

CS 489: Introduction to Natural Language Processing

Assignment 1: Data, Tokenization and Statistical Language Models

Instructor: Freda Shi (fhs@uwaterloo.ca)
TAs: Yixuan (Tom) Wang, Sheng Yao
Released: Fri, Sep 25, 2026
Due: Fri, Oct 9, 2026, 11:59 PM ET (Waterloo Time)
Weight: 9% of final grade

You are allowed to use AI tools however you can to assist your learning and growth, but you must declare any use of AI tools in your submission. You will be fully responsible for all submitted work, and we strongly encourage you to implement the core algorithms from scratch.

Warning: if we find evidence that you used AI tools without declaring them, you will receive 0 on this assignment.

1. Overview

This assignment walks the full **data** → **tokenization** → **language modeling** pipeline:

- **Data cleaning (Part 1):** turn *raw, uncleaned* Wikimedia dump pages into usable text.
- **Tokenization (Part 1):** implement a **byte-level** BPE tokenizer from scratch, and train a **unigram** tokenizer with an existing toolkit.
- **Compression through a model (Part 2):** measure how tokenization choices move the **bits-per-character** of a *provided, fixed* neural language model.
- **Statistical language modeling (Part 3):** build a token-level n -gram language model on Part-1 BPE tokens with $\text{add-}\alpha$ smoothing.

All downstream parts run on your **cleaned plaintext** produced in Part 1. For held-out test metrics in every part, the TAs use text cleaned by the course standard solution rather than running your cleaning script on the test files. You are encouraged to use the **Google Colab free tier** as your development environment (see Section 2.1) since it offers free GPU access for Part 2.

Total points: 100.

2. Data, Compute, and Environment

2.1 Compute and Environment

Every part is designed to run on the **Google Colab free tier** (T4 GPU when needed; 2-core CPU; ~12h sessions with idle timeouts). Only Part 2 requires a GPU.

All code must run in a **uv**-managed environment. Your submission must include a `pyproject.toml` and a checked-in `uv.lock` at the project root; pin your Python version (3.11 or later) and every dependency you use. Graded metrics come from your artifacts, but we may rebuild your environment with `uv sync` and re-run your entry points via `uv run python <script>` to verify reproducibility.

2.2 Corpus and Languages

The corpus covers seven typologically diverse languages drawn from Wikipedia. Below are the line counts (including blank lines) in the raw train and development files:

Language	Morphological type	Script	Train lines	Dev lines
Finnish	Agglutinative	Latin	161,873	14,400
Hungarian	Agglutinative	Latin	192,788	20,424
German	Fusional	Latin	173,672	15,471
Russian	Fusional	Cyrillic	172,063	18,756
Arabic	Fusional (root-and-pattern)	Arabic	199,150	19,364
Mandarin Chinese	Isolating	Han characters	449,463	47,120
Vietnamese	Isolating	Latin (diacritics)	180,349	18,435

2.3 What you are given

You receive **raw, uncleaned dumps**: wiki markup, templates, HTML entities, reference markers, tables, and inter-language link clutter are all left in. Turning this into “clean” plaintext is **Part 1’s first subtask**. Download the public corpus from [the CS 489 A1 dataset on Hugging Face](https://huggingface.co/datasets/cs489-f26-a1) into your project root before starting Part 1. For example, from your project root run:

```
uvx --from huggingface_hub hf download comping/cs489-f26-a1 \
  --repo-type dataset --local-dir data
```

This creates `data/raw/` and `data/clean_reference/`. The starter code is released at the following GitHub address:

<https://github.com/waterloo-nlp/CS489-F26/tree/main/a1/starter>

Overall, the files you receive are organized as follows:

- `data/raw/{lang}/`: uncleaned Wikimedia dumps.
- `data/clean_reference/{lang}/dev.txt`: the first 50 lines of the development corpus after cleaning with the standard solution. This sample serves as a reference for the expectation for your cleaning.
- `starter/`: starter code (see its `README.md` for full documentation).
 - `part2_transformer_lm.py`, `part2_train.py`, `part2_eval_bpc.py`: the fixed Part-2 Transformer, its trainer, and the bits-per-character evaluator.
 - `part3_artifacts.py`: save/load helpers that define the Part-3 model format.
 - `data.py`: corpus and token-id IO helpers.
 - `part2_tokenizer.py`: the Part-2 tokenizer interface stub you must implement.
 - `part2_config.example.json` and `pyproject.toml`: templates for the Part-2 config and the uv environment.

Copy the starter files into your project. The fixed files (`part2_transformer_lm.py`, `part2_train.py`, `part2_eval_bpc.py`, `data.py`, `part3_artifacts.py`, and the non-tokenizer parts of the config file) should not be modified; we restore these fixed files before grading your assignment. Your implemented `part2_tokenizer.py` and extended `pyproject.toml` are part of your submission.

3. Part 1: Data Cleaning and byte-BPE Tokenizer (35 pts)

Background

Byte-level Byte-Pair Encoding (BPE) learns a vocabulary of merge rules from raw text. Starting from a base vocabulary of the 256 symbols corresponding to **256 possible byte values**, it iteratively finds the most frequent adjacent pair of symbols and merges it into a new symbol, until the target vocabulary size V is reached.

To evaluate a tokenizer, **compression ratio** measures the number of tokens per original UTF-8 byte:

$$\text{Compression Ratio} = \frac{\text{length of tokenized text (tokens)}}{\text{length of original UTF-8 text (bytes)}}$$

A lower ratio is better; at $V=256$, byte-level BPE has a compression ratio of 1: each byte exactly corresponds to one token.

Subtasks

1. (6 pts) **Data cleaning.** Turn the raw Wikimedia dumps into plaintext. Keep only visible and meaningful content on wiki pages: remove wiki markup, standard page templates, reference markers, table markup, and interlanguage links; decode HTML entities and normalize whitespace. Document your cleaning decisions and report the UTF-8 byte counts of the cleaned train and dev files for each language in the table below. The raw byte counts include markup; the Finnish cleaned counts are the course standard solution's reference example.

Language	Raw train	Clean train	Raw dev	Clean dev
Finnish	17,029,153	10,088,172	1,678,953	1,007,274
Hungarian	17,379,397	_____	1,790,559	_____
German	18,347,716	_____	1,749,689	_____
Russian	30,104,329	_____	3,137,332	_____
Arabic	30,075,317	_____	2,987,657	_____
Mandarin Chinese	54,629,674	_____	5,226,036	_____
Vietnamese	21,539,469	_____	2,208,053	_____

The six languages other than Finnish are worth one point each: you receive the point when *both* cleaned train and dev byte counts are within 10% of the course standard solution's counts for that language.

Deliverables:

- `part1.1-clean.py`
 Command: `uv run python part1.1-clean.py --input data/raw/ --output artifacts/part1/cleaned/`
 For every file found under the input directory (recursively), it writes the cleaned text to the file at the same relative path under the output directory. For example, the cleaned version of `data/raw/fi/train.txt` must appear at `artifacts/part1/cleaned/fi/train.txt`.
- `part1.1-stats.py`
 Command: `uv run python part1.1-stats.py --raw-input data/raw/ --clean-input artifacts/part1/cleaned/`
 It reads both directories and prints raw and cleaned train/dev byte counts for each language; the output format is up to you.
- `writeup.pdf`: include your completed table and cleaning decisions, and submit both scripts. **Do not submit the large cleaned files under `artifacts/`.**

Tips

- When processing Chinese, there might be multiple variations of the same word marked by `-{ }-`. The standard solution prioritizes keeping `zh-hans`, followed by `zh-cn`, and then `zh-sg`.

- When counting bytes, the new line character `\n` is counted as one byte as well.
2. (7 pts) **Byte-level BPE merge loop.** Operate on the **UTF-8 byte stream** (base vocabulary 256). Treat the cleaned **training data of all seven languages as a single corpus**: concatenate every `train.txt` file under the input directory in sorted order of their relative paths. You may decide whether BPE merges can cross line boundaries during training and encoding; apply the same convention consistently, and explain your choice and reasoning in `writeup.pdf`. Part 3 still requires tokenizing each line independently, regardless of your choice here. Your implementation must count adjacent byte-pair frequencies, select and merge the most frequent pair, update the corpus and vocabulary, and repeat until size V . Store the ordered list of merge rules so the tokenizer can be serialized and reloaded.

Deliverables:

- `part1.2-bpe.py`
 Command: `uv run python part1.2-bpe.py --input artifacts/part1/cleaned --output artifacts/part1/bpe/ --vocab-size $V`
 It must write the following two artifact files under the output directory:
 - `bpe_vocab.txt`: one token per line in token-id order (the first line is token id 0), V lines in total. Each line is the lowercase hex encoding of the token's byte sequence; for example, the token `ing` appears as `696e67`.
 - `bpe_merges.txt`: one merge per line in learning order, $V - 256$ lines in total. Each line contains the two merged byte sequences as lowercase hex strings separated by a single space; for example, the line `696e 67` merges the tokens `in` and `g` into `ing`.
 - Submit the script together with the two artifact files under `artifacts/part1/bpe/`. The two artifacts are produced with $V = 50,000$. The TAs will load them to compute the compression ratio on the held-out test set; you receive full credit if your ratio is no more than 5% above the course standard solution's ratio.
3. (8 pts) **Encoder and decoder.** Apply the learned merges in order to new text; decode a token sequence back to the *exact* original bytes. Because every byte value is in the base vocabulary, byte-level BPE has **no out-of-vocabulary problem**: explain why this is an advantage over character-level BPE.

Deliverables:

- (3 pts) `part1.3-encode.py`
 Command: `uv run python part1.3-encode.py --input $FILENAME --bpe artifacts/part1/bpe/ --output artifacts/part1/encoded/$FILENAME`
 It loads the vocabulary and merges produced by `part1.2-bpe.py` and encodes the single input file into a token-id sequence, written to the output file.
 Output format: one token id (decimal integer) per line, in order.
- (3 pts) `part1.3-decode.py`
 Command: `uv run python part1.3-decode.py --input artifacts/part1/encoded/$FILENAME --bpe artifacts/part1/bpe/ --output artifacts/part1/decoded/$FILENAME`
 It decodes the token-id sequence in the single input file back to bytes, written to the output file. Running the two scripts back to back on a cleaned file must reproduce that file byte-for-byte (check with, e.g., `diff`).
- (2 pts) `writeup.pdf`: explain the no-out-of-vocabulary advantage and submit both scripts.

4. (4 pts) **Unigram tokenizer via sentencepiece.** Train a *unigram* tokenizer (Kudo, 2018) on the same corpus (all seven languages' cleaned train.txt files concatenated) and at the same vocabulary size ($V = 50,000$) as your submitted BPE tokenizer. You must use the `sentencepiece` library for this subtask; you may also use it wherever a unigram tokenizer is needed in this assignment, including Parts 1.5 and 2. The tokenizer must be lossless: `decode(encode(text)) == text` for any cleaned input text, including line breaks, repeated whitespace, and unseen characters.

Deliverables:

- `part1.4-unigram.py`
Command: `uv run python part1.4-unigram.py --input artifacts/part1/cleaned --output artifacts/part1/unigram/ --vocab-size $V`
It trains the unigram tokenizer and writes `unigram.model` under the output directory.
- `unigram.model`: the native model file written by `sentencepiece` to the path `artifacts/part1/unigram/`; it loads with `sentencepiece.SentencePieceProcessor`. Submit it with the script; the TAs will load it and verify your results. Full credit requires a loadable, lossless `unigram.model` with a compression ratio no more than 5% above the course standard solution's ratio.

5. (10 pts) **Analysis: compression across morphological types.**

For each $V \in \{5,000, 10,000, 20,000, 50,000\}$, train *both* your BPE and unigram tokenizers on the same single corpus as subtask 2, and measure compression ratio on the cleaned `dev.txt` file of at least three languages. Plot compression ratio against V for both tokenizers. Do agglutinative languages (Finnish, Hungarian) compress differently than fusional (German, Russian, Arabic) or isolating languages (Mandarin, Vietnamese)? Relate your findings to morphological typology features of the languages.

Deliverables:

- `part1.5-analysis.py`
Command: `uv run python part1.5-analysis.py --input artifacts/part1/cleaned --output artifacts/part1/analysis/`
It trains both tokenizers at every sweep size (reusing your code from subtasks 2 and 4) and writes the plot files (PDF/PNG/etc.) under the output directory.
- Include the plots and linguistic analysis in `writeup.pdf`. Full credit requires a compression-ratio plot (both tokenizers, at least three languages, and all four vocabulary sizes) and reasonable analysis.

4. Part 2: Optimize Bits-per-Character with a Fixed Transformer (35 pts)

Background

You are given a small GPT-style Transformer language model *and its training script as a fixed opaque box* (`part2_transformer_lm.py`, `part2_train.py`), together with the fixed bits-per-character evaluator (`part2_eval_bpc.py`). **You may not modify these files nor the model configuration.** You may change only the tokenization algorithm and vocabulary size; use the same Part-1 cleaned training and development text across your Part-2 runs. A lower bits-per-character score reflects a better tokenization choice under this model training setup. Bits-per-character is

$$\text{bpc} = -\frac{1}{N_{\text{chars}}} \sum_t \log_2 P_{\Theta}(w_t | w_{<t}) = \underbrace{\left(\frac{\text{bits}}{\text{token}} \right)}_{\text{model quality}} \times \underbrace{\left(\frac{\text{tokens}}{\text{char}} \right)}_{\text{tokenizer}}.$$

Here Θ denotes the model parameters, and the sum covers all token positions. Because the denominator is the character count, bpc is comparable across different tokenizations. Part 2 is about *tokenization quality measured through a model*, not about the model itself.

Subtasks

1. (5 pts) **Baseline run.** Train the provided model on your Part-1 tokenization (your submitted byte-BPE tokenizer at $V = 50,000$); report baseline bpc on the dev set (all seven languages' cleaned dev.txt files concatenated) and confirm your pipeline runs end-to-end. Full credit requires a correct end-to-end run and a reported baseline dev bpc with at most 10% relative error, using the course standard solution's baseline dev bpc as the reference.

Deliverables:

- writeup.pdf: report your baseline dev bpc.

2. (20+10 pts) **Optimize bpc.** Minimize bits-per-character by changing *only* the vocabulary size and tokenization algorithm (including but not limited to BPE and Unigram LM and their variants). Any tokenizer is permitted if it converts the cleaned text into token IDs that the provided Transformer can use. The encoding must be **lossless**: `decode(encode(text)) == text` for any cleaned input text, including line breaks, repeated whitespace, and unseen characters. Tokenization that replaces unseen characters with an unknown token is not lossless. Tune and verify on the dev set, then submit your trained checkpoint, config, and tokenizer that achieve the best bpc (Section 6). You receive full credit if your test bpc is no more than 5% above that by the course standard solution (byte-level BPE with the canonical cleaning setup).

Extra credit (up to +10 points): if your test bpc is lower than the standard solution's, you receive extra credit based on your improvement.

Deliverables:

- part2.py
Command: `uv run python part2.py --input artifacts/part1/cleaned --output artifacts/part2/ --vocab-size $V`
It trains your chosen tokenizer at the requested vocabulary size, trains the provided Transformer on the Part-1 cleaned training text, and writes the artifacts below under the output directory.
- model.pt: a checkpoint that loads into the provided `part2_transformer_lm.py`.
- config.{json,yaml}: every value needed to rebuild the run, including the model hyperparameters expected by `part2_transformer_lm.py`, the vocabulary size, and your tokenizer algorithm and settings.
- tokenizer/: all files needed to reload your chosen tokenizer (for example, BPE vocabulary and merge files or a SentencePiece model).
- part2_tokenizer.py: define `load_tokenizer(path)` to load `tokenizer/` and return an object with `encode(text)` that maps a cleaned text string to a list of token IDs in $\{0, \dots, V - 1\}$. The TAs use this encoder with your checkpoint to compute bpc on the standard-cleaned held-out test corpus.

3. (10 pts) **Analysis: the tokenization-compression tradeoff.** Using the decomposition above, fix the tokenization algorithm and sweep at least four vocabulary sizes, plot bpc vs. V , and identify the sweet spot. Connect this to the raw compression results from Part 1: does better compression always mean lower bpc?

Deliverables:

- writeup.pdf: include the bpc-vs- V plot, the sweet spot, and the connection to Part-1 compression results.
- part2_analysis/: put the code used for this analysis in this folder; no fixed file names, command-line interface, or output format is required.

Compute: We need GPU to train the Transformer; the provided model training process is able to fit in a Colab T4 GPU with 16 GB of memory.

5. Part 3: N-gram Language Model from Scratch (30 pts)**Background**

An n -gram language model estimates the probability of a sequence by decomposing it into overlapping n -grams under a Markov assumption of order $n - 1$:

$$P(w_1, \dots, w_T) = \prod_{t=1}^T P(w_t | w_{t-n+1}, \dots, w_{t-1}), \quad \text{PPL} = 2^{-\frac{1}{T} \sum_{t=1}^T \log_2 P(w_t | w_{t-n+1:t-1})}.$$

Work at the **token level** using your submitted Part-1 byte-BPE tokenizer ($V = 50,000$), where each w_t is a BPE token. All models will be trained on the same corpus as Part 1 (all seven languages' cleaned train.txt files). The data should be processed **line by line**: each line is tokenized independently, n -grams and contexts never cross line boundaries, and a dedicated end-of-sequence token EOS (token id 50,000, one past the BPE vocabulary) is appended to every line and scored like any other token. Prepending a beginning-of-sequence token BOS (token id 50,001) to every line is optional and left to you; if you use one, record it in your submitted model and count it in the vocabulary size (see starter part3_artifacts.py). Lower token-level perplexity is better; bits-per-character is $\text{bpc} = (T/N_{\text{chars}}) \log_2 \text{PPL}_{\text{token}}$, where T is the number of scored tokens (including one EOS per line) and N_{chars} is the number of characters in the cleaned text.

Subtasks

1. (10 pts) **Maximum-likelihood n -gram model.** Implement a token-level n -gram LM with configurable order $n \in \{2, 3, 4, 5\}$: $\hat{P}_{\text{ML}}(w_t | w_{t-n+1:t-1}) = C(w_{t-n+1:t}) / C(w_{t-n+1:t-1})$. This subtask and the next are graded from your source code and the dev-set results reported in your writeup. If any dev token has zero probability under the unsmoothed model, report the dev perplexity as $+\infty$, and write down a factor (i.e., the missing n -gram) that caused the zero probability in your writeup.

Deliverables:

- part3_ngram_lm.py
Command: `uv run python part3_ngram_lm.py --input artifacts/part1/cleaned --bpe artifacts/part1/bpe/ --order $N --smoothing none`
It trains an order- N MLE model on the cleaned train files using the submitted BPE tokenizer and prints dev perplexity to standard output.
- writeup.pdf: report dev-set results for the tested orders.

2. (8 pts) **Add- α smoothing.** Implement add- α smoothing and tune α on the dev set. Report the tuned α and dev perplexity in your writeup.

Deliverables:

- `part3_ngram_lm.py`
Command: `uv run python part3_ngram_lm.py --input artifacts/part1/cleaned --bpe artifacts/part1/bpe/ --order $N --smoothing add-alpha --alpha $ALPHA`
It trains an order- N add- α model on the cleaned train files using the submitted BPE tokenizer and prints dev perplexity to standard output.
- `writeup.pdf`: report the tuned α and dev perplexity.

3. (5 pts) **Perplexity, bpc, and best-system selection.** Compute perplexity and bits-per-character on the dev set for all tested (n, α) settings, select your best configuration, serialize the trained model with the provided save helper (starter `part3_artifacts.py`), and submit it (Section 6) with a short justification. You receive full credit if the test perplexity of your submitted model is no more than 10% above the course standard solution's.

Deliverables:

- `part3.py`
Command: `uv run python part3.py --input artifacts/part1/cleaned --bpe artifacts/part1/bpe/ --output artifacts/part3/`
It tokenizes the cleaned training text line by line with your submitted Part-1 byte-BPE tokenizer, trains your selected order n and add- α setting (documented in `README.md`), and writes `model.json.gz` under the output directory.
- `model.json.gz`: your best token-level n -gram model serialized with the provided save helper. The file must record everything needed to score new text: the order n , the smoothing parameter α , the vocabulary size including special tokens, the model tables, and the BOS token if you use one (see starter `part3_artifacts.py` for the exact format). The TA loads it with the starter loader and scores the held-out test corpus line by line (appending EOS) with your Part-1 BPE tokenizer to compute perplexity and bits-per-character.
- `writeup.pdf`: include dev perplexity and bpc for the tested settings and a justification of your selection.

4. (7 pts) **Analysis.**

- Plot the best token-level perplexity vs. n for at least two languages using add- α smoothing (you may tune the values of α), evaluating your model trained on the combined training data on each language's cleaned `dev.txt` file separately with the Part-1 BPE tokenizer. What does each curve's shape say about the language, and at what n is smoothing most important?
- Reason conceptually (no additional n -gram training required): does a larger BPE vocabulary make smoothing more or less important for a token-level n -gram LM?

Deliverables:

- `writeup.pdf`: include the perplexity-vs- n plots for at least two languages and answers to both analysis questions.

6. Submission File Tree

```

a1/
  pyproject.toml      # uv project definition
  uv.lock             # locked env ('uv sync')
  part1.1-clean.py    # clean train and dev files
  part1.1-stats.py    # report raw and cleaned bytes
  part1.2-bpe.py      # byte-BPE training
  part1.3-encode.py   # byte stream -> token ids
  part1.3-decode.py   # token ids -> byte stream
  part1.4-unigram.py  # unigram training
  part1.5-analysis.py # compression plots
  part2.py            # fixed Transformer training
  part2_tokenizer.py  # Part-2 tokenizer loader
  part2_analysis/     # Part-2.3 analysis code (any format)
  part3.py            # token-level n-gram training
  part3_ngram_lm.py   # token-level n-gram model
  artifacts/
    part1/
      bpe/
        bpe_vocab.txt # one token per line (hex), in id order
        bpe_merges.txt # one merge per line, in order
      unigram/
        unigram.model # trained sentencepiece model
    part2/
      model.pt         # provided-model checkpoint
      config.{json,yaml} # config for the best bpc run
      tokenizer/       # best-run tokenizer files
    part3/
      model.json.gz    # serialized by the starter helper
  writeup.pdf          # analysis, one section per part
  README.md            # commands to reproduce everything

```

Pack the above directory tree into a single .zip archive and submit it on LEARN by the deadline.

7. Policies

Collaboration and academic integrity

All tasks are **individual work**. You may discuss high-level ideas with classmates, but you must not share or copy code, solutions, or write-up text from other students.

You may use AI tools (e.g., GitHub Copilot, ChatGPT, Claude), but you take full responsibility for all submitted work. You must be able to explain your code and every claim in your write-up. If you use generative AI tools beyond grammar checking, you must cite them in your submission, per the course collaboration and citation policy. If we find evidence that you used AI tools without declaring them, you will receive 0 on this assignment.

See the [University of Waterloo Academic Integrity Policy \(Policy 71\)](#) for more details.

Late policy

All assignments are due at **11:59 PM ET on the due date**. A universal **72-hour grace period** applies: submissions within the grace period are accepted without penalty, and no submissions are accepted after the grace period ends. Extensions beyond the grace period are considered only for medical emergencies: a valid Verification of Illness Form (VIF) must be submitted before the original due date and must cover both the original due date and the entire grace period.

Submission

Everything is submitted on LEARN by the deadline as a single .zip archive containing your code, artifacts, and write-up in the structure shown in Section 6.

The raw corpora and clean-reference excerpts are distributed via [Hugging Face](#); starter code and the Part-2 Transformer/trainer are released at [the course GitHub starter directory](#).