as private interfaces (stored in folders and files that start with an underscore).
- A pyproject.toml file indicating the module's metadata, dependency packages and code style practices.
- A brief README.md describing its use and development notes.

Additionally, the module may have the following folders:

- tests: A number of test files to validate the workings of the module.
- data: A sample (publishable) data set to test the workings of the module.

3.6.5.2 Key Responsibilities

- Implementation of filters and other transforms as pipeline stages.

- Testing of modules to ensure proper implementation of concepts discussed during development iterations.
- Tracking versions of each module.
- Publication of modules to PyPI registries.

3.6.5.3 Open-Sourcing Notes

The modules should all be suitable for release as open-source libraries without a particular preference on which of them would be prioritized first, as they are all part of the same repository. Technically, each module can be published individually on a public index, but this only obscures the work, and we instead intend to make every component open-source at the same time.

For improvement of code quality, we keep some standards related to linting and code complexity, as well as trying out Sigrid CI for tracking maintainability issues of the code.

The modules are a good showcase of how we implement our own filters based on research material and make use of interfaces of external providers, for example for personally identifying information.

3.6.6 Open-Source Strategy

GPT-NL promises to be as open and transparent as possible. Although the models and data sets may not be completely available under a permissive license, we publish the code that resulted in a trained, curated model.

For the publication of the code, we target GitHub, as this is where the open-source community is at its largest. For this purpose, we have geared internal policies and configurations of repositories and CI/CD pipelines to keep open-source in mind and perform mirroring of development work that went through internal review and approval processes from GitLab to GitHub.

Code review is enforced through protected branches, merge request templates, review resolution, mandatory approvals and successful CI pipelines.

We make sure that our repositories are clean of internal information, secret credentials such as registry access tokens, and copies of data sets which are not suitable for publication under the permissive license scheme. Contributors have to sign off on these restrictions in open-sourced repositories, with checklists that include these validation steps. For legacy repositories where development may have taken place without this merge request flow, we perform code cleanup by resolving any outstanding merge requests and removing instances of data that are not suitable for publishing. Next, we copy the current state of the repository to an open-source variant, archive the old repository for historic logging purposes, and set up the mirroring on the new, open-source repository.

One part of the cleanup involves tracking which repositories have useful material for publication. Some code is barely used, outdated or in a format that is not suitable for reuse, such as a one-time Python notebook. Inclusions of forks should be instead brought to review at the upstream open-source repository for inclusion in the wider community. This means that initial readiness checks and administrative changes do not encompass all development time spent on making the repository open-source.

We intend to only publish code under a permissive license with clear clauses related to patents, in our case Apache 2.0. We further aim to not include any dependencies that are under more restrictive licenses, such as proprietary code or GPL libraries. In some cases, proprietary code such as those from NVIDIA may be used after review from legal teams.

Quality standards should be maintained, including consistent documentation, reuse of README templates, and code style checks (e.g., linters, CI pipelines).

4 Appendices

This chapter provides a collection of technical reference materials, including hardware specifications, detailed software stack evaluations, assessment results, and formal data and model format definitions. These resources support the system architecture activities but are too detailed to include in the main body of the document.

Each section in this chapter consolidates essential technical information on topics related to the system architecture work. They serve as reference points for the main sections of the report, offering detailed substantiation for their content.

The following appendices are included:

- GPT-NL data curation at SURF's HPC Snellius
- Assessing and monitoring energy at the curation and training pipeline
- SW Stacks and Framework for GPT-NL
- Croissant Format for Curation datasets

4.1 Assessing and Monitoring Energy at the Curation and Training Pipeline

The data curation and training phases of large language models (LLMs) are highly computational and demand substantial energy resources. Measuring energy consumption during these stages is essential for transparency, efficiency, and sustainability in model development. This focus is particularly relevant for GPT-NL, which is committed to a fair and transparent development cycle for LLMs.

Accurate energy profiling helps identify energy-intensive operations, optimize resource usage, and reduce the environmental footprint of large-model training. In addition, systematic energy monitoring provides valuable benchmarks for comparing model architectures, training strategies, and hardware configurations in terms of their energy efficiency.

For GPT-NL, understanding the energy cost of both data curation and pre-training is a strong requirement. The data curation process—encompassing large-scale data collection, cleaning, and filtering—can be as energy-intensive as the training phase itself. Quantifying the energy impact of these operations promotes the responsible use of computational infrastructure and supports the adoption of best practices for sustainable AI development. Given current milestones and budget constraints, GPT-NL focuses on measuring and reporting energy consumption rather than implementing optimization strategies. This approach ensures alignment with the project's goals of transparency and open knowledge sharing.

To achieve this, GPT-NL employs the EAR software library to measure and analyse energy consumption throughout its data curation and training pipeline. In this section, we discuss generic information on the EAR library and later the status of our energy assessment with this tool.

4.1.1 The EAR Software Framework

EAR (Energy Aware Runtime) software library is an energy management and energy monitoring framework designed to measure, analyze, and optimize energy consumption in high-performance computing (HPC) environments. EAR integrates seamlessly with cluster management systems and parallel job schedulers (such as the SLURM system used at Snellius) to provide per-job and per-node energy metrics. It collects real-time data from hardware energy counters and exposes this information through an API and visualization tools, allowing developers and system administrators to monitor consumption at multiple levels of granularity.

At its core, EAR employs a hierarchical architecture that separates monitoring, analysis, and control components. The EAR Daemon runs at the system level, collecting energy data from the hardware sensors and performance counters. This data is then processed by the EAR Library, which can be linked to applications to provide fine-grained, application-level measurements. EAR also supports adaptive power management, dynamically adjusting frequency and power limits to balance performance and energy efficiency.

In addition to monitoring, EAR provides an analytics layer that stores energy metrics in a central database for post-processing. This layer enables statistical analysis, trend detection, and comparison across workloads. EAR's modular design allows it to be extended to new architectures and integrated into complex workflows, making it a suitable choice for measuring energy consumption in large-scale AI training systems such as GPT-NL.

4.1.2 Use of EAR in GPT-NL

GPT-NL utilized the EAR system to measure energy consumption during both the data curation and pre-training stages. The necessary steps are well documented in the Snellius EAR Introduction Website. However, in practice, enabling the EAR monitoring system sometimes introduced instability in the SLURM job scheduler. This issue was particularly noticeable in long-running jobs, such as extended data processing or full training epochs. As a result, for certain extensive data curation and training runs, the EAR-based energy monitoring had to be disabled to ensure uninterrupted execution.

To address these limitations, GPT-NL is developing an alternative measure-and-estimate system that remains based on EAR but focuses on shorter, controlled runs. These shorter measurements are then used to fit an estimation model that predicts the total energy consumption for longer jobs. This hybrid approach balances measurement accuracy with system reliability, ensuring that energy tracking does not compromise training throughput.

At the time of writing, this energy estimation approach is still in validation and results for the curation stages of each dataset were still being compiled. The GPT-NL team will publish these results in a dedicated report after analysis, contributing to transparent and reproducible reporting of LLM development's environmental cost.

4.2 GPT-NL data curation at SURF’s HPC *Snellius*

Snellius serves as the national supercomputer managed by SURF for the Dutch high-performance computing (HPC) community. Designed to support both academic and industrial research, Snellius delivers cutting-edge, heterogeneous computing capabilities—from CPU-only nodes leveraging AMD’s Rome and Genoa architectures to GPU-accelerated configurations with NVIDIA A100 and H100 devices. This system plays a pivotal role in enabling large-scale, data-intensive simulations, machine learning applications, and scientific computing across the Netherlands. With robust SLURM-based job scheduling, flexible partitioning, and precise accounting in System Billing Units (SBUs), Snellius empowers users to maximize computational throughput while maintaining transparency and efficiency—making it a cornerstone of Dutch HPC infrastructure.

4.2.1 Hardware Configuration Overview

These are the key Snellius Partitions used in GPT-NL project.

4.2.1.1 Standard nodes

4.2.1.1.1 rome (alias thin)

- **Node type**: Thin compute nodes (tcn)
- **CPU**: AMD Rome, 128 cores/node
- **Memory**: 224 GiB usable RAM/node
- **Allocation granularity**: 1/8 node ≈ 16 cores + 28 GiB RAM

4.2.1.1.2 genoa

- **Node type**: Thin compute nodes (tcn)
- **CPU**: AMD Genoa, 192 cores/node
- **Memory**: 336 GiB usable RAM/node
- **Allocation granularity**: 1/8 node ≈ 24 cores + 42 GiB RAM
- **Main usage in GPT-NL**: Tests, development, and data curation

4.2.1.2 GPU-Accelerated Partitions

4.2.1.2.1 gpu_A100

- **Node type**: GPU compute nodes (gcn)
- **CPU**: Intel Xeon Platinum 8360Y, 72 cores/node
- **Memory**: 480 GiB RAM/node
- **GPU**: 4 × NVIDIA A100 (40 GB each)
- **Allocation granularity**: 1/4 node ≈ 18 cores + 1 GPU + 120 GiB RAM
- **Main usage in GPT-NL**: Tests, development, model training and data curation

4.2.1.2.2 gpu_H100

- **Node type**: GPU compute nodes (gcn)
- **CPU**: AMD EPYC 9334, 64 cores/node
- **Memory**: 720 GiB RAM/node

- **GPU**: 4 × NVIDIA H100 (94 GiB each)
- **Allocation granularity**: 1/4 node ≈ 16 cores + 1 GPU + 180 GiB RAM
- **Main usage in GPT-NL**: Tests, development, model training and data curation. Most of the pre-training and fine-tuning phases used the gpu_H100 partition with exclusive reservations of up to 22 nodes for longer training batches.

These configurations enable flexible, high-performance computing suitable for a wide range of scientific and engineering applications, reflecting Snellius's role as a versatile and advanced national HPC asset.

4.3 SW Stacks and Framework for GPT-NL

The following tables summarize and compare several SW stacks and AI-Frameworks that can be used for the data curation, training, and execution phases of the LLM created in GPT-NL.

The sections are separated for the different focal activities:

- SW stacks and frameworks for the Data Creation phase
- SW stacks and frameworks for LLM Training
- Support SW and frameworks for implementation in Snelius
- Other support SW

At the end of this section, we include a list of references for the tools and SW packages in the comparison.

4.3.1 Training Software

About	PyTorch	HuggingFace	Microsoft DeepSpeed (comm)	NVidia NeMo	TensorFlow	CUDA
Role within GPT-NL	Framework for machine learning	Umbrella term for transformers (Framework for machine learning, extends PyTorch/TensorFlow/JAX), [datasets](https://huggingface.co/docs/datasets/index) (Creating, loading, sharing of datasets as well as preparing for ML training), accelerate (Wrapper of PyTorch distributed machine learning algorithms) and more	Deep learning optimization framework for distributed training and inference	Generative AI framework for dataset curation, machine learning training and distributing inference in an end-to-end fashion	Framework for machine learning (direct alternative to PyTorch)	Unified processing and utilization of NVIDIA GPUs

General description	**From Wikipedia:** PyTorch is a machine learning library based on the Torch library, used for applications such as computer vision and natural language processing, originally developed by Meta AI and now part of the Linux Foundation umbrella. It is recognized as one of the two most popular machine learning libraries, offering free and open-source software released under the modified BSD license. PyTorch provides two high-level features:	It's an open source data science and machine learning platform. It acts as a hub for AI experts and enthusiasts—like a GitHub for AI. You can browse and use models created by other people, search for and use datasets. It is therefore NOT the same nature as PyTorch because it is not a programming library per se. It is interesting as a repository of *possibly available* models and modules.		NVIDIA NeMo Framework is an end-to-end, cloud-native framework to build, customize, and deploy generative AI models anywhere. It allows researchers and model developers to build their own neural network architectures using reusable components called Neural Modules (NeMo). It includes training and inferencing frameworks, guardrailing toolkits, data curation tools, and pretrained models, offering enterprises an easy, cost-effective, and fast way to adopt generative AI.	**From Wikipedia:** TensorFlow is a free and open-source software library for machine learning and artificial intelligence. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks. Its flexible architecture allows for the easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices.	**From Wikipedia:** CUDA (Compute Unified Device Architecture) is a proprietary and closed-source parallel computing platform and application programming interface (API) that allows software to use certain types of graphics processing units (GPUs) for general-purpose processing, an approach called general-purpose computing on GPUs (GPGPU). CUDA is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements for the execution of compute kernels.
Primary	PyTorch	Hugging Face	Deep Speed	Nvidia NeMo	TensorFlow	CUDA
Licensing	No specific licensing model. *Redistribution and use in source and binary forms, with or without modification, are permitted provided that the some conditions are met. See* License file	Licenses in Hugging Face are particular to the models and modules you use. So a case-by-case review is needed depending on what we import for use.	Apache 2.0	Apache 2.0	Apache 2.0	Proprietary and closed-source (NVidia)
Dependen-		Depends on models to be imported.			Full list of dependencies. (Python wheels or rpm packages)	Full list of dependencies and discussion here

4.3.2 Optimization for Snellius

About	General Description	Remarks

EasyBuild.io	EasyBuild is a software build and installation framework that allows you to manage (scientific) software on High Performance Computing (HPC) systems in an efficient way.	SURF is familiar with the use of this package. Mostly important for the deployment at the HPC.

Best practices for Data Formats in Deep Learning (SURF pages!!!)

4.3.3 Negative Recommendations

- **Do not use TensorFlow** :
  - The ML community seems to be more PyTorch oriented, with better support and documentation.
  - TensorFlow was/is historically more complicated to use, older technology, only later adopted graph execution. More cumbersome.
- **Do not use NVidia NeMo (weak recomendation)**:
  - NeMo offers a end-to-end platform for all stages (curation and training). That sounds interesting, but we get dependent on NVidia.
  - Corporative solution, probably involve extra costs
  - Interesting to be investigated if we want to build a more final commercial product
  - Not open source
  - Strong dependency to NVidia
- **Prefer Parquet over HDF5**:
  - HDF5 is not memory mapped, which may imply less performance
  - HDF5 is a standard for scientific data, but strong on numeric

4.3.4 Positive Recommendations

- **Use PyTorch** :
  - Large community, largely accepted, and well documented.
  - Many packages we intend to use (e.g. Data-Juicer) has PyTorch as dependency.
- **For storage, go for Parquet with Apache Arrow and PySpark on top**:
  - Parquet is a very performant storage format, concurrent, and largely used in LLM
  - Apache Arrow is for fast in-memory data processing
  - Apache Parquet for efficient on-disk storage
  - PySpark has modes to merge columnar data (expanding of data structure) on the fly. [3:57 PM] Apache Spark's is good for distributed data processing
  - Alternative to PySpark here would be to use Ray

- **Build of SW deployments/configuration in Snellius**:
  - We follow here the recommendation of SURF: EasyBuild.io
  - No objections

4.3.5 References

- EasyBuild.io
- Nvidia NeMo GitHub
- Gruener, R., Cheng, O., and Litvin, Y. (2018) Introducing Petastorm: Uber ATG's Data Access Library for Deep Learning
- QCon.ai 2019: "Petastorm: A Light-Weight Approach to Building ML Pipelines".

4.4 Croissant Format During Curation

This Appendix details how the croissant format is updated after each stage.

- String Normalization
    - Format remains the same. No changes in table format.
- Language Detector

```
{
 "@type": "sc:Dataset",
 "name": "gpt_nl_curated_dataset",
 "description": "The curated dataset contains data after the heuristic stage.",
 "license": "All rights reserved. License to be defined.",
 "url": "https://example.com/dataset <TO BE DEFINED>",
 "distribution": [
   {
     "@type": "cr:FileObject",
     "@id": "unique_id of the dataset",
     "name": "name.pdf",
     "contentUrl": "data/name.pdf",
     "encodingFormat": "text/csv"
   }
 ],
 "recordSet": [
   {
     "@type": "cr:RecordSet",
     "name": "gpt_nl_curated_recordset",
     "description": "Record set of the curated dataset.",
     "field": [
       {
         "@type": "cr:Field",
         "name": "uid",
         "description": "The unique_id of the data record. Use an ULID string representation followed by the '_gpt_nl' su
ffix.",
         "dataType": "sc:String",
         "references": {
           "fileObject": {
             "@id": "unique_id of the data record <DO NOT KNOW WHAT THIS IS. FIX LATER>"
           },
           "extract": {
             "column": "uid"
           }
         }
       },
       {
         "@type": "cr:Field",
         "name": "text",
         "description": "The third column contains the raw text of the data record",
         "dataType": "sc:String",
         "references": {
           "fileObject": {
             "@id": "name.pdf"
           },
           "extract": {
             "column": "text"
           }
         }
       },
       {
         "@type": "cr:Field",
         "name": "meta",
         "description": "Metadata associated with each record.",
         "dataType": "sc:struct",
         "field": [
           {
             "@type": "cr:Field",
             "name": "source",
             "description": "A human readable identifier for the data source.",
             "dataType": "sc:String",
             "references": {
               "fileObject": {
                 "@id": "name.pdf"
               },
               "extract": {
                 "column": "source"
               }
             }
           },
           {
             "@type": "cr:Field",
             "name": "source_url",
             "description": "A human readable identifier for the data source.",
             "dataType": "sc:URL",
             "references": {
               "fileObject": {
                 "@id": "name.pdf"
               },
               "extract": {
                 "column": "source_url"
               }
             }
           },
           {
             "@type": "cr:Field",
             "name": "timestamp",
             "description": "Timestamp of the datasource extraction. String representing a datetime UTC timestamp.",
             "dataType": "sc:Timestamp",
             "references": {
               "fileObject": {
                 "@id": "name.pdf"
               },
               "extract": {
                 "column": "timestamp"
               }
             }
           },
           {
             "@type": "cr:Field",
             "name": "language",
             "description": "Language of the text.",
             "dataType": "sc:String",
             "value": "nl"
           },
           {
             "@type": "cr:Field",
             "name": "language_score",
             "description": "Score indicating the confidence of the language detection.",
             "dataType": "sc:Float64",
             "value": 0.8238464593887329
           }
       }
     ]
   }
 ]
}
```

- Quality Filters

```
{
  "@type": "sc:Dataset",
  "name": "gpt_nl_curated_dataset",
  "description": "The curated dataset contains data after the heuristic stage.",
  "license": "All rights reserved. License to be defined.",
  "url": "https://example.com/dataset <TO BE DEFINED>",
  "distribution": [
    {
      "@type": "cr:FileObject",
      "@id": "unique_id of the dataset",
      "name": "name.pdf",
      "contentUrl": "data/name.pdf",
      "encodingFormat": "text/csv"
    }
  ],
  "recordSet": [
    {
      "@type": "cr:RecordSet",
      "name": "gpt_nl_curated_recordset",
      "description": "Record set of the curated dataset.",
      "field": [
        {
          "@type": "cr:Field",
          "name": "uid",
          "description": "The unique_id of the data record. Use an ULID string representation followed by the '_gpt_nl' su
ffix.",
          "dataType": "sc:String",
          "references": {
            "fileObject": {
              "@id": "unique_id of the data record <DO NOT KNOW WHAT THIS IS. FIX LATER>"
            },
            "extract": {
              "column": "uid"
            }
          }
        },
        {
          "@type": "cr:Field",
          "name": "text",
          "description": "The third column contains the raw text of the data record",
          "dataType": "sc:String",
          "references": {
            "fileObject": {
              "@id": "name.pdf"
            },
            "extract": {
              "column": "text"
            }
          }
        },
        {
          "@type": "cr:Field",
          "name": "meta",
          "description": "Metadata associated with each record.",
          "dataType": "sc:struct",
          "field": [
            {
              "@type": "cr:Field",
              "name": "source",
              "description": "A human readable identifier for the data source.",
              "dataType": "sc:String",
              "references": {
                "fileObject": {
                  "@id": "name.pdf"
                },
                "extract": {
                  "column": "source"
                }
              }
            },
            {
              "@type": "cr:Field",
              "name": "source_url",
              "description": "A human readable identifier for the data source.",
              "dataType": "sc:URL",
              "references": {
                "fileObject": {
                  "@id": "name.pdf"
                },
                "extract": {
                  "column": "source_url"
      }
    }
  },
  {
    "@type": "cr:Field",
    "name": "timestamp",
    "description": "Timestamp of the datasource extraction. String representing a datetime UTC timestamp.",
    "dataType": "sc:Timestamp",
    "references": {
      "fileObject": {
        "@id": "name.pdf"
      },
      "extract": {
        "column": "timestamp"
      }
    }
  },
  {
    "@type": "cr:Field",
    "name": "alpha_words_ratio",
    "description": "Ratio of alpha words in the text.",
    "dataType": "sc:Float64",
    "value": 0.928030303030303
  },
  {
    "@type": "cr:Field",
    "name": "avg_word_length",
    "description": "Average length of words in the text.",
    "dataType": "sc:Float64",
    "value": 4.780487804878049
  },
  {
    "@type": "cr:Field",
    "name": "bullet_lines_ratio",
    "description": "Ratio of bullet lines in the text.",
    "dataType": "sc:Float64",
    "value": 0.013157894736842105
  },
  {
    "@type": "cr:Field",
    "name": "digit_char_ratio",
    "description": "Ratio of digit characters in the text.",
    "dataType": "sc:Float64",
    "value": 0.0006934812760055479
  },
  {
    "@type": "cr:Field",
    "name": "dup_line_char_frac",
    "description": "Fraction of duplicate line characters.",
    "dataType": "sc:Float64",
    "value": 0.0
  },
  {
    "@type": "cr:Field",
    "name": "dup_line_frac",
    "description": "Fraction of duplicate lines.",
    "dataType": "sc:Float64",
    "value": 0.0
  },
  {
    "@type": "cr:Field",
    "name": "dup_n_grams",
    "description": "List of duplicate n-grams.",
    "dataType": "sc:List",
    "value": [
      [5.0, 0.0],
      [6.0, 0.0],
      [7.0, 0.0],
      [8.0, 0.0],
      [9.0, 0.0],
      [10.0, 0.0]
```

```
 ]
},
{
 "@type": "cr:Field",
 "name": "dup_para_char_frac",
 "description": "Fraction of duplicate paragraph characters.",
 "dataType": "sc:Float64",
 "value": 0.0
},
{
 "@type": "cr:Field",
 "name": "dup_para_frac",
 "description": "Fraction of duplicate paragraphs.",
 "dataType": "sc:Float64",
 "value": 0.0
},
{
 "@type": "cr:Field",
 "name": "ellipsis_lines_ratio",
 "description": "Ratio of lines with ellipses.",
 "dataType": "sc:Float64",
 "value": 0.0
},
{
 "@type": "cr:Field",
 "name": "ellipsis_ratio",
 "description": "Ratio of ellipses in the text.",
 "dataType": "sc:Float64",
 "value": 0.0
},
{
 "@type": "cr:Field",
 "name": "hash_ratio",
 "description": "Hash ratio of the text.",
 "dataType": "sc:Float64",
 "value": 0.0
},
{
 "@type": "cr:Field",
 "name": "language",
 "description": "Language of the text.",
 "dataType": "sc:String",
 "value": "nl"
},
{
 "@type": "cr:Field",
 "name": "language_score",
 "description": "Score indicating the confidence of the language detection.",
 "dataType": "sc:Float64",
 "value": 0.8238464593887329
},
{
 "@type": "cr:Field",
 "name": "mean_med_char",
 "description": "Mean median character length.",
 "dataType": "sc:Float64",
 "value": 16.355263157894736
},
{
 "@type": "cr:Field",
 "name": "mean_med_word",
 "description": "Mean median word length.",
 "dataType": "sc:Float64",
 "value": 3.736842105263158
},
{
 "@type": "cr:Field",
 "name": "n_char",
 "description": "Number of characters in the text.",
 "dataType": "sc:Int64",
 "value": 1194
```

```
              },
              {
                "@type": "cr:Field",
                "name": "n_non_symbol_words",
                "description": "Number of non-symbol words.",
                "dataType": "sc:Int64",
                "value": 246
              },
              {
                "@type": "cr:Field",
                "name": "stop_words_count",
                "description": "Count of stop words in the text.",
                "dataType": "sc:Int64",
                "value": 69
              },
              {
                "@type": "cr:Field",
                "name": "top_n_grams",
                "description": "List of top n-grams.",
                "dataType": "sc:List",
                "value": [
                  [2.0, 0.01525659],
                  [3.0, 0.02357836],
                  [4.0, 0.01317614]
                ]
              }
            ],
            "references": {
              "fileObject": {
                "@id": "name.pdf"
              },
              "extract": {
                "column": "meta"
              }
            }
          }
        ]
      }
  ]
}
```

- Identification and removal of personal information

```
{
  "@type": "sc:Dataset",
  "name": "gpt_nl_curated_dataset",
  "description": "The curated dataset contains data after the heuristic stage.",
  "license": "All rights reserved. License to be defined.",
  "url": "https://example.com/dataset <TO BE DEFINED>",
  "distribution": [
    {
      "@type": "cr:FileObject",
      "@id": "unique_id of the dataset",
      "name": "name.pdf",
      "contentUrl": "data/name.pdf",
      "encodingFormat": "text/csv"
    }
  ],
  "recordSet": [
    {
      "@type": "cr:RecordSet",
      "name": "gpt_nl_curated_recordset",
      "description": "Record set of the curated dataset.",
      "field": [
        {
          "@type": "cr:Field",
          "name": "uid",
          "description": "The unique_id of the data record. Use an ULID string representation followed by the '_gpt_nl' su
ffix.",
          "dataType": "sc:String",
          "references": {
```

```
          "fileObject": {
            "@id": "unique_id of the data record <DO NOT KNOW WHAT THIS IS. FIX LATER>"
          },
          "extract": {
            "column": "uid"
          }
        }
      },
      {
        "@type": "cr:Field",
        "name": "text",
        "description": "The third column contains the raw text of the data record",
        "dataType": "sc:String",
        "references": {
          "fileObject": {
            "@id": "name.pdf"
          },
          "extract": {
            "column": "text"
          }
        }
      },
      {
        "@type": "cr:Field",
        "name": "meta",
        "description": "Metadata associated with each record.",
        "dataType": "sc:struct",
        "field": [
          {
            "@type": "cr:Field",
            "name": "source",
            "description": "A human readable identifier for the data source.",
            "dataType": "sc:String",
            "references": {
              "fileObject": {
                "@id": "name.pdf"
              },
              "extract": {
                "column": "source"
              }
            }
          },
          {
            "@type": "cr:Field",
            "name": "source_url",
            "description": "A human readable identifier for the data source.",
            "dataType": "sc:URL",
            "references": {
              "fileObject": {
                "@id": "name.pdf"
              },
              "extract": {
                "column": "source_url"
              }
            }
          },
          {
            "@type": "cr:Field",
            "name": "timestamp",
            "description": "Timestamp of the datasource extraction. String representing a datetime UTC timestamp.",
            "dataType": "sc:Timestamp",
            "references": {
              "fileObject": {
                "@id": "name.pdf"
              },
              "extract": {
                "column": "timestamp"
              }
            }
          },
          {
```

```
        "@type": "cr:Field",
        "name": "entity_types",
        "description": "List of entity type names.",
        "dataType": "sc:String",
        "value": ["PERSON", "ORG", "LOCATION", "..."]
      },
      {
        "@type": "cr:Field",
        "name": "entity_type_counts",
        "description": "Count of each entity type.",
        "dataType": "sc:Int64",
        "value": [123, 45, 67, "..."]
      },
      {
        "@type": "cr:Field",
        "name": "failed_chunks",
        "description": "Chunks that failed during processing.",
        "dataType": "sc:Int64",
        "value": [5, 12, 19]
      },
      {
        "@type": "cr:Field",
        "name": "not_removed_entities",
        "description": "Entities that were not removed.",
        "dataType": "sc:String",
        "value": ["entity1", "entity2", "..."]
      },
      {
        "@type": "cr:Field",
        "name": "not_removed_entity_counts",
        "description": "Counts of not removed entities.",
        "dataType": "sc:Int64",
        "value": [3, 7, "..."]
      },
      {
        "@type": "cr:Field",
        "name": "processed_entities",
        "description": "Processed entity names.",
        "dataType": "sc:String",
        "value": ["entityA", "entityB", "..."]
      },
      {
        "@type": "cr:Field",
        "name": "processed_entity_counts",
        "description": "Counts of processed entities.",
        "dataType": "sc:Int64",
        "value": [10, 20, "..."]
      },
      {
        "@type": "cr:Field",
        "name": "processed_entity_types",
        "description": "Types of processed entities.",
        "dataType": "sc:String",
        "value": ["ORG", "PERSON", "..."]
      },
      {
        "@type": "cr:Field",
        "name": "processed_entity_replacements",
        "description": "Replacement values for processed entities.",
        "dataType": "sc:String",
        "value": ["[REDACTED]", "[MASKED]", "..."]
      }
    }
   ]
  }
 ]
}
```

- Harmful Language

```
{
 "@type": "sc:Dataset",
 "name": "gpt_nl_curated_dataset",
 "description": "The curated dataset contains data after the heuristic stage.",
 "license": "All rights reserved. License to be defined.",
 "url": "https://example.com/dataset <TO BE DEFINED>",
 "distribution": [
   {
     "@type": "cr:FileObject",
     "@id": "unique_id of the dataset",
     "name": "name.pdf",
     "contentUrl": "data/name.pdf",
     "encodingFormat": "text/csv"
   }
 ],
 "recordSet": [
   {
     "@type": "cr:RecordSet",
     "name": "gpt_nl_curated_recordset",
     "description": "Record set of the curated dataset.",
     "field": [
       {
         "@type": "cr:Field",
         "name": "uid",
         "description": "The unique_id of the data record. Use an ULID string representation followed by the '_gpt_nl' su
ffix.",
         "dataType": "sc:String",
         "references": {
           "fileObject": {
             "@id": "unique_id of the data record <DO NOT KNOW WHAT THIS IS. FIX LATER>"
           },
           "extract": {
             "column": "uid"
           }
         }
       },
       {
         "@type": "cr:Field",
         "name": "text",
         "description": "The third column contains the raw text of the data record",
         "dataType": "sc:String",
         "references": {
           "fileObject": {
             "@id": "name.pdf"
           },
           "extract": {
             "column": "text"
           }
         }
       },
       {
         "@type": "cr:Field",
         "name": "meta",
         "description": "Metadata associated with each record.",
         "dataType": "sc:struct",
         "field": [
           {
             "@type": "cr:Field",
             "name": "source",
             "description": "A human readable identifier for the data source.",
             "dataType": "sc:String",
             "references": {
               "fileObject": {
                 "@id": "name.pdf"
               },
               "extract": {
                 "column": "source"
               }
             }
           },
           {
```

```
       "@type": "cr:Field",
       "name": "source_url",
       "description": "A human readable identifier for the data source.",
       "dataType": "sc:URL",
       "references": {
        "fileObject": {
         "@id": "name.pdf"
        },
        "extract": {
         "column": "source_url"
        }
       }
      },
      {
       "@type": "cr:Field",
       "name": "timestamp",
       "description": "Timestamp of the datasource extraction. String representing a datetime UTC timestamp.",
       "dataType": "sc:Timestamp",
       "references": {
        "fileObject": {
         "@id": "name.pdf"
        },
        "extract": {
         "column": "timestamp"
        }
       }
      },
      {
        "@type": "cr:Field",
        "name": "toxic_sentences",
        "description": "Sentences identified as toxic based on prediction label and score threshold.",
        "dataType": "sc:String",
        "value": [
        "This is a toxic sentence.",
        "Another harmful statement.",
        "..."
        ]
      },
      {
        "@type": "cr:Field",
        "name": "toxic_sentence_start_indices",
        "description": "Start indices of toxic sentences in the original text.",
        "dataType": "sc:Int64",
        "value": [15, 102, "..."]
      },
      {
        "@type": "cr:Field",
        "name": "toxic_labels",
        "description": "Explainable labels assigned to toxic sentences.",
        "dataType": "sc:String",
        "value": ["Hate Speech", "Insult", "..."]
      },
      {
        "@type": "cr:Field",
        "name": "toxicity_scores",
        "description": "Confidence scores for toxicity predictions.",
        "dataType": "sc:Float64",
        "value": [0.91, 0.87, "..."]
      }

    }
   ]
  }
 ]
}
```