Uzushio: A Distributed Huge Corpus Processor for the LLM Era

Arseny Tolmachev[1] Masayoshi Hayashi[1] Takuro Niitsuma[2] Rintaro Enomoto[3]
Hao Wang[3] Shuhei Kurita[4] Daisuke Kawahara[3] Kazuma Takaoka[1] Yoshitaka Uchida[1]
[1]Works Applications [2]The Asahi Shimbun Company [3]Waseda Univesity [4]RIKEN
arseny@kotonoha.ws hayashi_mas@worksap.co.jp niitsuma-t@asahi.com
re9484@akane.waseda.jp conan1024hao@akane.waseda.jp shuhei.kurita@riken.jp
dkw@waseda.jp takaoka_k@worksap.co.jp uchida_yo@worksap.co.jp

Abstract

We introduce Uzushio — a distributed huge corpus preprocessing tool, which was used to create the Japanese part of v2 training corpus for the LLM-jp[1)] project. It uses paragraph-centric text extraction and near-duplicate detection approaches with sampling-based deduplication. Running the detection and filtering pipeline on the ABCI system takes approximately 10 hours for 1.4B Japanese documents extracted from the Common Crawl dumps.

1 Introduction

We introduce Uzushio[2)]: an Apache Spark [1]-based huge corpus preprocessing tool which was in development from the summer of 2023. Large language and other foundational models often require a multi-billion-token training corpus, and web data, e.g. Common Crawl [2], is often used as a training corpus for these models. Because of the large scale, processing such corpora requires significant resources. Additionally, some required stages for processing such corpora, e.g. near-duplicate detection, can be non-trivial. In this paper, we give a brief introduction to Uzushio and its processing steps.

The main processing pipeline of Uzushio consists of the following stages.

- Document text extraction
- Near-duplicate detection
- Duplicate statistics merging
- Filtering

We separate duplicate detection and deduplication to enable better scalability and flexibility. Corpus deduplication is done during the filtering step, sampling from the larger corpus using the statistics computed during the near-duplicate detection step. Distributed task execution can have spurious errors and problems, thus we design the most computationally expensive part — near-duplicate detection — to be executable for the parts of the whole corpus independently of other parts, and merging the computed results later. This additionally allows us to mix and match task sizes based on the availability of computational resources, making Uzushio very flexible to the system requirements. All stages can be executed both in cloud (e.g. in AWS EMR[3)]) and in HPC environments (we provide configuration for the ABCI system).

2 Document Text Extraction

The objective of the first stage of Uzushio is to produce paragraph-separated text documents from HTML documents. Our current implementation uses WARC [3] files as input and outputs zstd [4]-compressed parquet files.

WARC Format Parsing Uzushio uses webarchive-commons[4)] library to parse WARC files. We ignore all non-response documents and filter the content of parsed WARC messages to HTTP responses with text content.

Character Encoding Detection We use the following algorithm to detect character encoding of the HTTP body. We try encodings from different sources as specified below. With each encoding, we try to decode 16k bytes of the body and look for unmappable characters. If an unmappable character exists, we try the next encoding in the list. If everything fails, we skip the current document.

1. Sniff the `<meta http-equiv="content-type" ...>` tag from the body of the document. Use the encoding

1) https://llm-jp.nii.ac.jp/
2) https://github.com/WorksApplications/uzushio
3) https://aws.amazon.com/emr/
4) https://github.com/iipc/webarchive-commons

specified in the `content=...` attribute.
2. Use the value of the `Content-Type` HTTP header.
3. Try the juniversalchardet[5] library to guess the encoding of the body.
4. Try UTF-8

Language Detection We use the language-detector [5] library to detect languages. We use at most 20 kilobytes of document body to decode at most 4k characters, while skipping all characters with the code lower than 127. The exception to the rule is to keep a single space in place of any sequence of whitespace characters. The main idea here is to skip all markup while forming the buffer for language detection.

HTML Parsing We parse the content of HTML documents using Apache Tika[6], which is widely used to extract text content from various documents. The parsing step outputs a list of document paragraphs. We define paragraphs based on the structure of the HTML document using tags which are usually block-related, e.g. `<p>` or `<div>`. The content of `<br>` tags is replaced with `\n` symbols. After parsing we remove adjacent whitespace and delete all empty or whitespace-only lines or paragraphs.

CSS selectors For each paragraph, we also record its path in the HTML document as a CSS selector. For all parent tags of the current paragraph-like tag, we record tag name, id if it exists, and classes if they exist.

Links An HTML document contains a lot of links, and pages which contain a lot of them are often not useful for LLM training. Because of this, we mark text which is link content. Namely, we surround the text content of `<a>` tags with `0x02` and `0x03` characters in the extracted text.

3 Near-Duplicate Detection

We formulate the near-duplicate detection step as follows. For each paragraph, we compute its duplication frequency: the number of times the paragraph or its near-duplicates occur in the corpus.

For each paragraph, we track two hashes: the paragraph's own hash and near-duplicate group hash. During the duplicate detection procedure, we update group hashes to be the minimal hash for the whole group. Then the group duplication frequency would be the sum of the duplication frequency for all paragraphs.

5) https://github.com/albfernandez/juniversalchardet
6) https://tika.apache.org/

SimHash Algorithm The core part of our near-duplicate detection implementation is the SimHash[6] algorithm. It is a locality-sensitive hashing algorithm based on Gaussian projections. By being locality-sensitive, similar items will get similar hashes. We use these hashes as signatures to find candidates for near-duplicate detection.

The core idea of SimHash is to multiply a feature vector by a matrix of random numbers drawn from the Gaussian distribution, basically utilizing a random Gaussian projection algorithm. The signs of the projected vector will be bits of the hash value. In our application, we use character n-gram hashes as the features.

Breaking Down Corpus into Paragraphs We compute deduplication statistics paragraph-wise, so this step breaks down documents into paragraphs. We additionally strip all link and CSS path metadata from paragraphs, resulting only in text content for near-duplicate detection.

Exact Duplication Frequencies Exact deduplication frequencies are the number of times each paragraph occurs in the original corpus. We use the standard Spark SQL aggregation functionality to compute this number. Because of this step, we perform duplicate detection once for each paragraph instance in the original corpus.

SimHash Signatures We use the SimHash algorithm to compute signatures for each paragraph. Namely, for each paragraph, we compute hashes of 2,3,4-grams and sample from the Gaussian distribution using the hash values as seeds. We use 128-bit signatures as the default size.

Sorting Corpus by SimHash Signatures Sorting by signatures puts the paragraphs with similar signature prefixes together in the sort order. In the case of distributed sort, the paragraphs with similar signature prefixes are put in the same partitions as well.

Duplicates in a Fixed Window The main duplicate detection logic checks all pairs of paragraphs in a fixed-sized window, reassigning group hashes if two paragraphs were detected as near-duplicates. We detect if two paragraphs are duplicates using the Levenstein distance for very short paragraphs (if the average length is less than 30 characters) or n-gram overlap for the longer ones.

We perform this operation independently for each partition without taking into account partition boundaries, so there could be false negatives on the partition boundaries. Repeating the process several times with different sortings mitigates this problem.

Evaluating all pairs, even in a fixed window, is still computationally expensive, so we use several tricks to reduce computational costs. Instead of comparing n-grams directly, we compute bitsets based on hashes of character n-grams and compute overlap using bitsets directly. This implementation can undercount the number of matches in the case of hash collisions but is much faster than comparing n-grams directly.

Also, we treat paragraphs as non-similar if their length differs by more than 30% or 50 characters, whichever of the two numbers is smaller. We keep paragraphs in several lists bucketed by different lengths. This allows us to completely skip processing paragraph pairs which would be regarded to be non-similar by their lengths alone.

Repeating Sorting and Checking Several Times We repeat sorting paragraphs by SimHash signatures and checking steps 5 times (by default). This helps us mitigate problems with SimHash being a non-precise algorithm and a lack of duplicate detection over the partition boundaries. Each iteration of the algorithm propagates duplicate groups by updating group hashes of the paragraphs to the minimum paragraph hash in the group.

Compute Duplicate Statistics and Finalize Statistics After the final duplicate groups are computed, we aggregate per-paragraph full duplication frequencies into per-group frequencies by summing per-group counts. This produces duplicate statistics.

4 Merging Duplicate Statistics

Because the algorithm to compute duplicate statistic data is deterministic, it is possible to merge the duplication statistics computed from two different datasets. The merge procedure corrects group hashes and the related near-duplicate frequencies.

5 Filtering

Uzushio filters form a filter chain. A filter can edit contents of a document or delete it, or some of its paragraphs. Documents that were deleted by a filter will not have downstream filters in the filter chain applied to them.

Uzushio can output all documents, grouping them by the filter which has deleted the document. For example, the best data would be documents that pass all filters, and the second best data would remain on the last filter, and so on.

Most of the filters are described in detail below. We also provide filters which rewrite parts of the document like headers or lists to Markdown syntax, which is often used by LLMs.

Duplicate Documents Subsampling The main objective of this filter is to perform corpus deduplication. For this, we estimate the duplication count of the document from the paragraph near-duplicate frequency and then sample documents so the expected number of the documents would be equal to the provided number.

Estimation is done by using percentiles, not means, because percentiles are more robust to outliers and most web documents contain several extremely frequent paragraphs.

High-Frequency Subsequent Paragraph Trimming We trim paragraphs if multiple successive ones have a frequency higher than a specified threshold. The goal of this filter is to remove text that is present on a large number of pages, like navigation or advertisements, but leave in place the main content, which should be less duplicated over the corpus. As a safety mechanism to prevent over-deletion of text, this filter supports removing paragraphs only if a specified and subsequent number of them have high near frequencies.

Document Compression Rate One good idea for filtering is based on the observation that, especially in a web corpus, texts with different characteristics have different compression rates. Namely, we compute the ratio of compressed text to uncompressed text and filter out documents that have lower or higher compression rates than the provided thresholds. We use the LZ4 algorithm to compress document text data.

Hiragana Ratio Another good heuristic to distinguish low-quality Japanese documents from other ones is the ratio of hiragana characters in text. While regular text contains hiragana, lists, advertisements, and other low-quality text contains a very low percentage of it. For this filter, we measure the ratio of hiragana characters to the total number of characters in a document.

Document Link Ratio The percentage of link text is another useful criterion for distinguishing low-quality documents. The text extraction step of Uzushio records the spans of text that were link bodies in original documents. We use that information to compute the link ratio as the number of characters that were in the links to the total number of characters in the document.

Documents that contain many links are mostly low-

Filter	Size, TB	Percentage
DuplicateDocs(5%, 5)	2.088	41.6%
Hiragana.Low	0.929	18.5%
Links.High	0.458	9.1%
DuplicatePars	0.380	7.6%
DupRatio(5%, 5)	0.151	3.0%
DupRatio(5%, 2.5)	0.151	3.0%
DupRatio(10%, 1.5)	0.463	9.2%
Total	5.015	100.0%

Table 1: Common Crawl filtering results

quality advertisement spaces or link farms and regular pages do not contain that many links. This filter helps us remove such pages.

Word List Filtering Documents can contain bad words which are not suitable for training LLMs. For example, the web contains a high number of adult-related sites. Such documents are usually removed using word lists.

We provide two types of such filters: one counts all instances of words, another counts only types of included words. However, we do not do any other filtering on word detection and any substring inclusion will be counted as a word. For example, HojiChar [7] checks that there is a different character category on the detected word boundaries. We adopt word lists developed by the HojiChar project, removing words that caused a lot of false positives.

N-gram Based Language Model Filters One of the strongest filters we implement is the n-gram language model filter which allows users to evaluate per-document or per-paragraph average perplexity and delete documents or paragraphs for which the evaluated perplexity is larger than the provided thresholds. The filter uses Sudachi for tokenization and the KenLM [8, 9] library for the n-gram language model implementation.

We provide two variations on the filter: per-document and per-paragraph ones. In addition to that, we provide an option to ignore outliers when evaluating perplexity: a certain percentage of tokens with the highest tokenwise perplexity are ignored.

6 Filtering and Extraction for LLM-jp

We extracted the text data from all Common Crawl archives until the end of 2023. There were 5.7PB of gzipped WARC archives, from which we got 5.95TB of extracted text with paragraph CSS selectors and marked link text, compressed with zstd. The extraction took about two weeks on the AWS EMR Serverless platform in the us-west-1 region, where the Common Crawl S3 bucket is located, running two extraction tasks in parallel with 1,000 executors for each task, using ARM64 executors with Java 17 execution backend and 6 GB of RAM per 4 executors. We were billed about $7,000 for the whole extraction process, including temporary S3 storage and traffic egress.

Table 1 shows the results of filtering of the extracted data. In total, there are 5.015 TB of gzipped json documents, without CSS selectors which were contained in the extracted text. The process was done on the ABCI system, using 10 rt_F nodes in a distributed manner for each job execution, submitting all jobs for the particular stage at once. Near-duplicate detection and filtering were done per segment for years after 2017 (inclusive) and per year for years before 2016 (inclusive). It took around 6 hours for near-duplicate detection, 1 hour for duplicate statistic data merging and 3 hours for filtering, costing us approximately 300 ABCI points.

The top 4 filters were deduplication (using the 5th percentile as a marker, downsampled to 5 instances), low hiragana ratio (less than 15% of hiragana), high link ratio (more than 40% of link text) and successive duplicate paragraphs with a high duplication count.

We produce three sets of data for LLM training. The first one contains downsampled documents with an expected duplication count of 1.5, evaluated at the 10th percentile. It contains almost no duplicate documents and is of the best quality. The remaining two can contain duplicate documents, but should still contain mostly novel text. They are downsampled to an expected count of 2.5 and 5, respectively, both of which use the 5th percentile as a marker.

7 Conclusion

Uzushio is a distributed tool for processing very large corpora built on the Apache Spark distributed computation framework. It adopts a paragraph-centric approach: both text extraction and deduplication treat paragraphs as the main unit. This allows Uzushio to delete not only duplicate documents but navigation-like content as well in an unsupervised manner. Uzushio can be run on general HPC systems like ABCI or in cloud environments.

Acknowledgements

The resource extraction part of this study was supported by JST PRESTO JPMJPR20C2 and JSPS KAKENHI JP22K17983.

The idea of using the document compression rate for filtering low-quality content was initially proposed by Yusuke Oda during one of the LLM-jp corpus workgroup meetings.

References

[1] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. **Commun. ACM**, Vol. 59, No. 11, p. 56–65, oct 2016.

[2] Common crawl: An open repository of web crawl data. https://commoncrawl.org/.

[3] The WARC format 1.0. https://iipc.github.io/warc-specifications/specifications/warc-format/warc-1.0/.

[4] Yann Collet and Murray Kucherawy. RFC8878: Zstandard compression and the 'application/zstd' media type. https://datatracker.ietf.org/doc/html/rfc8878.

[5] Nakatani Shuyo, Kessler Fabian, Roland Francois, and Robert Theis. language-detector: Language detection library for java. https://github.com/optimaize/language-detector.

[6] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In **Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing**, STOC '02, p. 380–388, New York, NY, USA, 2002. Association for Computing Machinery.

[7] Shinzato Kenta and Shun Kiyono. Hojichar: The text processing pipeline. https://github.com/HojiChar/HojiChar.

[8] Kenneth Heafield. KenLM: Faster and smaller language model queries. In **Proceedings of the Sixth Workshop on Statistical Machine Translation**, pp. 187–197, Edinburgh, Scotland, July 2011. Association for Computational Linguistics.

[9] Kenneth Heafield, Ivan Pouzyrevsky, Jonathan H. Clark, and Philipp Koehn. Scalable modified Kneser-Ney language model estimation. In **Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)**, pp. 690–696, Sofia, Bulgaria, August 2013. Association for Computational Linguistics.

A Extraction Statistics

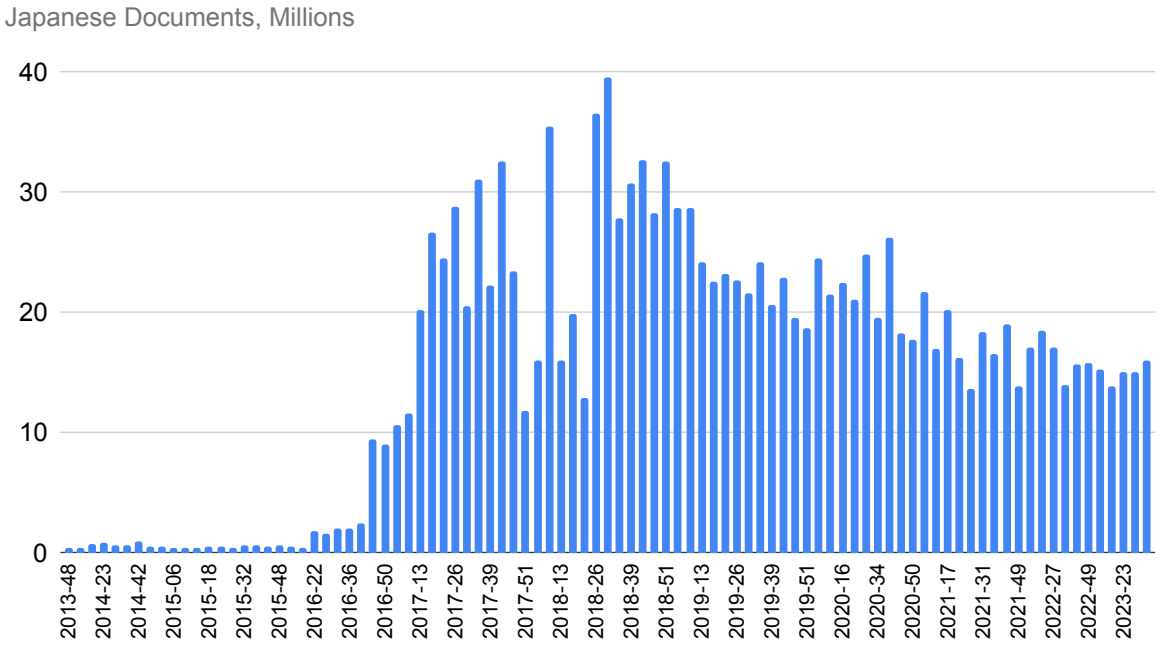

Figure 1: Number of extracted Japanese documents per Common Crawl segment

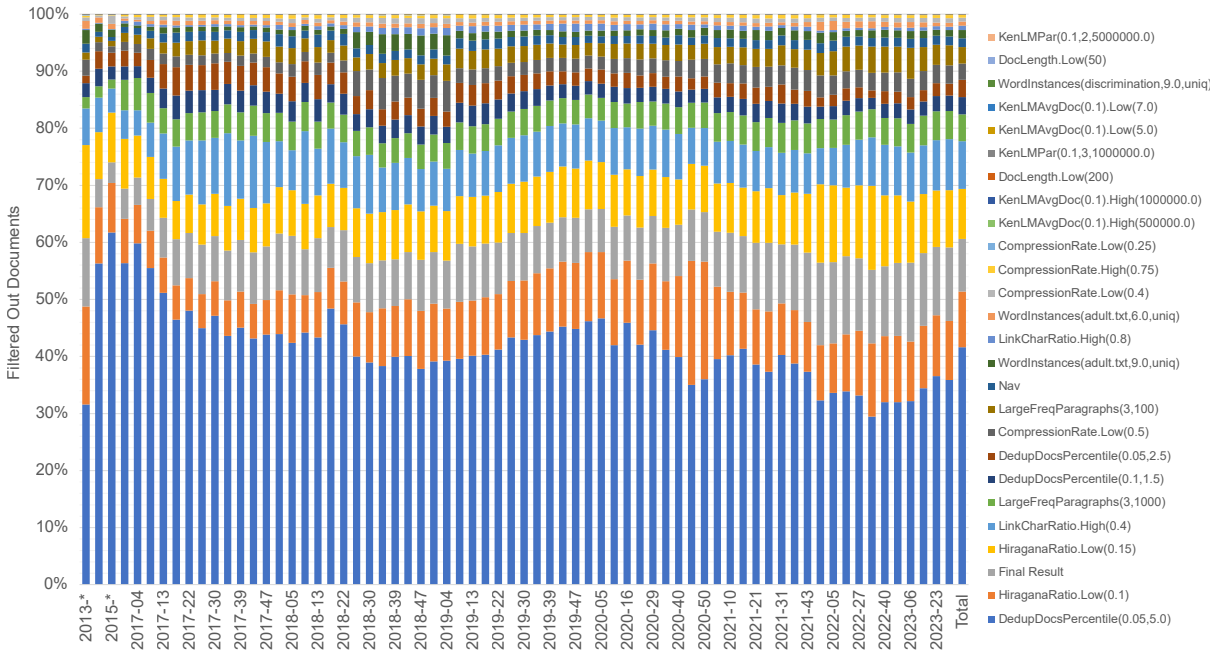

Figure 2: Percentage of documents or document content that were removed by individual filters. Processing for 2013-2016 was done by the whole year, not by individual dumps. Filter names use internal definitions. **Final Result** is documents which were not filtered out by any filter. **Filter.**{**High, Low**} means that the metric was higher or lower than the specified threshold. **DedupDocsPercentile** filter was applied 3 times, first one (0.05, 5.0) at the beginning of the pipeline, where it removed duplicate documents from further processing, and the remaining two times (0.05, 2.5), (0.1, 1.5) in the very end of the pipeline to create sub-datasets of different quality, as described in Section 6.