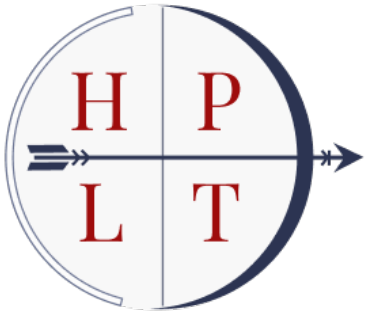
Co-funded by the European Union



HPLT: High Performance Language Technologies

Translation Models for Select Language Pairs

Deliverable number: 5.1

Version 1.0

Horizon Europe

UK Research and Innovation

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

Translation Models for Select Language Pairs	
Lead author:	Barry Haddow (UEDIN)
Contributing authors:	Nikolay Bogoychev (UEDIN) Pinzhen Chen (UEDIN) Jindřich Helcl (CUNI) Jelmer van der Linde (UEDIN) Bhavitvya Malik (UEDIN) Pavel Stepachev (UEDIN) Dušan Variš (CUNI)
Internal reviewers:	David Samuel (UiO) Sampo Pyysalo (UTURKU)
Deliverable number:	5.1
Dissemination level:	Public (PU)
Contractual Delivery Date:	Feb 29, 2024
Actual Delivery Date:	Feb 29, 2024
Number of pages:	23

Document history

Version	Date	Changes
1.0	Feb 29, 2024	Original Submission

Abstract

In HPLT, we have been developing tools for automated model building (OpusCleaner, OpusTrainer and OpusPocus) as well mining the Internet Archive for parallel data. In this deliverable we describe a set of translation models that we built using the HPLT tools, and the HPLT data. The models cover several lower resource language pairs, and demonstrate the value of HPLT data in improving translation performance, over baseline models trained on all of the Opus data.

Contents

1. Executive summary

This deliverable describes the initial HPLT translation model release. We aimed to train translation models with all language pairs in the first release of the HPLT parallel data, using the data preparation and training tools that we are developing in the project. We describe OpusTrainer (a tool for managing the training data for MT systems) and OpusPocus (a tool for orchestrating model training for MT). We then describe the architecture and process for model training using these tools. In the evaluation section we show the degree to which HPLT data affects translation performance when added to the large Opus training corpus.

CHARLES UNIVERSITY

UNIVERSITY OF OSLO

UNIVERSITY OF EDINBURGH

UNIVERSITY OF TURKU

UNIVERSITY OF HELSINKI

PROMPSIT

CESNET

SIGMA2

2. Introduction

The main aim of the first HPLT MT model release was to bring together all the tools that we have been developing for the MT model pipeline, and to show that they are capable of building a suite of MT models in a mostly automated fashion. The model building also helped us to examine the quality of the first HPLT data release (Tiedeman et al., 2023), and to see if it influences performance when combined with the much larger Opus parallel data collection. For this reason, we aimed to build bilingual models for all the language pairs included in the first HPLT parallel data release.

The tooling for the model-building pipeline includes OpusCleaner (for selecting and cleaning training data), OpusTrainer (a data scheduling and data augmenting tool), and OpusPocus (for managing the training process itself). The first tool is described in a different deliverable (Ramírez-Sánchez, 2024) whereas the other two tools are described in Section 3.

The actual release of the trained models is through Hugging Face (in the HPLT organisation https://huggingface.co/HPLT, under the MT model collection https://huggingface.co/collections/HPLT/machine-translation-models-65dba9a92f6d2dfc2755cd52). We also have a repository (https://github.com/hplt-project/mt-models) which contains the configuration required to download and process the data, and train and evaluate the models. The idea is that a third party should be able to use this repository, together with our tool chain, to completely reproduce our model building. We discuss the model building itself in Section 4, referring to the repository when appropriate.

Finally, to evaluate the models, we used the FLoRes-200 (NLLB team et al., 2022) and NTREX (Federmann et al., 2022) data sets, both of which consist of English source texts translated into many different target languages, including all the language pairs in our release. The evaluation results are in Section 5.

3. Tools

3.1. OpusTrainer

This section is an extract from Bogoychev et al. (2023)

Training high-quality machine translation systems requires carefully combining parallel data from different sources and quality levels; applying on-the-fly modifications to it and more.

This is challenging to achieve with neural network toolkits that make use of static training data, because ideally, we want to modify the data mixture and potentially augment it on the fly, without having to *prepare* the data first and write it to disk which is wasteful.

Multilingual model training The problem is exacerbated when training many-to-many or English-to-many multilingual models where high-resource languages would often have orders of magnitude more data than low-resource languages. In order for a multilingual model to train well in this setting, the MT toolkit needs to see balanced data from all languages (Freitag and Firat, 2020). Doing this by concatenating and upsampling data (in order to get equal amounts of data seen for all languages), could waste multiple terabytes of disk space.

3.1.1. Data Scheduling

OpusTrainer solves this problem by streaming and mixing data from multiple sources. OpusTrainer uses a simple YAML configuration file where the user can declare all of their data sources and a desired mix of them for different stages of training. OpusTrainer then reads in the data from different sources and then outputs the desired mix to *stdout*. OpusTrainer is meant to be used with neural network toolkits that support reading data from *stdin* such as Marian (Junczys-Dowmunt et al., 2018), but it can also output the desired data mix to a file, making it usable with all toolkits. An example configuration that describes a full training run with various data mixings for different stages of training can be seen in Figure 3.1.

3.1.2. Data Augmentation

Humans are very robust to decoding noisy texts, but this can pose a major challenge to machine translation systems due to the way we collect our training data:

- Title Case and Upper Case parallel data is quite rare in training data and is sometimes regularised during acquisition.
- Typos are also comparatively rare in training data, because either we use clean sources or we perform spellchecking on web-crawled sources.
- Emojis, which human readers expect to be copied over from the source to the target, are not seen during training because typically lines containing emojis are removed from the training data at preprocessing steps.

```yaml
datasets:
  clean: clean.gz # 2.4 GB
  cleanish: clean.med.gz # 1.3 GB
  medium: medium.gz # 1.8 GB
  dirty: dirty.gz # 33 GB

stages:
  - start
  - mid
  - end

start:
  - clean 0.65
  - cleanish 0.25
  - medium 0.1
  - dirty 0
  - until clean 1 # Until 1 epoch of clean

mid:
  - clean 0.25
  - cleanish 0.1
  - medium 0.05
  - dirty 0.6
  - until dirty 1

end:
  - clean 0.4
  - cleanish 0.15
  - medium 0.15
  - dirty 0.3
  - until dirty inf

seed: 1111
```

Figure 3.1: OpusTrainer basic configuration defining the data scheduling for training a model.

In order to alleviate these issues, OpusTrainer provides multiple data modifiers that can be applied on the fly, at random on the training data:

- UpperCaser and TitleCaser
- Typo modifier, which inserts typos in words during training
- Merge modifier, which randomly merges several input sentences together to help the model be more robust to longer sentences.
- Noise modifier, that generates random sentences consisting of unicode noise, identical on both the source and the target side. This modifier teaches the model to copy unknown strings to the target side.
- Inline Noise modifier: A more complicated version of the above that uses word alignments in order to *inject* noisy Unicode characters (including Emoji) in approximately the same logical place on both the source and the target side. This modifier teaches the model that unknown sequences of <*unk*> characters should be just copied on the target side.

All of those modifiers are applied to each sentence in the training data with a user-defined probability as shown in Figure 3.2.

```yaml
modifiers:
  - UpperCase: 0.05
  - TitleCase: 0.05
  - Typos: 0.05
  - Tags: 0.005
    augment: 1 # Augment with inline unicode noise
  - Noise: 0.0005
    min_word_legnth: 2 # Minumum lenght of each fake word
    max_word_length: 5 # Maximum lenght of each fake word
    max_words: 4       # Maximu mnumber of fake words
```

Figure 3.2: Different modifiers specified in YAML format to be used during training.

3.1.3. Terminology

OpusTrainer is able to leverage word alignment information to produce terminology-augmented systems, precisely as the one described by Bogoychev and Chen (2023). This is achieved by finding bijective word alignment mappings between the source and the target sentences and at randomly injecting terminology hints in the source, precisely like the one shown on 3.3.

Where is the airport? ↔ Wo ist der Flughafen?
Where is the airport __*target*__ *Flughafen* __*done*__? ↔ Wo ist der Flughafen?

Figure 3.3: Terminology augmentation in practice. During training, it is hinted that the target word *Flughafen* corresponds to *Airport*, so that at inference when providing the model with terminology hints it will know how to incorporate them at the output.

These terminology hints can then be used at inference time, and the model will know how to incorporate the desired terminology hint at the target side. The relevant training options are shown in figure 3.4

```yaml
modifiers:
  - Tags: 0.04
    custom_detok_src: null
    custom_detok_trg: null
    template: "{src} __target__ {trg} __done__"
```

Figure 3.4: Tag modifier is used to add terminology hints to the source during training. Values of 3% to 7% seem to work well in practice.

OpusTrainer is open source and available on GitHub,[1] with ample documentation and examples. OpusTrainer is designed to be used mainly with neural network toolkits that read in training input on *stdin*, as it takes care of shuffling between epochs, resuming training and all other functions normally done by the data module of a neural network toolkit. It can, however, also be used to write a preprocessed training corpus on disk so toolkits that do not support reading *stdin* can also make use of it.

[1] https://github.com/hplt-project/OpusTrainer

3.2. OpusPocus

This section describes the OpusPocus NMT training pipeline manager.[2] The main goal of the pipeline manager is to abstract and automate the repetitive parts of the training process such as data preparation, model training, generation of synthetic parallel data by backtranslation and the model fine-tuning. A new user should be able to run the default training pipelines without any knowledge about the implementation details, simply by plugging-in their training data (possibly manually inspected and assessed for cleaning) and running a pipeline execution command.

The pipeline manager is inspired by utilities such as GNU Make[3] or Snakemake[4]. These utilities help represent pipeline steps and their dependencies by building a directed acyclic graph (DAG) of tasks. Before the execution of a given step, the pipeline checks the state of the dependencies and executes any unfinished prerequisite steps. After that, the execution of the current step can proceed.

While these tools can be used for NMT training, they are too general and pipelines written using these tools usually become too difficult to maintain, and they do not work well in HPC (high performance computing) environments. On the other hand, NMT training often adheres to a small number of specific scenarios that can benefit from a customised pipeline manager utility. Furthermore, NMT training within the HPLT project poses unique challenges related to training using the LUMI HPC cluster which come with their own set of limitations for the user, e.g. job runtime limits and a limit on the number of submitted jobs. These challenges need to be addressed for efficient scaling of the NMT training up to tens of language pairs, with hundreds of experiments needed to run in relatively short time.

The main features of OpusPocus are:

- **Implementation in Python** for better maintainability due to a larger user base (compared to bash). In the future, even the task-related scripts should be completely in Python.
- **Modularity.** Each step of the pipeline is isolated from the others and only requires the outputs from its dependencies to be executed.
- **Separation of pipeline execution and monitoring.** The pipeline manager prepares the input and step dependencies and allows execution of the pipeline. The manager terminates after execution – the process monitoring is done separately by either a new manager execution or by a separate user script.
- **Separation of task definition and task execution.** We plan to support various execution methods. Currently, the support for bash and SLURM is implemented and HyperQueue[5] support is partially implemented. The execution method implementation should be independent of the structure of individual tasks.

[2] https://github.com/hplt-project/OpusPocus
[3] https://www.gnu.org/software/make/
[4] https://snakemake.readthedocs.io/en/stable/
[5] https://github.com/It4innovations/hyperqueue

3.2.1. General Architecture

The two main building blocks of OpusPocus are implemented via `OpusPocusStep` and `OpusPocusPipeline`. A user chooses a specific pipeline implementation based on its description, sets a correct set of pipeline parameters (configurable via YAML configuration files), and executes the pipeline. These steps are possible to perform without any prior knowledge about the code structure and the implementation of the individual pipeline steps.

The pipeline step implementation is reusable – new pipelines can be implemented by arranging the existing steps and their dependencies in various ways, as long as the required dependencies of the pipeline steps are plugged-in. Furthermore, additional pipeline steps can be defined by inheriting from the abstract `OpusPocusStep` class and overwriting its abstract methods. The `OpusPocusStep` implements a state attribute to indicate the current step state. The currently available step states are:

- `INITED` – the step was successfully initialized (the directory structure and the step hyper-parameters are available)
- `RUNNING` – the step has been scheduled for execution or already being executed
- `FAILED` – the step execution has failed
- `DONE` – the step execution has finished successfully

3.2.1.1. Main Program

The pipeline manager can be invoked by the `go.py` entry script. The script currently implements 3 main sub-commands: `init`, `run`, and `traceback`.

The `init` command. The `init` command invokes the pipeline initialization, and proceeds in four steps: First, the manager checks whether all pipeline hyper-parameters are provided (using default values for non-required parameters). Second, the manager creates a dependency DAG as defined in the pipeline implementation. Third a directory structure is created within the directory specified using the `--pipeline-dir` argument. The sub-directories in that location represent workspaces of the individual pipeline steps. The step directory names are derived automatically based on the step implementation. Fourth, the pipeline hyper-parameters are saved for later execution.

The `run` command. This command executes the pipeline using the chosen environment (e.g., bash, slurm, etc.). The manager follows the dependency graph and starts the execution either with steps with no dependencies, or with steps that have their dependencies already satisfied (i.e. they are in the `DONE` state – this can happen when re-running after a partially successful execution). Then, the following steps are scheduled for execution after their dependencies have finished successfully (slurm) or wait in a subprocess until all dependencies are done (bash).

traceback This command provides a visualisation of the pipeline dependency graph and the step hyper-parameters of the current pipeline instance.

3.2.1.2. The `OpusPocusPipeline` class

Each pipeline implementation, describing new pipeline workflow, needs to inherit from the `OpusPocusPipeline` abstract class. A pipeline implementation defines its own set of pipeline parameters using the `add_parser` method. The resulting pipeline parameterization enables launching multiple instances of the same pipeline with specific modifications. Furthermore, the derived class must overwrite the `build_pipeline_graph` method. This method should contain all the graph-building logic specific to the individual pipeline implementation (corpus cleaning process, training, fine-tuning and its dependencies of the intermediate backtranslation steps). Finally every implementation needs to be registered using the `opuspocus.pipelines.register_pipeline` decorator to be visible to the pipeline manager. Note that the pipeline implementations are separated from the actual executions methods - they only describe the workflow and the parameters of the individual steps.

3.2.1.3. The `OpusPocusStep` class

Every pipeline step implementation inherits from `OpusPocusStep` abstract class. The step parameters are defined directly via the derived class constructor and are automatically saved during initialization. Each derived class must overwrite the `step_name` method defining a naming scheme for the respective step instance directories. Besides the step initialization and execution implementations, it also defines various helper methods, for example for loading and saving the step parameters, or accessing the step output and log directory.

The step class implements a code generation method (`compose_command`) which is used to create the step execution script.[6] We split the command definition into several logical parts, such as scheduler directives, variable definitions, or exception handling. These parts can be reused in different step definitions. The only step-specific code generation is defined in the `_cmd_body_str` abstract method which needs to be implemented by the individual derived step classes.

Due to logical similarity, we provide a `CorpusStep` abstract class, inheriting from `OpusPocusStep`. Every step implementation that modifies or creates the corpus data (corpus preprocessing, corpus translation) should inherit from this class. The abstract class provides additional functionality, such as output sharding or indication of the available datasets. Furthermore, it enables more specific dependency restrictions and type checking for the pipelines.

Finally, each step needs again to be registered using the `opuspocus.pipeline_steps.register_step` decorator to be visible to the `.build_step`, `.load_step` factory methods.

3.2.2. Repository Structure

The OpusPocus repository contains the following structure:

- `go.py` – Pipeline manager entry script.
- `opuspocus/utils.py` – Various helper methods.

[6] Currently written in bash.

- `opuspocus/command_utils.py` – Helper methods for various methods of pipeline execution (bash, slurm, etc.)
- `opuspocus/pipelines/` – implementation of the abstract `OpusPocusPipeline` class and its derivations implementing a specific pipeline structures.
- `opuspocus/pipeline_steps/` – implementation of the abstract `OpusPocusStep` class and its derivations implementing a individual pipeline steps

All new pipeline and pipeline step class implementations need to be located in the `opuspocus/pipelines` and `opuspocus/pipeline_steps` directories respectively and registered using the related `@register_` decorator to be available to the manager.

In future, we consider replacing the methods in `opuspocus/command_utils.py` with more robust class-based implementations, similart to `OpusPocusStep` and `OpusPocusPipeline`.

3.2.3. Execution Examples

The following code is an example script for a pipeline initialization and execution:

```
#!/usr/bin/env bash
SRC="en"
TGT="fr"

PIPELINE_DIR="my_pipeline"

RAW_DATA_DIR="data/$SRC-$TGT/raw"
VALID_DIR="data/$SRC-$TGT/"
TEST_DIR="data/$SRC-$TGT/test"

MARIAN_CONFIG="config/marian.simple.yml"

./go.py init \
    --pipeline simple \
    --pipeline-dir $PIPELINE_DIR \
    --pipeline-config config/pipeline.simple.yml \
    --src-lang $SRC \
    --tgt-lang $TGT \
    --raw-data-dir $RAW_DATA_DIR \
    --valid-data-dir $VALID_DIR \
    --test-data-dir $TEST_DIR \
    --marian-config $MARIAN_CONFIG \
    --log-level info
./go.py run \s
    --runner sbatch \
    --runner-opts '--account=project_465000574 --partition=small-g --time=48:00:00' \
```

```
    --pipeline-dir $PIPELINE_DIR \
    --log-level info
```

The scripts invokes a *simple* pipeline and creates the pipeline directory structure inside the `$PIPELINE_DIR`. The pipeline is then scheduled using SLURM. The `$MARIAN_CONFIG` is a standard MarianNMT configuration file.

After the initialization, the user can check the dependency structure using the traceback sub-command:

```
$ ./go.py traceback --pipeline-dir "my_pipeline"
```

The command prints the following output:

```
Target 0: s.train_model.ca-en
+ s.train_model.ca-en: RUNNING
|=+ vocab_step
  + s.generate_vocab.ca-en: RUNNING
  |=+ corpus_step
    + s.gather.ca-en: RUNNING
    |=+ previous_corpus_step
      + s.decontaminate.ca-en: RUNNING
      |=+ previous_corpus_step
        + s.clean.ca-en: RUNNING
        |=+ previous_corpus_step
          + s.raw.ca-en: RUNNING
          |=+ previous_corpus_step
            + None
|=+ train_corpus_step
  + s.gather.ca-en: RUNNING
  |=+ previous_corpus_step
    + s.decontaminate.ca-en: RUNNING
    |=+ previous_corpus_step
      + s.clean.ca-en: RUNNING
      |=+ previous_corpus_step
        + s.raw.ca-en: RUNNING
        |=+ previous_corpus_step
          + None
|=+ model_init_step
  + None

Target 1: s.train_model.en-ca
[...]
```

The pipeline has two targets (`s.train_model.ca-en` and `s.train_model.en-ca`) with shared dependencies (both targets use the same training parallel corpus and vocabulary). All steps are marked as running – they will later change state to DONE or FAILED based on the execution result.

3.2.4. Discussion and Future Work

We described the current state of the implementation of the OpusPocus NMT training pipeline manager. The basic architecture design is finished together with a simple pipeline implementation.

In the following months, we will focus on the following tasks:

- Finish the HyperQueue support implementation. This is currently the main requirement for pipelines that require repeating larger dataset processing, e.g. parallelized backtranslation of the monolingual data.
- Finish the Iterative Backtranslation pipeline implementation. Based on Popel et al. (2020), we aim to reproduce the original results and test the proposed iterative backtranslation method on a much larger scale using the parallel data gathered during the HPLT project.
- Support for OpusTrainer. The current version of OpusPocus invokes MarianNMT directly. As a next step, we plan to implement the support for OpusTrainer which will enable a more nuanced curriculum learning.
- Additional features. The initial tests of the pipeline manager at scale showed that some essential features that could improve the user experience are still missing. To name a few, we plan to implement an improved pipeline status reporting or a more robust failure recovery.

4. Models

4.1. Data Selection and Preparation

Our aim for this first round of model building was to produce models for the language pairs included in the first HPLT data release, in both en-XX and XX-en directions. Due to constraints on computational resources, we have trained models for 16/18 language pairs at the time of writing the report. We trained models using three different data conditions, with all data derived from the Opus collection (Tiedemann, 2012):[1]

HPLT Only the HPLT parallel data. We use v1.1 from Opus, but for parallel data, this is identical to v1.2.

OPUS A concatenation of all datasets from Opus, for the given language pair, excluding data derived from HPLT. For many language pairs, this usually includes other large-scale crawled datasets like CCMatrix/NLLB, CCAligned, and ParaCrawl (Schwenk et al., 2021; NLLB team et al., 2022; El-Kishky et al., 2020; Bañón et al., 2020).

HPLT+OPUS A concatenation of the data from the first two sources.

We report the data sizes at the sentence pair level in Table 4.1 for the three conditions. In addition, we show the % size of HPLT relative to OPUS. The parallel data in this HPLT release are generally smaller than the existing OPUS collection, with the highest being 30.9% for Traditional Chinese and the lowest being 0.24% for Bosnian.

For each corpus, we applied a basic set of cleaning rules available in OpusCleaner and did a manual inspection to ensure that the cleaning rules were working sensibly. Since we do not speak most of the languages in the HPLT data release, the manual inspection was limited to a cursory sanity check in many cases. We leave a full exploration of OpusCleaner cleaning options to future work. The `json` files recording data composition and filter rules and values are included in our release GitHub repository.[2]

[1] https://opus.nlpl.eu/

[2] https://github.com/hplt-project/HPLT-MT-Models

Pair	Number of sentence pairs			%
	HPLT	OPUS	HPLT+OPUS	
ar-en	10580773	93310073	103890991	11.339
bs-en	199810	82956680	83156490	0.241
ca-en	7172951	26546268	33719219	27.021
en-et	4950818	31404411	36355229	15.765
en-eu	531025	2815628	3346695	18.86
en-fi	20468388	125702855	146171243	16.283
en-ga	865846	15355019	16220865	5.639
en-gl	974882	16497221	17472103	5.909
en-hi	9655965	57602567	67258457	16.763
en-hr	8495511	74520266	77624591	11.4
en-is	1741021	18438611	20179293	9.442
en-mt	670912	14850871	15521783	4.518
en-nn	44376	264606	308982	16.771
en-sq	1496269	64945148	66441508	2.304
en-sw	1316875	13367808	14684683	9.851
en-zh_Hant	4447104	14380235	18827339	30.925

Table 4.1: Dataset statistics for MT model training. This shows the number of parallel sentence pairs in the cleaned corpora, i.e. the corpora that were used for training. The "%" column specifies the proportion of HPLT sentence pairs to the number in OPUS

4.2. Training

To ensure reasonably rapid training, allowing us to validate and debug the pipeline, we used the modestly-sized Transformer-base (Vaswani et al., 2017) architecture preset in Marian (Junczys-Dowmunt et al., 2018). Since we adopted a simple curriculum, where all datasets were concatenated and shuffled together, we called Marian directly from OpusPocus as opposed to training via OpusTrainer. We opted for a joint sentencePiece vocabulary for the source and target languages (Kudo and Richardson, 2018). We store the full training configuration and training logs in the model release GitHub.

The training pipeline that we used for model building has the following steps:

Raw Copying the data files into place.

Clean Applying OpusCleaner cleaning rules.

Decontaminate Removing any overlap with test data.

Gather Concatenating data into a single file.

Generate-vocab Creating a SentencePiece vocabulary.

Train Training translation models in both directions.

5. Evaluation

5.1. Test Sets and Metrics

For each trained model, we perform inference on two test sets: FLORES200 (NLLB team et al., 2022) and NTREX (Federmann et al., 2022). Both test sets originated in English and translated to other languages, by professional translators. Following conventions in the field, we choose a beam size of 6 and a maximum generation length of 512 tokens. Detailed decoding configurations are also supplied in our GitHub repository.

The model outputs are scored using three different reference-based metrics that are commonly used for automatic translation evaluation: BLEU (Papineni et al., 2002), ChrF++ (Popović, 2017), and COMET (Rei et al., 2020). BLEU computes n-gram accuracy with a penalty if the output is too short; ChrF++ is based on character n-gram precision and recall enhanced with word n-grams; COMET is a neural metric that derives a score from source, output, and reference sentence embeddings. For the first two metrics, we use the `sacrebleu` (Post, 2018) implementation.[1] The sacrebleu signature is `nrefs:1|case:mixed|eff:yes|nc:6|nw:2|space:no|version:2.3.1`; where we use `zh` as the tokenization method for Traditional Chinese and `13a` for the rest of the languages. COMET scores are from the reference-based version `wmt-22-da`.[2] Full results are shown in Tables 6.1 (BLEU), 6.2 (chrF++), and 6.3 (COMET). We represent translation directions in a source-target format using language codes. In Table A.1 in the appendix we outline the mapping between language codes and language names.

5.2. Result Discussions

From the results, we observe that the highest metric scores are obtained when both HPLT and OPUS data are used in many cases, compared with using HPLT or OPUS data alone. Such a trend is consistent across the two test sets and three evaluation metrics. This underscores the inherent value that HPLT parallel data contributes to the current open-source landscape in machine translation. For languages that do not benefit from the addition of HPLT data, it could be that the HPLT data is to small (relative to OPUS) to make a difference, or that the HPLT is too noisy (which requires investigation) or out-of-domain. We do not observe any cases where adding the HPLT significantly reduces performances, according to our metrics.

The performance of HPLT-only configurations typically falls short when compared to OPUS-only setups across most language pairs. However, it is noteworthy that the attained results remain commendable, given that the HPLT data are usually just around 10% of the sizes of OPUS. Specifically, HPLT surpasses OPUS in Traditional Chinese despite having only 30% of its size. HPLT data alone results in very low numbers for Norwegian but seems to be beneficial when mixed with OPUS data.

[1] https://github.com/mjpost/sacrebleu
[2] https://huggingface.co/Unbabel/wmt22-comet-da

6. Conclusion

We have described the tools developed from translation model building in HPLT, and the initial set of models. The models are available from our Hugging Face organisation, and the model building helps to validate our tools and data sets. In the second half of the project we expect to release an improved set of models, using more and better data, adding back-translation and multingual MT, as well as developing the robustness and usability of the tool-chain.

	FLORES-200			**NTREX**		
Pair	HPLT	OPUS	HPLT+OPUS	HPLT	OPUS	HPLT+OPUS
ar-en	35.0	**40.2**	40.1	28.6	**35.1**	34.7
en-ar	26.8	28.9	**29.2**	23.9	25.6	**25.8**
bs-en	12.8	**39.3**	37.7	9.4	**36.0**	35.1
en-bs	4.7	**28.1**	27.9	4.1	**24.2**	23.6
ca-en	41.0	**44.9**	44.5	31.3	**36.3**	35.7
en-ca	38.4	**42.8**	**42.8**	32.0	**35.8**	**35.8**
et-en	30.6	30.9	**32.7**	24.3	26.1	**27.4**
en-et	23.7	25.1	**25.2**	22.0	24.2	**24.5**
eu-en	19.4	23.4	**24.5**	15.7	20.4	**21.2**
en-eu	12.1	14.7	**16.2**	9.0	11.5	**12.0**
fi-en	29.0	29.6	**30.9**	22.5	**25.3**	24.8
en-fi	21.7	**23.4**	22.5	16.7	**18.7**	17.9
ga-en	29.9	**38.6**	38.4	24.7	**34.3**	33.8
en-ga	27.3	**33.1**	33.0	21.2	**25.7**	25.5
gl-en	31.4	37.4	**38.0**	27.8	34.1	**34.8**
en-gl	27.9	32.2	**32.5**	27.6	**32.7**	**32.7**
hi-en	35.2	**38.9**	38.2	27.4	**33.0**	32.6
en-hi	32.8	**34.8**	34.5	26.2	27.6	**27.7**
hr-en	33.1	35.4	**35.7**	30.1	37.0	**37.1**
en-hr	28.4	**30.5**	**30.5**	28.3	**32.7**	32.5
is-en	25.3	30.9	**31.5**	23.2	29.5	**30.1**
en-is	20.6	**24.0**	23.9	18.7	**23.0**	22.6
mt-en	41.4	51.8	**51.9**	34.2	**46.6**	**46.6**
en-mt	30.6	39.1	**39.6**	24.1	31.6	**32.1**
nn-en	1.4	21.0	**23.1**	1.1	18.3	**21.1**
en-nn	0.6	14.7	**16.6**	0.4	14.0	**15.7**
sq-en	31.7	**38.3**	37.8	30.7	**40.2**	39.1
en-sq	27.8	**30.8**	30.7	29.2	**32.9**	32.7
sw-en	27.2	37.4	**38.2**	27.0	36.6	**37.1**
en-sw	28.4	**32.8**	**32.8**	30.5	34.8	**34.9**
zh_hant-en	20.3	18.2	**21.0**	18.2	16.9	**19.9**
en-zh_hant	**25.4**	18.5	23.4	**21.3**	14.2	19.6

Table 6.1: BLEU results

Pair	FLORES-200			NTREX		
	HPLT	OPUS	HPLT+OPUS	HPLT	OPUS	HPLT+OPUS
ar-en	58.5	**63.2**	63.1	54.6	**59.1**	58.9
en-ar	55.0	56.5	**56.9**	50.6	52.0	**52.2**
bs-en	38.0	**62.7**	61.7	34.0	**59.5**	58.8
en-bs	26.0	**54.9**	54.6	23.9	**50.4**	50.0
ca-en	64.4	**66.8**	66.5	57.7	**60.6**	60.2
en-ca	61.7	**64.8**	**64.8**	56.2	58.9	**59.0**
et-en	56.6	56.7	**58.1**	52.0	53.3	**54.1**
en-et	53.4	54.4	**54.8**	51.1	53.0	**53.1**
eu-en	45.7	49.3	**51.1**	41.4	46.3	**47.4**
en-eu	43.4	47.3	**49.1**	38.4	42.8	**44.0**
fi-en	55.2	54.9	**56.4**	49.7	51.3	**51.5**
en-fi	51.6	**53.1**	52.3	46.8	**48.6**	47.6
ga-en	54.9	**61.4**	61.2	51.1	**58.1**	57.7
en-ga	52.6	56.9	**57.1**	47.4	**51.3**	51.2
gl-en	57.2	61.6	**61.9**	54.1	58.4	**59.0**
en-gl	54.0	57.2	**57.5**	52.8	56.6	**56.8**
hi-en	59.9	**62.4**	62.1	54.6	**58.0**	57.9
en-hi	55.5	**57.3**	57.1	49.6	**51.0**	50.9
hr-en	58.3	60.0	**60.2**	56.5	**60.6**	**60.6**
en-hr	54.9	56.4	**56.5**	53.9	**57.1**	57.0
is-en	50.0	54.9	**55.5**	49.1	54.1	**54.6**
en-is	45.1	**48.8**	48.7	43.8	**48.0**	47.8
mt-en	64.5	71.4	**71.6**	59.8	**68.1**	68.0
en-mt	60.7	**67.2**	67.1	54.3	60.5	**60.9**
nn-en	18.0	44.3	**47.3**	16.7	41.1	**44.6**
en-nn	15.6	37.7	**39.8**	14.7	36.4	**38.3**
sq-en	58.3	**61.9**	**61.9**	57.5	**62.2**	61.8
en-sq	54.6	**56.9**	56.6	53.4	**56.3**	56.1
sw-en	51.0	59.3	**60.0**	50.4	57.8	**58.1**
en-sw	54.6	58.3	**58.5**	55.2	58.8	**59.0**
zh_hant-en	47.7	44.8	**47.8**	44.9	42.4	**45.8**
en-zh_hant	**18.9**	16.4	18.5	**21.6**	15.4	20.4

Table 6.2: ChrF++ results

Pair	FLORES-200			NTREX		
	HPLT	OPUS	HPLT+OPUS	HPLT	OPUS	HPLT+OPUS
ar-en	0.8396	**0.8664**	0.8645	0.8194	**0.8442**	0.8426
en-ar	0.8439	0.8540	**0.8542**	0.8062	0.8226	**0.8245**
bs-en	0.5882	**0.8690**	0.8663	0.5640	**0.8553**	0.8518
en-bs	0.4314	0.8809	**0.8821**	0.4178	**0.8446**	0.8435
ca-en	0.8676	**0.8778**	0.8771	0.8398	**0.8569**	0.8545
en-ca	0.8461	**0.8689**	0.8685	0.7897	**0.8227**	0.8220
et-en	0.8611	0.8646	**0.8707**	0.8335	0.8422	**0.8483**
en-et	0.8664	**0.8857**	0.8843	0.8087	**0.8417**	0.8352
eu-en	0.7810	0.8202	**0.8361**	0.7430	0.7957	**0.8081**
en-eu	0.7674	0.8111	**0.8255**	0.7114	0.7677	**0.7796**
fi-en	0.8742	0.8706	**0.8788**	0.8507	0.8581	**0.8632**
en-fi	0.8861	**0.8976**	0.8897	0.8313	**0.8587**	0.8402
ga-en	0.7653	0.8236	**0.8257**	0.7370	**0.8019**	0.7976
en-ga	0.7561	0.7903	**0.7919**	0.7142	**0.7570**	0.7530
gl-en	0.8236	0.8628	**0.8638**	0.7926	0.8379	**0.8416**
en-gl	0.8033	**0.8432**	0.8420	0.7342	0.7946	**0.7962**
hi-en	0.8741	**0.8852**	0.8838	0.8485	0.8616	**0.8620**
en-hi	0.7621	**0.7882**	0.7858	0.7231	**0.7502**	0.7465
hr-en	0.8575	0.8683	**0.8692**	0.8431	0.8622	**0.8628**
en-hr	0.8664	**0.8872**	0.8856	0.8092	**0.8546**	0.8512
is-en	0.7815	0.8372	**0.8407**	0.7797	0.8402	**0.8450**
en-is	0.7651	**0.7969**	0.7929	0.7161	**0.7652**	0.7597
mt-en	0.7601	0.8156	**0.8161**	0.7336	**0.7983**	0.7982
en-mt	0.6995	**0.7234**	0.7215	0.6773	0.6994	**0.7022**
nn-en	0.4071	0.6620	**0.7042**	0.4002	0.6370	**0.6690**
en-nn	0.5113	0.6445	**0.6749**	0.4848	0.5978	**0.6204**
sq-en	0.8468	**0.8703**	0.8685	0.8440	**0.8701**	0.8680
en-sq	0.8509	**0.8780**	0.8761	0.8023	**0.8537**	0.8517
sw-en	0.7542	0.8217	**0.8249**	0.7638	0.8256	**0.8267**
en-sw	0.7743	**0.8112**	0.8104	0.7572	**0.8082**	0.8080
zh_hant-en	0.8182	0.8007	**0.8259**	0.7900	0.7704	**0.8000**
en-zh_hant	**0.8017**	0.7335	0.7896	**0.7492**	0.6702	0.7350

Table 6.3: COMET results

Bibliography

Marta Bañón, Pinzhen Chen, Barry Haddow, Kenneth Heafield, Hieu Hoang, Miquel Esplà-Gomis, Mikel L. Forcada, Amir Kamran, Faheem Kirefu, Philipp Koehn, Sergio Ortiz Rojas, Leopoldo Pla Sempere, Gema Ramírez-Sánchez, Elsa Sarrías, Marek Strelec, Brian Thompson, William Waites, Dion Wiggins, and Jaume Zaragoza. 2020. ParaCrawl: Web-scale acquisition of parallel corpora. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4555–4567, Online. Association for Computational Linguistics.

Nikolay Bogoychev and Pinzhen Chen. 2023. Terminology-aware translation with constrained decoding and large language model prompting. In *Proceedings of the Eighth Conference on Machine Translation*, pages 890–896, Singapore. Association for Computational Linguistics.

Nikolay Bogoychev, Jelmer van der Linde, Graeme Nail, Barry Haddow, Jaume Zaragoza-Bernabeu, Gema Ramírez-Sánchez, Lukas Weymann, Tudor Nicolae Mateiu, Jindřich Helcl, and Mikko Aulamo. 2023. OpusCleaner and OpusTrainer, open source toolkits for training machine translation and large language models. ArXiv preprint.

Ahmed El-Kishky, Vishrav Chaudhary, Francisco Guzmán, and Philipp Koehn. 2020. CCAligned: A massive collection of cross-lingual web-document pairs. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5960–5969, Online. Association for Computational Linguistics.

Christian Federmann, Tom Kocmi, and Ying Xin. 2022. NTREX-128 – news test references for MT evaluation of 128 languages. In *Proceedings of the First Workshop on Scaling Up Multilingual Evaluation*, pages 21–24, Online. Association for Computational Linguistics.

Markus Freitag and Orhan Firat. 2020. Complete multilingual neural machine translation. In *Proceedings of the Fifth Conference on Machine Translation*, pages 550–560, Online. Association for Computational Linguistics.

Marcin Junczys-Dowmunt, Roman Grundkiewicz, Tomasz Dwojak, Hieu Hoang, Kenneth Heafield, Tom Neckermann, Frank Seide, Ulrich Germann, Alham Fikri Aji, Nikolay Bogoychev, André F. T. Martins, and Alexandra Birch. 2018. Marian: Fast neural machine translation in C++. In *Proceedings of ACL 2018, System Demonstrations*, pages 116–121, Melbourne, Australia. Association for Computational Linguistics.

Taku Kudo and John Richardson. 2018. 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*, pages 66–71, Brussels, Belgium. Association for Computational Linguistics.

NLLB team, Marta R. Costa-jussà, James Cross, Onur Çelebi, Maha Elbayad, Kenneth Heafield, Kevin Heffernan, Elahe Kalbassi, Janice Lam, Daniel Licht, Jean Maillard, Anna Sun, Skyler Wang, Guillaume Wenzek, Al Youngblood, Bapi Akula, Loic Barrault, Gabriel Mejia Gonzalez, Prangthip Hansanti, John Hoffman, Semarley Jarrett, Kaushik Ram Sadagopan, Dirk Rowe, Shannon Spruit, Chau Tran, Pierre Andrews, Necip Fazil Ayan, Shruti Bhosale, Sergey Edunov, Angela Fan, Cynthia Gao, Vedanuj Goswami, Francisco Guzmán, Philipp Koehn, Alexandre Mourachko, Christophe

Ropers, Safiyyah Saleem, Holger Schwenk, and Jeff Wang. 2022. No Language Left Behind: Scaling Human-Centered Machine Translation. ArXiv preprint.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Martin Popel, Markéta Tomková, Jakub Tomek, Łukasz Kaiser, Jakob Uszkoreit, Ondřej Bojar, and Zdenek Zabokrtsky. 2020. Transforming machine translation: a deep learning system reaches news translation quality comparable to human professionals. *Nature Communications*, 11:4381.

Maja Popović. 2017. chrF++: words helping character n-grams. In *Proceedings of the Second Conference on Machine Translation*, pages 612–618, Copenhagen, Denmark. Association for Computational Linguistics.

Matt Post. 2018. A call for clarity in reporting BLEU scores. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 186–191, Brussels, Belgium. Association for Computational Linguistics.

Gema Ramírez-Sánchez. 2024. D3.1: software for cleaning data sets. HPLT deliverable.

Ricardo Rei, Craig Stewart, Ana C Farinha, and Alon Lavie. 2020. COMET: A neural framework for mt evaluation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2685–2702, Online. Association for Computational Linguistics.

Holger Schwenk, Guillaume Wenzek, Sergey Edunov, Edouard Grave, Armand Joulin, and Angela Fan. 2021. CCMatrix: Mining billions of high-quality parallel sentences on the web. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 6490–6500, Online. Association for Computational Linguistics.

Jörg Tiedeman, Nikolay Arefev, Andrey Kutuzov, Stephan Oepen, Jaume Zaragoza-Bernabeu, Mikko Aulamo, Ona de Gibert Bonet, and Pavel Straňák. 2023. D2.1: initial release of monolingual and parallel data sets. HPLT deliverable.

Jörg Tiedemann. 2012. Parallel data, tools and interfaces in OPUS. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*, pages 2214–2218, Istanbul, Turkey. European Language Resources Association (ELRA).

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

A. Appendix

A.1. Language codes

Code	Language
ar	Arabic
bs	Bosnian
ca	Catalan
en	English
et	Estonian
eu	Basque
fi	Finnish
ga	Irish
gl	Galician
hi	Hindi
hr	Croatian
is	Icelandic
mt	Maltese
nn	Norwegian
sq	Albanian
sw	Swahili
zh_hant	Traditional Chinese

Table A.1: Mapping between language codes and language names.