CS336 Assignment 4 (data): Filtering Language Modeling Data

Version `1.0.2`

CS336 Staff

Spring 2025

1 Assignment Overview

In this assignment, you will gain some hands-on experience with filtering web crawls to create data for language modeling.

What you will implement.

1. Convert Common Crawl HTML to text.
2. Filter the extracted text with various methods (e.g., harmful content, personal identifiable information, etc.).
3. Deduplicate the training data.

What you will run.

1. Train language models on different datasets to better understand the impact of particular processing decisions on performance.

What the code looks like. All the assignment code as well as this writeup are available on GitHub at:

`github.com/stanford-cs336/assignment4-data`

Please `git clone` the repository. If there are any updates, we will notify you and you can `git pull` to get the latest.

1. `cs336-basics/*`: This folder contains code for training the model we built in assignment 1. The model has been slightly optimized, replacing some of our hand-crafted components from assignment 1 with their PyTorch-native counterparts (e.g., using the PyTorch built-in cross-entropy kernel). We've also included a training script that supports multi-GPU distributed data parallel training. You will use this script to train a model on your filtered data and make a leaderboard submission—more on this later.
2. `cs336_data/*`: This is where you'll write your code for assignment 4. We created an empty module named `cs336_data`. Note that there's no code in here, so you should be able to do whatever you want from scratch.
3. `tests/*.py`: This contains all the tests that you must pass. These tests invoke the hooks defined in `tests/adapters.py`. You'll implement the adapters to connect your code to the tests. Writing more tests and/or modifying the test code can be helpful for debugging your code, but your implementation is expected to pass the original provided test suite.
4. `README.md`: This file contains more details about the expected directory structure, as well as some basic instructions on setting up your environment.

How to submit. You will submit the following files to Gradescope:

- `writeup.pdf`: Answer all the written questions. Please typeset your responses.
- `code.zip`: Contains all the code you've written.

To submit to the leaderboard, submit a PR to:

`github.com/stanford-cs336/assignment4-data-leaderboard`

See the `README.md` in the leaderboard repository for detailed submission instructions.

2 Filtering Common Crawl

While large language models are primarily trained on Internet data, most researchers don't build a web crawler to source training data for their models. Instead, they use publicly available crawls. The most popular public webcrawl comes from Common Crawl, a non-profit that provides a free corpus of web pages, making available "over 250 billion pages spanning 17 years".[1]

However, as we'll see, turning the Common Crawl (CC) dumps into usable data for language model training takes significant work. For example, the raw data from web pages is in HTML, and we want to extract text from it. In addition, many pages might be of low quality, exact or almost duplicates, have harmful content, or contain sensitive information, and we might want to filter those pages out or remove undesirable parts of their content from our dataset. In this assignment, we will set up a pipeline that performs several of these steps, turning raw Internet data into a usable training set for language models.

2.1 Looking at the data

Before implementing anything, it is always useful to look at the raw data and get a sense of it. The CC data is made available in three formats:

WARC ("Web ARChive format") files contain the raw CC data, which includes page IDs and URLs, metadata and HTTP request details (e.g., date and time of the request, server IP address), and of course the raw content of the page (e.g., HTML).

WAT ("Web Archive Transformation") files contain higher-level metadata, extracted from WARC files and dumped as a JSON object. For example, for HTML pages, this includes a list of links from that page and the page title.

WET ("Web Extracted Text") files contain extracted plain text from the raw HTML pages.

For the following problem, we'll look into one WARC file and its corresponding WET file. These files are part of the April 2018 crawl, and were downloaded as follows:

```
# Download a sample WARC file.
$ wget https://data.commoncrawl.org/crawl-data/CC-MAIN-2025-18/segments/1744889135610.12/
warc/CC-MAIN-20250417135010-20250417165010-00065.warc.gz
# Download its corresponding WET file.
$ wget https://data.commoncrawl.org/crawl-data/CC-MAIN-2025-18/segments/1744889135610.12/
wet/CC-MAIN-20250417135010-20250417165010-00065.warc.wet.gz
```

On the Together cluster, these files are available at:

- `/data/CC/example.warc.gz`

[1] `https://commoncrawl.org/`

- `/data/CC/example.warc.wet.gz`

WARNING: These files contain completely unfiltered Internet pages, which may include a large volume of potentially harmful content. If you see a document that you'd rather not read, feel free to keep scrolling and skip to another one.

Problem (`look_at_cc`): 4 points

(a) Download the WARC file above, or find the copy we provide on the cluster. Let's look at the first page in this file. This is a gzipped file, and you can browse its contents with:

```
$ zcat /data/CC/example.warc.gz | less
```

`less` lets you browse the file using keyboard arrows, Page Up, Page Down. To exit, press "q".

Look at the very first web page. What is its URL? Is it still accessible? Can you tell what the page seems to be about by looking at the raw HTML?

Deliverable: A 2-3 sentence response.

(b) Let's now look at the corresponding WET file:

```
$ zcat /data/CC/example.warc.wet.gz | less
```

Note that the WET files contain HTTP headers (e.g., `Content-Length`) that are not part of the extracted text contents. If you look at the first example, you will see that it contains text that was extracted from the raw HTML you just saw.

Notice that much of the extracted text is reminiscent of the HTML structure, and not actually the page's main content. Are there parts of the text you see that you think should have been filtered out by the extractor? Think about the quality of this text as training data: what might go wrong in training a model on text that looks like this? Conversely, what useful information can a model potentially extract from this page?

Deliverable: A 3-4 sentence response.

(c) What makes a good training example is highly contextual. Describe an application domain for which this example might be useful to have in the training data, and one where it might not be.

Deliverable: A 1-2 sentence response.

(d) Let's look at some more examples to get a better sense of what's in the Common Crawl. Look through 25 more WET records. For each record, very briefly comment on the document's language (if you can identify it), the domain name, what type of page it is, etc. How many examples does it take until you see what you'd deem a "high-quality" webpage?

Deliverable: Brief annotations of 25 documents with the document's language, domain, type of page, and any other miscellaneous notes about the document. The number of examples it takes until you see a high-quality example.

2.2 HTML to text conversion

As you might have realized from looking at the WARC and WET files above, extracting text from HTML is challenging. Typically, any extraction procedure will look for visible content in the HTML (such as `<p>` tags, which are supposed to contain blocks of text). But that can still extract much more than what we would perceive as the main content of a page when opening it in a Web browser. For example, when opening

StackOverflow, the main content is in the question and answers, but technically the menu options, links to unrelated pages in other StackExchange, footer, links to sign up or log in — those are all visible text, and it is challenging to distinguish those reliably from the page's main content.

Many tools implement text extraction pipelines. In this assignment, we will use the Resiliparse[2] library for performing text extraction. Resiliparse will also help with an even more basic problem: detecting the text encoding of the bytes containing the raw content. Although most pages on the Web are encoded in UTF-8 (98.2%, according to Wikipedia), our text extraction pipeline should be robust to other encodings as well.

Note: We suggest using the FastWARC library to iterate over records in each WARC file. Specifically, the following classes may be helpful:

```
from fastwarc.warc import ArchiveIterator, WarcRecordType
```

Problem (`extract_text`): 3 points

(a) Write a function that extracts text from a byte string containing raw HTML. Use `resiliparse.extract.html2text.extract_plain_text` to perform the extraction. This function needs a string, so you will need to first decode the byte string into a Unicode string. Be aware that the input byte string might not be encoded in UTF-8, so your function should be able to detect the encoding in case UTF-8 fails. Resiliparse also offers `resiliparse.parse.encoding.detect_encoding()`, which might be useful.

Deliverable: A function that takes a byte string containing HTML and returns a string containing the extracted text. Implement the adapter `[run_extract_text_from_html_bytes]` and make sure it passes `uv run pytest -k test_extract_text_from_html_bytes`

(b) Run your text extraction function on a single WARC file. Compare its output to the extracted text in the corresponding WET file. What differences and/or similarities do you notice? Which extraction seems better?

Deliverable: 2-3 sentence response comparing and contrasting the text extracted by your own function versus the extracted text in the WET files.

2.3 Language identification

The web contains pages written in thousands of languages. But training a multilingual model that can effectively make use of such diverse data at scale is challenging at most compute budgets. Thus, many language modeling training sets derived from Common Crawl contain data from a limited set of languages.

A useful library for this purpose is fastText (`https://fasttext.cc`), which provides efficient text classifiers. The library provides both the infrastructure to train classifiers on your own data and a collection of pre-trained models, including for language identification. You can download the fastText language identification model `lid.176.bin` from `https://fasttext.cc/docs/en/language-identification.html`; it is also available at `/data/classifiers/lid.176.bin` on the Together cluster.

Typically, language filters use a score given by the classifier to decide whether to keep a given page. Use the fastText language identification classifier to implement a language identification filter, which should give a non-negative score for how confident it is in its prediction.

[2] `https://resiliparse.chatnoir.eu/en/stable/index.html`

Problem (`language_identification`): 6 points

(a) Write a function that will take a Unicode string and identify the main language that is present in this string. Your function should return a pair, containing an identifier of the language and a score between 0 and 1 representing its confidence in that prediction.

Deliverable: A function that performs language identification, giving its top language prediction and a score. Implement the adapter `[run_identify_language]` and make sure it passes both tests in `uv run pytest -k test_identify_language`. Note that these tests assume a particular string identifier for English ("en") and Chinese ("zh"), so your test adapter should perform any applicable re-mapping, if necessary.

(b) The behavior of language models at inference time largely depends on the data they were trained on. As a result, issues in the data filtering pipeline can result in problems downstream. What issues do you think could arise from problems in the language identification procedure? In a higher-stakes scenario (such as when deploying a user-facing product), how would you go about mitigating these issues?

Deliverable: A 2-5 sentence response.

(c) Run your language identification system on text extracted from the WARC files (via your previously-implemented text extraction function). Manually identify the language in 20 random examples and compare your labels with the classifier predictions. Report any classifier errors. What fraction of documents are English? Based on your observations, what would be a suitable classifier confidence threshold to use in filtering?

Deliverable: A 2-5 sentence response.

2.4 Personal identifiable information

The web contains large quantities of information that can be used to reach or identify individuals, such as email addresses, phone numbers, or IP addresses. We might not want a user-facing language model to output such information about real people, and a common step is to mask out these pieces of information in the training dataset.

You will now implement three procedures for masking out (a) email addresses, (b) phone numbers, and (c) IP addresses.

Problem (`mask_pii`): 3 points

1. Write a function to mask out emails. Your function will take a string as input, and replace all instances of email addresses with the string `"|||EMAIL_ADDRESS|||"`. To detect email addresses, you can look up regular expressions that do this reliably.

   Deliverable: A function that replaces all email addresses in a given string with the string `"|||EMAIL_ADDRESS|||"`, returning a pair containing both the new string and the number of instances that were masked. Implement the adapter `[run_mask_emails]` and make sure it passes all tests in `uv run pytest -k test_mask_emails`.

2. Write a function to mask out phone numbers. Your function will take a string as input, and replace all instances of phone numbers with the string `"|||PHONE_NUMBER|||"`. Doing this reliably can be extremely challenging, as phone numbers might be written in an extremely diverse set of

formats, but you should try to capture at least the most common phone number formats used in the United States, and be robust to minor syntactic deviations.

Deliverable: A function that replaces phone numbers in a given string with the string `"|||PHONE_NUMBER|||"`, returning a pair containing both the new string and the number of instances that were masked. Implement the adapter `[run_mask_phone_numbers]` and make sure it passes `uv run pytest -k test_mask_phones`.

3. Write a function to mask out IP addresses. For this problem, it is enough to focus on IPv4 addresses (4 numbers up to 255 separated by points). Your function will take a string as input, and replace all instances of IP addresses with the string `"|||IP_ADDRESS|||"`.

   Deliverable: A function that replaces IPv4 addresses in a given string with the string `"|||IP_ADDRESS|||"`, returning a pair containing both the new string and the number of instances that were masked. Implement the adapter `[run_mask_ips]` and make sure it passes `uv run pytest -k test_mask_ips`.

4. What problems do you think might arise downstream in a language model when these filters are naïvely applied on the training set? How might you mitigate these issues?

   Deliverable: A 2-5 sentence response.

5. Run your PII masking functions on text extracted from the WARC files (via your previously-implemented text extraction function). Look through 20 random examples where a replacement was made; give some examples of false positives and false negatives.

   Deliverable: A 2-5 sentence response.

2.5 Harmful content

Unfiltered dumps from the Web contain a large volume of text that we would not want a language model to repeat during inference. Some of these training examples can come even surprisingly from generally harmless websites such as Wikipedia — for instance, comments left by users on several pages can be rather toxic. Although it is essentially impossible to establish a clear line around what is harmful, many data filtering pipelines still do *some* filtering of pages that contain mostly harmful content.

There are many approaches for identifying such content, including counting words from a ban list, or building simple classifiers on labels given by human raters. In this part of the assignment, we will focus on identifying two broad categories of harmful content: "Not safe for work" (NSFW) (which includes pornography, profanity, or other potentially disturbing content), and toxic speech ("rude, disrespectful, or unreasonable language that is likely to make someone leave a discussion"[3]). We will use the `fasttext` pre-trained models made available by the Dolma project [Soldaini et al., 2024] to judge whether or not an input piece of text belongs to either of these categories. These classifiers were trained on the Jigsaw Toxic Comments dataset, which includes Wikipedia comments categorized under a variety of labels.[4]

The NSFW classifier is available for download at:
dolma-artifacts.org/fasttext_models/jigsaw_fasttext_bigrams_20230515/jigsaw_fasttext_bigrams_nsfw_final.bin

The hate speech classifier is available at:
dolma-artifacts.org/fasttext_models/jigsaw_fasttext_bigrams_20230515/jigsaw_fasttext_bigrams_hatespeech_final.bin
We've also placed these two classifiers in the Together cluster:

- `/data/classifiers/dolma_fasttext_hatespeech_jigsaw_model.bin` is a pre-trained classifier for hate and toxic speech
- `/data/classifiers/dolma_fasttext_nsfw_jigsaw_model.bin` is a pre-trained classifier for NSFW content

[3] `https://current.withgoogle.com/the-current/toxicity/`
[4] `https://paperswithcode.com/dataset/toxic-comment-classification-challenge`

Use these models to implement a function that takes a Unicode string containing the content of a page and returns a label (e.g., "toxic", "non-toxic"), along with a confidence score.

Problem (`harmful_content`): 6 points

---

1. Write a function to detect NSFW content.

   Deliverable: A function that labels a given string as containing NSFW content or not, returning a pair containing both the label and a confidence score. Implement the adapter `[run_classify_nsfw]` and make sure it passes `uv run pytest -k test_classify_nsfw`. Note that this test is just a sanity check, taken from the Jigsaw dataset, but by no means asserts that your classifier is accurate, which you should validate.

2. Write a function to detect toxic speech.

   Deliverable: A function that labels a given string as consisting of toxic speech or not, returning a pair containing both the label and a confidence score. Implement the adapter `[run_classify_toxic_speech]` and make sure it passes `uv run pytest -k test_classify_toxic_speech`. Again, this test is just a sanity check, also taken from Jigsaw.

3. What problems do you think might arise downstream in a language model when these filters are applied to create the training set? How might you mitigate these issues?

   Deliverable: A 2-5 sentence response.

4. Run your harmful content filters on text extracted from the WARC files (via your previously-implemented text extraction function). Look through 20 random examples and compare the classifier predictions to your own judgments. Report any classifier errors. What fraction of documents are harmful? Based on your observations, what would be suitable classifier confidence threshold(s) to use in filtering?

   Deliverable: A 2-5 sentence response.

2.6 Quality Rules

Even after filtering pages by language and removing harmful content, a substantial fraction of the pages that remain will still be of low quality for language model training. Again, while "quality" is not easy to define, looking through Common Crawl examples is useful for identifying examples of low quality content, such as:

- Pages with pay-walled content
- Placeholder pages for broken links
- Log in, sign up or contact forms
- Pages with primarily non-textual content, which get lost during text extraction (e.g., photos, videos)

The Gopher paper [Rae et al., 2021] describes a set of simple quality filters to remove similar simple cases of low-quality text from web-scraped data. These filters consist of simple heuristic rules that are easily understood, and they can often cover many cases of trivially unsuitable examples. The Gopher quality filters include several criteria based on document length, word length, symbol-to-word ratios, and the presence of certain English stop words. For this assignment, you will implement a subset of the filters described in the Gopher paper [Rae et al., 2021]. Specifically, you should remove documents that:

- Contain less than 50 or more than 100,000 words.
- Have a mean word length outside the range of 3 to 10 characters.
- Have more than 30% of lines ending with an ellipsis ("...").
- Contain less than 80% of words with at least one alphabetic character.

For a full description of all the quality filters the Gopher paper used, please refer their Appendix A.

Problem (`gopher_quality_filters`): 3 points

(a) Implement (at least) the subset of the Gopher quality filters as described above. For tokenizing text into words, you might find the NLTK package useful (specifically `nltk.word_tokenize`), though you're not required to use it.

Deliverable: A function that takes a string as its only argument and returns a boolean indicating whether the text passes the Gopher quality filters. Implement the adapter `[run_gopher_quality_filter]`. Then, make sure your filters pass the tests in `uv run pytest -k test_gopher`.

(b) Run your rule-based quality filter on text extracted from the WARC files (via your previously-implemented text extraction function). Look through 20 random examples and compare the filter predictions to your own judgment. Comment on any cases where the quality filters differ from your judgments.

Deliverable: A 2-5 sentence response.

2.7 Quality Classifier

Now let's go beyond the simple syntactic criteria for quality captured by the Gopher rules. Language model training is not the first place where the need to rank content by quality appears. In particular, text quality is also a fundamental challenge in information retrieval. One of the classic signals that search engines leverage is the structure of links on the web: high-quality pages tend to link to other high-quality pages [Page et al., 1998]. OpenAI used a similar insight when constructing WebText, the dataset that GPT-2 [Radford et al., 2019] was trained on: they collected pages that were linked by Reddit comments above a minimum "karma" threshold. An alternative to Reddit is using Wikipedia as a source of high-quality links, since external sources linked from Wikipedia pages tend to be trusted pages [Touvron et al., 2023].

While using a controlled source generally leads to high-quality content, the resulting dataset ends up being small for today's standards (OpenWebText had 40GB of text, whereas The Pile is more than 20x larger). One approach is to use these reference pages as positive examples, (random) pages from Common Crawl as negative examples, and train a fastText classifier. This classifier gives a quality score that you can use to filter out pages from the entire Common Crawl. Setting the quality threshold trades off precision and recall.

In this part of the assignment, you will build a quality classifier. For your convenience, we have extracted the URLs of references pages from a recent Wikipedia dump and have placed them on the Together cluster at `/data/wiki/enwiki-20240420-extracted_urls.txt.gz`.[5] This file contains a list of 43.5M external links found on Wikipedia pages in the English language as of April of 2024, but we expect you to subsample these URLs to get positive examples of "high-quality" text for training your classifier. Note that these positive examples may still contain undesirable content, so it may be useful to apply the other primitives you've built

[5]The Wikipedia reference URLs are also available for download at `https://nlp.stanford.edu/data/nfliu/cs336-spring-2024/assignment4/enwiki-20240420-extracted_urls.txt.gz`.

(e.g., language identification, filtering rules, etc.) to further improve their quality. Given a file of URLs, you can use `wget` to scrape their contents in WARC format:

```
wget --timeout=5 \
    -i subsampled_positive_urls.txt \
    --warc-file=subsampled_positive_urls.warc \
    -O /dev/null
```

Problem (`quality_classifier`): 15 points

(a) Train a quality classifier that, given text, returns a numeric quality score.

Deliverable: A quality classifier for use in the next subproblem.

(b) Write a function that labels a page as high or low-quality, and provides a confidence score in the label.

Deliverable: A function taking a string as its only argument, and returning a pair with a label (high-quality or not) and a confidence score. Implement the adapter `[run_classify_quality]`. As a sanity check, make sure it correctly classifies the two examples we provide by running `uv run pytest -k test_classify_quality`.

3 Deduplication

The web contains a significant amount of duplicate content. Some pages are exact duplicates of each other — think about archives, or default pages generated by standard tools, like 404 pages from popular Web servers. But most of the duplication happens at a more granular level. For example, consider all the question pages on Stack Overflow. While each page has unique content (e.g., the question, comments, the answers themselves), all pages also have a substantial amount of redundancy, such as the header, menu options and footer, that will be repeated in exact form when all such pages are rendered. In the first part of this section, we will deal with this kind of exact duplication, and later see how to handle approximate duplicates.

3.1 Exact line deduplication

A simple approach for deduplicating exact repetition is to only keep the lines in a document that are unique in the corpus. This turns out to be sufficient to eliminate a large portion of redundancy, such as the header and menu options we mentioned above. In the simpler cases, when removing lines that are exactly repeated elsewhere, we are often left with the unique main content of each page (such as the question and answers on StackOverflow).

To do this, we can make one pass through the corpus to count how many occurrences of each line we observe. Then, in a second pass, we can rewrite each document by preserving only its unique lines.

Naïvely, the data structure to keep the counters can use as much space as it takes to store all the unique lines in the corpus. One simple memory efficiency trick is to instead use a hash of the line as the key, making the keys have a fixed size (instead of depending on the line's length). You will now implement this simple deduplication method.

Problem (`exact_deduplication`): 3 points

Write a function that takes a list of paths to input files and performs exact line deduplication on them. It should first count the frequency of each line in the corpus, using a hash to reduce memory,

and then rewrite each file by only keeping its unique lines.

Deliverable: A function that performs exact line deduplication. Your function should take two arguments: (a) a list of paths to its input files, and (b) an output directory. It should rewrite each input file to the output directory with the same name, but deduplicate the content by removing lines that occur more than once in the set of input files. For example, if the input paths are `a/1.txt` and `a/2.txt`, and the output directory is `b/`, your function should write the files `b/1.txt` and `b/2.txt`. Implement the adapter `[run_exact_line_deduplication]` and make sure it passes `uv run pytest -k test_exact_line_deduplication`.

3.2 MinHash + LSH document deduplication

Exact deduplication is useful for removing content that is repeated verbatim across multiple webpages, but does not handle cases where the document content slightly differs. For example, consider software license documents—the license document is often generated from a template that requires the year and the software author's name. As a result, the license file for one MIT-licensed project contains largely the same content as another MIT licensed-project, but they aren't *exact* duplicates. To remove this type of repeated, mostly-templated content, we need fuzzy deduplication. To efficiently perform fuzzy document-level deduplication, we will use minhash with locality sensitive hashing (LSH).[6]

To perform fuzzy deduplication, we will use a particular notion of similarity between documents: the Jaccard similarity between each document's ngrams. The Jaccard similarity between sets S and T is defined as $|S \cap T|/|S \cup T|$. To naively perform fuzzy deduplication, we could represent each document as a set of n-grams, and compute the Jaccard similarity between all pairs of documents, marking pairs as duplicates if they exceed a particular Jaccard similarity threshold. However, this method is impractical for large document collections (e.g., Common Crawl). Furthermore, naively storing a set of n-grams will take much more memory than a document.

MinHashing To address the memory concern, we replace our set of n-grams document representation with a *signature*. In particular, we'd like to construct signatures such that if we compare the signatures of two documents, we get an approximation of the Jaccard similarity between the documents' respective sets of ngrams. Minhash signatures fulfill these properties. To compute the minhash signature for a set of document n-grams $S = \{s_1, ..., s_n\}$, we will need k distinct hash functions $h_1, ..., h_k$.[7] Each hash function maps an n-gram to an integer. Given a hash function h_i, the minhash of the set of document ngrams S is $\text{minhash}(h_i, S) = \min(h_i(s_1), h_i(s_2), ..., h_i(s_n))$. The signature of the document n-grams S is a vector in $\mathbb{R}^k$, where each element i contains the minhash of S under the random hash function h_i, i.e, $[\text{minhash}(h_1, S), \text{minhash}(h_2, S), ..., \text{minhash}(h_k, S)]$.

It turns out that for two documents n-gram sets S_1 and S_2, the Jaccard similarity between the sets is approximated by the proportion of columns with the same minhash value (for a proof, see section 3.3.3 in chapter 3 of Leskovec et al., 2014). For example, given the documents signatures $[1, 2, 3, 2]$ and $[5, 2, 3, 4]$, the Jaccard similarity between the n-gram sets is approximated as $2/4$, since the second and third columns of these signatures have the same minhash value.

Locality-sensitive hashing (LSH) Although minhashing gives us a memory-efficient document representation that preserves the expected similarity between any document pair, we're still left with the need to compare all pairs of documents to find those with the greatest similarity. LSH provides a way to efficiently bucket documents that are likely to have high similarity. To apply LSH to our document signatures (now a vector in $\mathbb{R}^k$), we will divide the signature into b bands containing r minhashes, with $k = br$. For example, if we have 100-element document signatures (generated from 100 random hash functions), we can break this

[6] For a more in-depth treatment of LSH and minhashing, we refer the reader to Chapter 3 of Leskovec et al. [2014]. The chapter is available online at infolab.stanford.edu/ ullman/mmds/ch3n.pdf.

[7] These k distinct hash functions could be of the same family, but with different seeds. For example, MurmurHash3 is a family of hash functions, and using a particular seed instantiates a particular hash function within that family.

up into 2 bands of 50 minhashes each, or 4 bands of 25 minhashes, or 50 bands of 2 minhashes, etc. If two documents have the same hash values for a particular band, they will be clustered into the same bucket and will be considered as candidate duplicates. Thus, for a fixed number of signatures, increasing the number of bands will increase recall and decrease precision.

For a concrete example of the above, suppose that we have a document D_1 with a minhash signature $[1, 2, 3, 4, 5, 6]$ and another document D_2 with a minhash signature $[1, 2, 3, 5, 1, 2]$. If we use 3 bands with 2 minhashes each, then the first band of D_1 is $[1, 2]$, the second band of D_1 is $[3, 4]$, and the third band of D_1 is $[5, 6]$. Similarly, the first band of D_2 is $[1, 2]$, the second band of D_2 is $[3, 5]$, and the third band of D_2 is $[1, 2]$. Since the hashes in the first band match ($[1, 2]$ for both documents), D_1 and D_2 would be clustered together under the first band. They would not be clustered together under any of the other bands since the hashes do not match. However, note that since the documents are clustered together under at least one band, they'd be considered candidate duplicates regardless of whether the other bands match.

Once we've identified the candidate duplicates, we can process them in a variety of ways. For example, we could compute the Jaccard similarity between all candidate duplicates and label pairs that exceed a set threshold as duplicates.

Finally, we cluster duplicate documents across buckets. For example, suppose that documents A and B match in one bucket and have a true Jaccard similarity that's higher than our threshold, and that documents B and C match in another bucket and also have a true Jaccard similarity that's higher than our threshold. Then, we'd treat documents A, B, and C as a single cluster. We randomly remove all but one of the documents in each cluster.

Problem (`minhash_deduplication`): 8 points

Write a function that takes a list of paths to input files and performs fuzzy document deduplication with minhash and LSH. In particular, your function should compute minhash signatures for each document in the provided list of paths, use LSH with the provided number of bands to identify candidate duplicates, and then compute the true ngram Jaccard similarity between candidate duplicates and remove those that exceed a given threshold. To improve recall (following Penedo et al., 2023), normalize the text before computing minhash signatures and/or comparing Jaccard similarity by lowercasing, removing punctuation, normalizing whitespaces, and removing accents, and applying NFD unicode normalization.

Deliverable: A function that performs fuzzy document deduplication. Your function should take at least the following arguments: (a) a list of paths to its input files, (b) the number of hashes to use for computing minhash signatures, (c) the number of bands to use for LSH, (d) the n-gram length (in words) for computing minhash signatures, and (e) an output directory. You may assume that the number of hashes to use for computing minhash signatures is evenly divisible by the number of bands to use for LSH.

Your function should rewrite each input file to the output directory with the same name, but only writing documents that are either (a) not candidate duplicates and/or (b) are randomly selected to be retained from the clustered buckets. For example, if the input paths are `a/1.txt` and `a/2.txt`, and the output directory is `b/`, your function should write the files `b/1.txt` and `b/2.txt`. Implement the adapter `[run_minhash_deduplication]` and make sure it passes `uv run pytest -k test_minhash_deduplication`.

4 Leaderboard: filter data for language modeling

Now that we've implemented a variety of primitives for filtering web crawl data, let's put it to use and generate some language modeling training data.

In this part of your assignment, your goal is to filter a collection of CC WET files to produce language modeling training data. We've placed 5000 WET files at `/data/CC/CC*.warc.wet.gz` for you to use as a starting point.

In particular, your goal is to filter the CC dump to create language modeling data so that a trained Transformer language model minimizes validation perplexity on the C4 100 domains subset of the Paloma benchmark [Magnusson et al., 2023]. **You should not modify the model architecture or training procedure**, since the goal is to construct the best *data*. This dataset contains samples from the 100 most common domains in the C4 language modeling dataset [Raffel et al., 2020]. We've placed a copy of this data at `/data/paloma/tokenized_paloma_c4_100_domains_validation.npy` on the Together cluster (tokenized with the GPT-2 tokenizer)—you may find it useful to peruse the data to get a sense of what it looks like. Given your filtered dataset, you will train a GPT-2 small-shaped model for 200K iterations on this data and evaluate its perplexity on C4 100.

We note that you *are* allowed to make use of the Paloma validation data in constructing filters or classifiers to process the CC WET files, but are not allowed to literally copy any of the validation data into your training data. The language model should never see any data from the validation set.

Even 5000 WET files is a substantial amount of data, amounting to around 375GB of compressed text. To process this data efficiently, we recommend using multi-processing wherever possible. In particular, you may find the Python `concurrent.futures` or `multiprocessing` API to be helpful. Below, we present a minimal example of using `concurrent.futures` to parallelize a function across multiple processes:

```
import concurrent.futures
import os

from tqdm import tqdm

def process_single_wet_file(input_path: str, output_path: str):
    # TODO: read input path, process the input, and write the output to output_path
    return output_path

# Set up the executor
num_cpus = len(os.sched_getaffinity(0))
executor = concurrent.futures.ProcessPoolExecutor(max_workers=num_cpus)
wet_filepaths = ["a.warc.wet.gz", "b.warc.wet.gz", "c.warc.wet.gz"]
output_directory_path = "/path/to/output_directory/"

futures = []
for wet_filepath in wet_filepaths:
    # For each warc.wet.gz filepath, submit a job to the executor and get a future back
    wet_filename = str(pathlib.Path(wet_filepath).name)
    future = executor.submit(
        process_single_wet_file,
        wet_filepath,
        os.path.join(output_directory_path, wet_filename)
    )
    # Store the futures
    futures.append(future)

# Iterate over the completed futures as they finish, using a progress bar
# to keep track of progress.
for future in tqdm(
        concurrent.futures.as_completed(futures),
        total=len(wet_filepaths),
):
    output_file = future.result()
```

```
    print(f"Output file written: {output_file}")
```

To parallelize your data processing across the Slurm cluster, you can use the `submitit`[8] package, which provides a drop-in replacement to `concurrent.futures` that will handle launching jobs on a specified Slurm partition and collecting the results:

```
import os

import submitit
from tqdm import tqdm

def process_single_wet_file(input_path: str, output_path: str):
    # TODO: read input path, process the input, and write the output to output_path
    return output_path

# Set up the submitit executor
executor = submitit.AutoExecutor(folder="slurm_logs")
max_simultaneous_jobs = 16
wet_filepaths = ["a.warc.wet.gz", "b.warc.wet.gz", "c.warc.wet.gz"]
output_directory_path = "/path/to/output_directory/"
# Configure parameters of each job launched by submitit
executor.update_parameters(
    slurm_array_parallelism=max_simultaneous_jobs,
    timeout_min=15,
    mem_gb=2,
    cpus_per_task=2,
    slurm_account="cs336_user",
    slurm_partition="a4-cpu",
)
futures = []
# Use exector.batch() context manager to group all of the jobs in a Slurm array
with executor.batch():
    for wet_filepath in wet_filepaths:
        # For each WARC filepath, submit a job to the executor and get a future back
        wet_filename = str(pathlib.Path(wet_filepath).name)
        future = executor.submit(
            process_single_wet_file,
            wet_filepath,
            os.path.join(output_directory_path, wet_filename)
        )
        # Store the futures
        futures.append(future)

# Use tqdm to display progress
for future in tqdm(
        submitit.helpers.as_completed(futures),
        total=len(wet_filepaths),
    ):
    output_file = future.result()
    print(f"Output file written: {output_file}")
```

[8] https://github.com/facebookincubator/submitit

As you can see, the code is pretty similar when using `submitit` vs. the built-in `concurrent.futures` API. The main differences are (1) the need to configure the submitit executor parameters so it knows where to submit the Slurm jobs and each job's resource specifications, (2) using `executor.batch()` to group all of the Slurm jobs into a single job "array" (rather than `len(wet_filepaths)` individual jobs), which minimizes load on the Slurm scheduler, and (3) using `submitit.helpers.as_completed` when collecting the results.

We also suggest using the FastWARC library to iterate over records in each WET file, and the tldextract library to extract domains from URLs for filtering. In particular, these classes may be helpful:

```
from fastwarc.warc import ArchiveIterator, WarcRecordType
from tldextract import TLDExtract
```

Problem (`filter_data`): 6 points

(a) Write a script to filter language modeling data from a collection of Common Crawl WET files (located at `/data/CC/CC*.warc.wet.gz` on the Together cluster). You are free to apply any of the primitives we've implemented in earlier parts of the assignment, and you're also free to explore other filters and methods for generating data (e.g., filtering based on n-gram language model perplexity). Your goal is to produce data that, when trained on, minimizes the perplexity on the C4 100 domains subset of the Paloma benchmark.

Again, we note that you *are* allowed to make use of the Paloma validation data in constructing filters or classifiers to process the CC WET files, but are not allowed to literally copy any of the validation data into your training data.

Your script should report the number of examples kept by each filter that you've used, so you have a sense of how the filters are contributing to the final output data.

Deliverable: A script (or sequence of scripts) that filters the provided CC WET files in parallel to produce language modeling data. A written breakdown of what proportion of the discarded examples are removed by each filter step.

(b) How long does it take to filter the 5,000 WET files? How long would it take to filter the entire Common Crawl dump (100,000 WETs)?

Deliverable: Runtime of the data filtering pipeline.

Now that we've produced some language modeling data, let's take a look at it to better understand its contents.

Problem (`inspect_filtered_data`): 4 points

(a) Take five random examples from your filtered dataset. Comment on their quality and whether or not they'd be suitable for language modeling, especially given that our goal is to minimize perplexity on the C4 100 domains benchmark.

Deliverable: Five random examples from the final filtered data. Only showing pertinent excerpts of the data is fine, since the documents may be lengthy. For each example, a 1-2 sentence description of the example and whether or not it's worthwhile to use for language modeling.

(b) Take five CC WETs that were removed and/or modified by your filtering script. What part of your filtering process removed or modified these documents, and do you think that their removal and/or modification was justified?

Deliverable: Five random discarded examples from the original WETs. Only showing pertinent excerpts of the data is fine, since the documents may be lengthy. For each example, a 1-2 sentence description of the example and whether or not its removal was justified.

(c) If your analysis above motivates further changes to your data pipeline, feel free to make those changes before training your model. Report any changes and/or iterations of data that you experimented with.

Deliverable: A description of data changes and/or iterations that you experimented with.

Before training language models on our data, we need to tokenize it. Use the GPT-2 tokenizer via `transformers` to encode your filtered data into a sequence of integer IDs for training language models. Don't forget to include the GPT-2 end-of-sequence token `<|endoftext|>` after each document. Here's some starter code:

```
import multiprocessing

import numpy as np
from tqdm import tqdm
from transformers import AutoTokenizer

input_path = "path/to/your/filtered/data"
output_path = "path/to/your/tokenized/data"

tokenizer = AutoTokenizer.from_pretrained("gpt2")

def tokenize_line_and_add_eos(line):
    return tokenizer.encode(line) + [tokenizer.eos_token_id]

with open(input_path) as f:
    lines = f.readlines()

pool = multiprocessing.Pool(multiprocessing.cpu_count())
chunksize = 100
results = []
for result in tqdm(
    pool.imap(tokenize_line_and_add_eos, lines, chunksize=chunksize),
    total=len(lines),
    desc="Tokenizing lines"
):
    results.append(result)

pool.close()
pool.join()

# Flatten the list of ids and convert to numpy array
all_ids = [token_id for sublist in results for token_id in sublist]
print(f"Tokenized and encoded {input_path} into {len(all_ids)} tokens")
ids_array = np.array(all_ids, dtype=np.uint16)
ids_array.tofile(output_path)
```

Problem (`tokenize_data`): 2 points

Write a script to tokenize and serialize your filtered data. Make sure to serialize following the example code above, with `ids_array.tofile(output_path)`, where `ids_array` is a `np.uint16` numpy array of integer IDs. This ensures compatibility with the provided training script.

How many tokens are in your filtered dataset?

Deliverable: A script to tokenize and serialize your filtered data, and the number of tokens in your produced dataset.

Now that we've tokenized our dataset, we can train a model on it. We will train a GPT-2 small-shaped model for 200K iterations on the generated data, periodically measuring validation performance on the C4 100 domains dataset.

First, open the config file at `cs336-basics/configs/experiment/your_data.yaml` and set the `paths.train_bin` attribute to point to the file containing your tokenized training data. You should also set an appropriate `training.wandb_entity` and `training.wandb_project` attribute for logging.

Then, you will launch training using the `train.py` script available at `cs336-basics/scripts/train.py`.[9] You can view the hyperparameters that you will use in `cs336-basics/cs336_basics/train_config.py`. We will use 2 GPUs, data parallel, and batch size 128 per device. Our training run with this config finished in around 22 hours, so be sure to budget enough time. You may launch training with the following command:

```
uv run torchrun --standalone --nproc_per_node=2 scripts/train.py --config-name=experiment/your_data
```

after ensuring that you have set the config attributes mentioned above.

Once again, the goal of the assignment is to optimize *the data* to minimize validation loss, rather than trying to minimize loss by modifying the model and/or optimization procedure, so **do not modify the training config (other than the path and wandb attributes mentioned above) or the training script**.

When testing out your data, you may also find it helpful to set the `training.save_checkpoints` config argument to True, which will save a checkpoint whenever the validation loss is evaluated. This can be done by running

```
uv run torchrun --standalone --nproc_per_node=2 \
    scripts/train.py --config-name=experiment/your_data \
    +training.save_checkpoints=True
```

which will save checkpoints to `cs336-basics/output/your_data/step_N`. Then, you can draw samples from your saved model with the following command:

```
uv run python scripts/generate_with_gpt2_tok.py \
    --model_path cs336-basics/output/your_data/step_N
```

Problem (`train_model`): 2 points

Train a language model (GPT-2 small-shaped) on your tokenized dataset. Periodically measure the validation loss on C4 100 domains (this is already enabled by default in the config at `cs336-basics/cs336_basics/train_config.py`). What is the best validation loss that your model achieves? Submit this value to the leaderboard.

Deliverable: The best validation loss that was recorded, the associated learning curve, and a

[9] https://github.com/stanford-cs336/assignment4-data/blob/master/cs336-basics/scripts/train.py

description of what you did.

References

Luca Soldaini, Rodney Kinney, Akshita Bhagia, Dustin Schwenk, David Atkinson, Russell Authur, Ben Bogin, Khyathi Chandu, Jennifer Dumas, Yanai Elazar, et al. Dolma: An open corpus of three trillion tokens for language model pretraining research. *arXiv preprint arXiv:2402.00159*, 2024.

Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bring order to the web. In *Proc. of the 7th International World Wide Web Conf*, 1998.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and efficient foundation language models, 2023. arXiv:2302.13971.

Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, USA, 2nd edition, 2014. ISBN 1107077230.

Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Alessandro Cappelli, Hamza Alobeidli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay. The RefinedWeb dataset for Falcon LLM: Outperforming curated corpora with web data, and web data only, 2023.

Ian Magnusson, Akshita Bhagia, Valentin Hofmann, Luca Soldaini, Ananya Harsh Jha, Oyvind Tafjord, Dustin Schwenk, Evan Pete Walsh, Yanai Elazar, Kyle Lo, Dirk Groeneveld, Iz Beltagy, Hannaneh Hajishirzi, Noah A. Smith, Kyle Richardson, and Jesse Dodge. Paloma: A benchmark for evaluating language model fit, 2023. arXiv:2312.10523.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020. URL `http://jmlr.org/papers/v21/20-074.html`.