Co-funded by the European Union

HPLT: High Performance Language Technologies

HPLT Pipelines and Tools

Deliverable number: 7.2

Version 1.0

Funded by the European Union's Horizon Europe search and innovation programme under grant agreement No 101070350 and from UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee [grant number 10052546] programme

Project details

Project Acronym:	HPLT
Project Full Title:	HPLT: High Performance Language Technologies
Year of the Call:	2021
Type of Action:	HORIZON-IA (Innovation Action)
Grant Number:	101070350
Project URL:	https://hplt-project.org

Report details

Data Management Plan	
Lead author:	Nikolay Arefev (UiO)
Contributing authors:	Andrey Kutuzov (UiO) David Samuel (UiO) Stephan Oepen (UiO) Barry Haddow (UEDIN) Laurie Burchell (UEDIN) Ona de Gibert (UH) Jaume Zaragoza-Bernabeu (Prompsit)
Internal reviewers:	David Anton (CESNET) Jan Hajic (CUNI)
Deliverable number:	7.2
Dissemination level:	Public (PU)
Contractual Delivery Date:	Aug 31, 2024
Actual Delivery Date:	Aug 31, 2024
Number of pages:	26

Document history

Version	Date	Changes
1.0	Aug 31, 2024	Original Submission

Abstract

This report provides an overview of central software components developed in the HPLT project, organized in three main categories (a) data processing, (b) language model training and (c) translation model training. All HPLT software is available under permissive open source licenses and hosted in publicly accessible repositories, which serve as the primary means of software management and community engagement. This report provides a high-level inventory of the HPLT software ecosystem and pointers to installation and running instructions. The main purpose of this documentation is to aid replicability and reuse, i.e. support prospective users in obtaining, adapting, and using relevant components of the HPLT software stack.

Contents

1 Executive Summary

1.1 Introduction

This deliverable, named *HPLT Pipelines and Tools*, provides an overview of central software components developed in the HPLT project, two years into the project duration. The presentation is organized according to three main categories: (a) data processing, (b) language model training and (c) translation model training. This report provides a high-level inventory of the HPLT software ecosystem and pointers to installation and running instructions. Some key software components have been previously documented in other project deliverables; to avoid unnecessary duplication, this report will point to earlier project documentation where appropriate. The main purpose of this documentation is to aid replicability and reuse, i.e. support prospective users in obtaining, adapting, and using relevant components of the HPLT software stack.

1.2 Brief summary of the HPLT project

The EU-funded HPLT project applies high-performance computing to scale and advance language technologies. Leveraging upon recent advances in machine learning and astonishing storage capacities, HPLT has already successfully processed huge language data sets and produced language and translation models in numerous languages. The resulting models strongly focus on smooth integration, high accuracy, and regulatory compliance concerning privacy. Unwanted biases and ethical issues are also on the roadmap. The models and data sets have already had a substantial impact in the language service market in the EU and beyond. They are open, free and available from established language repositories for anyone interested in pursuing research or innovation projects and to support industry.

The project, coordinated by the Charles University in Prague (CUNI) with partners from five different universities two HPC centres and a private NLP company, has set in place partnerships with other projects, research institutions and companies for improved results.

CHARLES UNIVERSITY	UNIVERSITY OF OSLO	UNIVERSITY OF EDINBURGH	UNIVERSITY OF TURKU
UNIVERSITY OF HELSINKI	PROMPSIT	CESNET	SIGMA2

This report encompasses software resources developed across work packages 2–7 and contributions by all partners.

2 Reflections on Software Management

Due to huge computational requirements, the HPLT project needs to utilise heterogeneous systems to perform computations: starting for the beginning of the project, we used at least LUMI, Sigma2 NIRD, CESNET and Karolina clusters. To ensure consistency and replicability of the results, it is vital to maintain the exact same software stack on multiple HPC clusters, including LUMI and other systems, and to do it in a cost-efficient manner. This is necessary for replicability of our datasets, models, and other results. However, full automation of software downloading, building (if necessary), and installation is a complex undertaking. While the original intention of the HPLT project was to standardize on the EasyBuild framework[1] for automated software deployment wherever possible, we have faced difficulties in achieving this. These difficulties are in part related to evolving policies on LUMI, and in part to increasing use of the Python libraries ecosystem. EasyBuild deliberately disallows automated dependency resolution via `pip`, Python's native package installer. This indeed ensures replicability, but it comes with the price of manually (and often redundantly) having to specify Python dependencies into EasyBuild configuration files. Replicability here clashes with the ease of maintenance.

Thus, we had to give up the uniformity of approaches to automated software deployments across different HPC clusters, and to make use of different tools for various stages of data processing and model training, according to current state-of-the-art. In particular:

1. `warc2text` (our tool of choice for raw text extraction from WARC files, implemented in C++, later described in section 3.2.1.1) is deployed on the storage systems in the Czech Republic and Norway with EasyBuild.[2]
2. Further text processing, language identification, boilerplate removal, and normalization (so called 'Stage 2') is relying heavily on Python ecosystem and is deployed using `pip`.
3. Pipelines for training large language models also rely on Python internal mechanisms, but additionally make heavy use of Singularity containers, as recently encouraged by LUMI administrators[3]. Containers are well-suited for HPC environments, since they reduce strain on parallel file systems by minimizing the number of files, but containers challenge modularity and module reuse.

The specific deployment procedures are described in the respective GitHub repositories (see the sections below) There are still many open questions remaining regarding full automation of software download and installation on HPC clusters, but the HPLT project did learn a lot from this experience. We hope the recipes we provide will be helpful for other users.

[1] https://easybuild.io/
[2] https://github.com/bitextor/warc2text/tree/master/easyconfigs
[3] https://docs.lumi-supercomputer.eu/software/containers/singularity/

3 Data Processing Software

3.1 Data Ingestion

Internet Archive[1] is the main source of raw web crawls in the HPLT project. Additionally, we employ crawls from Common Crawl[2] which is a popular source of crawls for training language models. Both sources provide crawls in the Web ARChive (WARC) format among others, which is consumed by our data processing software.

3.1.1 Internet Archive

Internet Archive requires using a custom API for user authentication and data downloading. Command line tools[3] suggested to the users by Internet Archive provide low-level means of accessing their crawls, but lack the ability of automatic reliable downloading of a set of web crawls. To automate downloading of web crawls from Internet Archive we have developed special software.[4] It allows downloading crawls in many parallel threads, making as many attempts as required to download all files successfully.

Using this software, nine Internet Archive crawls with the total size of about 3.4 PB crawled over the years 2012-2020 have been downloaded as the source of data for the latest HPLT data release. We found it beneficial to run massively parallel downloading in up to 2000 parallel threads. The downloading process was running in two data centres, NIRD and CESNET. The speed varied from 12 TB/day to 35 TB/day depending on the number of threads and the current availability and overall load of the Internet Archive servers.

3.1.2 Common Crawl

For downloading Common Crawl (CC), no specific software was developed as it is publicly available for downloading through the standard HTTP protocol. We employed the command line download manager Aria2, which supports multi-thread downloading and automatic retries on failures.[5] It takes a file containing a list of URLs of WARC files to download, which in its turn can be downloaded directly from the Common Crawl website.[6] However, to facilitate downloading many crawls at once, helper scripts were developed that generate file lists for Aria2 from a list of crawls.[7]

Using the described software, 12 randomly selected CC versions crawled over the years 2014–2022 were downloaded, their total size is 738 TB. The downloading process was going in 100 parallel threads.[8] The downloading speed was 20 TB/day on average.

[1] https://archive.org/
[2] https://commoncrawl.org/
[3] https://archive.org/developers/internetarchive/cli.html
[4] https://github.com/hplt-project/ia-download
[5] https://aria2.github.io/, version 1.36.0 was used
[6] E.g. https://data.commoncrawl.org/crawl-data/CC-MAIN-2022-49/index.html provides files with metadata related to one of the CC crawls, including a list of URLs of the WARC files this crawl consists of.
[7] https://github.com/hplt-project/cc-download
[8] Due to the limitations on the number of simultaneously opened files in one of the data centres we were not able to significantly increase it.

3.2 Raw Text Extraction

Extraction of raw texts from WARC archives is performed in two stages.[9] In the first stage, WARC files are processed and relevant records are selected. From these records, web pages in HTML format and additional metadata are extracted and dumped to disk. This stage requires voluminous storage and fast access to WARC files but it is not computationally intensive. Thus, this stage was run in the data centers on the compute nodes directly attached to the storage with WARC files.

The second stage consists of parsing HTML pages, removing boilerplate and extracting text in natural language, and finally running language identification on the extracted text. This stage is significantly more computationally intensive, thus, it was run on the LUMI cluster.

3.2.1 Stage 1: WARC Parsing and Document Extraction

The goal of Stage 1 (a.k.a. warc2html) is to extract HTML code and additional information about documents in the downloaded crawls. Specifically, it selects relevant WARC records by their record type and content type, filters out documents from a black list of URLs, and finally dumps HTMLs and metadata to the output JSON files.

3.2.1.1 The warc2text tool

The tool warc2text[10] was previously developed and used in the ParaCrawl project. It was written in C++ with efficiency in mind to be able to scale to several petabytes of input data. For HPLT, it has been adapted to extract substantially more metadata (including document URLs, paths to the source WARC files and record positions inside, content types, timestamps) and support different output formats (in particular, ZSTD-compressed JSONlines used in the latest HPLT data release). Also, options to skip language identification and text extraction were added to adapt the tool to the next stages of text extraction. Finally, an optional processing step is added that dumps all WARC records having URLs ending with “robots.txt” to a separate WARC file for future filtering.

A more detailed list of the changes can be found on GitHub.[11]

3.2.1.2 Parallel processing of a set of crawls

To process a large volume of web crawls with warc2text a set of helper scripts were developed.[12] They help efficiently process all downloaded web crawls on several compute nodes directly attached to the storage in one of the data centres. This processing procedure consists of several steps.

1. Inspect the downloaded crawls and discover WARC files to process.

[9]The software for raw text extraction developed and used for preparation of the latest HPLT data release along with its technical description is published in https://github.com/hplt-project/warc2text-runner/tree/main/two.

[10]https://github.com/bitextor/warc2text

[11]Version 1.2.0 of warc2text was employed for the latest data release https://github.com/bitextor/warc2text/releases/tag/v1.2.0

[12]https://github.com/hplt-project/warc2text-runner/blob/main/two/README.MD#stage1-aka-warc2html

2. Combine the discovered WARC files into batches to reduce the number of output files in order to fit the limitations of the file system. We used batches of 1000 WARC files each. For each batch about 200 output folders are generated, one per detected language, and 3 files are dumped to each folder: `html.zst` contains extracted HTMLs, `metadata.zst` contains additional information about each document, `robotstxt.warc.gz` contains WARC records with robots.txt files for further filtering of documents disallowed for crawling.
3. Run warc2text in many parallel processes on several compute nodes. We used 2 nodes with 250 process on NIRD and 3 nodes with 60 processes on CESNET.

As the result of applying this procedure to the crawls downloaded for the latest HPLT data release 0.93 PB of data (mostly compressed HTMLs) were extracted from about 4.45 PB of web crawls, almost 5× reduction in size.

3.2.2 Stage 2: Text Extraction & Language Identification

The goal of Stage 2 (a.k.a. html2text) is to build a set of monolingual corpora for the selected natural languages from HTML pages extracted in Stage 1. More specifically, the following tasks are solved during Stage 2 of text extraction.

1. Parsing HTMLs and building their tree representations required in the following processing steps.
2. Removing boilerplate, i.e. parts of web pages that do not represent their main content, and extracting text in natural language from the remaining parts (see 3.2.3).
3. Identifying the language of each document and adding the extracted text to the corresponding corpus (see 3.2.4).
4. Detecting machine-translated texts by searching indicative tags and attributes in the HTML trees.[13]

These tasks are solved by the pipeline developed specifically for processing data for the latest HPLT data release on LUMI.[14] On the highest level the pipeline consists of two processing steps running in parallel: the first step[15] does HTML parsing and everything requiring the parsed tree, i.e. text extract with boilerplate removal and detection of machine-translated texts, while the second step[16] takes the extracted text and identifies its language. The first processing step is about 10 times slower than the second one. Thus, when running this pipeline for the second data release out of 250 workers on each LUMI node 10% were running the second step and the rest were running the first step. Within a single node, different workers were processing different documents from the same `html.zst` file. Parallelization across different compute nodes is trivial, with each node processing a dedicated set of `html.zst` files.

[13] The following patterns are used: https://github.com/hplt-project/warc2text-runner/blob/main/src/warc2text_runner/two/tagfilter/mt-filter-list.annotated

[14] Version 2.0.0 was employed: https://github.com/hplt-project/warc2text-runner/releases/tag/v2.0.0. See https://github.com/hplt-project/warc2text-runner/blob/main/two/README.MD#stage2-aka-html2text for more information about running this pipeline.

[15] https://github.com/hplt-project/warc2text-runner/blob/main/src/warc2text_runner/two/trafilatura/traf.py

[16] https://github.com/hplt-project/warc2text-runner/blob/main/src/warc2text_runner/two/fastertext_lid/proto_langid.py

Two important limitations we had to address are the limited storage space on LUMI that did not fit all outputs from Stage 1, and also much slower file transfer between data centres and LUMI compared to the speed of data processing on LUMI. To address these limitations, we have implemented an alternative method of running Stage 2 that simultaneously downloads and processes data in portions fitting into the available storage space. Another problem we faced was disproportionately long waiting time in the LUMI queue compared to the time of actual data processing. To address this problem, we had to carefully select the number of `html.zst` files processed by each task. On one hand, increasing this number reduces the total number of tasks, and thus, the waiting time in the LUMI queue. On the other hand, there is an upperbound because each task should fit into the LUMI maximum processing time of 2 days. Based on the observation that the processing time depends more on the total size rather than the number of input files, we split all `html.zst` files into batches of approximately the same size and assigned one batch to each task. We experimentally set the batch size to 1500 GB.

Running Stage 2 for the latest HPLT data release resulted in about 62 TB of compressed raw texts and 1 TB of compressed language information. This is roughly 15× reduction in size compared to outputs of Stage 1 and 75× reduction compared to the originally downloaded crawls.

3.2.3 Stage 2: Text Extraction with Boilerplate Removal

Text extraction with boilerplate removal is implemented using the Trafilatura[17] library [1]. The extraction algorithm is based on a bunch of rules and heuristics, and in case they did not work it falls back to using other text extraction libraries readability-lxml[18] and jusText[19].

We called the text extraction function of Trafilatura with non-default values of some arguments, they are listed in table 3.1. Namely, we skip text from HTML tables and comments, do not disable the fallbacks to other extraction libraries but disable the fall back to Trafilatura's own simple extraction baseline in case the extracted text is shorter than MIN_EXTRACTED_SIZE.[20]

Name	Value
include_comments	False
include_tables	False
no_fallback	False
MIN_EXTRACTED_SIZE	0

Table 3.1: Arguments of Trafilatura explicitly specified. All other arguments have their default values.

3.2.4 Stage 2: Document Language Identification

The language identification component is based primarily on work published in [2], incorporating further refinements described in [3].

[17] https://trafilatura.readthedocs.io, version 1.8.0
[18] https://github.com/buriy/python-readability
[19] https://github.com/miso-belica/jusText
[20] We found that the outputs of the baseline method are very noisy, so it is better to return nothing for a document rather fall back to this baseline.

3.2.4.1 Language identification model training data

We used the OpenLID dataset as training data for the language identification component [2], [21] with the addition of a corpus of Saudi dialect social media data to improve Arabic coverage [4]. All data sources in OpenLID are either released under an open license for research purposes or described as free to use.

We found that some of the sentences in non-Latin scripts in the original OpenLID dataset were not split correctly, so we wrote an updated version of the preprocessing script based on the LASER toolkit [22]. It uses a number of existing external sentence splitters to improve splitting for Ge'ez, Ol Chiki, Burmese, Khmer, Lao, and Tibetan. After sentence splitting, we ran a data pre-processing pipeline which normalised punctuation and replaced non-printing characters with the `sacremoses` library, normalised unicode data with the `unicodedata` library, and removed emoji. Further details are available in [3].

We noticed particularly poor performance for Arabic dialects compared even to other close language varieties (e.g. Croatian and Serbian). Analysis in [3] found that whilst Arabic dialects are easy to distinguish from other languages, it is difficult to distinguish Arabic dialects from each other. We therefore combined all Arabic dialects into one class (Arabic macrolanguage, (`ara_Arab`)). Otherwise, we followed the same language labels for language identification as used in [2]. Prior to training, we followed [5] and [6], and sampled proportionally to $p_l^{0.3}$, where p_l is the fraction of sentences in the dataset which are in language l. This aims to ameliorate class skew issues.

The final version of the training data contains 193 language classes and 121 million sentences. The mean number of sentences in each language class is 602K but there is significant skew: prior to sampling, the largest class (English) contains 7.5 million sentences and the smallest (South Azerbaijani) contains 532 sentences. A full list of the languages covered is available in [2].

3.2.4.2 Language identification model architecture

We used the above dataset to train a FastText language identification model, which is a standard tool for training such models and easily reproducible [7]. It embeds character-level n-grams from the input text, and then uses these as input to a multiclass linear classifier. We used the same hyperparameters as [6] given in table 3.2. Any hyperparameters not listed are set to FastText defaults. We used the Python interface of version 0.9.2 of the tool, which is the current stable version.[23]

3.2.4.3 Inference

At inference time, we kept preprocessing minimal for speed, only removing digits and any characters that were not space separators or included in the Unicode Word character category. For each valid input, the module returns a Python dictionary containing the three language labels with the highest predicted probabilities. Table 3.3 gives an example of input and output. In the case that the input is

[21] https://github.com/laurieburchell/open-lid-dataset
[22] https://github.com/facebookresearch/LASER/blob/main/utils/src/sentence_split.py
[23] https://github.com/facebookresearch/fastText/

Hyperparameter	Setting
Loss	Softmax
Epochs	2
Learning rate	0.8
Embedding dim.	256
Min. no. word occurrences	1000
Character n-grams	2–5
Word n-grams	1
Bucket size	1M
Threads	68

Table 3.2: Hyperparameters for the *fasttext* language identification model. Any hyperparameters not given are default.

empty text or None (as a result of preprocessing with Trafilatura; see above), the language identification module outputs `{lang: None}` as a Python dictionary.

Input:	`This is an example sentence.`
Output:	`{lang: [eng_Latn, sun_Latn, ibo_Latn],`
	`prob: [0.9939, 0.0021, 0.0020]}`

Table 3.3: An example input and output of the language module.

3.3 Monotexting: Cleaning & Filtering of Monolingual Text

The monotexting pipeline takes care of joining the metadata and text produced by Stage 2, deduplication of documents and adding more metadata for further cleaning of the data.

The main steps of the pipeline are:

1. Merge extracted text, WARC metadata fields and language identification metadata into a single JSON per document. The data after this step is divided into language directories and collection[24] directories.
2. Remove near-duplicates for each language within each collection.
3. Annotate each document with additional metadata for quality estimation and/or cleaning.
4. Create an additional cleaned version of the dataset, removing all the documents not meeting the cleaning criteria.[25]

For further explanation of the above steps, installation, execution of the pipeline and output format, please refer to the GitHub project page.[26]

The job submission logic for the monotexting pipeline has been substantially rewritten, compared to the first data release, to be more friendly to the LUMI cluster scheduler and filesystem. Therefore, the pipeline uses HyperQueue[27] for job submission of the deduplication and annotation steps. This

[24] A collection is a crawl or a set of crawls. For the Internet Archive, each crawl (wide5, wide6 etc.) is a collection itself. Instead, for Common Crawl, we join all the crawls within a year in a collection (all crawls from 2014 in cc14, all crawls from 2015 in cc15 and so on).
[25] `https://github.com/hplt-project/monotextor-slurm#cleaning`
[26] `https://github.com/hplt-project/monotextor-slurm/tree/v2.0`
[27] `https://github.com/It4innovations/hyperqueue/tree/v0.19.0`

alleviates the overload of the LUMI Slurm scheduler, reducing the number of submitted Slurm jobs from several thousand to a hundred jobs (or as many HyperQueue workers the user decides to use). Also, it drastically reduces the number of log files and requests to the metadata server of the LUMI filesystem because HyperQueue keeps a single log file for each of the steps. Finally, one last advantage is an easier job submission and monitoring for the user, being able to submit an entire step with a single command.

For the software deployment, monotexting pipeline now uses Singularity containers. This also reduces the amount of files, as all the installed packages are inside a single container file, compared to the previous software deployment in the first data release, where all the compiled Rust binaries and Python packages were installed in the cluster filesystem. Additionally, reproducibility of the software has dramatically increased, as the built packages inside the container do not depend on the cluster software stack available at the time the pipeline is deployed.

3.4 Bitexting: Discovery and Refinement of Parallel Texts

For parallel data discovery, there have not been any new software development. The pipeline that has been used (cirrus-scripts), however, comes from a previous European project: ParaCrawl. So the main job during this project has been adapting the installation procedure and environment to the LUMI cluster. The relevant changes can be found in the LUMI branch[28] of the pipeline repository.

Additionally to the repository README documentation, a setup guide[29] for LUMI has been written.

[28] https://github.com/paracrawl/cirrus-scripts/tree/d822f608c4ae5e1b0ead8c8c445d0c10e6b50c62
[29] https://github.com/paracrawl/cirrus-scripts/wiki/LUMI-setup

4 Language Model Training

This section describes the training framework for two types of language models used in the HPLT project – small encoder-only models and large decoder-only models.

4.1 Encoder-Only Language Models

We develop a suite of encoder-only language models for all 75 languages supported in the HPLT project initial release. These models are also known as masked language models (MLMs) after the training objective used in the first such model, BERT [8].[1] While these models can be used similarly to the generative decoder-only models [9], their primary use in the present days is as efficient representation learners – then they can be directly utilized to encode text segments (i.e. for document retrieval) or further trained on downstream tasks (such as morphosyntactic parsing, sentiment analysis or reading comprehension) – substantially outperforming decoder-only models of the same size for such use-cases [8, 10].

Model architecture As part of the HPLT project, we developed a data-efficient language model, which we call LTG-BERT [11]. We demonstrated that this model can be trained on a text corpus containing only 100 million words with comparable performance to the original BERT model trained on much larger dataset [11]. Therefore, we can support even the most low-resource languages covered by the HPLT project and train a monolingual masked language model for each of them.

Implementation The training pipeline for these models contains all steps from taking the raw HPLT corpus all the way up to releasing a Hugging Face-compatible language model (in our case available at `https://huggingface.co/HPLT`). We believe that it is important to not only openly release the model weights but also the source code that enables anyone to reproduce (and further improve upon) our results; the source code in available at `https://github.com/hplt-project/HPLT-WP4`. The entry-point of the repository is the file `schedule.sh`, which initializes a cascade of parallelized SLURM scripts that performs the following steps: ① sharding the raw corpus into manageable chunks, ② training a subword tokenizer, ③ tokenizing the shards, and ④ training an LTG-BERT model on the tokenized corpus. ⑤ Then the user can convert the model checkpoint into a Hugging Face-compatible directory, and ⑥ evaluate it on morphosyntactic parsing.

4.1.1 Corpus sharding

As the first step, we split the raw text corpus into chunks of manageable size. These chunks can then be all processed in parallel. We set the size of each chunk so that it can be fully loaded into working memory when training the language model. The entry point for corpus sharding is `preprocessing/shard_worker.sh`.

[1]Note that these two terms are not entirely equivalent – ‘*encoder-only*‘ refers to the bidirectional architecture, while ‘*masked*‘ refers to the fill-in-the-blanks training objective. But in practice (and in our case), they can be used interchangeably.

4.1.2 Training a subword tokenizer

All relevant modern language models ingest text encoded as subword-units, using a fixed vocabulary of possible subwords. This vocabulary has to first be initialized from a text corpus, in particular we use a greedy heuristic for searching for a semi-optimal vocabulary called WordPiece [8]. We use vocabulary size of $2^{15} = 32\,768$ subword tokens, other details are defined in `preprocessing/train_tokenizer.py`. A new tokenizer can be trained by running `preprocessing/train_tokenizer.sh`.

4.1.3 Tokenizing the corpus

We then use WordPiece encoding to segment each corpus shard into subwords (based on the trained subword vocabulary). This step has two reasons: to not slow down LM training by concurrent tokenization and to shrink the size of the training corpus. Tokenization can be called by running `preprocessing/tokenize_shards.sh`.

4.1.4 Language model training

The encoder-only language models are trained according to the LTG-BERT training recipe [11]. The model architecture and all training details are fully defined in the `encoder-only/` directory. The training script is written on top of the PyTorch framework.[2] It takes advantage of the computational resources on the LUMI supercomputer by distributing the training across 64 AMD MI250x GPUs, significantly speeding up the training process. It can be started by running `encoder-only/train.sh`.

4.1.5 Hugging Face conversion

The `--output_dir` argument of the previous script defines the directory with saved model checkpoints. The weights defined in these checkpoints can then be converted into Hugging Face-compatible models by running `encoder-only/convert_to_hf.py`.

4.1.6 Evaluation

The compatibility with Hugging Face API allows the user to easily evaluate the trained language models on numerous downstream tasks.[3] We use morphosyntactic parsing as an example, primarily because this task is available in a unified format for most of the 75 supported languages [12].[4] The directory in `evaluation/ud/` defines a morphosyntactic parser; `evaluation/ud/train.sh` finetunes a pretrained language model on morphosyntactic parsing and then evaluates its performance on a held-out dataset.

[2] https://pytorch.org/
[3] https://huggingface.co/docs/transformers/training
[4] https://universaldependencies.org/

4.2 Decoder-Only Language Models

Several iterations of software tools have been applied in the project for the purpose of training decoder-only (or causal) language models, building on the Megatron[5] and Megatron-DeepSpeed[6] frameworks. These tools have been used to create the FinGPT model family[7] and the 176B BLUUMI model[8] [13], the Poro 33B model[9] [14], as well as the Viking model family,[10] and are currently used to create models supporting all EU languages for a future release. All software is forked from Megatron-LM and therefore leverages its mechanics, such as model parallelism. The reason for variance in used software stacks is the throughput experimentation and feature availability in the LLM pretraining open-source software at the time of testing.

Model architecture The FinGPT model family and the BLUUMI model follow the model architecture of the GPT3 model family [10] as modified for the BLUUMI model [15], using ALiBi position embeddings [16] and additional layer normalization [17]. The FinGPT models range from 186M to 13B parameters, and Poro inherits the same architecture while increasing the number of parameters to 34B.

The Viking model family consists of multilingual models of sizes of 7B, 13B and 33B parameters, following the Llama 2 architecture [18]. This differs from the architecture of the FinGPT and Poro models in using rotary positional embeddings [19] instead of ALiBi, and a SwiGLU activation function instead of GELU. The 33B parameter model also uses Group Query Attention for inference efficiency [20].

Implementation The process to create the decoder-only language models involves a range of tools for data preprocessing, tokenizer creation, data formatting for training, model training and evaluation, as well as the conversion of the final checkpoints into a format compatible with the widely used Hugging Face `transformers` library[11] for release. We make the tools used for all of these steps available under open source licences in the repositories identified below, and detail the processing steps and the scripts used for specific steps in the following sections.

- The version of Megatron-DeepSpeed used to train the FinGPT and Poro models is available at `https://github.com/TurkuNLP/Megatron-DeepSpeed/tree/fingpt`
- The version of Megatron-DeepSpeed used to train the Viking models is available at `https://github.com/LumiOpen/Megatron-DeepSpeed`
- A version of Megatron-LM adapted for LUMI and used for training upcoming models is available at `https://github.com/LumiOpen/Megatron-LM-lumi`

[5] https://github.com/NVIDIA/Megatron-LM
[6] https://github.com/microsoft/Megatron-DeepSpeed
[7] https://huggingface.co/TurkuNLP/gpt3-finnish-13B
[8] https://huggingface.co/TurkuNLP/bloom-finnish-176b
[9] https://huggingface.co/LumiOpen/Poro-34B
[10] https://huggingface.co/LumiOpen/Viking-33B
[11] https://huggingface.co/docs/transformers

4.2.1 Training a BPE-tokenizer

All decoder-only models created in the project use Byte Pair Encoding (BPE) subword tokenizers trained on samples of the data used to pretrain their respective models. The script to train these tokenizers is available from https://github.com/TurkuNLP/finngen-tools/blob/main/create_bloom_tokenizer.py.

For the Poro tokenizer, we added special tokens for code as in StarCoder.[12] We also included special tokens of `<|user|>` and `<|assistant|>` to improve chat performance. The Viking tokenizer adopted OpenAI's Chat Markup Language (ChatML)-tokens to change these to `<|im_start|>` and `<|im_end|>`. As these changes are easy to apply to the given script by simply changing the contents of the `ADDITIONAL_SPECIAL_TOKENS`, we only provide the script matching the Poro-tokenizer.

All of the FinGPT, Poro and Viking-tokenizers utilize a unique vocabulary of $2^{17} = 131072$ tokens.

4.2.2 Tokenizing the corpus

For the purposes of the primary pretraining process, the source text corpora are first tokenized and converted from their original JSON-encoded plain text format to a binary input format that can be read efficiently in a random access manner by workers in the pretraining process. While this tokenization and conversion only requires CPU processing and is lightweight in comparison to the pretraining process, it is often convenient to convert parts of the source texts separately and then later merge the binary files into larger components for pretraining. The script for tokenization and conversion is made available at https://github.com/TurkuNLP/Megatron-DeepSpeed/blob/fingpt/tools/preprocess_data.py and the script for merging binary input files at https://github.com/TurkuNLP/Megatron-DeepSpeed/blob/fingpt/tools/merge_preprocessed_data.py.

4.2.3 Language model training

All decoder-only models are trained using the GPU cluster of the LUMI supercomputer, consisting of compute nodes with four MI250x GPUs each. The training configurations for the **FinGPT-** and **BLUUMI**-models mimic the configurations of GPT-3, which are detailed in the FinGPT paper [13]. Poro follows the original Llama training recipe [21], as detailed in the Poro paper [14]. FinGPT models were trained for 300 billion tokens and Poro for 1 trillion tokens, with training scaled up to 128 nodes (512 MI250x GPUs) for Poro. For decoder-only training, the main Python-script is `pretrain_gpt.py` and instructions for training on LUMI can be found from the `README.md` at https://github.com/TurkuNLP/Megatron-DeepSpeed/tree/fingpt.

The Viking family of models follows the recipe introduced in [18], and is trained for 2 trillion tokens. For Viking we adopted Microsoft's Megatron-DeepSpeed[13] to enable features such as FlashAttention[14]. The 33B model training was scaled to 256 nodes totalling to 1024 MI250x GPUs. As before, the

[12]https://huggingface.co/bigcode/starcoder
[13]https://github.com/LumiOpen/Megatron-DeepSpeed
[14]https://github.com/ROCm/flash-attention/tree/flash_attention_for_rocm

main Python script is `pretrain_gpt.py`. We also provide SLURM scripts for Viking models named `pretrain_viking_{7,13,33}B.sh` as a reference implementation.

4.2.4 HuggingFace conversion

Model checkpoints were converted into formats compatible with the Hugging Face `transformers` library using tools published with the libraries used for conversion. Specifically, for the FinGPT and Poro models, the script https://github.com/TurkuNLP/Megatron-DeepSpeed/blob/fingpt/tools/convert_checkpoint/deepspeed_to_transformers.py was used, and for Viking models, the script https://github.com/jonabur/Megatron-DeepSpeed/tree/main/tools/convert_checkpoint/viking_conversion/ was used.

4.2.5 Evaluation

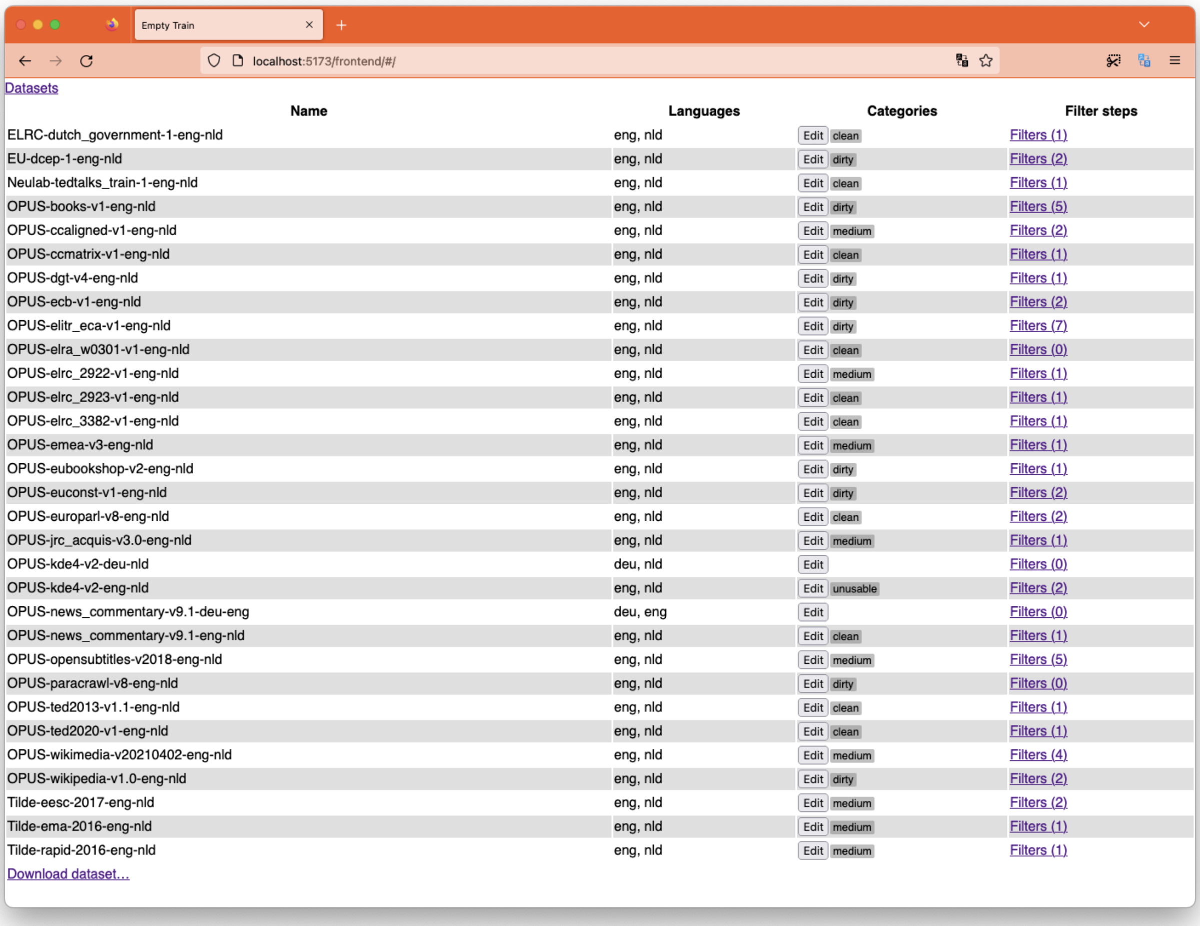
Decoder-only language models were evaluated conventionally using zero- and few-shot task settings, without finetuning the models after their initial pretraining. As the models were converted into a format compatible with the Hugging Face `transformers` library, it was possible to evaluate them using standard tools introduced by the community. Newly introduced evaluation datasets such as FIN-bench[15] were formatted to conform with the input specifications of the evaluation tools. For evaluating the FinGPT family of models, the BIG-bench evaluation tools[16] were used, and to evaluate the Poro, Viking, and subsequent models, the EleutherAI LM eval harness[17] was used.

[15] https://github.com/TurkuNLP/FIN-bench
[16] https://github.com/google/BIG-bench
[17] https://github.com/EleutherAI/lm-evaluation-harness

5 Translation Model Training

In this chapter, we describe our software tools for creating machine translation (MT) models. This includes tools for selecting and cleaning the data (5.1); for managing the training pipeline (5.2), for scheduling and feeding data (5.3) and for producing efficient models for deployment (5.4). So far our tools have targetted encoder-decoder architectures, but given the recent success of decoder-only architectures for MT [22, 23, 24], we are investigating how to extend our tools to support these systems too.

5.1 Data Selection and Cleaning: OpusCleaner



Name	Languages	Categories	Filter steps
ELRC-dutch_government-1-eng-nld	eng, nld	Edit clean	Filters (1)
EU-dcep-1-eng-nld	eng, nld	Edit dirty	Filters (2)
Neulab-tedtalks_train-1-eng-nld	eng, nld	Edit clean	Filters (1)
OPUS-books-v1-eng-nld	eng, nld	Edit dirty	Filters (5)
OPUS-ccaligned-v1-eng-nld	eng, nld	Edit medium	Filters (2)
OPUS-ccmatrix-v1-eng-nld	eng, nld	Edit clean	Filters (1)
OPUS-dgt-v4-eng-nld	eng, nld	Edit dirty	Filters (1)
OPUS-ecb-v1-eng-nld	eng, nld	Edit dirty	Filters (2)
OPUS-elitr_eca-v1-eng-nld	eng, nld	Edit dirty	Filters (7)
OPUS-elra_w0301-v1-eng-nld	eng, nld	Edit clean	Filters (0)
OPUS-elrc_2922-v1-eng-nld	eng, nld	Edit medium	Filters (1)
OPUS-elrc_2923-v1-eng-nld	eng, nld	Edit clean	Filters (1)
OPUS-elrc_3382-v1-eng-nld	eng, nld	Edit clean	Filters (1)
OPUS-emea-v3-eng-nld	eng, nld	Edit medium	Filters (1)
OPUS-eubookshop-v2-eng-nld	eng, nld	Edit dirty	Filters (1)
OPUS-euconst-v1-eng-nld	eng, nld	Edit dirty	Filters (2)
OPUS-europarl-v8-eng-nld	eng, nld	Edit clean	Filters (2)
OPUS-jrc_acquis-v3.0-eng-nld	eng, nld	Edit medium	Filters (1)
OPUS-kde4-v2-deu-nld	deu, nld	Edit	Filters (0)
OPUS-kde4-v2-eng-nld	eng, nld	Edit unusable	Filters (2)
OPUS-news_commentary-v9.1-deu-eng	deu, eng	Edit	Filters (0)
OPUS-news_commentary-v9.1-eng-nld	eng, nld	Edit clean	Filters (1)
OPUS-opensubtitles-v2018-eng-nld	eng, nld	Edit medium	Filters (5)
OPUS-paracrawl-v8-eng-nld	eng, nld	Edit dirty	Filters (0)
OPUS-ted2013-v1.1-eng-nld	eng, nld	Edit clean	Filters (1)
OPUS-ted2020-v1-eng-nld	eng, nld	Edit clean	Filters (1)
OPUS-wikimedia-v20210402-eng-nld	eng, nld	Edit medium	Filters (4)
OPUS-wikipedia-v1.0-eng-nld	eng, nld	Edit dirty	Filters (2)
Tilde-eesc-2017-eng-nld	eng, nld	Edit medium	Filters (2)
Tilde-ema-2016-eng-nld	eng, nld	Edit medium	Filters (1)
Tilde-rapid-2016-eng-nld	eng, nld	Edit medium	Filters (1)

Download dataset...

Figure 5.1: Listing datasets with OpusCleaner

Our tool for selecting and filtering parallel data for machine translation is OpusCleaner. This is available on GitHub[1] and also mentioned in our earlier deliverable (*D3.1 Software for cleaning data sets*)[2] and

[1]https://github.com/hplt-project/OpusCleaner
[2]https://hplt-project.org/HPLT_D3_1___Software_for_cleaning_data_sets.pdf

described in our technical report [25]. For detailed instructions on installing and running OpusCleaner we refer the reader to our GitHub repository; here we offer an overview of the purpose and operation of the tool.

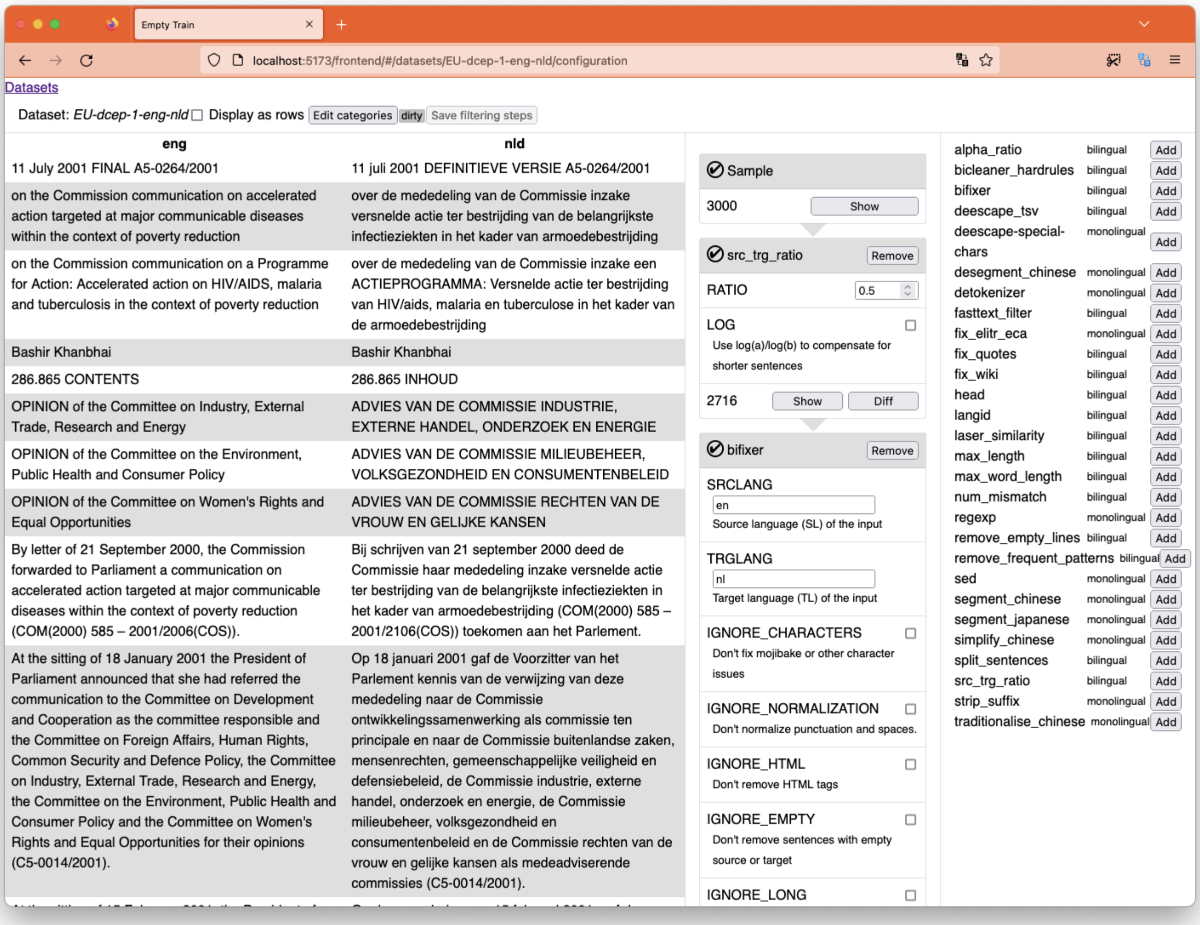
OpusCleaner consists of a frontend (written using the Vue[3] framework), an API and a command-line interface. An alternative, terminal-based frontend (Cli-a-NER) has also been developed in HPLT, and is also available in a separate repository[4]. The general idea is that the system builder can use the frontend to explore and select parallel datasets, to develop filters and observe their effects, and finally to save the results in a configuration file. This configuration file can then be used in the training pipeline to download datasets and run filters, enabling reproducible results.

In Figure 5.1 we show the interface for selecting datasets. The user supplies a language pair, and OpusCleaner will list the available parallel datasets from sources such as Opus. The user can then choose which datasets they would like to include in the project, and these are downloaded.

Once datasets are downloaded, OpusCleaner allows the user to explore the datasets (to assess their quality and topic, for instance) and to select and apply filters. For exploration, the OpusCleaner interface is able to show randomly selected portions of the datasets, and for filtering the interface allows the user to select filters from a large list and observe the effect. OpusCleaner supports all filters from the existing tool OpusFilter[5] and supplies some of its own. The user can label datasets according to their quality (labels that can be used later in OpusTrainer (5.3) and finally save the dataset and filter configuration file. Figure 5.2 shows the filter exploration interface of OpusCleaner.

Given an OpusCleaner configuration file (which uses the JSON format) the OpusCleaner command-line tools can be used to download the datasets referenced in the file, and apply the set of filters. The idea is that this can be used in a server-based workflow to build MT systems in a reproducible pipeline. OpusCleaner configuration files can be shared and published, enabling others to copy the data preparation steps, and to build on and extend the model-training pipelines.

[3]https://vuejs.org/
[4]https://github.com/hplt-project/clianer
[5]https://github.com/Helsinki-NLP/OpusFilter



Empty Train
localhost:5173/frontend/#/datasets/EU-dcep-1-eng-nld/configuration
Datasets

Dataset: *EU-dcep-1-eng-nld* ☐ Display as rows | Edit categories | dirty | Save filtering steps

eng	nld
11 July 2001 FINAL A5-0264/2001	11 juli 2001 DEFINITIEVE VERSIE A5-0264/2001
on the Commission communication on accelerated action targeted at major communicable diseases within the context of poverty reduction	over de mededeling van de Commissie inzake versnelde actie ter bestrijding van de belangrijkste infectieziekten in het kader van armoedebestrijding
on the Commission communication on a Programme for Action: Accelerated action on HIV/AIDS, malaria and tuberculosis in the context of poverty reduction	over de mededeling van de Commissie inzake een ACTIEPROGRAMMA: Versnelde actie ter bestrijding van HIV/aids, malaria en tuberculose in het kader van de armoedebestrijding
Bashir Khanbhai	Bashir Khanbhai
286.865 CONTENTS	286.865 INHOUD
OPINION of the Committee on Industry, External Trade, Research and Energy	ADVIES VAN DE COMMISSIE INDUSTRIE, EXTERNE HANDEL, ONDERZOEK EN ENERGIE
OPINION of the Committee on the Environment, Public Health and Consumer Policy	ADVIES VAN DE COMMISSIE MILIEUBEHEER, VOLKSGEZONDHEID EN CONSUMENTENBELEID
OPINION of the Committee on Women's Rights and Equal Opportunities	ADVIES VAN DE COMMISSIE RECHTEN VAN DE VROUW EN GELIJKE KANSEN
By letter of 21 September 2000, the Commission forwarded to Parliament a communication on accelerated action targeted at major communicable diseases within the context of poverty reduction (COM(2000) 585 – 2001/2006(COS)).	Bij schrijven van 21 september 2000 deed de Commissie haar mededeling inzake versnelde actie ter bestrijding van de belangrijkste infectieziekten in het kader van armoedebestrijding (COM(2000) 585 – 2001/2106(COS)) toekomen aan het Parlement.
At the sitting of 18 January 2001 the President of Parliament announced that she had referred the communication to the Committee on Development and Cooperation as the committee responsible and the Committee on Foreign Affairs, Human Rights, Common Security and Defence Policy, the Committee on Industry, External Trade, Research and Energy, the Committee on the Environment, Public Health and Consumer Policy and the Committee on Women's Rights and Equal Opportunities for their opinions (C5-0014/2001).	Op 18 januari 2001 gaf de Voorzitter van het Parlement kennis van de verwijzing van deze mededeling naar de Commissie ontwikkelingssamenwerking als commissie ten principale en naar de Commissie buitenlandse zaken, mensenrechten, gemeenschappelijke veiligheid en defensiebeleid, de Commissie industrie, externe handel, onderzoek en energie, de Commissie milieubeheer, volksgezondheid en consumentenbeleid en de Commissie rechten van de vrouw en gelijke kansen als medeadviserende commissies (C5-0014/2001).

Sample
3000 | Show

src_trg_ratio | Remove
RATIO 0.5
LOG ☐
Use log(a)/log(b) to compensate for shorter sentences
2716 | Show | Diff

bifixer | Remove
SRCLANG
en
Source language (SL) of the input
TRGLANG
nl
Target language (TL) of the input
IGNORE_CHARACTERS ☐
Don't fix mojibake or other character issues
IGNORE_NORMALIZATION ☐
Don't normalize punctuation and spaces.
IGNORE_HTML ☐
Don't remove HTML tags
IGNORE_EMPTY ☐
Don't remove sentences with empty source or target
IGNORE_LONG ☐

alpha_ratio	bilingual	Add
bicleaner_hardrules	bilingual	Add
bifixer	bilingual	Add
deescape_tsv	bilingual	Add
deescape-special-chars	monolingual	Add
desegment_chinese	monolingual	Add
detokenizer	monolingual	Add
fasttext_filter	bilingual	Add
fix_elitr_eca	monolingual	Add
fix_quotes	bilingual	Add
fix_wiki	bilingual	Add
head	bilingual	Add
langid	bilingual	Add
laser_similarity	bilingual	Add
max_length	bilingual	Add
max_word_length	bilingual	Add
num_mismatch	bilingual	Add
regexp	monolingual	Add
remove_empty_lines	bilingual	Add
remove_frequent_patterns	bilingual	Add
sed	monolingual	Add
segment_chinese	monolingual	Add
segment_japanese	monolingual	Add
simplify_chinese	monolingual	Add
split_sentences	bilingual	Add
src_trg_ratio	bilingual	Add
strip_suffix	monolingual	Add
traditionalise_chinese	monolingual	Add

Figure 5.2: Selecting and applying filters with OpusCleaner

5.2 Management of the Training Pipeline: OpusPocus

OpusPocus pipeline manager is a modular framework aimed at simplifying the scaling, deployment and execution of the NMT pipelines within the HPLT project. The initial pre-release version of OpusPocus used for training translation models was previously described in an earlier Deliverable (*D5.1: Translation models for select language pairs*[6]). In this deliverable, we provide an update on the current development progress and a brief summary of the future OpusPocus development. Since the end of August, the repository is now publicly accessible on GitHub.[7] We plan to introduce the pipeline manager during MT Marathon 2024 to gain at least a small user-base that can provide us feedback we can use to improve the overall user experience.

Figure 5.3 provides the overview of the basic OpusPocus workflow. The user first provides a pipeline configuration file with the pipeline DAG definitions and individual step parameters. Furthermore, the provided configuration can be overwritten using command-line options. This is useful for fast

[6]https://hplt-project.org/HPLT_D5_1___Translation_models_for_select_language_pairs.pdf
[7]https://github.com/hplt-project/OpusPocus

deployment of multiple variations of a single pipeline, i.e. using the pipeline with varying language pairs. The pipeline is then executed using the chosen execution method with parameters related to the machine to be executed on (number of CPUs, GPUs, available memory, etc.).

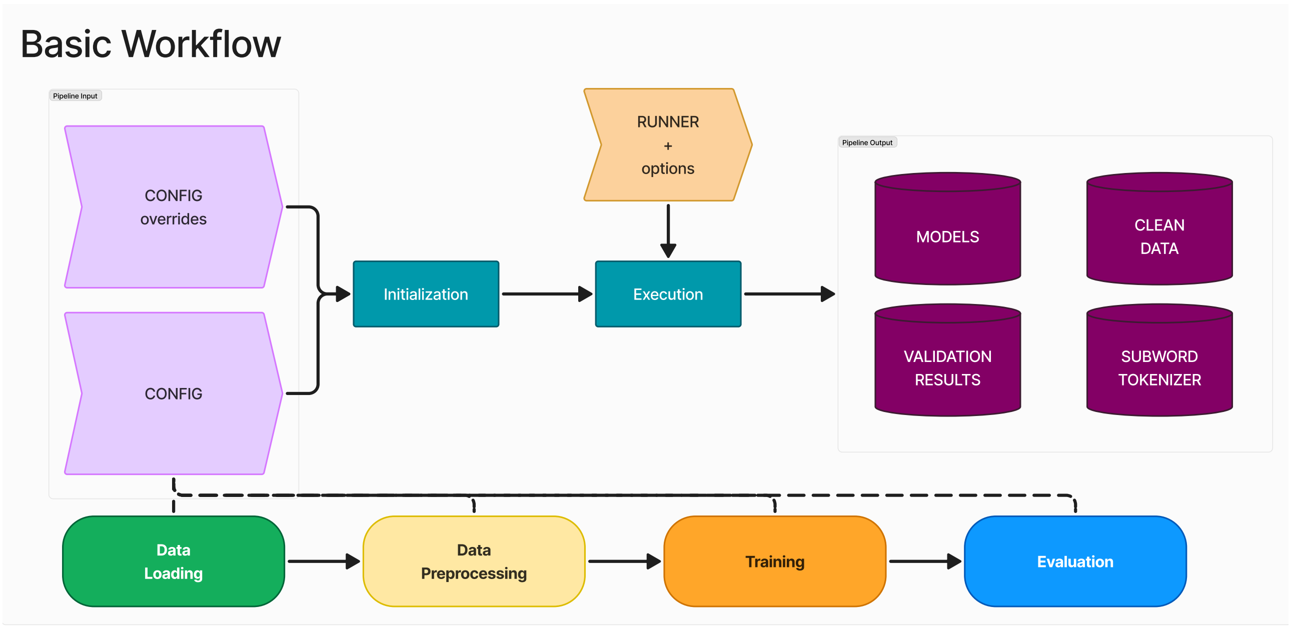


Figure 5.3: Overview of the basic OpusPocus workflow. The user provides the pipeline definition with individual parameters using the pipeline config. The pipeline is then initialized and executed using the chosen execution method, providing the output files at the end of successful execution.

The pipeline manager is highly modular, allowing the user to construct various existing training pipelines for NMT. Furthermore, the OpusPocus architecture design is general enough to allow further extension to other NLP tasks by implementation of the necessary pipeline steps. This can be done by implementing the abstract OpusPocusStep Python interface; while the implementation requires some level of Python coding skill, the interface is well documented to guide the contributor's implementation process.

The pipeline definition and execution is strictly separated, allowing execution either by running the pipeline code directly in the user terminal or scheduling the pipeline jobs via HPC scheduler (e.g. Slurm). We support dataset sharding, enabling parallel processing of large corpora. We are currently in process of extending the execution management features, including partial pipeline restart or rescheduling of failed tasks.

While still having progress in feature implementation since the release of the Deliverable D5.1 we heavily refocused on implementing a more thorough continuous delivery / continuous integration testing. This was required to guarantee reasonable test coverage before making the OpusPocus repository public and to make future development and maintenance more efficient. Although we still plan to further increase the test coverage, we plan to shift focus back to implementing missing features after the public release.

Current feature support focuses mainly on the standard NMT training and evaluation, including dataset cleaning, vocabulary inference and monolingual backtranslation. At the moment, we support model training directly using Marian NMT. However, we are also in progress of adding the OpusTrainer support to enable more sophisticated dataset mixing and more robust model training. We are also considering adding support for additional training frameworks, most likely ones that enable decoder-only model training. Based on the user feedback, we also plan to simplify the current pipeline configuration schemes to make them more user-friendly. While this is not a top priority, we hope to port the existing OpusDistillery into OpusPocus, allowing a full end-to-end, training + student distillation pipeline.

5.3 Training Data Scheduling: OpusTrainer

The purpose of OpusTrainer is to provide training data to an MT training engine such as Marian.[8] With OpusTrainer, the system builder can specify multiple data sources, provide details about how they should be mixed, and how this mixing should change as training progresses. This is all configured from a YAML file to enable reproducibility. OpusTrainer also supports a range of data augmentations which enables the training of models to better translate noisy input, and to handle terminology. These augmentations can inject case and spelling variations, and terminology tags, for instance.

OpusTrainer was described in an earlier Deliverable (*D5.1: Translation models for select language pairs*),[9] which in turn was an extract from our technical report [25]. The code and detailed instructions for installing and running OpusTrainer can be found in the GitHub repository.[10] We therefore refer the reader to those reports for more detail about OpusTrainer.

5.4 Producing Efficient Models for Deployment: OpusDistillery

OpusDistillery is a framework to streamline the Knowledge Distillation (KD) process of multilingual NMT models. OpusDistillery's main features are the integration of openly available teacher models from OPUS-MT and Hugging Face, comprehensive multilingual support and robust GPU utilization tracking.

The tool implements standard Sequence-Level Knowledge Distillation (Seq-KD) and its enhanced version, interpolated Seq-KD. Seq-KD [26] trains a student model on the sentence-level outputs produced by a teacher model. This process involves two main steps: (1) generating a synthetic dataset by forward translating the source text using the teacher model, and (2) training the student model on this generated data.

OpusDistillery is an extension of the Firefox Translation Training pipeline (FTT).[11] The final student is a quantized Marian model, fast at decoding and ready to be fed to the Bergamot-translator application.[12]

The pipeline works by feeding a YAML configuration file to Snakemake [27], a workflow management system that enables the definition of computational pipelines through rules specifying their input and output files. When the expected output files of a particular rule are absent, Snakemake systematically backtracks to identify and execute the necessary preceding rules in sequence to produce the required

[8] https://marian-nmt.github.io/
[9] https://hplt-project.org/HPLT_D5_1___Translation_models_for_select_language_pairs.pdf
[10] https://github.com/hplt-project/OpusTrainer
[11] https://browser.mt/
[12] https://github.com/browsermt/bergamot-translator

outputs. It is possible to run a specific rule or specify a target file for a given pipeline run. For NMT training, the tool uses SentencePiece [28] and Marian [29].

Our pipeline can be divided into five major steps: data preparation, synthetic dataset generation, student training, exporting and evaluation. A high-level overview of the steps is depicted in Figure 5.4.

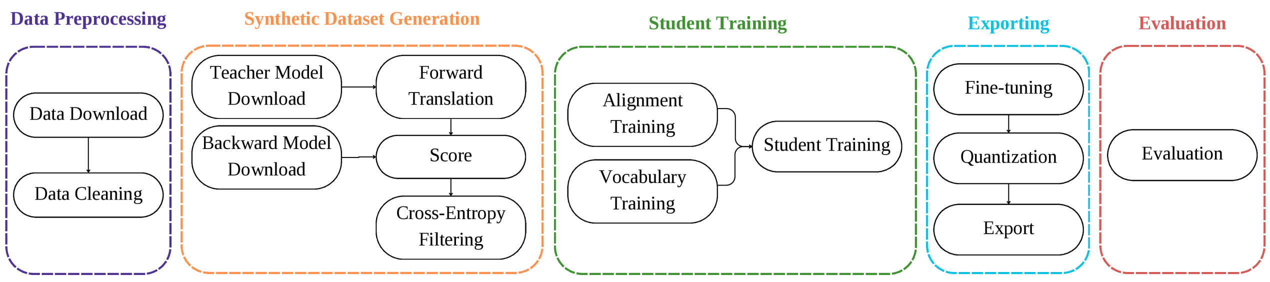


Figure 5.4: Overview of the OpusDistillery pipeline.

OpusDistillery is available at `https://github.com/Helsinki-NLP/OpusDistillery`.

Bibliography

[1] A. Barbaresi, "Trafilatura: A Web Scraping Library and Command-Line Tool for Text Discovery and Extraction," in *Proceedings of the Joint Conference of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing: System Demonstrations.* Association for Computational Linguistics, 2021, pp. 122–131. [Online]. Available: https://aclanthology.org/2021.acl-demo.15

[2] L. Burchell, A. Birch, N. Bogoychev, and K. Heafield, "An open dataset and model for language identification," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 865–879. [Online]. Available: https://aclanthology.org/2023.acl-short.75

[3] L. Burchell, "Improving natural language processing for under-served languages through increased training data diversity," Ph.D. dissertation, School of Informatics, University of Edinburgh, 2024.

[4] T. Tarmom, W. Teahan, E. Atwell, and M. A. Alsalka, "Compression versus traditional machine learning classifiers to detect code-switching in varieties and dialects: Arabic as a case study," *Natural Language Engineering*, vol. 26, no. 6, pp. 663–676, 2020.

[5] N. Arivazhagan, A. Bapna, O. Firat, D. Lepikhin, M. Johnson, M. Krikun, M. X. Chen, Y. Cao, G. Foster, C. Cherry *et al.*, "Massively multilingual neural machine translation in the wild: Findings and challenges," *arXiv preprint arXiv:1907.05019*, 2019.

[6] NLLB Team, "Scaling neural machine translation to 200 languages," *Nature*, pp. 1–6, 2024.

[7] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," in *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, M. Lapata, P. Blunsom, and A. Koller, Eds. Valencia, Spain: Association for Computational Linguistics, Apr. 2017, pp. 427–431. [Online]. Available: https://aclanthology.org/E17-2068

[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[9] D. Samuel, "Berts are generative in-context learners," 2024. [Online]. Available: https://arxiv.org/abs/2406.04823

[10] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle,

M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[11] D. Samuel, A. Kutuzov, L. Øvrelid, and E. Velldal, "Trained on 100 million words and still in shape: BERT meets British National Corpus," in *Findings of the Association for Computational Linguistics: EACL 2023*, A. Vlachos and I. Augenstein, Eds. Dubrovnik, Croatia: Association for Computational Linguistics, May 2023, pp. 1954–1974. [Online]. Available: https://aclanthology.org/2023.findings-eacl.146

[12] D. Zeman, J. Nivre, M. Abrams, E. Ackermann, N. Aepli, H. Aghaei, Ž. Agić, A. Ahmadi, L. Ahrenberg, C. K. Ajede, S. F. Akkurt, G. Aleksandravičiūtė, I. Alfina, A. Algom, K. Alnajjar, C. Alzetta, E. Andersen, L. Antonsen, T. Aoyama, K. Aplonova, A. Aquino, C. Aragon, G. Aranes, M. J. Aranzabe, B. N. Arıcan, Ĥ. Arnardóttir, G. Arutie, J. N. Arwidarasti, M. Asahara, K. Ásgeirsdóttir, D. B. Aslan, C. Asmazoğlu, L. Ateyah, F. Atmaca, M. Attia, A. Atutxa, L. Augustinus, M. Avelãs, E. Badmaeva, K. Balasubramani, M. Ballesteros, E. Banerjee, S. Bank, V. Barbu Mititelu, S. Barkarson, R. Basile, V. Basmov, C. Batchelor, J. Bauer, S. T. Bedir, S. Behzad, J. Belieni, K. Bengoetxea, I. Benli, Y. Ben Moshe, A. Berg, G. Berk, R. A. Bhat, E. Biagetti, E. Bick, A. Bielinskienė, E. F. Bilgin Taşdemir, K. Bjarnadóttir, V. Blaschke, R. Blokland, V. Bobicev, L. Boizou, J. Bonilla, E. Borges Völker, C. Börstell, C. Bosco, G. Bouma, S. Bowman, A. Boyd, A. Braggaar, A. Branco, K. Brokaitė, A. Burchardt, M. Campos, M. Candito, B. Caron, G. Caron, C. Carvalheiro, R. Carvalho, L. Cassidy, M. C. Castro, S. Castro, T. Cavalcanti, G. Cebiroğlu Eryiğit, F. M. Cecchini, G. G. A. Celano, S. Čéplö, N. Cesur, S. Cetin, Ö. Çetinoğlu, F. Chalub, L. Chamila, S. Chauhan, Y. Chen, E. Chi, T. Chika, Y. Cho, J. Choi, B. Chontaeva, J. Chun, J. Chung, A. T. Cignarella, S. Cinková, A. Collomb, Ç. Çöltekin, M. Connor, C. Corbetta, D. Corbetta, F. Costa, M. Courtin, B. Crabbé, M. Cristescu, V. Cvetkoski, I. L. Dale, P. Daniel, E. Davidson, L. F. de Alencar, M. Dehouck, M. de Laurentiis, M.-C. de Marneffe, V. de Paiva, M. O. Derin, E. de Souza, A. Diaz de Ilarraza, R. A. Díaz Hernández, C. Dickerson, A. Dinakaramani, E. Di Nuovo, B. Dione, P. Dirix, H. Do, K. Dobrovoljc, C. Döhmer, A. Doyle, T. Dozat, K. Droganova, M. S. Duran, P. Dwivedi, C. Ebert, H. Eckhoff, M. Eguchi, S. Eiche, R. Eiselen, M. Eli, A. Elkahky, B. Ephrem, O. Erina, T. Erjavec, S. Eslami, F. Essaidi, A. Etienne, W. Evelyn, S. Facundes, R. Farkas, F. Favero, J. Ferdaousi, M. Fernanda, H. Fernandez Alcalde, A. Fethi, J. Foster, T. Fransen, C. Freitas, K. Fujita, K. Gajdošová, D. Galbraith, E. Galy, F. Gamba, M. Garcia, M. Gärdenfors, T. Gaustad, E. E. Genç, F. F. Gerardi, K. Gerdes, L. Gessler, F. Ginter, G. Godoy, I. Goenaga, K. Gojenola, M. Gökırmak, Y. Goldberg, X. Gómez Guinovart, B. González Saavedra, B. Griciūtė, M. Grioni, L. Grobol, N. Grūzītis, B. Guillaume, K. Guiller, C. Guillot-Barbance, T. Güngör, N. Habash, H. Hafsteinsson, J. Hajič, J. Hajič jr., M. Hämäläinen, L. Hà Mỹ, N.-R. Han, M. Y. Hanifmuti, T. Harada, S. Hardwick, K. Harris, N. Hassert, D. Haug, J. Heinecke, O. Hellwig, F. Hennig, B. Hladká, J. Hlaváčová, F. Hociung, D. Hoefels, P. Hohle, Y. Huang, M. Huerta Mendez, J. Hwang, T. Ikeda, I. Iliadou, A. K. Ingason, R. Ion, E. Irimia, Ọ. Ishola, A. Islamaj, K. Ito, F. Iurescia, S. Jagodzińska, S. Jannat, T. Jelínek, A. Jha, K. Jiang, M. Jobanputra, A. Johannsen, H. Jónsdóttir, F. Jørgensen, M. Juutinen, H. Kaşıkara, N. Kabaeva, S. Kahane, H. Kanayama,

J. Kanerva, N. Kara, R. Karahóǧa, A. Kåsen, T. Kayadelen, S. Kengatharaiyer, V. Kettnerová, L. Kharatyan, J. Kirchner, E. Klementieva, E. Klyachko, P. Kocharov, A. Köhn, A. Köksal, K. Kopacewicz, T. Korkiakangas, M. Köse, A. Koshevoy, N. Kotsyba, B. Kovačić, J. Kovalevskaitė, S. Krek, P. Krishnamurthy, S. Kübler, A. Kuqi, O. Kuyrukçu, A. Kuzgun, S. Kwak, K. Kyle, K. Laan, V. Laippala, L. Lambertino, T. Lando, S. D. Larasati, A. Lavrentiev, J. Lee, P. Lê Hồng, A. Lenci, S. Lertpradit, H. Leung, M. Levina, L. Levine, C. Y. Li, J. Li, K. Li, Y. Li, Y. Li, K. Lim, B. Lima Padovani, Y.-J. J. Lin, K. Lindén, Y. J. Liu, N. Ljubešić, I. Lobzhanidze, O. Loginova, L. Lopes, S. Lusito, A.-M. Lutgen, A. Luthfi, M. Luukko, O. Lyashevskaya, T. Lynn, V. Macketanz, M. Mahamdi, J. Maillard, I. Makarchuk, A. Makazhanov, F. Mambrini, M. Mandl, C. Manning, R. Manurung, B. Marşan, C. Mărănduc, D. Mareček, K. Marheinecke, S. Markantonatou, H. Martínez Alonso, L. Martín Rodríguez, A. Martins, C. Martins, J. Mašek, H. Matsuda, Y. Matsumoto, A. Mazzei, R. McDonald, S. McGuinness, M. Mehta, P. A. Ménard, G. Mendonça, T. Merzhevich, P. Meurer, N. Miekka, E. Milano, A. Miller, K. Mischenkova, A. Missilä, C. Mititelu, M. Mitrofan, Y. Miyao, A. Mojiri Foroushani, J. Molnár, A. Moloodi, S. Montemagni, A. More, L. Moreno Romero, G. Moretti, S. Mori, T. Morioka, S. Moro, B. Mortensen, B. Moskalevskyi, K. Muischnek, R. Munro, Y. Murawaki, K. Müürisep, P. Nainwani, M. Nakhlé, J. I. Navarro Horñiacek, A. Nedoluzhko, G. Nešpore-Bērzkalne, M. Nevaci, L. Nguyễn Thị, H. Nguyễn Thị Minh, Y. Nikaido, V. Nikolaev, R. Nitisaroj, V. Norrman, A. Nourian, M. d. G. V. Nunes, H. Nurmi, S. Ojala, A. K. Ojha, H. Óladóttir, A. Olúòkun, M. Omura, E. Onwuegbuzia, N. Ordan, P. Osenova, R. Östling, A. Ott, L. Øvrelid, Ş. B. Özateş, M. Özçelik, A. Özgür, B. Öztürk Başaran, T. Paccosi, A. Palmero Aprosio, A. Panova, T. A. S. Pardo, H. H. Park, N. Partanen, E. Pascual, M. Passarotti, A. Patejuk, G. Paulino-Passos, G. Pedonese, A. Peljak-Łapińska, S. Peng, S. L. Peng, R. Pereira, S. Pereira, C.-A. Perez, N. Perkova, G. Perrier, S. Petrov, D. Petrova, A. Peverelli, J. Phelan, C. Pierre-Louis, J. Piitulainen, Y. Pinter, C. Pinto, R. Pintucci, T. A. Pirinen, E. Pitler, M. Plamada, B. Plank, A. Plum, T. Poibeau, L. Ponomareva, M. Popel, L. Pretkalniņa, R. Pretorius, S. Prévost, P. Prokopidis, A. Przepiórkowski, R. Pugh, T. Puolakainen, C. Purschke, S. Pyysalo, P. Qi, A. Querido, A. Rääbis, A. Rademaker, M. Rahoman, T. Rama, L. Ramasamy, C. Ramisch, J. Ramos, F. Rashel, M. S. Rasooli, V. Ravishankar, L. Real, P. Rebeja, S. Reddy, M. Regnault, G. Rehm, A. Riabi, I. Riabov, M. Rießler, E. Rimkutė, L. Rinaldi, L. Rituma, P. Rizqiyah, L. Rocha, E. Rögnvaldsson, I. Roksandic, M. Romanenko, R. Rosa, V. Roșca, D. Rovati, B. Rozonoyer, O. Rudina, J. Rueter, P. Ruffolo, K. Rúnarsson, S. Sadde, P. Safari, A. Sahala, S. Saleh, A. Salomoni, T. Samardžić, S. Samson, X. Sánchez-Rodríguez, M. Sanguinetti, E. Sanıyar, D. Särg, M. Sartor, A. Sarymsakova, M. Sasaki, B. Saulīte, A. Savary, Y. Sawanakunanon, S. Saxena, K. Scannell, S. Scarlata, E. Schang, N. Schneider, S. Schuster, L. Schwartz, D. Seddah, W. Seeker, S. Sellmer, M. Seraji, S. Shahzadi, M. Shen, A. Shimada, H. Shirasu, Y. Shishkina, M. Shohibussirri, M. Shvedova, J. Siewert, E. F. Sigurðsson, J. Silva, A. Silveira, N. Silveira, S. Silveira, M. Simi, R. Simionescu, K. Simkó, M. Šimková, H. B. Símonarson, K. Simov, D. Sitchinava, T. Sither, A. Smith, I. Soares-Bastos, P. E. Solberg, B. Sonnenhauser, S. Sourov, R. Sprugnoli, V. Stamou, S. Steingrímsson, A. Stella, A. Stephen, M. Straka, E. Strickland, J. Strnadová, A. Suhr, Y. L. Sulestio, U. Sulubacak, S. Suzuki, D. Swanson, Z. Szántó, C. Taguchi,

D. Taji, F. Tamburini, M. A. C. Tan, T. Tanaka, D. Tanaya, M. Tavoni, S. Tella, I. Tellier, M. Testori, G. Thomas, T. E. Tıraş, S. Tonelli, L. Torga, M. Toska, T. Trosterud, A. Trukhina, R. Tsarfaty, U. Türk, F. Tyers, S. Ĥórðarson, V. Ĥorsteinsson, S. Uematsu, R. Untilov, Z. Urešová, L. Uria, H. Uszkoreit, A. Utka, E. Vagnoni, S. Vajjala, S. Vak, R. van der Goot, M. Vanhove, D. van Niekerk, G. van Noord, V. Varga, U. Vedenina, G. Venturi, E. Villemonte de la Clergerie, V. Vincze, A. Vissamsetty, N. Vlasova, E. Vligouridou, A. Wakasa, J. C. Wallenberg, L. Wallin, A. Walsh, J. Wang, J. N. Washington, M. Wendt, P. Widmer, S. Wigderson, S. H. Wijono, V. B. Wille, S. Williams, M. Wirén, C. Wittern, T. Woldemariam, T.-s. Wong, A. Wróblewska, Q. Wu, M. Yako, K. Yamashita, N. Yamazaki, C. Yan, K. Yasuoka, M. M. Yavrumyan, A. B. Yenice, E. Yılandiloğlu, O. T. Yıldız, Z. Yu, A. Yuliawati, Z. Žabokrtský, S. Zahra, A. Zeldes, H. Zhou, H. Zhu, Y. Zhu, A. Zhuravleva, and R. Ziane, "Universal dependencies 2.14," 2024, LINDAT/CLARIAH-CZ digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University. [Online]. Available: http://hdl.handle.net/11234/1-5502

[13] R. Luukkonen, V. Komulainen, J. Luoma, A. Eskelinen, J. Kanerva, H.-M. Kupari, F. Ginter, V. Laippala, N. Muennighoff, A. Piktus, T. Wang, N. Tazi, T. Scao, T. Wolf, O. Suominen, S. Sairanen, M. Merioksa, J. Heinonen, A. Vahtola, S. Antao, and S. Pyysalo, "FinGPT: Large generative models for a small language," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing.* Association for Computational Linguistics, 2023, pp. 2710–2726. [Online]. Available: https://aclanthology.org/2023.emnlp-main.164

[14] R. Luukkonen, J. Burdge, E. Zosa, A. Talman, V. Komulainen, V. Hatanpää, P. Sarlin, and S. Pyysalo, "Poro 34b and the blessing of multilinguality," 2024. [Online]. Available: https://arxiv.org/abs/2404.01856

[15] T. Le Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé *et al.*, "Bloom: A 176b-parameter open-access multilingual language model," 2023.

[16] O. Press, N. A. Smith, and M. Lewis, "Train short, test long: Attention with linear biases enables input length extrapolation," *arXiv preprint arXiv:2108.12409*, 2021.

[17] T. L. Scao, T. Wang, D. Hesslow, L. Saulnier, S. Bekman, M. S. Bari, S. Biderman, H. Elsahar, N. Muennighoff, J. Phang *et al.*, "What language model to train if you have one million gpu hours?" *arXiv preprint arXiv:2210.15424*, 2022.

[18] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[19] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu, "Roformer: Enhanced transformer with rotary position embedding," *Neurocomputing*, vol. 568, p. 127063, 2024.

[20] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "Gqa: Training generalized multi-query transformer models from multi-head checkpoints," *arXiv preprint arXiv:2305.13245*, 2023.

[21] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal,

E. Hambro, F. Azhar *et al.*, “Llama: Open and efficient foundation language models,” *arXiv preprint arXiv:2302.13971*, 2023.

[22] H. Xu, Y. J. Kim, A. Sharaf, and H. H. Awadalla, “A Paradigm Shift in Machine Translation: Boosting Translation Performance of Large Language Models,” Sep. 2023, arXiv:2309.11674 [cs]. [Online]. Available: http://arxiv.org/abs/2309.11674

[23] D. M. Alves, J. Pombal, N. M. Guerreiro, P. H. Martins, J. Alves, A. Farajian, B. Peters, R. Rei, P. Fernandes, S. Agrawal, P. Colombo, J. G. C. de Souza, and A. F. T. Martins, “Tower: An Open Multilingual Large Language Model for Translation-Related Tasks,” Feb. 2024, arXiv:2402.17733 [cs]. [Online]. Available: http://arxiv.org/abs/2402.17733

[24] T. Kocmi, E. Avramidis, R. Bawden, O. Bojar, A. Dvorkovich, C. Federmann, M. Fishel, M. Freitag, T. Gowda, R. Grundkiewicz, B. Haddow, M. Karpinska, P. Koehn, B. Marie, K. Murray, M. Nagata, M. Popel, M. Popovic, M. Shmatova, S. Steingrímsson, and V. Zouhar, “Preliminary WMT24 Ranking of General MT Systems and LLMs,” Jul. 2024, arXiv:2407.19884 [cs]. [Online]. Available: http://arxiv.org/abs/2407.19884

[25] N. Bogoychev, J. van der Linde, G. Nail, B. Haddow, J. Zaragoza-Bernabeu, G. Ramírez-Sánchez, L. Weymann, T. N. Mateiu, J. Helcl, and M. Aulamo, “OpusCleaner and OpusTrainer, open source toolkits for training Machine Translation and Large language models,” Nov. 2023, arXiv:2311.14838 [cs]. [Online]. Available: http://arxiv.org/abs/2311.14838

[26] Y. Kim and A. M. Rush, “Sequence-level knowledge distillation,” *arXiv preprint arXiv:1606.07947*, 2016.

[27] F. Mölder, K. P. Jablonski, B. Letcher, M. B. Hall, C. H. Tomkins-Tinch, V. V. Sochat, J. Forster, S. Lee, S. O. Twardziok, A. Kanitz, A. Wilm, M. Holtgrewe, S. Rahmann, S. Nahnsen, and J. Köster, “Sustainable data analysis with snakemake,” *F1000Research*, vol. 10, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:234357363

[28] T. Kudo and J. Richardson, “Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing,” in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2018, pp. 66–71.

[29] M. Junczys-Dowmunt, K. Heafield, H. Hoang, R. Grundkiewicz, and A. Aue, “Marian: Cost-effective high-quality neural machine translation in C++,” in *Proceedings of the 2nd Workshop on Neural Machine Translation and Generation*, A. Birch, A. Finch, T. Luong, G. Neubig, and Y. Oda, Eds. Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 129–135. [Online]. Available: https://aclanthology.org/W18-2716