VILNIUS UNIVERSITY

FACULTY OF MATHEMATICS AND INFORMATICS

DATA SCIENCE STUDY PROGRAMME

Master's thesis

Hybrid retrieval methods and an attention mechanism in medical retrieval-augmented generation systems

Hibridiniai paieškos metodai ir dėmesio sutelkimo mechanizmas medicininėje informacijos paieškos sistemoje

Ugnius Byron Braun

Supervisor : assoc. prof. Linas Petkevičius

Scientific advisor : dr. Mindaugas Morkūnas

Vilnius
2026

Santrauka

Šiame darbe vertinami informacijos paieškos metodai medicininių dokumentų rinkiniams, taikant paieška papildyto teksto generavimo (*angl.* Retrieval-Augmented Generation, RAG) metodiką. Klinikiniai gairių dokumentai buvo iš anksto apdoroti, siekiant išgauti ir suvienodinti tekstinę informaciją iš pastraipų, lentelių ir paveikslėlių. Buvo įgyvendinti ir palyginti keli paieškos metodai. Eksperimentiniai rezultatai parodė, kad vien tik semantine paieška besiremiantys modeliai pasižymi ribotu tikslumu (33 %), tačiau skaidant tekstus į mažesnius teksto vienetus yra pagerinamas ankstyvųjų pozicijų rezultatas (42 %). Hibridinė, semantiniu ir leksiniu panašumu besiremianti paieška pasiekia geriausią tikslumą (88 %), o (*angl.* Native Sparse Attention) paremtas perrikiavimo metodas, apmokytas naudojant nedidelį klausimų-atsakymų rinkinį, reikšmingo patobulinimo nesuteikė (35 %).

Raktiniai žodžiai: Paieška papildytas teksto generavimas, dideli kalbos modeliai, vektorinė duomenų bazė, semantinis panašumas

Summary

This thesis evaluates information retrieval methods for medical document collections in a retrieval-augmented generation (RAG) setting. Clinical guideline documents were preprocessed to extract text, tables and images into a unified textual format, and multiple retrieval strategies were implemented and compared. Experimental results showed that dense semantic retrieval alone provides limited accuracy (33%), while subunit-based scoring improves early ranks (42%) and hybrid semantic-lexical retrieval achieves the strongest performance (88%). Native sparse attention reranking did not yield meaningful improvements, when trained with a small subset of question-answer pairs (35%).

Keywords: Retrieval-Augmented Generation, Large Language Models, Vector Database, Semantic Similarity.

List of Figures

List of Tables

Contents

Introduction

In the field of medicine, accurate clinical decision-making often relies on large volumes of information. Clinical guidelines, diagnostic protocols, and treatment recommendations are typically distributed across extensive collections of medical documents. In everyday clinical practice, manually locating the correct information within these document collections is both time-consuming and error-prone, increasing cognitive load and the risk of poor guidance. As a result, efficient information retrieval has become a critical practical challenge for medical professionals. Recent advances in artificial intelligence (AI) offer promising approaches to support medical workers by automating access to relevant clinical information.

In recent years, large language models (LLMs) have gained widespread attention and adoption. These models demonstrate strong performance across a variety of tasks and, in certain domains, approach or exceed human-level capabilities. As of April 2025, the conversational AI system ChatGPT reports approximately 800 million weekly active users[1], reflecting an incredibly rapid growth. The impact of this progress is evident across many fields, including medicine. Prior studies highlight the potential of AI to enhance clinical decision support through advanced computational capabilities [21], while other works identify AI as a pivotal component in modern medical decision-making processes [25].

Despite these advances, a fundamental limitation of LLM-based systems lies in their dependence on static training data. As medical knowledge evolves continuously, pretrained models may produce outdated or incomplete responses. To address this limitation, the concept of retrieval-augmented generation (RAG) was introduced [14]. In a RAG framework, a dedicated retrieval component supplies the language model with contextually relevant information, allowing responses to be grounded in up-to-date and domain-specific sources.

The **goal** of this work is to investigate how different retrieval-based methods can improve access to relevant information in medical document collections and to systematically assess the effectiveness of different retrieval strategies within a RAG system.

To achieve this goal, the following **objectives** are defined:

- Conduct a review of relevant literature to establish the theoretical background of AI-based information retrieval systems in the medical domain.
- Design and implement a baseline RAG pipeline for medical documents, enabling semantic retrieval of relevant content.
- Evaluate and compare multiple document processing and retrieval strategies by using a question-answer evaluation dataset.
- Investigate the applicability of DeepSeek's native sparse attention mechanism as a neural reranking approach within medical retrieval pipelines and compare its performance against established retrieval methods.

[1] https://www.demandsage.com/chatgpt-statistics

1 Literature review

1.1 Development of Large Language Models

To start at the very beginning, back in 1950 Alan Touring introduced the idea of whether or not machines can think [24]. 8 years after Touring's publication, the first statistical model of a neuron was constructed - a perceptron [18]. With an ability of learning linear decision boundaries, this model embodied one of the first implementations of a learning machine, thus proving that a machine could potentially be capable of learning, not only following pre-defined instructions. After years of slow progress and building more complicated structures people progressed to systems with multiple neurons, and eventually full networks of artificial neurons called neural networks. The next great breakthrough in this field had to do with training these networks and was called backpropagation [19]. This discovery not only enabled neural networks to move beyond linear separation, but popularized multilayer neural networks, highlighting their abilities and the potential of an emergent intelligence.

Standard feedforward networks still struggled with sequential or temporal information, keeping these AI models inferior to humans at most tasks. This limitation motivated the development of recurrent neural networks [7]. The Long Short-Term Memory model proposed by Hochreiter and Schmidhuber in 1997 enabled neural networks to retain information over long sequences. LSTMs became foundational for tasks such as language modeling and translation, forming the basis of early breakthroughs in the field of natural language processing.

Despite their success, recurrent models were inefficient and difficult to parallelize for more complicated tasks. This was motivation for trying to find better architectures free of recurrence. The first major step was the introduction of attention mechanisms [3], allowing models to concentrate on relevant parts of a sequence. This new methodology not only led to better overall results, but also reduced computational needs drastically. The next step was revolutionary and happened in 2017 - the introduction of the transformer [26]. Recurrence was replaced with self-attention mechanisms able to compute relationships between all tokens simultaneously. Context-dependent relationships could be learned, and this new architecture achieved unprecedented efficienty and scalability. This innovation allowed for parallelized computation, dramatically increasing training efficiency while preserving contextual understanding. The transformer architecture consists of two main components: an encoder and a decoder, each composed of multiple identical layers. Every encoder layer contains a multi-head self-attention mechanism and a feed-forward neural network, while each decoder layer incorporates an additional attention module that attends to the encoder's output. This way, the model learns complex, context-dependent representations that capture both local and global dependencies within a sequence.

As illustrated in Figure 1., the encoder–decoder design allows the transformer to process entire input sequences simultaneously. Positional encodings are added to preserve information about the order of tokens, enabling stable and efficient training for a variety of complicated tasks.

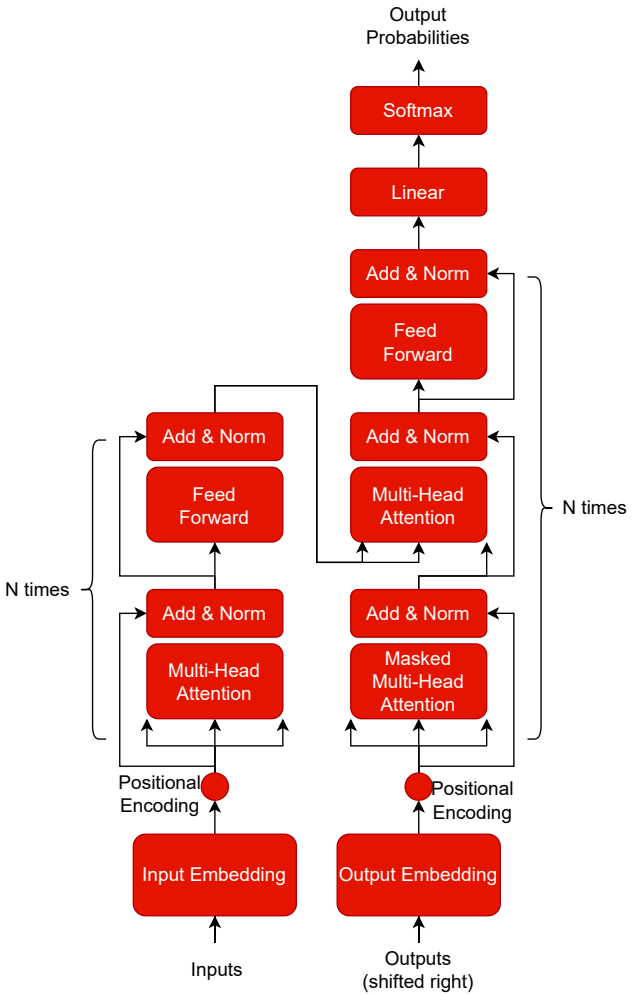

Figure 1. *Transformer architecture*
[26]

The impact of the transformer on the field of NLP has been profound. Within only three years, it was already acknowledged that transformer architectures had fundamentally changed how researchers approached natural language understanding [4]. Several state-of-the-art models like BERT [6] and GPT [17], being strongly based on the transformer architecture, were released shortly. BERT (Bidirectional Encoder Representations from Transformers) introduced the idea of pretraining transformer encoders using a masked-language objective, producing contextual embeddings that significantly improved downstream task performance. Meanwhile, GPT (Generative Pre-trained Transformer) models employed the decoder-only variant of the transformer to achieve high-quality text generation capabilities. The growth of these architectures led to large-scale models such as GPT-3, containing 175 billion parameters [5], marking what could be called the a new era in AI research. In fact, according to research [12], the performance of such models improves predictably when improving factors such as increased data, model size, and compute.

Given the immense computational cost of such large models, subsequent research focused on

efficiency and compression. DistilBERT [20] reduced the size of BERT by 40% and achieved a 60% speed improvement while maintaining 97% of its performance. Similarly, other authors [23] introduced even smaller variants—BERT-tiny, BERT-mini, BERT-small, and BERT-medium demonstrating that the transformer architecture can be effectively scaled down without substantial loss in accuracy. As a result, the transformer has evolved from a model architecture into a general computational framework underlying modern AI systems.

More recent developments have helped further progress, shifting emphasis from pure scaling toward efficient context processing. Models such as GPT-4 [1] demonstrated strong general reasoning capabilities, but also highlighted practical limitations related to context length. Newer architectures introduced extended context windows, as seen in models such as Gemini 1.5 [22], enabling the processing of entire documents within a single prompt. The very recently released gemini 3 [2] takes this even further, by supporting a context window of one million tokens, allowing it to ingest and analyze long form content such as text, videos, documents. It is capable of writing working code for large projects, generating realistic images and much more.

1.2 Understanding Retrieval Augmented Generation

Despite the constantly progressing capabilities of large language models, their performance could still be considerably constrained by the static nature of their training data. This can be crucial in fields such as medicine, where ongoing research continually generates new knowledge. Furthermore, the variability of guidelines across countries with differing levels of development highlights the critical role that training data plays. The obvious concern is that a static model can hardly keep up with the volume of data being generated. Another clear issue, underlined especially in high-stakes fields like medicine, is that models do sometimes generate misinformation. This limitation is particularly concerning, since medical factual accuracy and reliability is crucial. Even very large transformer-based models, such as GPT-4, exhibit a tendency to generate incorrect, but plausibly stated information (commonly known as hallucinations) [10]. This can happen more frequently when the model is asked about knowledge beyond its training data. Studies such as [15] have shown that GPT-4 can perform surprisingly well on medical reasoning tasks, highlighting the potential of harnessing such a tool. However, the possibility of errors is acknowledged, underscoring the need for methods that ensure usage of verified medical knowledge.

One of the most promising solutions to this problem is Retrieval-Augmented Generation (RAG).

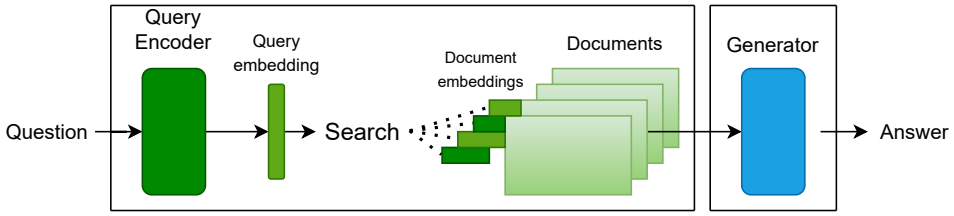

Figure 2. *RAG overview*
[14]

[2] https://blog.google/products/gemini/gemini-3/gemini-3

First proposed in 2020 [14], RAG combines information retrieval systems and language generation models. This way, an LLM can access external knowledge at the time of prompting - giving developers the ability to control the data models ground their answer with. A common approach is to have a retriever identifying semantically relevant documents from a database, which then passes this information as contextual input to a generator model. By ensuring that the text generation step relies on retrieved evidence, RAG not only reduces hallucination rates [2], but keeps the model up-to-date without requiring costly retraining.

To fully understand how RAG systems work, we must delve into how "semantic relevance" is computed. This process begins at the level of tokens - the basic units that language models process. A token typically represents a word, subword, or punctuation mark, depending on the tokenization scheme. BERT and similar transformer-based encoders rely on subword tokenization rather than full-word segmentation. This approach uses a predefined vocabulary (a collection of unique token units available to the model). If a word appears during inference but is not present in the vocabulary, it is decomposed into smaller, more frequent subword components. This strategy dramatically reduces the number of truly "unknown" tokens and enables the model to handle rare or domain-specific terminology with ease. BERT employs the WordPiece tokenization method [27], using a vocabulary of approximately 30,000 subword units [6]. This vocabulary size falls near the upper range recommended by the creators of WordPiece, balancing representational richness with computational efficiency. Subword-based vocabularies are particularly advantageous for biomedical applications, as complex medical terms can be segmented into meaningful components that the model has previously encountered.

The model's input design strategy allows for multiple sentences to be passed in. Furthermore, to support multi-segment input, BERT introduces several special tokens. The [CLS] token marks the beginning of every input sequence and provides a pooled representation for classification tasks. The [SEP] token denotes the boundary between sentences or the end of a single-sentence input. Additionally, BERT uses segment embeddings to indicate which sentence each token belongs to, and positional embeddings to encode the order of tokens within the sequence. The following illustration (Figure 3.) shows a visual representation of this embedding strategy.

In the process of preparing textual data and computing numerical representations, tokenization and embedding are two separate stages. The former is responsible for breaking the text up into subword units, while the latter converts these tokens into learned vector representations. As illustrated by the strategy used in BERT with token, segment and positional embeddings, these vectors can be very contextually dense.

A central part of RAG systems is the use of modern embedding models - typically responsible for both the tokenization phase and dense vector representations. While these vectors capture complicated contextual meaning, they also allow for important mathematical computations. A widely used metric is called cosine similarity (Equation 1) - allowing people to measure how closely two pieces of text relate in meaning by calculating dot products of vectors.

$$Cosine\ Similarity = \frac{\vec{a} \cdot \vec{b}}{||\vec{a}|| \cdot ||\vec{b}||} \tag{1}$$

Figure 3. *Example of embedding strategy applied in BERT-type models* [6]

Embeddings, combined with cosine similarity enable the construction of vector databases that support rapid similarity search through algorithms such as FAISS [11]. In the context of RAG, this methodology facilitates the ability to retrieve medical documents, which are conceptually related to the user's query, ensuring that responses are grounded in relevant, authoritative material.

While RAG can enhance factual accuracy, its performance is dependent on the relevance of data supplied by the retriever. General-purpose LLMs can often struggle with specialized vocabularies (especially with non-English text) and domain-specific syntax (for example language used in medicine). This problem has led to the development of domain-specific models, and we will focus on the medical field. The goal with domain-specific models is to train them on extensive data from one specific area, making these LLMs worse at general-purpose tasks, but allowing them to excel in their designated area. Some relevant models built for biomedical tasks are BioBERT [13], ClinicalBERT [8] - both fine-tuned versions of BERT, trained on specific medical data (PubMed abstracts, full-text biomedical articles, MIMIC-III dataset). More recently released in 2025 is Google's MedGemma collection - the company's most capable AI models for health AI development [3]. The combination of retrieval-based methods and domain-specialized LLMs has opened new possibilities for medical applications. RAG systems have been successfully applied to literature-based question answering [16]

[3] https://research.google/blog/medgemma-our-most-capable-open-models-for-health-ai-development/

with the help of a medicine-specialized embedding model, a vector database search framework, and a domain-specific fine-tuned LLM.

1.3 Reducing computational cost

As both general purpose and domain-specific LLMs get more sophisticated, the number of parameters keeps increasing - and so does the number of computations associated with training and running these models. This became a growing challenge in the field of natural language generation. A critical limitation of the standard transformer architecture is the computational complexity of its self-attention mechanism. In a sequence of length n, every token must compare itself to every other token, producing an $n \times n$ attention matrix. Formally, the attention scores are computed as:

$$Attention(Q, K, V) = softmax(\frac{QK^\top}{\sqrt{d_k}})V, \tag{2}$$

where Q, K and V represent the vectors of queries, keys and values.

Since this operation requires evaluating n^2 pairwise interactions, the computational and memory cost of self-attention scales quadratically with sequence length [26]. As important as computing all interactions is for these models to generate fluent text, this computational cost quickly becomes prohibitively expensive. Considering a simple query for a large language model, the input sequence is small: tens or perhaps hundreds of tokens, keeping the computation manageable. On the other hand, when a RAG system's retriever returns a relevant multi-page research article that gets appended to the context, the dense attention matrix with tens of thousands of tokens can become problematic.

These limitations have motivated the development of more efficient attention mechanisms capable of handling long sequences without incurring quadratic costs. Recent research has started exploring sparse attention mechanisms, able to reduce computational complexity by attending only to a subset of tokens rather than the full sequence. Instead of computing attention weights across all n^2 token pairs, sparse attention restricts the computation to strategically selected positions, lowering complexity to nearly linear levels while preserving contextual understanding. This class of techniques includes block-sparse patterns, sliding windows, and top-k token selection, all shown to reduce memory usage and speed up inference for long sequences.

One recent demonstration of sparse attention in large-scale models is DeepSeek's architecture, introduced in 2025 [9]. DeepSeek employs a mixture of dense and sparse attention layers, allowing the model to process long input sequences efficiently without substantial loss in accuracy. Its sparse modules prioritize tokens that carry semantic significance, enabling the model to focus on the most relevant features of the input while discarding redundant interactions. According to the authors, this design significantly improves scalability and retrieval performance for long-context tasks. By incorporating sparse attention mechanisms it may be possible to improve the efficiency of retrieval. Therefore, an experimental component of this thesis will explore how DeepSeek-style sparse attention influences retrieval efficiency when applied to biomedical document processing.

2 Experimental part

2.1 Initial data

The initial dataset used in the experimental part consists of a collection of medical documents obtained during work at Vilnius University Hospital Santaros Klinikos (VULSK). The set of validated documents consisted of 28 MS Word documents, representing relevant clinical material in different fields, mostly cancer-related, and used by workers and doctors of the hospital. The texts are unstructured, including diverse types of content: guidelines for specific illnesses, complications, theoretical information on coding schemes for medical terminology, classifications of severity for illnesses, dosages of specific medicine. Along with the high volume of text, many structured tables are present, as well as embedded images such as schemes or diagrams. These different types hinder the ability for text comprehension and will require additional preprocessing techniques. Tables often include dense clinical information (e.g., thresholds, medication dosages, risk categories), but require conversion into linear text to be suitable for embedding-based retrieval. Table 1. shows an example of tabular data found in one of the medical documents, regarding cancer stages based on categorization. In this document, information about stages is not presented in any other way, making it vital to extract this information.

Table 1. *Example of table found in medical document*

Stadija	T kategorija	N kategorija	M kategorija
0	Tis	N0	M0
IA	T1a	N0	M0
IB	T1b–T2	N0	M0
IIA	T3a	N0	M0
IIB	T3b	N0	M0
IIIB	T4	Bet kuris N	M0
	Bet kuris T	N2	M0
IV	Bet kuris T	Bet kuris N	M1

Images, on the other hand, may include vital information in the shape of arrowed graphs, visually presenting treatment recommendations to the human eye. One of the graphs found in these documents is presented in Figure 4., illustrating the need for machine understanding of arrows, cells, hierarchy and sequencing.

A completely different approach is required for parsing out textual information from certain images. Clearly, an essential step in the project will be the extraction and normalization of textual content, ensuring that all information, whether located in simple paragraphs, table cells, or figure captions, can be used together consistently. The extracted text will then serve multiple purposes throughout this study. First, it forms the basis for constructing the retrieval corpus used in the RAG system. Each document, consisting of guidelines for different illnesses may contain vital information and must be considered by the retriever. Second, the same source documents will be used to generate a question–answer (QA) dataset required for evaluating the system. This dataset will consist of clinically relevant queries paired with pointers to the text considered to contain the correct answer.

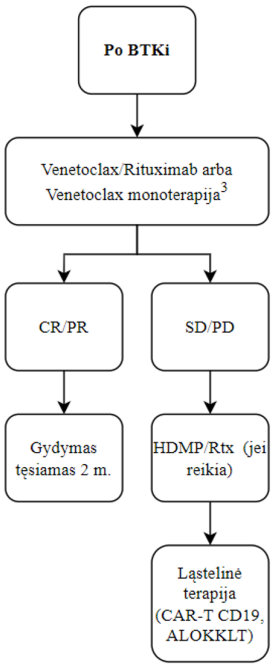

Figure 4. *Example of image found in medical documents*

The QA pairs will allow measurement of retrieval accuracy. The dataset of questions and answers will also be able to support the training of the sparse attention module based on DeepSeek. For this reason, a portion of the documents will be reserved for training, with the remaining portion kept for evaluation.

2.2 Computational Environment

The experimental part of this thesis, including code-based computations, was carried out on a remote high-performance computing (HPC) server (belonging to VULSK). The chosen integrated development environment was Microsoft's Visual Studio Code which, combined with the Remote-SSH extension, allowed Python-based experimentation directly on the server.

The server is equipped with two Intel Xeon Gold-6242 processors, each providing 16 physical cores with simultaneous multithreading enabled, for a total of 64 logical CPU cores. For workloads such as large-scale tokenization and embedding generation, CPU parallelism plays an important role. Tokenization is quite easily parallelizable across documents, while vector database construction benefits strongly from multi-core execution.

Model inference and embedding generation is accelerated using an NVIDIA Tesla-T4 GPU with 16 GB GDDR6 memory. The Tesla-T4 offers 2560 CUDA cores and dedicated tensor cores, enabling significant speedups for transformer-based models, particularly during matrix multiplications within attention layers and feed-forward networks. While the 16 GB memory is modest compared to modern data-center GPUs, it is sufficient for the medium-sized models used in this thesis. However, GPU memory limits directly constrain the batch size of embeddings and the maximum feasible context window for certain transformer architectures. Experiments were performed using Python-3.11.9 within a Linux environment.

2.3 Preprocessing

2.3.1 Creating textual representations of tables

The initial .docx files contain tables of varying complexity. Some are simple, standard $n\ x\ m$ tables, while others contain merged cells, note-like entries spanning the width of the table and other nonuniform structures. This section will include examples of the different types of tables present in the documents, and explanations of methods used in order to extract uniform, meaningful text out of all varieties.

To extract structured information from .docx files, the "python-docx" library was used. This library acts as a high-level interface for reading these kinds of documents, while internally each file is represented using Office Open XML (OOXML). A .docx file can be understood as a zipped collection of XML documents, and the main textual content is stored in "word/document.xml". The body of this file, represented by the "w:body" element, contains a sequence of XML nodes corresponding to paragraphs (`w:p`), tables (`w:tbl`), and other Word-specific elements, like embedded images. When "python-docx" loads a DOCX file, these XML structures are wrapped in Python objects, allowing the document to be traversed programmatically.

Identifying these tables is then quite simple: whenever a "child" of the "w:body" is of type "w:tbl", we know that no matter the formatting, based on the document's underlying structure that object is a table. Each table consists of rows ("w:tr") and cells ("w:tc") which allow for very convenient parsing of the internal contents.

The simplest and most convenient variant of a table will be addressed first - consisting of a single header row followed by data rows, without any merged cells. Since the structure is uniform and very clear, this kind of table is the easiest to flatten into a readable textual form. The contents can be extracted in a top-to-bottom, left-to-right manner for each row, while preserving "header: value" pairs:

$$Header_1 : Value_1 \mid Header_2 : Value_2 \mid Header_3 : Value_3 \mid ... \tag{3}$$

Each output line represents one logical entry, formatted like this:

Vaistas	Dozė	Vartojimas	Diena
Ciklofosfamidas	$750\,mg/m^2$	į.v.	1
Rituksimabas	$375\,mg/m^2$	į.v.	1

Textual representation:

`Vaistas: Ciklofosfamidas | Dozė: 750 mg/m`^{2} `| Vartojimas: į.v. | Diena: 1`

`Vaistas: Rituksimabas | Dozė: 375 mg/m`^{2} `| Vartojimas: į.v. | Diena: 1`

Table 2. *A table along with the generated textual representation*

This representation compresses the tabular information into a linear sequence while preserving the original associations between column names and their corresponding values. The resulting format is much more suitable for our tasks. However, clinical tables often contain irregular patterns

that require additional logic beyond simple row-wise extraction. Some of the simpler irregularities and the exact solutions are explained next.

- **Merged, title-like first row:** Some tables begin with a row where all the columns are merged together. Programatically, this usually results in all cells containing the same text. In practice, the purpose of this row is as that of a table title - it should not be addressed as a header, since that would result in all columns having the exact same header value. This issue is detected by checking whether the first row contains multiple identical non-empty values and, if so, treats it as a single logical field. That title text is meaningful and must be accessed and outputed in the representation. The "header: value" logic starts from the second row in cases like this.
- **Note-like rows with only one meaningful value:** Rows that contain many cells but only one non-empty entry usually represent comments or footnotes. Such rows are outputed as a single standalone line, not linked to any headers.
- **Merged rows or columns:** Quite often adjacent cells can be merged together (Table 1.). No matter if adjacent columns or rows are merged, the logic stays the same - in the textual representation, both variants spanning out from the merged cell must be repeated. In Table 1., since there are two separate combinations of T, N and M categories that represent the same stage, both must be repeated, while showing they are both linked to the same stage.
- **Empty or formatting-only rows.** Rows with no meaningful text are removed. These could appear due to spacing, copy-paste operations, or formatting issues.

A slightly more complicated format which is present in multiple documents, is "left-to-right" tables. An example of such a table is provided:

Table 3. *Part of a table from the initial documents, including complicated merging*

<table>
<tr><th>Stadija</th><th>Interven-cija</th><th>Rezultatas</th><th>Tolesnė taktika</th></tr>
<tr><td rowspan="4">IA stadija
T1a–1b, N0, M0, G1
IB stadija
T2a–b, N0, M0, G1</td><td rowspan="4">▷operacija</td><td>▷kraštas nuo naviko > 1.0 cm
arba nepažeista fascija</td><td>▷stebėjimas</td></tr>
<tr><td rowspan="3">▷kraštas nuo naviko ≤ 1.0 cm
(ir pažeista fascija)</td><td>▷Reoperacija</td></tr>
<tr><td>▷Jei IA stadija
▷stebėjimas</td></tr>
<tr><td>▷Jei IB stadija
▷spindulinė terapija</td></tr>
</table>

This is similar to a previously addressed issue of merging, however, much more rows are merged together here, and the table even has ▷markers indicating that it must be read from left to right.

For this exact table, we must “split” our textual representation at the column “Intervencija”, since the next column has two separate rows, and both must be shown to be linked to the value under “Intervencija”. Similarly, the bottom value under “Rezultatas” is linked to three separate values under “Tolesnė taktika”, and all these relationships must be preserved as well. Resulting, a table like this would have 4 separate textual lines - one for each cell under the row with the largest amount of cells.

The final kind of table that will be discussed here is a “top-to-bottom” table, which is similar to the previous example, except the relationships follow a downward path (also indicated by a ▽symbol).

Table 4. *Example of a table from the initial documents, including a downward logical path*

<table>
<tr><td colspan="4">Atsitiktinai rastas mazgas patikros metu + kylantis AFP</td></tr>
<tr><td colspan="2">▽
< 1 cm</td><td colspan="2">▽
> 1 cm</td></tr>
<tr><td colspan="2">▽
VOE + AFP kas 3-6 mėn</td><td colspan="2">▽
3-ių fazių KT arba MRT</td></tr>
<tr><td>▽
Stabilu</td><td>▽
Didėja</td><td>▽
Patvirtinama HCC diagnozė</td><td>▽
Jei nepatvirtinama HCC diagnozė, aptarti multidisciplinininiame aptarime dėl papildomo ištyrimo arba biopsijos arba tolimesnio stebėjimo kas 3-6 mėn.</td></tr>
<tr><td>▽
Tęsti stebėjimą kas 3-6 mėn. VOE ar KT + AFP</td><td>▽
Žiūrėti algoritmą > 1 cm</td><td>▽
MDT aptarimas</td><td></td></tr>
</table>

Due to the way tables are stored, a reliable method for parsing out the required textual representations was not found. From a human point of view, this table is understandable - it can be interpreted as a graph with arrows. The first cell that should be addressed is “<1 cm”, and starting with it, two separate pathways down must be displayed. A similar logic would explain the right side of the table as well. A procedure was used for extracting meaningful textual output, and it was implemented by communicating with the Gemini Vision API - based on a state-of-the-art AI model capable of interpreting both visual and textual data. Gemini Vision can analyze images and complicated structured text, generating human-readable descriptions and insights. The versatility and availability of this tool allows for processing of various inputs while providing consistent, high-quality summaries. As it is possible to retrieve the text underlying the complicated Table 4., this can be extracted, stripped of whitespace and sent to the Gemini Vision API as plain text, together with an instruction prompt specifying the desired output (and language). Lithuanian is chosen since all documents are in Lithuanian, and having only some tables explained in English could cause unwanted differences in the retrieval phase. The exact prompt used for extracting a summary out of Table 4. is

presented next, along with the actual produced summary.

Prompt: “Analyze the text from this table and describe its structure clearly. - Focus on the step-by-step sequence of connections while explaining how the connections work. - List all the labels exactly as written. - Do not include general introduction, start immediately with the explanation. - Provide the explanation in fluent LITHUANIAN.”

Summary: “ Ši diagrama aprašo algoritmą, skirtą valdyti kepenų mazgus, atsitiktinai aptiktus patikros metu, kai taip pat nustatomas padidėjęs alfa-fetoproteino (AFP) kiekis. Algoritmas prasideda nuo keturių identiškų įrašų: ”Atsitiktinai rastas mazgas patikros metu + kylantis AFP”. Tada šie įrašai išsiskaido pagal mazgo dydį į dvi kategorijas: ”<1 cm” ir ”>1 cm”.

Du mazgai, kurių dydis ”<1 cm”, nukreipiami į stebėjimą: ”VOE + AFP kas 3–6 mėn.”. (VOE greičiausiai reiškia ultragarso tyrimą). Po stebėjimo vertinama, ar mazgas yra stabilus: ”Stabilu”. Jei mazgas stabilus, nurodoma: ”Tęsti stebėjimą kas 3–6 mėn. VOE ar KT + AFP”. Jei mazgas didėja, t.y. ”Didėja”, nurodoma: ”Žiūrėti algoritmą >1cm”, taip perjungiant sprendimo seką į kitą šios diagramos šaką.

Du mazgai, kurių dydis ”>1 cm”, nukreipiami į diagnostinius tyrimus: ”3-ių fazių KT arba MRT”. Tuomet vertinama, ar tyrimai patvirtina kepenų ląstelių karcinomą (HCC): ”Patvirtinama HCC diagnozė”. Jei patvirtinama HCC diagnozė, nurodoma: ”MDT aptarimas”, tai yra multidisciplininio aptarimo poreikis. Jei HCC diagnozė nepatvirtinama, tai nurodoma: ”Jei nepatvirtinama HCC diagnozė, aptarti multidisciplinininiame aptarime dėl papildomo ištyrimo arba biopsijos arba tolimesnio stebėjimo kas 3-6mėn.”.”

2.3.2 Creating textual representations of images

A similar procedure is employed for image inputs. Each image is read in binary format, and its MIME type is determined to correctly inform the API about the file format. The extracted image is then sent to the same Gemini Vision API with an almost identical prompt (only emphasizing that this time, we are providing an image). The prompt, along with the retrieved summary of Figure 4. follows:

Prompt: “Analyze the text in this image and describe its structure clearly. - Focus on the step-by-step sequence of connections while explaining how the connections work. - List all the labels exactly as written. - Do not include general introduction, start immediately with the explanation. - Provide the explanation in fluent LITHUANIAN.”

Summary: “Diagrama atspindi žingsnis po žingsnio terapinį algoritmą.

Pradedama nuo bloko, pažymėto ”Po BTKi”. Iš šio bloko viena rodyklė veda į sekantį bloką ”Venetoclax/Rituximab arba Venetoclax monoterapija³”.

Iš šio antro bloko atsišakoja dvi rodyklės, vedančios į du skirtingus blokus: ”CR/PR” ir ”SD/PD”.

Iš bloko ”CR/PR” rodyklė veda į bloką ”Gydymas tęsiamas 2 m.”.

Iš bloko ”SD/PD” rodyklė veda į bloką ”HDMP/Rtx (jei reikia)”.

Galiausiai, iš bloko ”HDMP/Rtx (jei reikia)” rodyklė veda į bloką ”Ląstelinė terapija (CAR-T CD19, ALOKKLT)”.”

2.3.3 Generating Questions and Answers

A dedicated question–answer dataset was constructed for evaluation, parameter tuning as well as training. Each question was designed to reflect realistic clinical queries that could be posed when working with the original guideline documents. The process was carried out in collaboration with a practicing physician at the hospital, ensuring that the questions were both medically relevant and representative of the type of information clinicians would typically seek.

For every document, approximately 20 questions were formulated to test the retriever's ability to identify specific concepts, conditions, or decision points. For each question, one or more corresponding reference passages were manually selected from the document. These passages contained the key piece of information that must appear in the retrieved chunk in order for the result to be considered correct. The reference text varied depending on the context: in some cases it consisted of symptom lists or treatment criteria, in others a diagnostic or procedural code accompanied by its descriptive context. When the information originated from a table or image, the relevant caption and associated textual description were used instead. Some examples are presented in Table 5.:

Document_ID	Question	Result 1	Result 2	Result 3
ampulės_srities_vėžio...	koks yra ampulės srities vėžio mucininės adenokarcinomos morfologinis kodas pagal ICD-O-3?	8460/3	mucininė adenokarcinoma	–
įgytos_inhibitorinės_hemofilijos...	kokie yra įgytos inhibitorinės hemofilijos diagnostiniai kriterijai?	izoliuotai pailgėjęs ADTL	sumažėjęs FVIII:C	aptinkami antikūnai prieš FVIII
imunoterapijos_komplikacijų_gydymas...	kokie CIS simptomai pasireiškia galvos smegenų organų sistemai?	galvos skausmai, haliucinacijos, ...	2 lentelė. CIS simptomai	–

Table 5. *Examples from the question–answer dataset.*

Including surrounding contextual information in each reference passage was essential. Many documents contain repeated elements, such as medical codes or short recurring terms. These by themselves are insufficient for precise validation, as chunks from other documents could be wrongfully validated. Adding the immediate textual context ensured that a retrieved chunk could be reliably matched to the exact part of the document where the question was derived, also making sure that the document is correct as well. By structuring the dataset in this way, the retrieval evaluation became more meaningful and robust, accurately reflecting the model's ability to locate and return medically relevant content within the corpus.

2.4 Baseline Model

Here the implementation of a baseline model will be discussed, including all relevant parts of the process - extracting textual units from documents, embedding, storing units in a database, the ability to search the created database for relevant units based on a question as well as the evaluation process of this strategy.

2.4.1 Chunking strategy

As the available computational resources were slightly limited, using full medical documents as context for a the natively run LLM was unviable. In this baseline model, a carefully designed strategy for splitting documents into textual units - refered to as chunks - was implemented. The strategy is responsible for enabling efficient retrieval and processing of text, tables and figures from documents, while maintaining a computationally viable size for the textual units. After processing, a document is split into chunks, each of which serves as a segment that can later be embedded, indexed, and searched with the use of a vectorized database.

The procedure begins by loading a .docx file and scanning its internal XML structure. This way, different element types such as paragraphs, tables and embedded images are identified.

Paragraphs are collected in their original order and cleaned of formatting artifacts or hiden XML tags. Superscripts and other text anomalies are normalized using helper functions. Each paragraph is stored in a unified list of content blocks that maintain the sequential structure of the document. Tables are processed accordingly to previously explained methods, and the textual representation of each table is saved at this stage of the workflow for further use. The same is done for images - the description obtained from Gemini Vision API is saved for further use. For all tables and images that are sent to Gemini Vision API, unique hash-based filenames are created and saved locally, eliminating unnecessary API calls. Moreover, both tables and images are extracted from the initial document text. Since the textual representations are saved chronologically, this allows for a later reconstruction, based on caption-matching.

Next, the logic for “injecting” the table and image representations back into their surrounding paragraphs was implemented. Storing image or table descriptions as separate chunks in the database would cause them to lose essential contextual meaning provided by the adjacent text. For instance, the sentences immediately preceding an image often introduce or describe what is about to be shown. If these introductory sentences and the corresponding image description were placed in separate chunks, semantic search would be far less likely to retrieve the chunk containing the image description itself and it might instead match the preceding chunk, where the image is merely introduced. Additionally, keeping the image caption within the same chunk as the image description is advantageous for several later stages of the workflow, which are discussed further on.

After all textual and visual elements of the document have been collected, a pairing operation is performed, associating captions with their respective images or tables. This is achieved using regular expression patterns that detect Lithuanian caption forms such as “pav.”, “paveikslas”, “schema” or “lentelė”. As these kinds of captions can appear either before or after the visual element, both cases

are handled. Once captions are matched, the function constructs ordered lists of caption-image and caption-table pairs.

Finally, consecutive paragraphs (excluding the table/image descriptions, but including the captions), are merged together. This combined text is then split into smaller textual units, with a size of 200 words and an overlap of 50 words, ensuring that important context is not lost in the process. Within these chunks, according to the generated caption pairs, textual representations of both tables and images are inserted back into the text. For tables, a "***" marker pair wraps the representation, and for images, a "^^^" marker pair is used. These delimiters preserve the internal structure of the document and make it possible to later identify and handle these regions separately during embedding or search. If a chunk doesn't include any captions, and thus doesn't end up having any representation injected into it - the "type" variable for this chunk is set to "text_chunk". Other types used are "table" and "image".

The final output is a list of dictionaries, each representing a processed, ready-to-use chunk, including these fields:

- Type of the chunk ("text_chunk", "table" or "image").
- The original document name.
- The fully assembled textual content.

2.4.2 Building a Vectorized Database

To enable efficient semantic retrieval across the entire set of documents, a dedicated procedure was implemented to construct a vectorized database. The created python function identifies unprocessed documents, extracts their text representations using the previously described pipeline, and encodes them into numerical embeddings that can be stored and queried efficiently.

The procedure begins by scanning the designated document folder for all available files with the .docx extension. The full file paths are collected into a list and compared against the entries already present in the existing database table. If the table does not yet exist, all documents are treated as new. Otherwise, the function retrieves the list of already embedded document names and filters out any duplicates, ensuring that only unprocessed documents are sent through the embedding pipeline.

Each new document is then processed as explained in the previous section, resulting in a list of finalized chunks for every document. All chunks from all new documents are collected into a single list. Subsequently, the textual content of each chunk is encoded into vector embeddings using a predefined model. For the entirety of this work, the embedding model "embeddinggemma-300m-medical" was used. This model converts each text segment into a high-dimensional numerical representation that captures its semantic meaning. The embedding process is executed in batches to optimize performance and resource usage. The resulting embeddings are stored as NumPy arrays and converted to list format for database compatibility.

For each embedded chunk, a structured record is created containing the following information:

- The original document name.
- The type of the chunk.
- The full text of the chunk.
- The numerical embedding vector for the full text.

These records are then inserted into the vectorized database. Document-level metadata is saved within each record, enabling filtering and validation of retrieval results by source document. This ensures that search results remain both semantically accurate and contextually traceable.

2.4.3 Retrieval Phase

Once the vectorized database is built, an efficient search mechanism is required to identify the document chunks that best match a given query. In the baseline retrieval system, this process is fully semantic and relies on the same embedding model that was used for constructing the vectorized database. This ensures that both the query and the document chunks are represented within the same embedding space, allowing meaningful comparisons between them.

The retrieval process begins by preprocessing the input query using a cleaning and normalization function. This step includes operations such as lowercasing and removing unwanted symbols or stop words. Optionally, lemmatization can be applied to reduce inflected forms of words to their base versions, ensuring consistent representation. After preprocessing, the query is encoded into a dense vector embedding using the same model that was applied during database creation.

The embedding vector of the query is then used to perform a semantic search over the vectorized database. The search process leverages the “LanceDB” library in Python, which internally utilizes the DiskANN algorithm — a high-performance graph-based approximate nearest neighbor (ANN) method. This allows for fast and scalable similarity searches even in large embedding spaces.

The semantic search retrieves the top-m chunks that are most similar to the query according to cosine similarity. For testing purposes, a value of m=100 was used, providing a representative subset of semantically relevant passages from across the indexed documents. These retrieved candidates are then sorted in descending order based on their similarity scores, ensuring that the most semantically related chunks appear first in the ranked list.

Finally, the retrieval function outputs the top-k ranked chunks as the system’s search result. In practice, k=5 was used, as providing five chunks to the subsequent summarization model offers a good balance between completeness and efficiency. This number is large enough to capture relevant information while avoiding unnecessary context that could reduce summarization accuracy.

The resulting retrieval pipeline, consisting of the construction of the vectorized database, semantic search based on a user query, and the extraction of the top-k most relevant chunks, forms the core of the baseline retrieval system.

2.4.4 Illustration of retrieval for specific question

To illustrate the behavior of the retrieval models evaluated in this work, a concrete example of the retrieval output is presented. A randomly selected question from the constructed question–answer dataset is used as input, and the top m=5 retrieved text chunks are shown. For clarity of presentation, the retrieved texts are truncated.:

Question: Kas priklauso limfinės sistemos 2 sričiai?

Retrieved chunks:

- limfocitozė ≥ 5 x 109/l periferiniame kraujyje ir/ar > 30% kaulų čiulpuose. I stadija – kloninė B limfocitozė ≥ 5 x 109/l periferiniame kraujyje ir limfadenopatija. II stadija – kloninė B limfocitozė ≥ 5 x 109/l periferiniame kraujyje ir splenomegalija ir/ar hepatomegalija. III stadija – kloninė B l...
- Vizito metu gydytojas įvertina kraujodarą (atliekamas bendras kraujo tyrimas), ar nėra aktyvių infekcijų (matuojama kūno temperatūra, objektyvus ištyrimas), įvertina kepenų, inkstų veiklą (kreatininas, bilirubinas, ALT, AST, gamaGTP). Kiti tyrimai atliekami pagal klinikines indikacijas gydančio gydy...
- x 109/l, nepriklausomai nuo palpacinio sindromo buvimo. Limfinės 1, 2 sistemos sritys 1 sritis. Galvos, kaklo, Valderjerio žiedo limfmazgiai. 2 sritis. Pažastų limfmazgiai. 3 sritis. Kirkšnių limfmazgiai. 4 sritis. Čiuopiama blužnis. 5 sritis. Čiuopiamos kepenys. Pastabos: 1Limfmazgių padidėjimas tu...
- išemija ir įvykti perioperacinis miokardo infarktas, širdies ritmo sutrikimai; praeinantis ar negrįžtamas centrinės nervų sistemos pažeidimas (insultas), nulemiantis sunkų neįgalumą ar net mirtį; besitęsiantis kraujavimas sukelia krešėjimo sutrikimus, taip sukurdamas ydingą nesustojančio kraujavimo ...
- 121:2018 „Gydytojas onkologas chemoterapeutas“ patvirtinimo“. Lietuvos Respublikos sveikatos apsaugos ministro 2011 m. birželio 8 d. įsakymas V-591 „Dėl Lietuvos medicinos normos MN 28:2019 „Bendrosios praktikos slaugytojas“ patvirtinimo“. VšĮ Vilniaus universiteto ligoninės Santaros klinikų Vidaus ...

In this example, only one of the retrieved chunks contains the information required to answer the question, explicitly identifying the second region of the lymphatic system. The remaining chunks are deemed semantically related but clearly are not directly relevant to the query. This illustrates how retrieval-based systems surface a small set of candidate passages, from which the most relevant information can be selected and later provided to a language model for grounded response generation.

2.4.5 Evaluation Phase

To evaluate the effectiveness of the baseline retrieval system, a structured evaluation workflow was designed. This process makes use of the questions–answers dataset, where each question is linked to a specific document and one or more gold-standard text spans that represent the correct answer. These spans serve as ground truth references, allowing for a direct comparison between the retrieved chunks and the expected content.

Before evaluation, all gold-standard answer spans are preprocessed to ensure consistency. This normalization step includes removing differences in casing and whitespace to avoid mismatches caused by formatting rather than content. Each question may contain multiple answer spans, all of which must be located within a chunk for it to be considered a correct retrieval.

The dataset was divided into training, validation, and test subsets to support reproducibility and parameter tuning. Using a fixed random seed, the dataset was randomly shuffled and split into three parts: 100 samples for validation, 100 for testing, and the remaining 337 entries for training. These subsets were saved to guarantee consistent experimental conditions throughout all evaluation phases.

The evaluation procedure executes the full retrieval pipeline for each question in the test set and logs the results in a structured format. For every question, the search pipeline is initialized with the same embedding model and scoring configuration used in the main system, with several differences. The process begins by embedding the cleaned version of the question and performing a full semantic search across all chunks in the database. Here it is important to make sure all possible existing chunks are considered. A list of, in our case, 537 dictionaries is returned, where each dictionary consists of:

- The question used as the query.
- The name of the document from which the retrieved chunk originates.
- The full text of the chunk.
- The cosine similarity score between the query and the chunk.

This list is then sorted in descending order according to cosine similarity, ensuring that the most semantically related chunks appear at the top. For each question, the retrieval results are saved in a JSON format list, allowing detailed analysis and reproducibility.

After logging all detailed data regarding the retrieval procedure, a structured comparison was performed. For each question, the system identifies all retrieved chunks that (1) originate from the correct document, and (2) contain all required text spans within them. This dual condition guarantees that a chunk is only considered a valid match when it retrieves the correct content from the correct source document. With this clear condition in mind, it is possible to iterate through all questions, while recording the positions of the validated chunks throughout the lists.

Since the purpose of the retrieval phase is to prepare relevant data for subsequent summarization by a large language model, the ranking of retrieved chunks plays a crucial role in evaluating

retrieval performance. Ideally, the system should return the correct chunk within the top five results, as these are the ones provided to the summarization model. Including more than five chunks would risk overwhelming the model with irrelevant information and degrading its performance. Therefore, the key evaluation metric of this work is whether the correct chunk appears among the top five retrieved positions for each question. If at least one correct chunk appears within the top five retrieved results, the query is marked as a successful retrieval for top-5 accuracy. All matches and ranks are logged into a consolidated CSV file, which provides a compact overview of system performance across all questions.

2.4.6 Results

The retrieval performance of the baseline model was evaluated on the test set using several standard ranking metrics. These include Top-1 accuracy, Top-5 accuracy, Average rank, and Mean Reciprocal Rank (MRR). Each of these provides a different perspective on the retrieval quality and allows for a detailed understanding of how effectively the system identifies the correct document chunks.

Top-k accuracy measures the proportion of questions for which the correct chunk appears within the first k retrieved results. In the baseline model, the Top-1 accuracy reached 16%, indicating that for roughly one out of six questions, the most relevant chunk was ranked first. The Top-5 accuracy increased to 33%, showing that in one third of the test queries, the correct answer was located among the top five retrieved chunks. As stated earlier, the Top-5 accuracy directly reflects the proportion of queries for which the system successfully provides relevant information to the summarization model.

The correct passage appeared around the 64th position in the ranked list of candidates on average. Although this value might seem high, it reflects the fact that many documents contain a large number of semantically similar chunks, making fine-grained ranking challenging in a purely embedding-based setup.

The Mean Reciprocal Rank (MRR) offers a more interpretable, scale-independent metric for ranking performance. It is defined as:

$$\text{MRR} = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{r_i}, \tag{4}$$

where r_i denotes the rank position of the first correct result for query i, and N is the total number of evaluated queries. In this formulation, higher-ranked (lower r_i) correct matches contribute more strongly to the final score. The baseline model achieved an MRR of 0.248, indicating that correct answers tend to appear closer to the top of the ranked list rather than being uniformly distributed. MRR is particularly useful because it balances sensitivity to both early and late rankings and penalizes cases where the correct chunk is found only after many irrelevant results.

The distribution of match positions across all test questions is visualized in Figure 5. The histogram clearly shows a strong concentration of correct matches near the beginning of the ranked list, with the majority appearing within the first 100 positions. This confirms that while the base-

line model often retrieves relevant information near the top, there is still considerable variability in performance across queries. The long tail of higher ranks indicates cases where semantically similar but irrelevant chunks were prioritized, highlighting potential areas for improvement in retrieval precision.

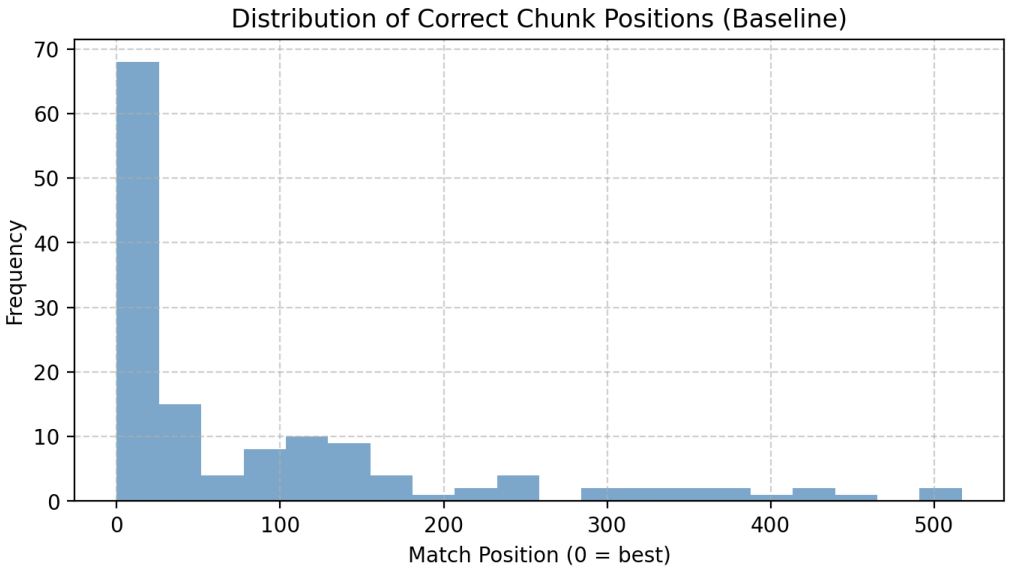

Figure 5. *Distribution of correct chunk positions for the baseline retrieval model. A lower rank (closer to zero) indicates that the correct passage was retrieved earlier.*

Overall, these results establish a solid foundation for subsequent experiments and model upgrades. While the baseline retrieval demonstrates reasonable semantic matching capability, the relatively low Top-1 and Top-5 accuracy values suggest that further refinements, such as introducing re-ranking mechanisms, could lead to performance improvements.

2.5 Subunits Improvement

The next stage focused on improving the semantic retrieval quality by refining how document chunks are represented internally. In the baseline model, each chunk was embedded and stored as a single dense vector, representing the entire passage as a single semantic unit. While this approach captures the overall meaning of a text segment, it may lose granularity in cases where a chunk contains multiple distinct concepts or where only a small portion of it is relevant to a given query. This problem becomes especially apparent in documents that contain a mix of paragraphs, tables, and figure descriptions, where one local section may be highly relevant while others are unrelated.

To address this limitation, a new version of the retrieval system was implemented. The central idea is to divide each parent chunk into smaller subunits and represent each of them as an individual embedding vector. This enables the retrieval mechanism to consider fine-grained similarities between the user query and specific portions of the document rather than relying solely on the global chunk-level representation. The overall document segmentation strategy remains the same as in the baseline model. However, additional information is stored for each chunk to facilitate subunit-level reasoning during retrieval.

2.5.1 Building a Vectorized Database

The procedure for constructing the vectorized database follows the same general workflow as in the baseline model but with several key modifications. Each document is first processed into chunks (here refered to as parent chunks) using the same chunking parameters as before, including predefined size and overlap values to ensure contextual continuity across boundaries.

Once each parent chunk is defined, its internal text content is split into smaller textual subunits. During this step, the text is also analyzed to detect non-paragraph elements such as tables and figures. For each of these, a textual description is extracted and stored as an independent subunit associated with its parent chunk. This ensures that non-textual content, which often carries important information, can also meaningfully contribute to this specific semantic matching strategy. After extracting textual representations for tables and images, the remaining text is split into subunits as well. These subunits are created by slicing the text according to a fixed maximum length and a small overlap percentage, similar to the logic used for chunking but applied at a finer scale. The goal is to produce self-contained, semantically interpretable units of text, each corresponding to 10 words.

For every parent chunk, a collection of subunits is thus generated. All subunit texts, along with the full parent text, are then passed in a single batch to the embedding model. This produces a parent-level embedding vector followed by a list of subunit embedding vectors. The resulting data are stored together in the database, with each record containing:

- The document name and chunk type.
- The full parent chunk text and its corresponding vector representation.
- A list of subunit texts derived from the parent chunk.
- A list of embedding vectors for each of those subunits.

By storing these additional fields directly in the database, it becomes possible to later perform retrieval operations that account for internal chunk structure without needing to reprocess or re-embed the documents.

2.5.2 Retrieval Phase

The retrieval phase of the subunit model follows the same general structure as in the baseline system but introduces an additional stage of subunit-level reasoning. Initially, a query is preprocessed and encoded into a dense embedding vector using the same model employed in all previous experiments. This query embedding is then used to perform an initial similarity search across all parent chunk embeddings stored in the database. The result is an ordered list of candidate chunks ranked by their cosine similarity to the query, identical to the baseline retrieval procedure.

However, in this version, since each retrieved chunk is internally represented not only by its parent embedding but also by a set of subunit embeddings, the system computes the cosine similarity between the query embedding and every subunit embedding associated with that chunk. In the simplest approach, one could take the arithmetic mean of these subunit similarity values to represent

the overall semantic similarity of the chunk. However, this would imply that all subunits are equally relevant to the query, which is rarely the case. Typically, only one or a few subunits of a chunk directly correspond to the user's question, while the rest may contain unrelated content. Averaging across all subunits therefore tends to dilute strong local matches and penalize chunks that contain a small but highly relevant section.

To address this, the final similarity score for each chunk is instead computed using a form of *soft-max pooling*. This approach assigns exponentially higher weights to subunits that are more similar to the query, thereby amplifying the influence of strong local matches while maintaining numerical stability. For a given parent chunk C with n subunits and corresponding similarity scores $s_1, s_2, \ldots, s_n$, the softmax-pooled similarity is defined as:

$$S_{\text{soft}}(C) = \frac{1}{\tau} \log \left(\frac{1}{n} \sum_{i=1}^{n} e^{\tau s_i} \right) \tag{5}$$

where τ is a temperature parameter controlling the sharpness of the weighting. As τ increases, the pooling behaves more like a hard maximum; as τ decreases, it approaches a simple mean. In this implementation, a value of $\tau = 10$ was used, which empirically provides a good balance between sensitivity to the best subunit and robustness to noise.

While the subunit-level pooling effectively captures localized semantic matches, it can occasionally overemphasize small fragments that are individually similar but do not represent the overall context of the parent chunk. To mitigate this effect, the final chunk score combines the softmax-pooled similarity with the parent-level cosine similarity, resulting in a blended score:

$$S(C) = (1 - \lambda)\, S_{\text{soft}}(C) + \lambda\, s_{\text{parent}} \tag{6}$$

where s_{parent} is the cosine similarity between the query and the parent chunk embedding, and $\lambda \in [0,1]$ controls the relative contribution of global versus local similarity. In the present work, the value $\lambda = 0.2$ was chosen, giving 80% weight to the fine-grained subunit similarities and 20% weight to the overall parent embedding.

This blended similarity formulation captures both the local relevance of specific subunits and the global coherence of the entire chunk. It is conceptually motivated by the observation that users typically formulate questions referring to a small part of a document, but correct retrieval still requires some consistency with the surrounding context. By ranking all candidate chunks according to $S(C)$, the system produces a final retrieval list that hopefully better reflects nuanced semantic relationships within the documents.

2.5.3 Evaluation Phase

To accurately assess the performance of the improved retrieval system, an extended evaluation workflow was implemented. The same questions–answers dataset used in the baseline evaluation was employed here to ensure direct comparability between the two approaches. The preprocessing of gold-standard data remains identical to that described for the baseline evaluation. The same train-

ing, validation, and test splits are used, ensuring that performance improvements can be attributed exclusively to the modified retrieval method rather than differences in data sampling. Each question from the test set is associated with a target document and a set of gold-standard answer spans. The system initializes with the same embedding model, database, and preprocessing steps as in the baseline configuration, but now applies the additional subunit scoring and reranking procedure. For each question, the process begins by embedding the cleaned query and performing a semantic search across all database chunks. This search produces the exact same list of 537 dictionaries as in the baseline model, where each dictionary contains:

- The question used as the query.
- The name of the document from which the retrieved text chunk originates.
- The full text of the chunk.
- The cosine similarity score between the query and the chunk.

This list is sorted in descending order based on cosine similarity, producing what will be referred to as the "raw" results list. Next, each of these retrieved entries is processed using the subunit scoring approach explained earlier. This list is then sorted in descending order by $S(C)$ (Equation 6), producing a new list, referred to as the "processed" results list.

For each question, both lists — the raw and the processed results — are saved as individual JSON files. This means that for every query, two files are produced: one containing the original semantic search results and one containing the reranked results afterthe subunit scoring. In total, 200 pairs of JSON files are generated, corresponding to all queries from the test set. These are then used to calculate the same metrics, as in the evaluation of the baseline model.

2.5.4 Results

The results were compared directly against the baseline semantic retrieval system to determine whether incorporating subunit-level similarity calculations improved the ranking of relevant document chunks.

Model	Top-1 Accuracy	Top-5 Accuracy	Average Rank	MRR
Baseline (raw)	0.160	0.330	64.40	0.248
Subunits (processed)	0.220	0.420	72.98	0.322

Table 6. *Retrieval performance comparison between the baseline and subunit-based models on the 100-question test set.*

Table 6. summarizes the main retrieval metrics obtained for both models. The subunit model achieved a clear improvement in top-rank accuracy: top-1 accuracy increased from 0.16 to 0.22, while top-5 accuracy rose from 0.33 to 0.42. The mean reciprocal rank (MRR) likewise improved from

0.248 to 0.322, confirming that on average the first correct chunk appeared earlier in the ranking. Furthermore, not included in the table, but a noteworthy satistic: the proportion of questions where the retrieved rank improved was 45%, indicating that almost half of the evaluated queries benefited from subunit-level reasoning.

Interestingly, the mean rank of the first correct chunk slightly worsened, increasing from 64.3 in the baseline to 73.0 in the subunit model. This seemingly contradictory result can be explained by the fact that, although the subunit-based approach successfully promoted relevant chunks to the top of the list more frequently, the cases where it failed tended to result in much lower ranks. In other words, the subunit model made stronger distinctions, excelling when a highly relevant subunit was detected, but occasionally producing large errors when none of the subunits closely matched the query.

Figure 6. presents a visual comparison of top-1 and top-5 retrieval accuracies between the baseline and the subunit model. The improvement in both metrics demonstrates that the softmax-pooled subunit aggregation leads to more accurate top-ranked retrievals. The difference is particularly visible for the top-5 metric, which better reflects the model's ability to bring relevant information within the small set of chunks later used for summarization.

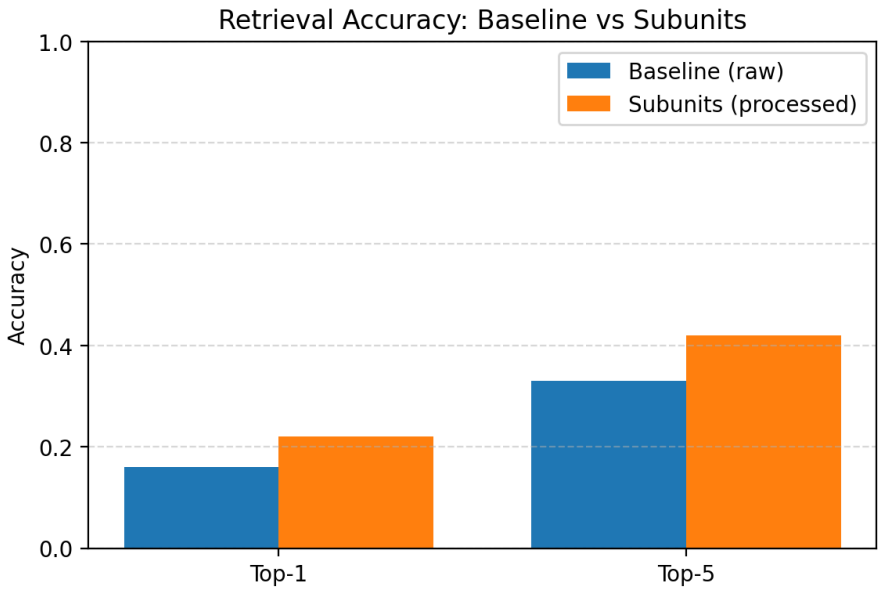

Figure 6. *Comparison of Top-1 and Top-5 retrieval accuracies between the baseline (raw) and subunit-based (processed) models.*

To further analyze ranking behavior, Figure 7. shows the distribution of the ranks of the first correct chunk for both systems. Both histograms exhibit a strong concentration of matches at low ranks, confirming that most queries retrieve relevant chunks within the top section of the ranked list. However, the subunit model (shown in orange) demonstrates a slightly heavier tail toward higher ranks. This reflects a higher variance in retrieval performance: when the subunit mechanism identifies a clearly relevant passage, it ranks it very highly, but when no subunit stands out, the final score can be substantially lower. Such cases contribute to the slight increase in mean rank despite the overall improvement in top-k performance.

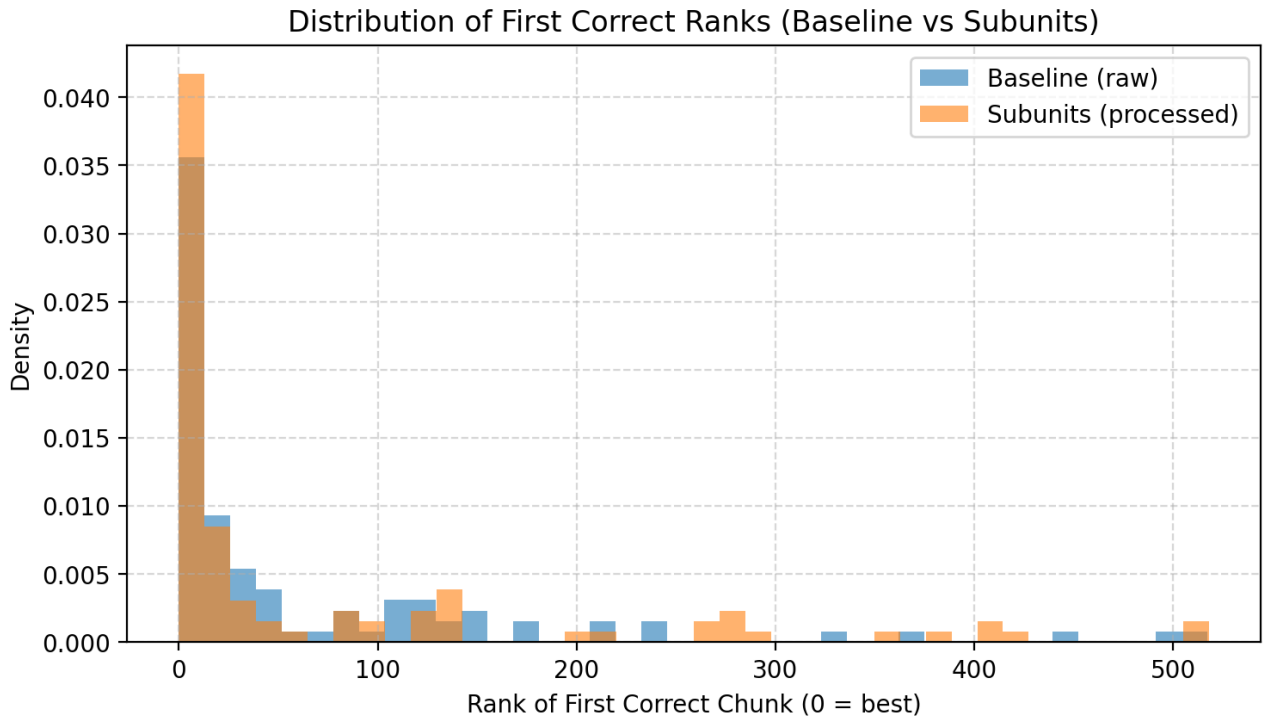

Figure 7. *Distribution of first correct chunk ranks for the baseline and subunit models.*

Overall, the results demonstrate that incorporating subunit-level semantic aggregation improves retrieval precision at the top of the ranking, which is crucial for downstream summarization. Although it introduces slightly higher variability in ranking behavior, the method more effectively identifies fine-grained relevance within longer document chunks, thereby increasing the likelihood that the correct information appears among the highest-ranked results.

2.6 Keyword matching improvement

Next, an additional enhancement was developed to improve retrieval precision. This extension introduces a hybrid approach that integrates explicit keyword-based matching with the existing semantic similarity mechanism. Keyword matching can be particularly helpful in domains characterized by recurring terminology, such as medicine.

The underlying database and embedding model remain identical to those used in the baseline system, and this improvement will be compared to the results from the baseline system. The same cleaning and normalization pipeline is applied to all queries, ensuring compatibility and consistency. This improvement is also implemented as a post-processing step, applied after the initial semantic search.

2.6.1 Retrieval Phase

Once the top-m semantically similar chunks are retrieved, as explained earlier, a refined scoring procedure is introduced. Each chunk is re-evaluated using both semantic similarity and lexical overlap with the query. First, the number of query keywords appearing in the chunk's text is counted using exact word-boundary matching to prevent partial matches. This count serves as a measure of explicit

lexical relevance. At the same time, the cosine similarity between the query embedding and the chunk embedding is recomputed to provide a continuous measure of semantic closeness. These two measures are then combined into a single weighted score, defined by the linear formula:

$$final\_score = \alpha \times keyword\_count + \beta \times similarity\_score \tag{7}$$

The coefficients $\alpha > 0$ and $\beta > 0$ determine the relative importance of lexical and semantic components. Increasing α makes the system more sensitive to direct keyword overlap, while a higher β emphasizes conceptual similarity. In this experiment, equal values were chosen, balancing both components equally.

After calculating the final scores, all candidate chunks are reranked in descending order based on the final score, and the top-k results are returned as the improved retrieval output. This hybrid scoring mechanism ensures that retrieved chunks are not only semantically related to the query but also contain direct lexical matches.

2.6.2 Evaluation Phase

For evaluation, firstly, the raw results are generated just like in the baseline model. Next, each of these retrieved entries is processed using the hybrid scoring approach. For every candidate, keyword counts are calculated and combined with the semantic similarity score according to the linear formula introduced earlier (Equation 7). This yields a final score for each chunk, representing the integrated relevance of semantic and lexical similarity. The list is then re-sorted in descending order according to these final scores, resulting in the processed results list, where this time, each dictionary contains:

- The question used as a query.
- The name of the document, from which the returned text chunk is.
- The full text of the chunk.
- The final score, calculated based on both semantic similarity and keyword matching.

JSON files are then generated in the same fashion, allowing for clear computation of the same metrics relevant for our comparison.

2.6.3 Results

The introduction of the keyword-matching mechanism resulted in a substantial improvement over the baseline model. The additional lexical signal allowed the system to prioritize chunks containing key medical terminology present in the question, reducing the number of irrelevant yet semantically similar results.

Table 7. summarizes the retrieval performance for both models. The keyword-enhanced model achieved a tTop-1 accuracy of 57%, more than three times higher than the baseline (16%). Similarly, the top-5 accuracy increased from 33% to 88%, demonstrating that in the vast majority of cases,

the correct answer chunk was ranked within the first few retrieved results. Ranking-based metrics also confirmed the improvement: the mean reciprocal rank (MRR) rose from 0.248 to 0.709, and the average rank of the first correct chunk dropped dramatically from 64.4 to just 3.4. Overall, 74% of questions saw an improved ranking when keyword information was incorporated.

Model	Top-1 Accuracy	Top-5 Accuracy	Avg. Rank	MRR
Baseline (raw)	0.160	0.330	64.40	0.248
Keyword Matching (processed)	0.570	0.880	3.39	0.709

Table 7. *Comparison of baseline and keyword-matching retrieval performance.*

Figure 8. illustrates the improvement in retrieval accuracy between the baseline and the keyword-matching model. The bar chart clearly shows the substantial jump in both Top-1 and Top-5 accuracy. This confirms that keyword matching provides a highly effective complementary signal to embedding-based similarity, especially in the biomedical domain where terminology is highly structured and domain-specific. By combining semantic and lexical cues, the model achieves a more precise understanding of query intent, substantially improving the relevance of the retrieved chunks.

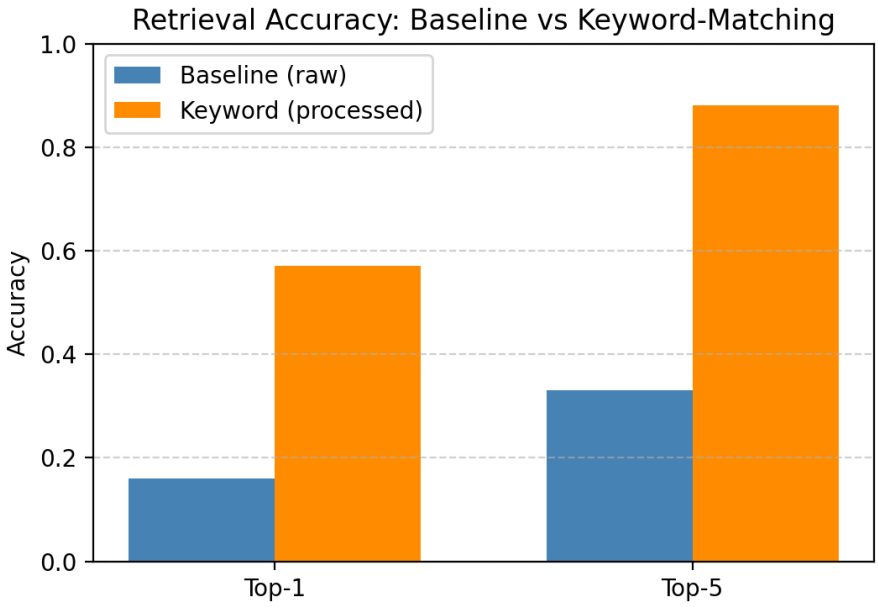

Figure 8. *Retrieval accuracy comparison between the baseline and the keyword-matching models.*

2.7 NSA Reranker

To try and further enhance the retrieval quality beyond the previously implemented baseline, keyword-based, and subunit-augmented retrievers, a neural reranking module based on the Native Sparse Attention (NSA) [9] mechanism was introduced. This component represents the novel contribution of the work. The reranker aims to refine the ranking of semantically retrieved chunks by introducing an additional neural attention layer that learns to distinguish between relevant and non-relevant passages based on contextual cues.

The core idea of this stage remains consistent with previous retrieval versions: all documents are chunked using the same segmentation pipeline, and their embeddings are stored in a vectorized database in the same way. The NSA reranker does not modify the underlying document structure or database format; rather, it operates as an additional neural layer on top of the existing retrieval pipeline. Its primary purpose is to re-evaluate the initially retrieved candidate chunks for a query, using a learned attention mechanism. With the limited amount of training data, the hope is that it will still be able to capture higher-order dependencies and contextual relevance beyond lexical overlap or static embedding similarity.

2.7.1 Methodology and Relevance

The Native Sparse Attention (NSA) architecture, introduced by DeepSeek in 2025, was designed as an alternative to conventional dense attention mechanisms, addressing the computational and representational inefficiencies that arise when modeling long sequences. In traditional transformer architectures, attention is computed exhaustively across all token pairs, resulting in quadratic time and memory complexity with respect to sequence length. While effective for short texts, such dense interactions become impractical when handling long medical documents or large sets of candidate chunks, where much of the contextual information is either redundant or weakly informative.

NSA reformulates the attention operation as a sparse and hierarchical selection process. Instead of attending to every token in the sequence, the model dynamically identifies and processes only the most informative regions of the context. This is achieved through a hybrid mechanism that combines three key components:

- **Local attention windows:** Each token attends densely to its immediate neighborhood, ensuring that short-range dependencies and local coherence are preserved. This maintains the ability to capture relations within compact text segments, such as the co-occurrence of medical terms, abbreviations, or code–definition pairs.
- **Compressed global blocks:** To retain awareness of the overall context, NSA constructs a set of compressed representations that summarize larger sections of the input. These act as coarse-grained context anchors, allowing the model to relate local features to the broader semantic structure of the document.
- **Selective sparse connections:** A limited number of attention heads are allocated to attend across block boundaries, enabling long-distance information flow between non-adjacent but semantically related elements (e.g., a symptom listed in one paragraph and its associated treatment guideline in another). These sparse connections are learned end-to-end, allowing the model to automatically discover which global relations are most informative for the task.

This design achieves a dual objective: it maintains sufficient expressiveness to model global dependencies, while remaining computationally efficient and hardware-aligned. The result is a scalable attention mechanism that can process long or structurally complex inputs without the exponential cost of dense self-attention.

In the context of retrieval-augmented generation systems, NSA can be viewed as a neural refinement layer that learns to weigh candidate information according to contextual relevance rather than surface similarity. In this work, NSA is repurposed from its original generative modeling application to a reranking setting, where it receives as input a query embedding and a set of candidate chunk embeddings produced by the baseline retriever. These embeddings can be interpreted as compact summaries of document content in a high-dimensional semantic space. The reranker uses sparse attention to model the interactions between the query and each candidate, as well as implicit relationships among the candidates themselves. This allows it to identify which chunks are not only semantically similar to the query, but also contextually appropriate given the overall candidate distribution.

The relevance of this approach to medical retrieval lies in the intrinsic properties of clinical text. Medical documents are dense, hierarchical, and often repetitive: key terms, codes, and laboratory measurements recur in multiple sections, but their importance depends heavily on the surrounding context. For example, a chunk mentioning “ALT” and “bilirubin” may occur in many documents, yet only in specific contexts does it indicate a diagnostic criterion for a particular condition of choice. Sparse attention allows the model to capture such context-dependent associations by learning where to focus attention within a large and heterogeneous candidate set.

Moreover, the hierarchical compression used in NSA aligns naturally with the structured nature of medical guidelines, where information is typically organized into subsections, tables, and enumerated lists. By combining local and global attention patterns, NSA can simultaneously preserve fine-grained details (e.g., a numeric laboratory threshold or procedural step) and broader thematic coherence (e.g., the overall diagnostic pathway). This combination is especially important for distinguishing near-duplicate chunks or repeated references to the same clinical code, a challenge that earlier retrieval versions struggled with.

Finally, NSA’s end-to-end trainability enables it to adapt directly to the domain-specific data distribution used in this work. Because the model learns from real question–answer pairs derived from medical documents, it can develop a specialized notion of relevance aligned with expert judgment. In effect, NSA learns to model the same kind of contextual reasoning that a human clinician might perform when linking a diagnostic query to the most informative guideline section. This ability to move beyond static similarity metrics and into learned context weighting is what highlights the potential of an NSA reranker.

2.7.2 Training of the NSA Reranker

The python code for the Native Sparse Attention architecture was found online[4], as no official DeepSeek implementations were available. The next step involved training the reranker model to optimize its ability to distinguish relevant text chunks among retrieved candidates.

For training and validation, the same structured question–answer dataset described in earlier sections was used. As the dataset was divided into training, validation and testing subsets, the predefined 100 validation entries were used along with the 337 training entries.

[4] https://github.com/lucidrains/native-sparse-attention-pytorch

During training data construction, each question was first embedded using the same medical-domain embedding model used throughout the retrieval pipeline of other model variants. Using this query embedding, all available document chunks were retrieved from the same LanceDB database, where records were saved by using the same chunking strategy as before. For each question, the full list of candidate chunks was collected, and the chunk that originated from the expected document and contained all gold spans was labeled as the positive candidate. All other retrieved chunks were treated as negatives.

The NSA reranker was trained for 50 epochs using a batch size of 8 and the Adam optimizer, with a learning rate of 5×10^{-5} and a weight decay of 10^{-5}. The model parameters were randomly initialized, and a fixed random seed of 42 ensured reproducibility across runs. Each training batch contained several question–candidate sets, and the model produced a scalar relevance score for each candidate. The loss function used was categorical cross-entropy, encouraging the model to assign the highest probability to the positive candidate within each set of retrieved chunks.

To prevent gradient explosion, the gradient norm was clipped at a maximum value of 1.0. The model's dimensionality was automatically matched to the embedding size used during retrieval, ensuring compatibility between the embedding model and the reranker network.

After each epoch, the model was evaluated on the validation split. The validation metrics included:

- Top-1 accuracy – percentage of questions where the correct chunk ranked first,
- Top-5 accuracy – percentage where it appeared within the top five,
- Mean Reciprocal Rank (MRR) – average reciprocal rank of the first correct chunk across all validation questions.

Model snapshots for the best model based on top-5 accuracy on the validation set were saved automatically every ten epochs, allowing for a comparison of 5 versions, each of which were trained for a different number of epochs (10, 20, 30, 40, 50).

Figure 9. shows the development of the training loss and validation metrics across 50 epochs. The training loss rapidly decreases within the first few epochs and reaches near-zero values by around epoch 10, indicating that the model quickly minimizes the objective on the training data. However, this sharp decline is not followed by any meaningful improvement in validation performance. All three validation metrics (top-1 accuracy, top-5 accuracy, and MRR) increase slightly in the early stages but then stabilize at relatively low levels, showing no upward trend for the remainder of the training. This behaviour suggests that the model is able to memorize training samples but fails to generalize beyond them.

The plateauing validation curves indicate that the reranker does not meaningfully improve after the first ten epochs, even though optimization continues. This stagnation implies that the NSA model, in its current configuration, is unable to effectively learn discriminative features that separate relevant from irrelevant chunks within the validation set. While the top-5 accuracy values approach 0.5, this remains considerably lower than the performance achieved by other methods presented earlier, confirming that the reranker's training outcome is suboptimal.

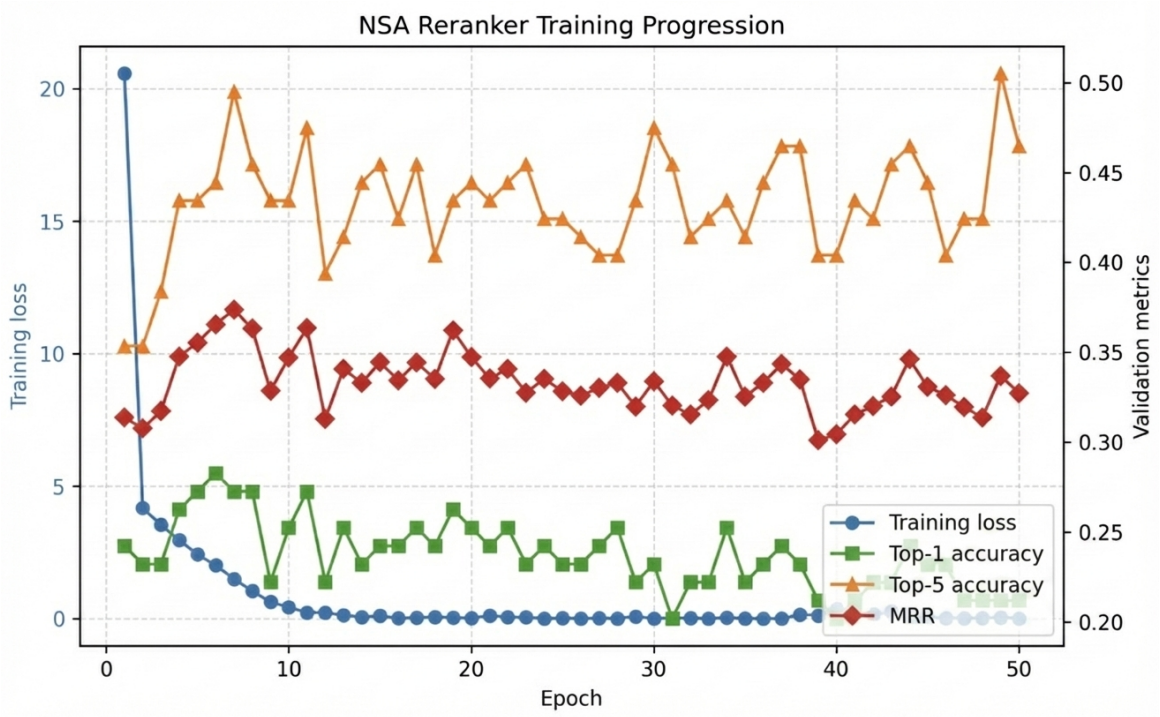

Figure 9. *Training progression of the NSA reranker*

2.7.3 Evaluation Phase

The evaluation procedure followed the same structure as in the previous retrieval experiments. The same question–answer dataset was used for testing, and the evaluation relied on the same metrics: Top-1 and Top-5 accuracy, Mean Reciprocal Rank (MRR), and the average rank of the first correct match. As before, for each question, all available document chunks were ranked by semantic similarity to the query, producing the baseline (raw) retrieval order.

The NSA reranker was then applied as a secondary ranking stage. For each query, the same set of candidate chunks retrieved by the baseline model was passed through the reranker, which produced a refined ordering based on the learned attention weights. This process was repeated for several checkpoints of the trained model, corresponding to 10, 20, 30, 40, and 50 epochs. Each checkpoint represented a distinct reranking configuration, allowing a direct comparison of how model performance evolved during training.

All reranked lists were evaluated against the same ground-truth dataset, using the same automated procedure as in previous sections. The comparison of the NSA reranker outputs with the baseline rankings provided insight into whether the learned sparse-attention mechanism was able to meaningfully improve retrieval quality beyond the original semantic similarity scores.

2.7.4 Results

The NSA reranker was evaluated on the same testing set of 100 questions used in previous experiments. Each model checkpoint (after 10, 20, 30, 40, and 50 training epochs) was applied to the raw retrieval results to produce a reranked list for each query. Table 8. summarizes the obtained metrics for all versions of the model.

Model	Top-1 acc.	Top-5 acc.	Avg. rank	MRR
Baseline (raw)	0.160	0.330	64.40	0.248
NSA (epoch 10)	0.150	0.350	40.75	0.248
NSA (epoch 20)	0.150	0.350	40.75	0.248
NSA (epoch 30)	0.150	0.350	40.75	0.248
NSA (epoch 40)	0.150	0.350	40.75	0.248
NSA (epoch 50)	0.150	0.350	40.75	0.248

Table 8. *Performance of the NSA reranker on the testing set of 100 questions.*

The results in show that the NSA reranker offers only a minor improvement in retrieval performance compared to the baseline model. The Top-5 accuracy increases slightly from 0.33 to 0.35, and the average rank of the first correct chunk improves from 64.4 to 40.8. However, Top-1 accuracy barely decreases and the mean reciprocal rank (MRR) remains unchanged, suggesting that the reranker has not meaningfully improved the ranking of the most relevant chunks.

All NSA checkpoints yield identical results after the 10^{th} epoch, consistent with the earlier training plots that showed full stagnation in validation metrics. This indicates that the model reaches its limit early and fails to learn a stronger ranking signal beyond the initial phase of training. Despite the theoretically higher representational capacity of the sparse-attention mechanism, the NSA reranker in its current form does not generalize well enough to meaningfully outperform other tested retrieval strategies.

Results

1. The baseline retrieval model, relying exclusively on dense embeddings and cosine similarity, achieved a Top-1 accuracy of 16%, Top-5 accuracy of 33% and an MRR of 0.248. Although relevant chunks were often assigned quite high on the list, many correct ones appeared far from the top.

2. Introducing subunit-based embeddings with softmax pooling increased Top-1 accuracy to 22% and Top-5 accuracy to 42%, with an MRR of 0.332. Improvement in Top-5 accuracy was visible in 45% of the cases. The average rank, however, got worse and changed from 64 to 73.

3. The keyword-matching retrieval model achieved a Top-1 accuracy of 57%, Top-5 accuracy of 88%, MRR of 0.709 and an average rank of 3.39. Compared to the baseline, 74% of queries showed an improvement.

4. During training, the NSA reranker's training loss decreased rapidly within the first 10 epochs, while Top-1 accuracy, Top-5 accuracy as well as MRR tested on the validation set plateaued early and showed no further improvement across the 50 epochs.

5. All NSA checkpoints (after training for 10-50 epochs) produced identical results: Top-1 accuracy, compared to the baseline model, decreased from 22% to 15%, while Top-5 accuracy increased very slightly to 35%. While the MRR didn't change, the average rank decreased to 40.75.

Conclusions

1. The baseline results show that embedding-only similarity frequently fails to rank clinically relevant passages among the top retrieved results.

2. A more fine-grained subunit-based scoring system improves key accuracy metrics, but also increases ranking variance. The increase in average rank indicates less stable behavior when no strongly matching subunit is present.

3. The hybrid semantic-lexical approach consistently outperformed all other evaluated methods across all ranking metrics, indicating that explicit lexical overlap can be highly informative in a medical RAG system.

4. The NSA reranker failed to improve key ranking metrics beyond baseline performance, suggesting that sparse-attention rerankers require substantially larger datasets and perhaps more diverse and meaningful questions to be effective.

5. The largest improvements were achieved through relatively simple scoring modifications rather than additional neural layers, highlighting the importance of alighning retrieval mechanisms with document structure and domain characteristics.

Limitations and Future Work

The main limitation of this work is the relatively small size of the annotated question–answer dataset, which limited the effectiveness of neural reranking and led to early overfitting in the NSA model. While subunit-based retrieval improved top-ranked accuracy, its increased ranking variance suggests that more robust aggregation or query-aware subunit selection could further stabilize results. The keyword-matching approach relied on a simple linear scoring scheme, and future work could explore more advanced hybrid or learned weighting strategies. Despite these limitations, the results show that careful preprocessing and retrieval design can substantially improve retrieval accuracy in real-world medical document collections.

References and sources

[1] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, et al. “Gpt-4 technical report.” In: *arXiv preprint arXiv:2303.08774* (2023).

[2] O. Ayala, P. Bechard. “Reducing hallucination in structured outputs via Retrieval-Augmented Generation.” In: *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*. 2024, pages 228–238.

[3] D. Bahdanau, K. Cho, Y. Bengio. “Neural machine translation by jointly learning to align and translate.” In: *arXiv preprint arXiv:1409.0473* (2014).

[4] A. M. Braşoveanu, R. Andonie. “Visualizing transformers for nlp: a brief survey.” In: *2020 24th international conference information visualisation (IV)*. IEEE. 2020, pages 270–279.

[5] T. Brown, B. Mann, N. Ryder, M. Subbiah, et al. “Language models are few-shot learners.” In: *Advances in neural information processing systems* 33 (2020), pages 1877–1901.

[6] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova. “Bert: Pre-training of deep bidirectional transformers for language understanding.” In: *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 2019, pages 4171–4186.

[7] S. Hochreiter, J. Schmidhuber. “Long short-term memory.” In: *Neural computation* 9.8 (1997), pages 1735–1780.

[8] K. Huang, J. Altosaar, R. Ranganath. “Clinicalbert: Modeling clinical notes and predicting hospital readmission.” In: *arXiv preprint arXiv:1904.05342* (2019).

[9] J. Yuan, H. Gao, D. Dai, J. Luo, et al. “Native sparse attention: Hardware-aligned and natively trainable sparse attention.” In: *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2025, pages 23078–23097.

[10] Z. Ji, N. Lee, R. Frieske, T. Yu, et al. “Survey of hallucination in natural language generation.” In: *ACM computing surveys* 55.12 (2023), pages 1–38.

[11] J. Johnson, M. Douze, H. Jégou. “Billion-scale similarity search with GPUs.” In: *IEEE Transactions on Big Data* 7.3 (2019), pages 535–547.

[12] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, et al. “Scaling laws for neural language models.” In: *arXiv preprint arXiv:2001.08361* (2020).

[13] J. Lee, W. Yoon, S. Kim, D. Kim, S. Kim, C. H. So, J. Kang. “BioBERT: a pre-trained biomedical language representation model for biomedical text mining.” In: *Bioinformatics* 36.4 (2020), pages 1234–1240.

[14] P. Lewis, E. Perez, A. Piktus, F. Petroni, et al. “Retrieval-augmented generation for knowledge-intensive nlp tasks.” In: *Advances in neural information processing systems* 33 (2020), pages 9459–9474.

[15] H. Nori, N. King, S. M. McKinney, D. Carignan, E. Horvitz. “Capabilities of gpt-4 on medical challenge problems.” In: *arXiv preprint arXiv:2303.13375* (2023).

[16] M. A. Quidwai, A. Lagana. “A rag chatbot for precision medicine of multiple myeloma.” In: *MedRxiv* (2024), pages 2024–03.

[17] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. “Language models are unsupervised multitask learners.” In: *OpenAI blog* 1.8 (2019), page 9.

[18] F. Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), page 386.

[19] D. E. Rumelhart, G. E. Hinton, R. J. Williams. “Learning representations by back-propagating errors.” In: *nature* 323.6088 (1986), pages 533–536.

[20] V. Sanh, L. Debut, J. Chaumond, T. Wolf. “DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter.” In: *arXiv preprint arXiv:1910.01108* (2019).

[21] L. L. Scientific. “REVOLUTIONIZING HEALTHCARE WITH LARGE LANGUAGE MODELS: ADVANCEMENTS, CHALLENGES, AND FUTURE PROSPECTS IN AI-DRIVEN DIAGNOSTICS AND DECISION SUPPORT.” In: *Journal of Theoretical and Applied Information Technology* 103.9 (2025).

[22] G. Team, P. Georgiev, V. I. Lei, R. Burnell, et al. “Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context.” In: *arXiv preprint arXiv:2403.05530* (2024).

[23] I. Turc, M.-W. Chang, K. Lee, K. Toutanova. “Well-read students learn better: On the importance of pre-training compact models.” In: *arXiv preprint arXiv:1908.08962* (2019).

[24] A. M. Turing. “Computing machinery and intelligence (1950).” In: *Mind* 59.236 (2021), pages 33–60.

[25] V. K. Uppalapati, D. S. Nag. “A comparative analysis of AI models in complex medical decision-making scenarios: evaluating ChatGPT, Claude AI, Bard, and Perplexity.” In: *Cureus* 16.1 (2024).

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin. “Attention is all you need.” In: *Advances in neural information processing systems* 30 (2017).

[27] Y. Wu. “Google’s neural machine translation system: Bridging the gap between human and machine translation.” In: *arXiv preprint arXiv:1609.08144* (2016).

Appendix 1: AI Usage

During the preparation of this thesis, generative AI tools (ChatGPT, Gemini) were used. These tools helped to assist with minor language corrections, grammatical clarity, as well as for LaTeX formatting support and code presentation. As explained in the work, Gemini was also used as part of the methodology of the chunking strategy. The exact way this tool was used, with an example of prompts and exact answers from the model is included in the work.

Appendix 2: Python code

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import re
import json
import glob
import torch
import stopwordsiso as stopwords
import stanza
import lancedb
from sentence_transformers import SentenceTransformer
import logging
from docx import Document
from docx.table import Table as DocxTable
import mimetypes
from google import genai
from google.genai import types
from itertools import zip_longest
from itertools import product
import time
import hashlib
from collections import deque
from pathlib import Path

CHUNK_SIZE = 200
CHUNK_OVERLAP = 50
SUBUNIT_SIZE = 10
SUBUNIT_OVERLAP = 4

TMP_IMG_FOLDER = os.path.expanduser("~/tmp_images")
MODEL_NAME = "gemini-2.0-flash"
PROMPT_TABLE = (
    """Analyze the text from this table and describe its structure clearly.
    - Focus on the step-by-step sequence of connections while explaining how the
    ↪  connections work.
    - List all the labels exactly as written.
    - Do not include general introduction, start immediately with the explanation.
    - Provide the explanation in fluent LITHUANIAN."""
)
PROMPT = (
    """Analyze the text in this image and describe its structure clearly.
    - Focus on the step-by-step sequence of connections while explaining how the
    ↪  connections work.
    - List all the labels exactly as written.
    - Do not include general introduction, start immediately with the explanation.
```

```
    - Provide the explanation in fluent LITHUANIAN."""
)
CONTEXT_WINDOW = 50
LOOKBACK_PARAGRAPHS = 3
logging.getLogger("stanza").setLevel(logging.ERROR)
nlp = stanza.Pipeline("lt", processors="tokenize, lemma", use_gpu=True, verbose=False)
lt_stopwords = stopwords.stopwords("lt")
docs_folder = os.path.expanduser("~/test_docs")
docs_folder_full = os.path.expanduser("~/test_docs_full")
db_path = os.path.abspath(os.path.expanduser("~/lancedb_data"))
logging.getLogger("sentence_transformers").setLevel(logging.ERROR)
db = lancedb.connect(db_path)

def chunk_text(text, chunk_size=CHUNK_SIZE, overlap=CHUNK_OVERLAP):

    """Splits text into fixed-size overlapping chunks."""

    words = text.split() # splits at whitespaces
    chunks = []
    start = 0
    while start < len(words):
        end = start + chunk_size
        chunk_words = words[start:end]
        chunks.append(" ".join(chunk_words))
        start += chunk_size - overlap
    return chunks


def split_into_subunits(text, subunit_size=SUBUNIT_SIZE,
↪   subunit_overlap=SUBUNIT_OVERLAP):

    """Splits a text chunk into smaller sub-units."""

    words = text.split()
    subunits = []
    start = 0
    while start < len(words):
        end = start + subunit_size
        subunit_words = words[start:end]
        subunits.append(" ".join(subunit_words))
        start += subunit_size - subunit_overlap
    return subunits


def process_chunk_for_subunits(chunk_text):
    """
    1. Extracts Tables (***) and Images (^^^) as standalone subunits.
    2. Sends the remaining text to the user's `split_into_subunits` function.
    Returns a list of strings (all subunits).
    """
```

```
    tokens = re.split(r'(\*\*\*|\^\^\^)', chunk_text)
    all_subunits = []
    i = 0
    while i < len(tokens):
        token = tokens[i]

        if token == "***":
            # The NEXT token is the table content
            if i + 1 < len(tokens):
                table_content = tokens[i+1].strip()
                if table_content:
                    all_subunits.append(table_content)
                i += 2 # Skip content and the next marker (handled by logic or next loop)
            else:
                i += 1

        elif token == "^^^":
            # The NEXT token is the image description
            if i + 1 < len(tokens):
                image_content = tokens[i+1].strip()
                if image_content:
                    all_subunits.append(image_content)
                i += 2
            else:
                i += 1

        else:
            # This is normal text (or empty string from split)
            clean_text = token.strip()
            if clean_text:
                text_parts = split_into_subunits(clean_text)
                all_subunits.extend(text_parts)
            i += 1

    return all_subunits


def clean_query(query: str, lemmatize: bool = True) -> str:

    """Preprocesses a Lithuanian query: lowercase + tokenize + lemmatize(optional) +
    ↪  remove stopwords."""

    doc = nlp(query.lower())
    result = []
    for sent in doc.sentences:
        for word in sent.words: # iterates through each word in each sentence
            token = word.lemma if lemmatize else word.text # based on the parameter
            ↪  lemmatize, either uses lemmatized form or not
```

```
            if token.isalpha() and token not in lt_stopwords: # only keep non-stopword
            ↪  tokens composed of only letters
                result.append(token)
    return " ".join(result)

def fix_superscripts(text: str) -> str:

    """Ensures that superscripts are not stuck to words when extracted from word
    ↪  files."""

    return re.sub(r'([A-Za-zĄČĘĖĮŠŲŪŽąčęėįšųūž])(\d+)', r'\1 \2', text)


def describe_image_with_gemini(image_path):

    """Send one image to Gemini Vision API and get description. Cache result in
    ↪  image_path + '.gemini.txt'"""

    client = genai.Client(api_key=os.getenv("GEMINI_API_KEY"))
    cache_path = image_path + ".gemini.txt"
    if os.path.exists(cache_path): # check if cache for this specific image already
    ↪  exists (prevents useless gemini calls)
        with open(cache_path, "r", encoding="utf-8") as f:
            return f.read().strip() # returns image description without calling gemini

    mime = mimetypes.guess_type(image_path)[0] or "image/png"
    with open(image_path, "rb") as f: # open image file in binary mode ("rb")
        data = f.read()
    contents = [
        types.Part.from_bytes(data=data, mime_type=mime), # preparing image data to send
        ↪  to gemini
        PROMPT, # this comes from config.py (predefined prompt for gemini)
    ]
    response = client.models.generate_content( # actually sends image to gemini
        model=MODEL_NAME, # also comes from config.py
        contents=contents,
        config=types.GenerateContentConfig(max_output_tokens=800),
    )
    text = response.text.strip()
    with open(cache_path, "w", encoding="utf-8") as f:
        f.write(text) # overwrites description in image path
    return text


def describe_table_with_gemini(table):
    """
    Convert a DOCX table with vertical markers into structured text,
    then send it to Gemini for description using the same caching logic
    as describe_image_with_gemini().
```

```
    - Preserves caching to table_hash.gemini.txt
    """

    # --- Generate deterministic hash based on table contents ---
    table_lines = []
    for row in table.rows:
        row_text = []
        for cell in row.cells:
            text = cell.text.strip()
            if text:
                row_text.append(text)
        table_lines.append(" | ".join(row_text))

    table_text = "\n".join(table_lines)
    table_hash = hashlib.md5(table_text.encode("utf-8")).hexdigest()
    cache_path = os.path.join(TMP_IMG_FOLDER, f"table_{table_hash}.gemini.txt")
    os.makedirs(TMP_IMG_FOLDER, exist_ok=True)

    # --- Return cached description if it exists ---
    if os.path.exists(cache_path):
        with open(cache_path, "r", encoding="utf-8") as f:
            return f.read().strip()

    # --- Prepare content for Gemini ---
    table_bytes = table_text.encode("utf-8")

    client = genai.Client(api_key=os.getenv("GEMINI_API_KEY"))

    contents = [
        types.Part.from_bytes(data=table_bytes, mime_type="text/plain"),  # Send table
        ↪  text instead of image
        PROMPT_TABLE,  # Predefined prompt for Gemini
    ]

    response = client.models.generate_content(
        model=MODEL_NAME,
        contents=contents,
        config=types.GenerateContentConfig(max_output_tokens=1500),
    )

    description = response.text.strip()

    # --- Cache the description ---
    with open(cache_path, "w", encoding="utf-8") as f:
        f.write(description)

    return description

def _table_text_from_docx_v4(table): # +- veikia su -> tipo lentelemis.
```

```
"""
v4: stable version that preserves v3 behavior and implements the
left-to-right / top-to-bottom splitting rules with correct last-column join.
"""

rows = []
for row in table.rows:
    row_cells = [cell.text.strip() for cell in row.cells]
    orig_len = len(row_cells)

    unique_nonempty = {c for c in row_cells if c.strip()}
    if len(unique_nonempty) == 1 and orig_len > 1:
        filtered_cells = [unique_nonempty.pop()]
    else:
        filtered_cells = [c for c in row_cells]
    rows.append({'cells': filtered_cells, 'orig_len': orig_len})

rows = [r for r in rows if any(c.strip() for c in r['cells'])]
if not rows:
    return ""

out_lines = []
first = rows[0]
merged_first = (
    first['orig_len'] > 1
    and len(first['cells']) == 1
    and first['cells'][0].strip() != ""
)
if merged_first and len(rows) > 1:
    out_lines.append(first['cells'][0])
    headers = rows[1]['cells']
    data_rows = [r['cells'] for r in rows[2:]]
else:
    headers = first['cells']
    data_rows = [r['cells'] for r in rows[1:]] # up to here same as v3


logical_headers = [] # distinct header values ( for headers that span multiple
↪  sub-columns )
header_counts = [] # corresponding counts
for h in headers:
    if logical_headers and h == logical_headers[-1]:
        header_counts[-1] += 1
    else:
        logical_headers.append(h)
        header_counts.append(1)
if not logical_headers:
    return first['cells'][0] or "" # return 'title' or "" if no headers found !!
    ↪  again input for changes if actually no headers exist !!
```

```
total_header_cells = sum(header_counts)

def map_row_to_grouped(row): # takes list of physical cell strings

    row_extended = list(row) + [""] * max(0, total_header_cells - len(row)) # pads
    ↪ row up to total_header_cells with ""
    grouped = {}
    idx = 0
    for h, cnt in zip(logical_headers, header_counts):
        slice_vals = row_extended[idx: idx + cnt] # physical cell strings
        ↪ corresponding to logical_header
        if h == logical_headers[0]: # first logical_header is kept as a single field
        ↪ (no splitting)
            joined = " | ".join(s for s in slice_vals if s and s.strip())
            grouped[h] = [joined] if joined.strip() else []
        else:
            vals = [s for s in slice_vals if s and s.strip()]
            grouped[h] = vals # list of non-empty cells under specifric header (will
            ↪ be expanded)
        idx += cnt
    extras = [s for s in row[idx:] if s and s.strip()] if len(row) > idx else [] # if
    ↪ more text shows up without a designated header
    return grouped, extras

for phys_row in data_rows:
    if not any((c and c.strip()) for c in phys_row):
        continue
    grouped, extras = map_row_to_grouped(phys_row)

    total_nonempty_values = sum(len(v) for v in grouped.values())
    if total_nonempty_values == 1 and len(phys_row) > 1: # finding merged note rows
        for vals in grouped.values():
            if vals:
                out_lines.append(vals[0]) # if note row, just append it
                break
        continue

    last_header = logical_headers[-1]
    last_list = list(grouped.get(last_header, [])) + list(extras) # adding extras to
    ↪ final column, which will also not be expanded anymore

    seen = set()
    last_final = []
    for item in last_list:
        if item and item not in seen:
            seen.add(item)
            last_final.append(item)
    last_text = ", ".join(last_final)
```

```
        prefix_headers = logical_headers[:-1] # headers for splitting (excluding last
        ↪ one)
        lists_for_product = []
        for h in prefix_headers:
            vals = grouped.get(h, [])
            lists_for_product.append(vals if vals else [""]) # list of lists for each
            ↪ header

        for combo in product(*lists_for_product): # combo is a tuple aligned with
        ↪ prefix_headers

            parts = []
            for h_name, val in zip(prefix_headers, combo):
                if val and val.strip():
                    parts.append(f"{h_name}: {val}") # same header: value pairs
            if last_text:
                parts.append(f"{last_header}: {last_text}") # last header
            if parts:
                out_lines.append(" | ".join(parts)) # add to out_lines

    return "\n".join(out_lines) # join everything



def _table_text_from_docx_v5(table):
    """
    v5: identical to v4, except:
        - If any cell contains the vertical marker ' ',
          the table is sent to Gemini Vision for description (using
          ↪ describe_with_gemini_table()).
        - Otherwise, behaves exactly like v4.
    """
    # Check for vertical structure markers
    has_vertical_marker = any(" " in cell.text for row in table.rows for cell in
    ↪ row.cells)

    if has_vertical_marker:
        return describe_table_with_gemini(table)

    # Otherwise, fall back to normal v4 logic
    return _table_text_from_docx_v4(table)

def gemini_rate_limit():
    """
    Ensures we don't exceed the Gemini API quota.
    Waits until the 1-minute window resets if too many calls were made.
    """
    # --- CONFIG ---
    MAX_CALLS_PER_MINUTE = 19
    WAIT_SECONDS = 60
```

```
    # --- Rate limiter ---
    GEMINI_CALL_COUNT = 0
    GEMINI_CALL_RESET = time.time()

    elapsed = time.time() - GEMINI_CALL_RESET
    if elapsed > WAIT_SECONDS:
        # Reset every minute
        GEMINI_CALL_COUNT = 0
        GEMINI_CALL_RESET = time.time()
        elapsed = 0

    if GEMINI_CALL_COUNT >= MAX_CALLS_PER_MINUTE:
        remaining = WAIT_SECONDS - elapsed
        print(f"Gemini rate limit reached ({MAX_CALLS_PER_MINUTE}/min). Sleeping
        ↪  {int(remaining)}s...")
        for sec in range(int(remaining), 0, -5):
            print(f"   ...{sec}s remaining...")
            time.sleep(5)
        # Sleep remainder (in case of non-multiple of 5)
        if remaining % 5:
            time.sleep(remaining % 5)

        # Reset after wait
        GEMINI_CALL_COUNT = 0
        GEMINI_CALL_RESET = time.time()

    GEMINI_CALL_COUNT += 1


# --- Caption detector ---

def read_docx_v5(file_path, table_reader_version):
    """
    Extracts text, tables, and images from DOCX for v5 pipeline.

    - Tables processed via `table_reader_version`
    - Images described via Gemini with rate limiting
    - Supports captions both *after* ("schema") and *before* ("paveikslas") the image
    """
    doc = Document(file_path)
    doc_name = os.path.basename(file_path)

    # --- Extract all images ---
    rel_image_map = {}
    for rel_id, rel in doc.part.rels.items():
        if rel.is_external:
            continue
        target_ref = str(rel.target_ref)
        content_type = getattr(rel.target_part, "content_type", "")
```

```
    if "image" in target_ref.lower() or "image" in content_type:
        image_data = rel.target_part.blob
        ext = (
            mimetypes.guess_extension(content_type)
            or os.path.splitext(target_ref)[1]
            or ".png"
        )
        os.makedirs(TMP_IMG_FOLDER, exist_ok=True)
        image_hash = hashlib.sha256(image_data).hexdigest()[:12] # makes the image
        ↪  filenames actually distinct, was causing problems
        image_filename = os.path.join(TMP_IMG_FOLDER, f"{image_hash}{ext}")
        if not os.path.exists(image_filename):
            with open(image_filename, "wb") as f:
                f.write(image_data)
        rel_image_map[rel_id] = image_filename

para_map = {p._p: p.text for p in doc.paragraphs}
para_by_element = {p._p: p for p in doc.paragraphs}
blocks = []
# --- Collect elements in order ---
for child in doc.element.body.iterchildren():
    tag = child.tag.split("}")[-1]

    if tag == "p":
        para_obj = para_by_element.get(child)
        text = para_map.get(child)
        if not text:
            text = " ".join("".join(child.itertext()).split())
        text = fix_superscripts(text)

        xml = getattr(child, "xml", "")
        rids = re.findall(r'(?:[a-zA-Z0-9]+:)?embed=["\'](rId\d+)["\']', xml)
        if rids:
            for rid in rids:
                if rid in rel_image_map:
                    image_filename = rel_image_map[rid]
                    try:
                        gemini_rate_limit()
                        description = describe_image_with_gemini(image_filename)
                    except Exception as e:
                        description = f"[image description unavailable: {e}]"
                    blocks.append({
                        "type": "i",
                        "image_path": image_filename,
                        "image_description": description,
                    })

        if text.strip():
            blocks.append({"type": "p", "text": text})
```

```
    elif tag == "tbl":
        table = DocxTable(child, doc)
        has_vertical_marker = any(" " in cell.text for row in table.rows for cell in
        ↪  row.cells)
        if has_vertical_marker:
            gemini_rate_limit()
        ttext = table_reader_version(table)
        if ttext.strip():
            blocks.append({"type": "t", "table_text": ttext})

# --- Pair images with captions ---

CAPTION_RE = re.compile(
    r'^\s*(\d+)\s*(pav\.|paveikslas\.|schema\.|grafikas\.)',
    re.IGNORECASE
)
caption_q = deque()
image_q = deque()
image_caption_pairs = []

for blk in blocks:
    if blk["type"] == "p":
        text = (blk.get("text") or "")
        # find *all* captions in this paragraph, in order
        for m in CAPTION_RE.finditer(text):
            cap = text.strip() #m.group(0).strip()
            if image_q:
                # there was an image earlier waiting for its caption
                img_desc = image_q.popleft()
                image_caption_pairs.append((cap, img_desc))
            else:
                caption_q.append(cap)

    elif blk["type"] == "i":
        img_desc = blk["image_description"]
        if caption_q:
            # there was a caption waiting; pair now
            cap = caption_q.popleft()
            image_caption_pairs.append((cap, img_desc))
        else:
            image_q.append(img_desc)

table_caption_pairs = []
for i, blk in enumerate(blocks):
    if blk["type"] == "t":

        if i > 0 and blocks[i-1]["type"] == "p" and
        ↪  looks_like_caption(blocks[i-1]["text"]):
            table_caption_pairs.append((blocks[i-1]["text"], blk["table_text"]))
```

```
        elif i + 1 < len(blocks) and blocks[i+1]["type"] == "p" and
        ↪  looks_like_caption(blocks[i+1]["text"]):
            table_caption_pairs.append((blocks[i+1]["text"], blk["table_text"]))

# --- Combine text into chunks ---
paragraph_texts = [b["text"] for b in blocks if b["type"] == "p"]
full_text = "\n".join(paragraph_texts)
text_chunks = chunk_text(full_text)
TABLE_MARKER = "***"
IMAGE_MARKER = "^^^"

chunks = []
pair_used = [False] * len(image_caption_pairs)
for chunk_text_str in text_chunks:
    chunk_type = "text_chunk"
# --- Inject tables ---

    for caption, table_text in table_caption_pairs:
        if caption and caption in chunk_text_str:
            pattern = re.escape(caption)
            chunk_text_str = re.sub(
                pattern,
                f"{caption}\n{TABLE_MARKER}\n{table_text}\n{TABLE_MARKER}",
                chunk_text_str,
                count=1,
            )
            chunk_type = "table"
    for idx, (caption, image_desc) in enumerate(image_caption_pairs):
        if pair_used[idx] or not caption:
            continue
        if caption in chunk_text_str:
            pattern = re.escape(caption)
            chunk_text_str = re.sub(
                pattern,
                f"{caption}\n{IMAGE_MARKER}\n{image_desc}\n{IMAGE_MARKER}",
                chunk_text_str,
                count=1,
            )
            pair_used[idx] = True
            chunk_type = "image"

    chunks.append({
        "doc_name": doc_name,
        "type": chunk_type,
        "text": chunk_text_str
    })

return chunks
```

```python
def add_new_documents(docs_folder_version, read_docx_version, embedder, table_name,
↪  table=None):

    """Finds all not yet embedded documents and embeds them, adding to vectorized DB."""

    file_paths = [
        os.path.join(docs_folder_version, f)
        for f in os.listdir(docs_folder_version)
        if f.endswith(".docx") #or f.endswith(".doc")
    ] # builds a list of full file paths for all .docx files found in 'docs_folder'

    if table is None:
        new_files = file_paths # No documents have been embedded yet, so all files are
        ↪  new
    else:
        df = table.to_pandas() # table already exists
        existing_docs = set(df["doc_name"].tolist()) # extract and filter out existing
        ↪  documents
        new_files = [p for p in file_paths if os.path.basename(p) not in existing_docs] #
        ↪  now we know which ones are new

    if not new_files:
        print("No new files to add.")
        return table # table is already complete
    all_chunks = []
    for path in new_files: # for all not yet embedded documents
        #path = ensure_docx_format(path)
        chunks = read_docx_version(path) # sends them to the chosen 'read_docx' function
        all_chunks.extend(chunks) # flattens into a single list of dictionaries for each
        ↪  chunk

    texts = [chunk["text"] for chunk in all_chunks] # raw text strings for each chunk
    embeddings = embedder.encode(
        texts,
        batch_size=32,
        show_progress_bar=True,
        convert_to_numpy=True
    ) # does the embedding with 'EMBEDDING_MODEL' defined in config.py

    records = []
    for chunk, emb in zip(all_chunks, embeddings):
        records.append({
            "doc_name": chunk["doc_name"],
            "type": chunk["type"],
            "text": chunk["text"],
            "vector": emb.tolist()
        }) # records for the database, including metadata and embeddings

    if table is None: # user did not pass an existing table
        if table_name in db.table_names(): # if table already existed
```

```
            table = db.open_table(table_name)
            table.add(records) # we only add the additional embedded documents
        else:
            table = db.create_table(table_name, data=records) # if it didn't exist, we
            ↪  create a table with records (which is all documents in this case)
    else: # user opened a table and wants to add new embeddings to it
        table.add(records)
        table.optimize()
        print(f"Added {len(new_files)} new document(s) to vectorized DB.")

    return table

def add_new_documents_v5(docs_folder_version, read_docx_version, embedder, table_name,
↪  table=None):
    """
    Ingests docs. Stores:
    - Parent Chunk Vector (for initial Top-100 retrieval).
    - Subunit Vectors (saved in metadata for reranking).
    """

    file_paths = [
        os.path.join(docs_folder_version, f)
        for f in os.listdir(docs_folder_version)
        if f.endswith(".docx")
    ]

    # --- Identify New Files ---
    if table is None:
        new_files = file_paths
    else:
        try:
            df = table.to_pandas()
            if not df.empty:
                existing_docs = set(df["doc_name"].tolist())
                new_files = [p for p in file_paths if os.path.basename(p) not in
                ↪  existing_docs]
            else:
                new_files = file_paths
        except Exception:
            # Fallback if table is empty or new
            new_files = file_paths

    if not new_files:
        print("No new files to add.")
        return table

    records = []

    for path in new_files:
        doc_name = os.path.basename(path)
```

```
        print(f"Processing {doc_name}...")

        # 1. Read Chunks
        # Note: Ensure read_docx_version returns the dict with 'text', 'type', etc.
        chunks = read_docx_version(path, table_reader_version=None)

        for chunk in chunks:
            parent_text = chunk["text"]

            # 2. Split into Subunits using the logic defined in Step 1
            subunits_text_list = process_chunk_for_subunits(parent_text)

            # Fallback: if splitting results in nothing, use parent text
            if not subunits_text_list:
                subunits_text_list = [parent_text]

            # 3. Create Embeddings
            # Construct a batch: [Parent, Subunit_1, Subunit_2, ...]
            texts_to_embed = [parent_text] + subunits_text_list

            vectors = embedder.encode(
                texts_to_embed,
                convert_to_numpy=True,
                show_progress_bar=False
            )

            parent_vector = vectors[0]
            subunit_vectors = vectors[1:] # All remaining vectors

            # 4. Prepare Record
            # Store subunit vectors as a simple list of lists so the DB accepts it as
            ↪ JSON/Array
            records.append({
                "doc_name": doc_name,
                "type": chunk["type"], # 'text_chunk', 'table', or 'image'
                "text": parent_text,   # The full context
                "vector": parent_vector.tolist(), # MAIN VECTOR for search

                # METADATA FOR RERANKING
                "subunits_texts": subunits_text_list,
                "subunits_vectors": [v.tolist() for v in subunit_vectors]
            })

    # --- Add to DB ---
    if not records:
        print("No valid records found in new files.")
        return table

    if table is None:
        if table_name in db.table_names():
```

```
            table = db.open_table(table_name)
            table.add(records)
        else:
            table = db.create_table(table_name, data=records)
    else:
        table.add(records)

    print(f"Added {len(new_files)} documents to {table_name}.")
    return table

def search(embedder, query, table, k, alpha, beta, allowed_types=None, lemmatize=True):

    """Finds top-k matching chunks based on keywords and embedding similarity."""

    cleaned_query = clean_query(query, lemmatize) # applied preprocessing to the query
    query_keywords = cleaned_query.split()
    query_embedding = embedder.encode(cleaned_query).tolist() # get embedding of the
    ↪ query
    results = table.search(query_embedding).limit(100).to_list() # use embedding to
    ↪ retrieve 100 similar chunks

    scored_results = []
    query_emb = np.array(query_embedding)
    for r in results:
        if allowed_types and r["type"] not in allowed_types:
            continue # disregards the chunk if it's type isn't allowed by user
        text_lower = r["text"].lower()
        keyword_count = sum(1 for kw in query_keywords if re.search(r'\b' + re.escape(kw)
        ↪ + r'\b', text_lower)) # number of query keywords that appear in the chunk
        chunk_embedding = np.array(r["vector"])
        similarity_score = float(np.dot(chunk_embedding, query_emb) /
        ↪ (np.linalg.norm(chunk_embedding) * np.linalg.norm(query_emb))) # exact
        ↪ similarity between query and chunk

        if keyword_count > 0 or similarity_score > 0:
            highlighted_text = highlight_keywords(r["text"], query_keywords) # highlights
            ↪ keyword matches in red using function
            scored_results.append({
                "doc_name": r["doc_name"],
                "type": r["type"],
                "text": highlighted_text,
                "keyword_count": keyword_count,
                "similarity_score": similarity_score,
                "final_score": alpha * keyword_count + beta * similarity_score # weighted
                ↪ sum, we can control the importance of keyword vs semantic matching
            })

    scored_results.sort(key=lambda x: -x["final_score"])

    return scored_results[:k]
```

```python
def search_subunit_rerank(embedder, query, table, k=5, alpha=0.5, beta=0.5,
↪  initial_top_k=100):
    """
    1. Retrieval: Gets top 100 chunks based on PARENT similarity.
    2. Reranking: For those 100, calculates a new score based on the BEST subunit match.

    Formula: Final_Score = (alpha * keyword_score) + (beta * max_subunit_similarity)
    """

    # 1. Pre-computation for Query
    query_vec = embedder.encode(query, convert_to_numpy=True)
    cleaned_query = clean_query(query, lemmatize=True) # Assuming you have clean_query
    query_keywords = cleaned_query.split()

    # 2. Initial Retrieval (Top 100 Parents)
    candidates = table.search(query_vec).limit(initial_top_k).to_list()

    scored_candidates = []

    for cand in candidates:
        # --- A. Calculate Similarity (MaxSim of Subunits) ---
        sub_vecs = np.array(cand["subunits_vectors"])
        sub_texts = cand["subunits_texts"]

        similarity_score = 0.0
        best_subunit_text = ""

        if len(sub_vecs) > 0:

            q_norm = np.linalg.norm(query_vec)
            s_norms = np.linalg.norm(sub_vecs, axis=1)

            # Safe division
            denom = s_norms * q_norm
            denom[denom == 0] = 1e-9

            scores = np.dot(sub_vecs, query_vec) / denom

            # AGGREGATE: We take the MAX score (Best matching subunit)
            best_idx = np.argmax(scores)
            similarity_score = float(scores[best_idx])
            best_subunit_text = sub_texts[best_idx]
        else:
            # Fallback if no subunits (shouldn't happen)
            best_subunit_text = cand["text"]
            # Fallback similarity to parent similarity
            p_vec = np.array(cand["vector"])
            similarity_score = float(np.dot(p_vec, query_vec) /
            ↪  (np.linalg.norm(p_vec)*np.linalg.norm(query_vec)))
```

```
        # --- B. Calculate Keyword Score ---
        # Strategy: Check in the BEST SUBUNIT. If the vector matched that specific
        ↪  sentence,
        # we want to know if the keywords are ALSO in that specific sentence.

        txt_lower = best_subunit_text.lower()
        keyword_count = sum(1 for kw in query_keywords if kw in txt_lower)

        kw_score_norm = min(keyword_count, 5.0) / 5.0

        # --- C. Final Weighted Score ---
        final_score = (alpha * kw_score_norm) + (beta * similarity_score)

        scored_candidates.append({
            "doc_name": cand["doc_name"],
            "type": cand["type"],
            "text": cand["text"], # The full parent text
            "highlight_subunit": best_subunit_text, # The specific part that matched
            "final_score": final_score,
            "sim_score": similarity_score,
            "kw_score": kw_score_norm
        })

    # 3. Sort by Final Score and Slice
    scored_candidates.sort(key=lambda x: x["final_score"], reverse=True)

    return scored_candidates[:k]



# CONFIG
data_path = "/home/ugniusbb/bioinfo_project/evaluation/questions_answers_updated.xlsx"
out_dir = Path("/home/ugniusbb/bioinfo_project/evaluation/splits")
out_dir.mkdir(parents=True, exist_ok=True)

RANDOM_SEED = 42
TEST_SIZE = 100
DEV_SIZE = 100


df = pd.read_excel(data_path)

shuffled = df.sample(frac=1.0, random_state=RANDOM_SEED).reset_index(drop=True)

test_df = shuffled.iloc[:TEST_SIZE].copy()
val_df = shuffled.iloc[TEST_SIZE:TEST_SIZE + DEV_SIZE].copy()
train_df = shuffled.iloc[TEST_SIZE + DEV_SIZE:].copy()

test_idx_path = out_dir / "test_indices.csv"
```

```python
val_idx_path = out_dir / "val_indices.csv"
train_idx_path = out_dir / "train_indices.csv"

test_df.to_csv(test_idx_path, index=False)
val_df.to_csv(val_idx_path, index=False)
train_df.to_csv(train_idx_path, index=False)

print(f"Saved splits: train={len(train_df)}, val={len(val_df)}, test={len(test_df)}")


def evaluate_and_log_retrieval_v1(
    question,
    doc_id,
    identifier,
    question_index,
    table,
    out_dir,
    alpha=1.0,
    beta=1.0,
    allowed_types=None,
    lemmatize=False,
):

    # ------------------------------
    # 1. Run raw semantic search
    # ------------------------------
    cleaned_query = clean_query(question, lemmatize)
    query_embedding = EMBEDDING_MODEL.encode(cleaned_query).tolist()
    num_total_chunks = len(table)
    raw_results = table.search(query_embedding).limit(num_total_chunks).to_list()

    # compute query norm for cosine similarity
    query_emb = np.array(query_embedding)
    query_norm = np.linalg.norm(query_emb)

    # ------------------------------
    # 2. Save RAW results
    # ------------------------------
    filename_base = f"eval_{question_index}_{identifier}"
    raw_log_path = os.path.join(out_dir, f"{filename_base}_raw.json")

    raw_serializable = []
    for rank, r in enumerate(raw_results, start=1):
        vec = np.array(r["vector"])
        cos_sim = float(np.dot(vec, query_emb) / (np.linalg.norm(vec) * query_norm))

        raw_serializable.append({
            "rank": rank-1,
            "question": question,
            "doc_name": r["doc_name"],
```

```
            "type": r.get("type", None),
            "text": r["text"],
            "cosine_similarity": cos_sim
        })

with open(raw_log_path, "w", encoding="utf-8") as f:
    json.dump(raw_serializable, f, ensure_ascii=False, indent=2)

# ----------------------------
# 3. Process results using keyword scoring
# ----------------------------
cleaned_query = clean_query(question, lemmatize)
query_keywords = cleaned_query.split()

processed = []
for r in raw_results:
    if allowed_types and r["type"] not in allowed_types:
        continue

    text_lower = r["text"].lower()
    keyword_count = sum(
        1 for kw in query_keywords
        if re.search(r'\b' + re.escape(kw) + r'\b', text_lower)
    )

    emb = np.array(r["vector"])
    sim = float(np.dot(emb, query_emb) / (np.linalg.norm(emb) * query_norm))

    if keyword_count > 0 or sim > 0:
        processed.append({
            "doc_name": r["doc_name"],
            "type": r["type"],
            "text": r["text"],
            "keyword_count": keyword_count,
            "similarity_score": sim,
            "final_score": alpha * keyword_count + beta * sim,
            "question": question,
            "cosine_similarity": sim,
        })

processed.sort(key=lambda x: -x["final_score"])
top_processed = processed

# --- ADD RANKS TO processed entries ---
for rank, entry in enumerate(top_processed, start=1):
    entry["rank"] = rank

# ----------------------------
# 4. Save PROCESSED results
# ----------------------------
```

```
    processed_log_path = os.path.join(out_dir, f"{filename_base}_processed.json")

    with open(processed_log_path, "w", encoding="utf-8") as f:
        json.dump(top_processed, f, ensure_ascii=False, indent=2)

    # ------------------------------
    # 5. Return summary
    # ------------------------------
    return {
        "question": question,
        "doc_id": doc_id,
        "num_raw": len(raw_results),
        "num_processed": len(top_processed),
    }



def evaluate_and_log_retrieval_v2(
    question,
    doc_id,
    identifier,
    question_index,
    table,
    out_dir,
    allowed_types=None,
    lemmatize=False,
    tau=10.0,           # temperature for softmax pooling
    lambda_parent=0.2   # weight for parent similarity
):
    """
    Retrieves based on parent embeddings, then re-ranks based on subunit-level
    semantic similarity using softmax pooling + parent similarity blending.

    Final score = (1 -  ) * softmax_pool(subunit_scores) +   * parent_similarity
    """

    # ------------------------------
    # 1. Query preprocessing
    # ------------------------------
    cleaned_query = clean_query(question, lemmatize)
    query_embedding = EMBEDDING_MODEL.encode(cleaned_query, convert_to_numpy=True)
    query_norm = np.linalg.norm(query_embedding)

    # ------------------------------
    # 2. Raw semantic search (parent level)
    # ------------------------------
    num_total_chunks = len(table)
    raw_results = table.search(query_embedding).limit(num_total_chunks).to_list()

    # ------------------------------
```

```python
# 3. Save raw (parent) results
# ------------------------------
filename_base = f"eval_{question_index}_{identifier}"
raw_log_path = os.path.join(out_dir, f"{filename_base}_raw.json")

raw_serializable = []
for rank, r in enumerate(raw_results, start=1):
    p_vec = np.array(r["vector"])
    p_norm = np.linalg.norm(p_vec)
    denom = p_norm * query_norm
    cos_sim = float(np.dot(p_vec, query_embedding) / denom) if denom > 1e-9 else 0.0

    raw_serializable.append({
        "rank": rank,
        "question": question,
        "doc_name": r["doc_name"],
        "type": r.get("type", None),
        "text": r["text"],
        "cosine_similarity": cos_sim
    })

with open(raw_log_path, "w", encoding="utf-8") as f:
    json.dump(raw_serializable, f, ensure_ascii=False, indent=2)

# ------------------------------
# 4. Define helper: softmax pooling
# ------------------------------
def softmax_pool(scores, tau=10.0):
    """Smooth max pooling over subunit scores."""
    if scores.size == 0:
        return 0.0
    m = np.max(scores)
    exps = np.exp((scores - m) * tau)
    return float((np.log(exps.mean()) + m * tau) / tau)

# ------------------------------
# 5. Subunit-level re-ranking
# ------------------------------
processed = []

for r in raw_results:
    if allowed_types and r["type"] not in allowed_types:
        continue

    # Retrieve subunit data
    sub_vecs = np.array(r.get("subunits_vectors", []))
    parent_vec = np.array(r["vector"])
    p_norm = np.linalg.norm(parent_vec)
    denom = p_norm * query_norm
```

```
        parent_sim = float(np.dot(parent_vec, query_embedding) / denom) if denom > 1e-9
        ↪  else 0.0

        if sub_vecs.size == 0:
            # Fallback: no subunits → use parent similarity directly
            semantic_score = parent_sim
        else:
            # Compute cosine similarity for each subunit
            s_norms = np.linalg.norm(sub_vecs, axis=1)
            dot_products = np.dot(sub_vecs, query_embedding)
            denoms = s_norms * query_norm
            denoms[denoms == 0] = 1e-9
            subunit_scores = dot_products / denoms

            # Softmax pooling
            pooled_sim = softmax_pool(subunit_scores, tau=tau)

            # Combine with parent similarity
            semantic_score = (1 - lambda_parent) * pooled_sim + lambda_parent *
            ↪  parent_sim

        processed.append({
            "doc_name": r["doc_name"],
            "type": r.get("type", "text"),
            "text": r["text"],
            "semantic_score": semantic_score,
            "question": question,
            "rank": 0
        })

    # ------------------------------
    # 6. Sort and save processed results
    # ------------------------------
    processed.sort(key=lambda x: -x["semantic_score"])
    for rank, entry in enumerate(processed, start=1):
        entry["rank"] = rank

    processed_log_path = os.path.join(out_dir, f"{filename_base}_processed.json")
    with open(processed_log_path, "w", encoding="utf-8") as f:
        json.dump(processed, f, ensure_ascii=False, indent=2)

    # ------------------------------
    # 7. Return summary
    # ------------------------------
    return {
        "question": question,
        "doc_id": doc_id,
        "num_raw": len(raw_results),
        "num_processed": len(processed),
        "top_match_score": processed[0]["semantic_score"] if processed else 0.0
```

```
    }



def evaluate_and_log_retrieval_v3(
    question,
    doc_id,
    identifier,
    question_index,
    table,
    out_dir,
    nsa_model_path,
    allowed_types=None,
    lemmatize=False,
    top_m=None,
    device=None
):
    """
    Evaluate retrieval and NSA reranking for a single question.

    - RAW: baseline pure vector similarity from LanceDB (no keyword weighting)
    - NSA: same candidates reranked using trained NSA model
    - Saves *_raw.json and *_nsa.json for each question.
    """
    os.makedirs(out_dir, exist_ok=True)

    # ------------------------------
    # 0. Setup and model loading
    # ------------------------------
    if device is None:
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    if not hasattr(evaluate_and_log_retrieval_v3, "nsa_model"):
        print(f"[INFO] Loading NSA model from {nsa_model_path}")
        head_data = table.head(1)
        sample = head_data.to_pylist()[0] if hasattr(head_data, "to_pylist") else
        ↪  head_data.to_list()[0]
        D = len(sample["vector"])
        nsa_model = NSARerankerModel(dim=D).to(device)
        nsa_model.load_state_dict(torch.load(nsa_model_path, map_location=device))
        nsa_model.eval()
        evaluate_and_log_retrieval_v3.nsa_model = nsa_model
    else:
        nsa_model = evaluate_and_log_retrieval_v3.nsa_model

    # ------------------------------
    # 1. Run raw semantic search
    # ------------------------------
    cleaned_query = clean_query(question, lemmatize)
```

```
query_embedding = EMBEDDING_MODEL.encode(cleaned_query).tolist()
num_total_chunks = len(table)
limit_val = top_m if top_m and top_m > 0 else num_total_chunks

raw_results = table.search(query_embedding).limit(limit_val).to_list()

if len(raw_results) == 0:
    print(f"[WARN] No candidates found for question {question_index}")
    return None

# Filter by allowed types if provided
if allowed_types:
    raw_results = [r for r in raw_results if r["type"] in allowed_types]

# ------------------------------
# 2. Save RAW semantic results
# ------------------------------
raw_json = []
for rank, r in enumerate(raw_results, start=1):
    raw_json.append({
        "rank": rank,
        "question": question,
        "doc_name": r["doc_name"],
        "type": r.get("type", None),
        "text": r["text"],
        "similarity_score": float(r["_distance"]) if "_distance" in r else None
    })

raw_path = os.path.join(out_dir, f"eval_{question_index}_{identifier}_raw.json")
with open(raw_path, "w", encoding="utf-8") as f:
    json.dump(raw_json, f, ensure_ascii=False, indent=2)

# ------------------------------
# 3. NSA RERANKING
# ------------------------------
query_emb = np.array(query_embedding, dtype=np.float32)
cand_embs = np.stack([np.array(r["vector"], dtype=np.float32) for r in raw_results])

q_emb_t = torch.tensor([query_emb], dtype=torch.float32).to(device)
c_emb_t = torch.tensor([cand_embs], dtype=torch.float32).to(device)

with torch.no_grad():
    logits = nsa_model(q_emb_t, c_emb_t)[0].cpu().numpy()

# Normalize logits → probabilities
probs = stable_softmax_1d(logits)

# Combine with metadata
reranked = []
for i, r in enumerate(raw_results):
```

```
            reranked.append({
                "rank": i + 1,  # temporary
                "question": question,
                "doc_name": r["doc_name"],
                "type": r.get("type", None),
                "text": r["text"],
                "nsa_logit": float(logits[i]),
                "nsa_prob": float(probs[i]),
            })

        # Sort by NSA probability (descending)
        reranked.sort(key=lambda x: -x["nsa_prob"])

        # Reassign ranks
        for rank, r in enumerate(reranked, start=1):
            r["rank"] = rank

        nsa_path = os.path.join(out_dir, f"eval_{question_index}_{identifier}_nsa.json")
        with open(nsa_path, "w", encoding="utf-8") as f:
            json.dump(reranked, f, ensure_ascii=False, indent=2)

        return {
            "question": question,
            "doc_id": doc_id,
            "num_raw": len(raw_results),
            "raw_json": raw_path,
            "nsa_json": nsa_path,
        }

from evaluation.eval_functions import evaluate_and_log_retrieval_v1
from configs.config import db, docs_folder_full, EMBEDDING_MODEL
from core.doc_readers import read_docx_v5, _table_text_from_docx_v5
from core.rag_core import add_new_documents


splits_dir = Path("/home/ugniusbb/bioinfo_project/evaluation/splits")
test_csv = splits_dir / "test_indices.csv"
test_df = pd.read_csv(test_csv)

pipeline_version = "original_v1"
table_name = f"doc_chunks_{pipeline_version}"

if table_name in db.table_names():
    table = db.open_table(table_name)
else:
    table = None

read_docx_with_table_v5 = lambda path, **kwargs: read_docx_v5(path,
↪  table_reader_version=_table_text_from_docx_v5)
```

```
table = add_new_documents(docs_folder_full, read_docx_with_table_v5, EMBEDDING_MODEL,
↪  table_name, table)


for qid, row in test_df.iterrows():

    question = row["question"]
    doc_id = row["doc_id"]

    summary = evaluate_and_log_retrieval_v1(
    question=question,
    doc_id=doc_id,
    identifier=pipeline_version,
    question_index=qid,
    table=table,
    out_dir="/home/ugniusbb/bioinfo_project/evaluation/evaluation_v1_JSONs"
    )
from evaluation.eval_functions import evaluate_and_log_retrieval_v2
from configs.config import db, docs_folder_full
from core.doc_readers import read_docx_v5, _table_text_from_docx_v5
from core.rag_core import add_new_documents_v5


splits_dir = Path("/home/ugniusbb/bioinfo_project/evaluation/splits")
test_csv = splits_dir / "test_indices.csv"
test_df = pd.read_csv(test_csv)

pipeline_version = "original_v2"
table_name = f"doc_chunks_{pipeline_version}"

if table_name in db.table_names():
    table = db.open_table(table_name)
else:
    table = None

read_docx_with_table_v5 = lambda path, **kwargs: read_docx_v5(path,
↪  table_reader_version=_table_text_from_docx_v5)
table = add_new_documents_v5(docs_folder_full, read_docx_with_table_v5, table_name,
↪  table)


for qid, row in test_df.iterrows():

    question = row["question"]
    doc_id = row["doc_id"]

    summary = evaluate_and_log_retrieval_v2(
    question=question,
    doc_id=doc_id,
    identifier=pipeline_version,
```

```
    question_index=qid,
    table=table,
    out_dir="/home/ugniusbb/bioinfo_project/evaluation/evaluation_v2_JSONs"
    )

from evaluation.eval_functions import evaluate_and_log_retrieval_v3
from configs.config import db, docs_folder_full
from core.doc_readers import read_docx_v5, _table_text_from_docx_v5
from core.rag_core import add_new_documents_v5

# ------------------------------
# Paths and configuration
# ------------------------------
splits_dir = Path("/home/ugniusbb/bioinfo_project/evaluation/splits")
test_csv = splits_dir / "test_indices.csv"
test_df = pd.read_csv(test_csv)

# Vectorized table
table_name = "doc_chunks_original_v1"
if table_name in db.table_names():
    table = db.open_table(table_name)
else:
    table = None

read_docx_with_table_v5 = lambda path, **kwargs: read_docx_v5(
    path, table_reader_version=_table_text_from_docx_v5
)
table = add_new_documents_v5(docs_folder_full, read_docx_with_table_v5, table_name,
↪  table)

# ------------------------------
# Define model checkpoints
# ------------------------------
models_dir = Path("/home/ugniusbb/bioinfo_project/evaluation/nsa_training")
model_paths = {
    "epoch10": models_dir / "nsa_reranker_epoch10_r50.434.pt",
    "epoch20": models_dir / "nsa_reranker_epoch20_r50.444.pt",
    "epoch30": models_dir / "nsa_reranker_epoch30_r50.475.pt",
    "epoch40": models_dir / "nsa_reranker_epoch40_r50.404.pt",
    "epoch50": models_dir / "nsa_reranker_epoch50_r50.465.pt",
    "best_mrr": models_dir / "nsa_reranker.pt",
}

# Base output folder
base_out_dir = Path("/home/ugniusbb/bioinfo_project/evaluation/evaluation_v3_JSONs")

# ------------------------------
# Evaluate each model
# ------------------------------
for label, model_path in model_paths.items():
```

```
    print(f"\n===== Evaluating NSA model: {label} =====")
    pipeline_version = f"nsa_{label}"
    out_dir = base_out_dir / pipeline_version
    out_dir.mkdir(parents=True, exist_ok=True)

    for qid, row in test_df.iterrows():
        question = row["question"]
        doc_id = row["doc_id"]

        evaluate_and_log_retrieval_v3(
            question=question,
            doc_id=doc_id,
            identifier=pipeline_version,
            question_index=qid,
            table=table,
            out_dir=out_dir,
            nsa_model_path=str(model_path)
        )

    print(f"Results for model '{label}' saved to: {out_dir}")

def normalize(s):
    if not isinstance(s, str):
        return ""
    s = unicodedata.normalize("NFKC", s)
    s = s.lower()
    s = re.sub(r"\s+", "", s)
    return s

def chunk_contains_all_spans(text, spans):
    text_norm = normalize(text)
    for span in spans:
        span_norm = normalize(span)
        if span_norm not in text_norm:
            return False
    return True


def evaluate_v1(csv_path, logs_dir, identifier, out_csv="eval_v1_results.csv"):

    df = pd.read_csv(csv_path)

    # Columns that may contain answer spans
    gold_cols = ["A"] + [chr(c) for c in range(ord("B"), ord("Z")+1)] + ["A1"]

    def extract_spans(row):
        spans = []
        for col in gold_cols:
            val = row.get(col)
            if isinstance(val, str) and val.strip():
```

```
            spans.append(val.strip())
    return spans

df["gold_spans"] = df.apply(extract_spans, axis=1)

results = []

# ------------------------------------------------
# Loop through ALL questions in the Excel file
# ------------------------------------------------
for qid, row in df.iterrows():

    question = row["question"]
    expected_doc = row["doc_id"]  # <-- doc to restrict matches to
    gold_spans = row["gold_spans"]

    base = f"eval_{qid}_{identifier}"

    raw_path = os.path.join(logs_dir, f"{base}_raw.json")
    proc_path = os.path.join(logs_dir, f"{base}_processed.json")

    if not (os.path.exists(raw_path) and os.path.exists(proc_path)):
        print(f"Skipping q{qid}: logs not found ({base})")
        continue

    with open(raw_path, "r", encoding="utf-8") as f:
        raw_data = json.load(f)

    with open(proc_path, "r", encoding="utf-8") as f:
        proc_data = json.load(f)

    # ----------------------------------------
    # Find indexes where ALL spans appear
    # AND the chunk belongs to the correct document
    # ----------------------------------------
    raw_match_idxs = [
        i for i, entry in enumerate(raw_data)
        if entry.get("doc_name", "").strip().lower() == expected_doc.strip().lower()
        and chunk_contains_all_spans(entry["text"], gold_spans)
    ]

    proc_match_idxs = [
        i for i, entry in enumerate(proc_data)
        if entry.get("doc_name", "").strip().lower() == expected_doc.strip().lower()
        and chunk_contains_all_spans(entry["text"], gold_spans)
    ]

    # ----------------------------------------
    # top-5 pass/fail
    # ----------------------------------------
```

```
        raw_top5_pass = any(i < 5 for i in raw_match_idxs)
        proc_top5_pass = any(i < 5 for i in proc_match_idxs)

        results.append({
            "question_index": qid,
            "question": question,
            "doc_id": expected_doc,
            "gold_spans": gold_spans,
            "raw_match_positions": raw_match_idxs,
            "processed_match_positions": proc_match_idxs,
            "raw_top5_pass": raw_top5_pass,
            "processed_top5_pass": proc_top5_pass,
        })

    # Save CSV
    out_df = pd.DataFrame(results)
    out_df.to_csv(out_csv, index=False, encoding="utf-8")
    print(f"Saved evaluation to {out_csv}")

    return out_df


# -------------------------- RUN -----------------------------
evaluate_v1(
    csv_path="/home/ugniusbb/bioinfo_project/evaluation/splits/test_indices.csv",
    logs_dir="/home/ugniusbb/bioinfo_project/evaluation/evaluation_v1_JSONs",
    identifier="original_v1",
    out_csv="/home/ugniusbb/bioinfo_project/evaluation/eval_v1_results.csv"
)

def evaluate_v2(csv_path, logs_dir, identifier, out_csv="eval_v2_results.csv"):

    df = pd.read_csv(csv_path)

    # Columns that may contain answer spans
    gold_cols = ["A"] + [chr(c) for c in range(ord("B"), ord("Z")+1)] + ["A1"]

    def extract_spans(row):
        spans = []
        for col in gold_cols:
            val = row.get(col)
            if isinstance(val, str) and val.strip():
                spans.append(val.strip())
        return spans

    df["gold_spans"] = df.apply(extract_spans, axis=1)

    results = []

    # ------------------------------------------------
```

```
# Loop through ALL questions in the Excel file
# ------------------------------------------------
for qid, row in df.iterrows():

    question = row["question"]
    expected_doc = row["doc_id"]  # <-- doc to restrict matches to
    gold_spans = row["gold_spans"]

    base = f"eval_{qid}_{identifier}"

    raw_path = os.path.join(logs_dir, f"{base}_raw.json")
    proc_path = os.path.join(logs_dir, f"{base}_processed.json")

    if not (os.path.exists(raw_path) and os.path.exists(proc_path)):
        print(f"Skipping q{qid}: logs not found ({base})")
        continue

    with open(raw_path, "r", encoding="utf-8") as f:
        raw_data = json.load(f)

    with open(proc_path, "r", encoding="utf-8") as f:
        proc_data = json.load(f)

    # ----------------------------------------
    # Find indexes where ALL spans appear
    # AND the chunk belongs to the correct document
    # ----------------------------------------
    raw_match_idxs = [
        i for i, entry in enumerate(raw_data)
        if entry.get("doc_name", "").strip().lower() == expected_doc.strip().lower()
        and chunk_contains_all_spans(entry["text"], gold_spans)
    ]

    proc_match_idxs = [
        i for i, entry in enumerate(proc_data)
        if entry.get("doc_name", "").strip().lower() == expected_doc.strip().lower()
        and chunk_contains_all_spans(entry["text"], gold_spans)
    ]

    # ----------------------------------------
    # top-5 pass/fail
    # ----------------------------------------
    raw_top5_pass = any(i < 5 for i in raw_match_idxs)
    proc_top5_pass = any(i < 5 for i in proc_match_idxs)

    results.append({
        "question_index": qid,
        "question": question,
        "doc_id": expected_doc,
        "gold_spans": gold_spans,
```

```
            "raw_match_positions": raw_match_idxs,
            "processed_match_positions": proc_match_idxs,
            "raw_top5_pass": raw_top5_pass,
            "processed_top5_pass": proc_top5_pass,
        })

    # Save CSV
    out_df = pd.DataFrame(results)
    out_df.to_csv(out_csv, index=False, encoding="utf-8")
    print(f"Saved evaluation to {out_csv}")

    return out_df


# ------------------------- RUN ----------------------------
evaluate_v2(
    csv_path="/home/ugniusbb/bioinfo_project/evaluation/splits/test_indices.csv",
    logs_dir="/home/ugniusbb/bioinfo_project/evaluation/evaluation_v2_JSONs",
    identifier="original_v2",
    out_csv="/home/ugniusbb/bioinfo_project/evaluation/eval_v2_results.csv"
)

def evaluate_v3_all_models(csv_path, base_logs_dir, model_folders, out_csv):
    """
    Evaluate retrieval for multiple NSA models (epoch10, epoch20, etc.)
    using a single baseline 'raw' reference.
    The output CSV matches the previous evaluate_v3 structure,
    but adds one column-pair (match_positions + top5_pass) per model.
    """

    df = pd.read_csv(csv_path)

    # Extract gold spans
    gold_cols = ["A"] + [chr(c) for c in range(ord("B"), ord("Z")+1)] + ["A1"]
    def extract_spans(row):
        spans = []
        for col in gold_cols:
            val = row.get(col)
            if isinstance(val, str) and val.strip():
                spans.append(val.strip())
        return spans
    df["gold_spans"] = df.apply(extract_spans, axis=1)

    results = []

    # Pick one folder to extract RAW results from (they're identical)
    base_raw_model = model_folders[0]
    base_raw_dir = Path(base_logs_dir) / base_raw_model

    print(f"\n[INFO] Using '{base_raw_model}' folder for RAW results")
```

```python
# Loop through all questions
for qid, row in df.iterrows():
    question = row["question"]
    expected_doc = row["doc_id"]
    gold_spans = row["gold_spans"]

    base = f"eval_{qid}_{base_raw_model}"

    raw_path = base_raw_dir / f"{base}_raw.json"
    if not raw_path.exists():
        print(f"Skipping q{qid}: raw JSON not found in {base_raw_model}")
        continue

    with open(raw_path, "r", encoding="utf-8") as f:
        raw_data = json.load(f)

    # Find raw matches
    raw_match_idxs = [
        i for i, entry in enumerate(raw_data)
        if entry.get("doc_name", "").strip().lower() == expected_doc.strip().lower()
        and chunk_contains_all_spans(entry["text"], gold_spans)
    ]
    raw_top5_pass = any(i < 5 for i in raw_match_idxs)

    # Initialize row with baseline data
    row_dict = {
        "question_index": qid,
        "question": question,
        "doc_id": expected_doc,
        "gold_spans": gold_spans,
        "raw_match_positions": raw_match_idxs,
        "raw_top5_pass": raw_top5_pass,
    }

    # Now evaluate each NSA model
    for model_name in model_folders:
        logs_dir = Path(base_logs_dir) / model_name
        proc_path = logs_dir / f"eval_{qid}_{model_name}_nsa.json"

        if not proc_path.exists():
            # no results for this question in this model
            row_dict[f"{model_name}_match_positions"] = []
            row_dict[f"{model_name}_top5_pass"] = False
            continue

        with open(proc_path, "r", encoding="utf-8") as f:
            proc_data = json.load(f)

        proc_match_idxs = [
```

```
                i for i, entry in enumerate(proc_data)
                if entry.get("doc_name", "").strip().lower() ==
                ↪  expected_doc.strip().lower()
                and chunk_contains_all_spans(entry["text"], gold_spans)
            ]
            top5_pass = any(i < 5 for i in proc_match_idxs)

            row_dict[f"{model_name}_match_positions"] = proc_match_idxs
            row_dict[f"{model_name}_top5_pass"] = top5_pass

        results.append(row_dict)

    # Save output CSV
    out_df = pd.DataFrame(results)
    out_df.to_csv(out_csv, index=False, encoding="utf-8")
    print(f"\n  Saved combined evaluation to {out_csv}")

    # Print quick summary
    summary_cols = [f"{m}_top5_pass" for m in model_folders]
    summary = out_df[summary_cols + ["raw_top5_pass"]].mean().to_frame("Top-5
    ↪  Accuracy").T
    print("\n===== Summary: Top-5 accuracy per model =====")
    print(summary.to_string(index=False))

    return out_df


# --------------------- RUN ---------------------

csv_path = "/home/ugniusbb/bioinfo_project/evaluation/splits/test_indices.csv"
base_logs_dir = "/home/ugniusbb/bioinfo_project/evaluation/evaluation_v3_JSONs"

model_folders = [
    "nsa_epoch10",
    "nsa_epoch20",
    "nsa_epoch30",
    "nsa_epoch40",
    "nsa_epoch50",
    "nsa_best_mrr"
]

out_csv = "/home/ugniusbb/bioinfo_project/evaluation/eval_v3_all_models_wide.csv"

evaluate_v3_all_models(csv_path, base_logs_dir, model_folders, out_csv)

def compute_metrics_v1(csv_path):
    df = pd.read_csv(csv_path)

    # ------------------------------------------------------
    # Parse the list columns safely
```

```python
# ------------------------------------------------------
def parse_list(x):
    if isinstance(x, str) and x.startswith("["):
        try:
            return ast.literal_eval(x)
        except Exception:
            return []
    return []

df["raw_match_positions"] = df["raw_match_positions"].apply(parse_list)
df["processed_match_positions"] = df["processed_match_positions"].apply(parse_list)

# ------------------------------------------------------
# BASIC METRICS
# ------------------------------------------------------
# Questions with no match at all
df["no_match_anywhere"] = df.apply(
    lambda row: len(row["raw_match_positions"]) == 0
                and len(row["processed_match_positions"]) == 0,
    axis=1
)
num_no_match = df["no_match_anywhere"].sum()
total_qs = len(df)

# Top-5 accuracy (already computed in the CSV)
raw_top5_accuracy = df["raw_top5_pass"].mean()
processed_top5_accuracy = df["processed_top5_pass"].mean()

# ------------------------------------------------------
# ADVANCED METRICS
# ------------------------------------------------------
def first_or_none(lst):
    return lst[0] if lst else None

df["raw_first_hit"] = df["raw_match_positions"].apply(first_or_none)
df["processed_first_hit"] = df["processed_match_positions"].apply(first_or_none)

# Filter valid hits
raw_hits = df["raw_first_hit"].dropna().astype(float)
proc_hits = df["processed_first_hit"].dropna().astype(float)

# --- Top-1 accuracy (correct chunk at rank 0) ---
raw_top1_accuracy = (df["raw_first_hit"] == 0).mean()
processed_top1_accuracy = (df["processed_first_hit"] == 0).mean()

# --- Average rank (1-based) ---
raw_avg_rank = (raw_hits + 1).mean() if not raw_hits.empty else None
proc_avg_rank = (proc_hits + 1).mean() if not proc_hits.empty else None

# --- Mean Reciprocal Rank (MRR) ---
```

```
raw_mrr = (1 / (raw_hits + 1)).mean() if not raw_hits.empty else None
proc_mrr = (1 / (proc_hits + 1)).mean() if not proc_hits.empty else None

# --- Percentage of questions where processed improved rank ---
df["processed_better"] = df.apply(
    lambda row:
        row["processed_first_hit"] is not None
        and (row["raw_first_hit"] is None or row["processed_first_hit"] <
        ↪  row["raw_first_hit"]),
    axis=1
)
percent_processed_better = df["processed_better"].mean()

# ------------------------------------------------------
# PRINT RESULTS
# ------------------------------------------------------
print("\n====== Baseline Retrieval Evaluation Metrics ======\n")
print(f"Total questions evaluated: {total_qs}")
print("----------------------------------------")
print(f"Questions with **no match at all**: {num_no_match}
↪  ({num_no_match/total_qs:.1%})\n")

no_match_df = df[df["no_match_anywhere"] == True][["question_index", "question",
↪  "doc_id"]]
print("\nQuestions with no match at all:")
if no_match_df.empty:
    print("  All questions matched at least once.")
else:
    for _, row in no_match_df.iterrows():
        print(f"  • Q{row['question_index']}: {row['question'][:100]}... (Doc:
        ↪  {row['doc_id']})")

print("\n--- Top-k Accuracy ---")
print(f"Raw Top-1 accuracy:         {raw_top1_accuracy:.3f}")
print(f"Processed Top-1 accuracy:   {processed_top1_accuracy:.3f}")
print(f"Raw Top-5 accuracy:         {raw_top5_accuracy:.3f}")
print(f"Processed Top-5 accuracy:   {processed_top5_accuracy:.3f}\n")

print("--- Ranking Metrics ---")
print(f"Raw average rank:           {raw_avg_rank:.2f}")
print(f"Processed average rank:     {proc_avg_rank:.2f}")
print(f"Raw MRR:                    {raw_mrr:.3f}")
print(f"Processed MRR:              {proc_mrr:.3f}\n")

print(f"Percentage where processed improved ranking:
↪  {percent_processed_better:.1%}")
print("\n==========================================\n")

# ------------------------------------------------------
# Return dictionary for optional plotting or reporting
```

```
    # ------------------------------------------------------
    return {
        "num_questions": total_qs,
        "num_no_match": num_no_match,
        "raw_top1_accuracy": raw_top1_accuracy,
        "processed_top1_accuracy": processed_top1_accuracy,
        "raw_top5_accuracy": raw_top5_accuracy,
        "processed_top5_accuracy": processed_top5_accuracy,
        "raw_avg_rank": raw_avg_rank,
        "processed_avg_rank": proc_avg_rank,
        "raw_mrr": raw_mrr,
        "processed_mrr": proc_mrr,
        "percent_processed_better": percent_processed_better,
    }

compute_metrics_v1("/home/ugniusbb/bioinfo_project/evaluation/eval_v1_results.csv")
def compute_metrics_v2(csv_path):
    """
    Computes retrieval evaluation metrics for the subunit-based model.
    Works on eval_v2_results.csv produced by evaluate_v2.py.
    """

    df = pd.read_csv(csv_path)

    # ------------------------------------------------------
    # Parse list-like columns safely
    # ------------------------------------------------------
    def parse_list(x):
        if isinstance(x, str) and x.startswith("["):
            try:
                return ast.literal_eval(x)
            except Exception:
                return []
        return []

    df["raw_match_positions"] = df["raw_match_positions"].apply(parse_list)
    df["processed_match_positions"] = df["processed_match_positions"].apply(parse_list)

    # ------------------------------------------------------
    # BASIC COUNTS
    # ------------------------------------------------------
    total_qs = len(df)

    df["no_match_anywhere"] = df.apply(
        lambda row: len(row["raw_match_positions"]) == 0
                    and len(row["processed_match_positions"]) == 0,
        axis=1
    )
    num_no_match = df["no_match_anywhere"].sum()
```

```python
# ------------------------------------------------------
# TOP-K ACCURACY
# ------------------------------------------------------
raw_top5_accuracy = df["raw_top5_pass"].mean()
proc_top5_accuracy = df["processed_top5_pass"].mean()

# ------------------------------------------------------
# RANKING METRICS
# ------------------------------------------------------
def first_or_none(lst):
    return lst[0] if lst else None

df["raw_first_hit"] = df["raw_match_positions"].apply(first_or_none)
df["processed_first_hit"] = df["processed_match_positions"].apply(first_or_none)

raw_hits = df["raw_first_hit"].dropna().astype(float)
proc_hits = df["processed_first_hit"].dropna().astype(float)

# --- Top-1 Accuracy ---
raw_top1_accuracy = (df["raw_first_hit"] == 0).mean()
proc_top1_accuracy = (df["processed_first_hit"] == 0).mean()

# --- Average Rank (1-based for readability) ---
raw_avg_rank = (raw_hits + 1).mean() if not raw_hits.empty else None
proc_avg_rank = (proc_hits + 1).mean() if not proc_hits.empty else None

# --- Mean Reciprocal Rank (MRR) ---
raw_mrr = (1 / (raw_hits + 1)).mean() if not raw_hits.empty else None
proc_mrr = (1 / (proc_hits + 1)).mean() if not proc_hits.empty else None

# --- Improvement Indicator ---
df["processed_better"] = df.apply(
    lambda row:
        row["processed_first_hit"] is not None
        and (row["raw_first_hit"] is None or row["processed_first_hit"] <
        ↪  row["raw_first_hit"]),
    axis=1
)
percent_processed_better = df["processed_better"].mean()

# ------------------------------------------------------
# PRINT RESULTS
# ------------------------------------------------------
print("\n====== Subunit Model Retrieval Evaluation Metrics ======\n")
print(f"Total questions evaluated: {total_qs}")
print("----------------------------------------")
print(f"Questions with **no match at all**: {num_no_match}
↪  ({num_no_match/total_qs:.1%})\n")
```

```
    no_match_df = df[df["no_match_anywhere"] == True][["question_index", "question",
    ↪  "doc_id"]]
    print("Questions with no match at all:")
    if no_match_df.empty:
        print("  All questions matched at least once.\n")
    else:
        for _, row in no_match_df.iterrows():
            print(f"  • Q{row['question_index']}: {row['question'][:100]}... (Doc:
            ↪  {row['doc_id']})")

    print("--- Top-k Accuracy ---")
    print(f"Raw Top-1 accuracy:         {raw_top1_accuracy:.3f}")
    print(f"Subunit Top-1 accuracy:     {proc_top1_accuracy:.3f}")
    print(f"Raw Top-5 accuracy:         {raw_top5_accuracy:.3f}")
    print(f"Subunit Top-5 accuracy:     {proc_top5_accuracy:.3f}\n")

    print("--- Ranking Metrics ---")
    print(f"Raw average rank:           {raw_avg_rank:.2f}")
    print(f"Subunit average rank:       {proc_avg_rank:.2f}")
    print(f"Raw MRR:                    {raw_mrr:.3f}")
    print(f"Subunit MRR:                {proc_mrr:.3f}\n")

    print(f"Percentage where subunit retrieval improved ranking:
    ↪  {percent_processed_better:.1%}")
    print("\n========================================\n")

    # ------------------------------------------------------
    # RETURN DICTIONARY FOR LOGGING / PLOTTING
    # ------------------------------------------------------
    return {
        "num_questions": total_qs,
        "num_no_match": num_no_match,
        "raw_top1_accuracy": raw_top1_accuracy,
        "subunit_top1_accuracy": proc_top1_accuracy,
        "raw_top5_accuracy": raw_top5_accuracy,
        "subunit_top5_accuracy": proc_top5_accuracy,
        "raw_avg_rank": raw_avg_rank,
        "subunit_avg_rank": proc_avg_rank,
        "raw_mrr": raw_mrr,
        "subunit_mrr": proc_mrr,
        "percent_subunit_better": percent_processed_better,
    }
compute_metrics_v2("/home/ugniusbb/bioinfo_project/evaluation/eval_v2_results.csv")
def compute_metrics_all_models(csv_path):
    df = pd.read_csv(csv_path)

    # --- Helper to parse lists safely ---
    def parse_list(x):
        if isinstance(x, str) and x.startswith("["):
            try:
```

```
            return ast.literal_eval(x)
        except Exception:
            return []
    return []

# --- Parse all *_match_positions columns ---
match_cols = [c for c in df.columns if c.endswith("_match_positions")]
for c in match_cols:
    df[c] = df[c].apply(parse_list)

# Identify model names (e.g. raw, nsa_epoch10, ...)
models = [c.replace("_match_positions", "") for c in match_cols]

print("\n====== Retrieval Evaluation Metrics (All Models) ======\n")
print(f"Detected models: {models}\n")

summary_rows = []

for model in models:
    match_col = f"{model}_match_positions"
    top5_col = f"{model}_top5_pass"

    if match_col not in df.columns:
        continue

    # Helper: first hit rank (None if no match)
    def first_or_none(lst):
        return lst[0] if lst else None

    df[f"{model}_first_hit"] = df[match_col].apply(first_or_none)
    hits_only = df[f"{model}_first_hit"].dropna()

    # --- Basic counts ---
    total_qs = len(df)
    no_match = df[f"{model}_first_hit"].isna().sum()

    # --- Top-1 and Top-5 accuracy ---
    top1_accuracy = (df[f"{model}_first_hit"] == 0).mean()
    top5_accuracy = df[top5_col].mean() if top5_col in df.columns else np.nan

    # --- Average rank and MRR (1-based indexing) ---
    if not hits_only.empty:
        avg_rank = (hits_only + 1).mean()
        mrr = (1.0 / (hits_only + 1)).mean()
    else:
        avg_rank, mrr = np.nan, np.nan

    summary_rows.append({
        "model": model,
        "num_questions": total_qs,
```

```
            "num_no_match": no_match,
            "top1_accuracy": top1_accuracy,
            "top5_accuracy": top5_accuracy,
            "avg_rank": avg_rank,
            "mrr": mrr,
        })

    # --- Print results nicely ---
    print("------------------------------------------------------------")
    for r in summary_rows:
        print(f"Model: {r['model']}")
        print(f"  Total questions:              {r['num_questions']}")
        print(f"  No match at all:              {r['num_no_match']}
        ↪  ({r['num_no_match']/r['num_questions']:.1%})")
        print(f"  Top-1 accuracy:               {r['top1_accuracy']:.3f}")
        print(f"  Top-5 accuracy:               {r['top5_accuracy']:.3f}")
        print(f"  Average rank (1-based):       {r['avg_rank']:.2f}")
        print(f"  Mean Reciprocal Rank (MRR):   {r['mrr']:.3f}")
        print("------------------------------------------------------------")

    summary_df = pd.DataFrame(summary_rows)
    print("\n========================================\n")
    return summary_df


# Example usage
summary_df = compute_metrics_all_models(
    "/home/ugniusbb/bioinfo_project/evaluation/eval_v3_all_models_wide.csv"
)
summary_df.to_csv(
    "/home/ugniusbb/bioinfo_project/evaluation/eval_v3_metrics_summary.csv",
    index=False
)

def create_accuracy_plots_baseline(csv_path, out_dir="plots_baseline"):
    # Ensure output directory exists
    os.makedirs(out_dir, exist_ok=True)

    # Load evaluation results
    df = pd.read_csv(csv_path)

    # Parse stored lists of match positions
    def parse_list(s):
        try:
            return ast.literal_eval(s) if isinstance(s, str) else []
        except Exception:
            return []

    df["raw_match_positions"] = df["raw_match_positions"].apply(parse_list)
    df["raw_top5_pass"] = df["raw_top5_pass"].astype(int)
```

```
    # Compute Top-5 accuracy
    raw_acc = df["raw_top5_pass"].mean()

    # -------------------------
    # BAR PLOT: Baseline Top-5 accuracy
    # -------------------------
    plt.figure(figsize=(5, 4))
    plt.bar(["Baseline"], [raw_acc], color="steelblue")
    plt.ylabel("Top-5 Accuracy")
    plt.title("Baseline Retrieval Accuracy (Top-5)")
    plt.ylim(0, 1)

    out_path = os.path.join(out_dir, "baseline_top5_accuracy.png")
    plt.savefig(out_path, dpi=200, bbox_inches="tight")
    plt.close()
    print(f"Saved: {out_path}")

    # -------------------------
    # HISTOGRAM: Match position distribution
    # -------------------------
    raw_positions = [pos for lst in df["raw_match_positions"] for pos in lst]

    plt.figure(figsize=(8, 4))
    plt.hist(raw_positions, bins=20, alpha=0.7, color="steelblue")
    plt.xlabel("Match Position (0 = best)")
    plt.ylabel("Frequency")
    plt.title("Distribution of Correct Chunk Positions (Baseline)")
    plt.grid(True, linestyle="--", alpha=0.6)

    out_path = os.path.join(out_dir, "baseline_match_position_histogram.png")
    plt.savefig(out_path, dpi=200, bbox_inches="tight")
    plt.close()
    print(f"Saved: {out_path}")

    return {"baseline_top5_accuracy": raw_acc}


# Example usage
create_accuracy_plots_baseline(
    csv_path="/home/ugniusbb/bioinfo_project/evaluation/eval_v1_results.csv",
    out_dir="/home/ugniusbb/bioinfo_project/evaluation/plots_baseline"
)

def create_accuracy_plots_v1(csv_path, out_dir="plots_v1_keyword"):
    """
    Compare baseline vs keyword-matching model retrieval performance.
    Generates:
        1) Bar chart of Top-1 & Top-5 accuracy
        2) Log-scale histogram of first correct ranks
```

```
    3) CDF (cumulative distribution) of ranks
"""
os.makedirs(out_dir, exist_ok=True)

# ------------------------------
# Load and parse CSV
# ------------------------------
df = pd.read_csv(csv_path)

def parse_list(s):
    try:
        return ast.literal_eval(s) if isinstance(s, str) else []
    except Exception:
        return []

df["raw_match_positions"] = df["raw_match_positions"].apply(parse_list)
df["processed_match_positions"] = df["processed_match_positions"].apply(parse_list)
df["raw_top5_pass"] = df["raw_top5_pass"].astype(int)
df["processed_top5_pass"] = df["processed_top5_pass"].astype(int)

# ------------------------------
# Compute metrics
# ------------------------------
def top1_acc(match_lists):
    return sum(len(lst) > 0 and lst[0] == 0 for lst in match_lists) /
    ↪  len(match_lists)

raw_top1 = top1_acc(df["raw_match_positions"])
proc_top1 = top1_acc(df["processed_match_positions"])

raw_top5 = df["raw_top5_pass"].mean()
proc_top5 = df["processed_top5_pass"].mean()

# ------------------------------
# BAR PLOT: Accuracy Comparison
# ------------------------------
labels = ["Top-1", "Top-5"]
baseline_vals = [raw_top1, raw_top5]
keyword_vals = [proc_top1, proc_top5]

x = np.arange(len(labels))
width = 0.35

plt.figure(figsize=(6, 4))
plt.bar(x - width/2, baseline_vals, width, label="Baseline (raw)", color="steelblue")
plt.bar(x + width/2, keyword_vals, width, label="Keyword (processed)",
↪  color="darkorange")
plt.xticks(x, labels)
plt.ylabel("Accuracy")
plt.title("Retrieval Accuracy: Baseline vs Keyword-Matching")
```

```
plt.ylim(0, 1)
plt.legend()
plt.grid(axis="y", linestyle="--", alpha=0.6)

out_path = os.path.join(out_dir, "v1_keyword_accuracy_comparison.png")
plt.savefig(out_path, dpi=200, bbox_inches="tight")
plt.close()
print(f"Saved: {out_path}")

# ------------------------------
# Extract rank lists
# ------------------------------
def first_or_none(lst):
    return lst[0] if len(lst) > 0 else None

df["raw_first_hit"] = df["raw_match_positions"].apply(first_or_none)
df["proc_first_hit"] = df["processed_match_positions"].apply(first_or_none)

raw_ranks = [r for r in df["raw_first_hit"].dropna()]
proc_ranks = [r for r in df["proc_first_hit"].dropna()]

# ------------------------------
# HISTOGRAM: Log-scale Rank Distribution
# ------------------------------
plt.figure(figsize=(8, 4))
plt.hist(raw_ranks, bins=50, alpha=0.6, label="Baseline (raw)", color="steelblue",
↪  density=True)
plt.hist(proc_ranks, bins=50, alpha=0.6, label="Keyword (processed)",
↪  color="darkorange", density=True)
plt.xscale("log")
plt.xlabel("Rank of First Correct Chunk (log scale)")
plt.ylabel("Density")
plt.title("Distribution of First Correct Ranks (Baseline vs Keyword)")
plt.legend()
plt.grid(True, linestyle="--", alpha=0.6)

out_path = os.path.join(out_dir, "v1_keyword_rank_distribution_log.png")
plt.savefig(out_path, dpi=200, bbox_inches="tight")
plt.close()
print(f"Saved: {out_path}")

# ------------------------------
# CDF: Cumulative Rank Distribution
# ------------------------------
def cdf_data(ranks):
    sorted_ranks = np.sort(ranks)
    y = np.arange(1, len(sorted_ranks) + 1) / len(sorted_ranks)
    return sorted_ranks, y

raw_x, raw_y = cdf_data(raw_ranks)
```

```
    proc_x, proc_y = cdf_data(proc_ranks)

    plt.figure(figsize=(8, 4))
    plt.plot(raw_x, raw_y, label="Baseline (raw)", color="steelblue", linewidth=2)
    plt.plot(proc_x, proc_y, label="Keyword (processed)", color="darkorange",
    ↪  linewidth=2)
    plt.xscale("log")
    plt.xlabel("Rank (log scale)")
    plt.ylabel("Cumulative Fraction of Queries")
    plt.title("Cumulative Rank Distribution (Baseline vs Keyword)")
    plt.legend()
    plt.grid(True, linestyle="--", alpha=0.6)

    out_path = os.path.join(out_dir, "v1_keyword_rank_cdf.png")
    plt.savefig(out_path, dpi=200, bbox_inches="tight")
    plt.close()
    print(f"Saved: {out_path}")

    # ------------------------------
    # Return summary
    # ------------------------------
    return {
        "raw_top1_accuracy": raw_top1,
        "processed_top1_accuracy": proc_top1,
        "raw_top5_accuracy": raw_top5,
        "processed_top5_accuracy": proc_top5
    }


# Example usage
create_accuracy_plots_v1(
    csv_path="/home/ugniusbb/bioinfo_project/evaluation/eval_v1_results.csv",
    out_dir="/home/ugniusbb/bioinfo_project/evaluation/plots_v1_keyword"
)

def create_accuracy_plots_v2_from_single_csv(
    csv_path,
    out_dir="plots_subunit_comparison",
    max_rank_for_hist=None,  # e.g., 300 to clip long tails; None = no clip
    bins=40
):
    # Ensure output dir
    os.makedirs(out_dir, exist_ok=True)

    # Load results
    df = pd.read_csv(csv_path)

    # --- Helpers ---
    def parse_list(s):
        try:
```

```
        return ast.literal_eval(s) if isinstance(s, str) else []
    except Exception:
        return []

def first_or_none(lst):
    return lst[0] if lst else None

# Parse list-like columns
df["raw_match_positions"] = df["raw_match_positions"].apply(parse_list)
df["processed_match_positions"] = df["processed_match_positions"].apply(parse_list)

# Ensure booleans are numeric for mean()
if "raw_top5_pass" in df.columns:
    df["raw_top5_pass"] = df["raw_top5_pass"].astype(int)
if "processed_top5_pass" in df.columns:
    df["processed_top5_pass"] = df["processed_top5_pass"].astype(int)

# ---------- Metrics (single CSV has both) ----------
# Top-5 (from CSV)
raw_top5 = df["raw_top5_pass"].mean()
proc_top5 = df["processed_top5_pass"].mean()

# Top-1 (compute from first match position)
df["raw_first_hit"] = df["raw_match_positions"].apply(first_or_none)
df["proc_first_hit"] = df["processed_match_positions"].apply(first_or_none)

raw_top1 = (df["raw_first_hit"] == 0).mean()
proc_top1 = (df["proc_first_hit"] == 0).mean()

# For histograms: use only the *first* correct rank when it exists
raw_first_hits = df["raw_first_hit"].dropna().astype(int).tolist()
proc_first_hits = df["proc_first_hit"].dropna().astype(int).tolist()

# Optionally clip long tails for visibility (does not affect metrics)
if max_rank_for_hist is not None:
    raw_first_hits = [min(r, max_rank_for_hist) for r in raw_first_hits]
    proc_first_hits = [min(r, max_rank_for_hist) for r in proc_first_hits]

# ---------- Plot 1: Accuracy comparison (Top-1 & Top-5) ----------
plt.figure(figsize=(6.5, 4.2))
labels = ["Top-1", "Top-5"]
baseline_vals = [raw_top1, raw_top5]
subunit_vals = [proc_top1, proc_top5]
x = np.arange(len(labels))
width = 0.38

plt.bar(x - width/2, baseline_vals, width, label="Baseline (raw)")
plt.bar(x + width/2, subunit_vals, width, label="Subunits (processed)")

plt.xticks(x, labels)
```

```
    plt.ylim(0, 1)
    plt.ylabel("Accuracy")
    plt.title("Retrieval Accuracy: Baseline vs Subunits")
    plt.grid(axis="y", linestyle="--", alpha=0.5)
    plt.legend()

    out_path = os.path.join(out_dir, "accuracy_comparison_v2.png")
    plt.savefig(out_path, dpi=200, bbox_inches="tight")
    plt.close()
    print(f"Saved: {out_path}")

    # ---------- Plot 2: Distribution of first correct ranks ----------
    plt.figure(figsize=(8.5, 4.5))
    plt.hist(raw_first_hits, bins=bins, alpha=0.6, density=True, label="Baseline (raw)")
    plt.hist(proc_first_hits, bins=bins, alpha=0.6, density=True, label="Subunits
    ↪  (processed)")
    plt.xlabel("Rank of First Correct Chunk (0 = best)")
    plt.ylabel("Density")
    title = "Distribution of First Correct Ranks (Baseline vs Subunits)"
    if max_rank_for_hist is not None:
        title += f" (capped at {max_rank_for_hist})"
    plt.title(title)
    plt.grid(True, linestyle="--", alpha=0.5)
    plt.legend()

    out_path = os.path.join(out_dir, "rank_distribution_comparison_v2.png")
    plt.savefig(out_path, dpi=200, bbox_inches="tight")
    plt.close()
    print(f"Saved: {out_path}")

    return {
        "raw_top1": float(raw_top1),
        "raw_top5": float(raw_top5),
        "processed_top1": float(proc_top1),
        "processed_top5": float(proc_top5),
        "n_raw_first_hits": len(raw_first_hits),
        "n_proc_first_hits": len(proc_first_hits),
    }

# Example usage:
create_accuracy_plots_v2_from_single_csv(
    csv_path="/home/ugniusbb/bioinfo_project/evaluation/eval_v2_results.csv",
    out_dir="/home/ugniusbb/bioinfo_project/evaluation/plots_subunit_comparison"
)

def plot_nsa_training_metrics(csv_path, out_path=None):
    """
    Visualize NSA reranker training progression:
      - Training loss
      - Validation Recall@1
```

```
    - Validation Recall@5
    - Validation MRR
    """

    # Load metrics
    df = pd.read_csv(csv_path)
    if "epoch" not in df.columns:
        df["epoch"] = range(1, len(df) + 1)

    # Create figure
    plt.figure(figsize=(8, 5))
    ax1 = plt.gca()

    # ---- Left y-axis: training loss ----
    ax1.plot(df["epoch"], df["train_loss"], color="tab:blue", marker="o", label="Training
    ↪  loss")
    ax1.set_xlabel("Epoch")
    ax1.set_ylabel("Training loss", color="tab:blue")
    ax1.tick_params(axis="y", labelcolor="tab:blue")
    ax1.grid(True, linestyle="--", alpha=0.5)

    # ---- Right y-axis: validation metrics ----
    ax2 = ax1.twinx()
    ax2.plot(df["epoch"], df["val_recall@1"], color="tab:green", marker="s",
    ↪  label="Recall@1")
    ax2.plot(df["epoch"], df["val_recall@5"], color="tab:orange", marker="^",
    ↪  label="Recall@5")
    ax2.plot(df["epoch"], df["val_mrr"], color="tab:red", marker="D", label="MRR")
    ax2.set_ylabel("Validation metrics", color="black")
    ax2.tick_params(axis="y", labelcolor="black")

    # ---- Combined legend ----
    lines1, labels1 = ax1.get_legend_handles_labels()
    lines2, labels2 = ax2.get_legend_handles_labels()
    ax2.legend(lines1 + lines2, labels1 + labels2, loc="lower right", frameon=True)

    # ---- Title and layout ----
    plt.title("NSA Reranker Training Progression")
    plt.tight_layout()

    # ---- Save or show ----
    if out_path is None:
        out_path = os.path.join(os.path.dirname(csv_path), "nsa_training_progress.png")

    plt.savefig(out_path, dpi=300, bbox_inches="tight")
    plt.close()
    print(f"Saved plot to: {out_path}")

# Example usage:
plot_nsa_training_metrics("/home/ugniusbb/bioinfo_project/evaluation/nsa_training/training_metrics.cs
```

```
SPLITS_DIR = "/home/ugniusbb/bioinfo_project/evaluation/splits"
TRAIN_CSV = os.path.join(SPLITS_DIR, "train_indices.csv")
VAL_CSV   = os.path.join(SPLITS_DIR, "val_indices.csv")

# Where to write logs & artifacts
OUT_DIR = "/home/ugniusbb/bioinfo_project/evaluation/nsa_training"
os.makedirs(OUT_DIR, exist_ok=True)

# LanceDB table with embedded chunks
TABLE_NAME = "doc_chunks_original_v1"

# Candidate retrieval settings

# Training settings
RANDOM_SEED = 42
BATCH_SIZE = 8
EPOCHS = 50
LR = 5e-5
WEIGHT_DECAY = 1e-5
GRAD_CLIP_NORM = 1.0
NUM_WORKERS = 0  # DataLoader workers (set >0 if CPU allows)

# Model save / logs
MODEL_PATH = os.path.join(OUT_DIR, "nsa_reranker.pt")
METRICS_CSV = os.path.join(OUT_DIR, "training_metrics.csv")
LOSS_PLOT = os.path.join(OUT_DIR, "loss_curve.png")
VAL_HIST_PLOT = os.path.join(OUT_DIR, "val_prob_hist.png")
CONFIG_JSON = os.path.join(OUT_DIR, "train_config.json")

# =========================
# === PROJECT IMPORTS =====
# =========================
try:
    from configs.config import db, EMBEDDING_MODEL
except Exception as e:
    raise RuntimeError("Could not import db or EMBEDDING_MODEL from configs.config") from
    ↪  e

# NSA import (your package/repo)
try:
    from native_sparse_attention_pytorch import SparseAttention
except Exception:
    # try relative repo path (as in your previous script)
    repo_path = os.path.join(os.path.dirname(__file__),
    ↪  "native-sparse-attention-pytorch")
    if os.path.exists(repo_path) and repo_path not in os.sys.path:
        os.sys.path.append(repo_path)
    from native_sparse_attention_pytorch import SparseAttention
```

```python
# ========================
# ====== UTILITIES =======
# ========================

def set_seed(seed=RANDOM_SEED):
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

def open_table(name):
    return db.open_table(name)

def clean_query(q: str, lemmatize: bool = False) -> str:
    # Minimal cleaner (aligns with your earlier usage)
    q = q.strip()
    # (optional) add your own lemmatizer here if needed
    return q

def normalize(s: str) -> str:
    """Maximally permissive normalization for matching across Excel + DOCX + LanceDB."""
    if not isinstance(s, str):
        return ""
    s = s.strip().lower()
    s = unicodedata.normalize("NFKD", s)
    # unify dash variants
    s = s.replace("-", "-").replace("-", "-").replace("-", "-")
    # remove newlines, all spaces (including non-breaking)
    s = re.sub(r"[\s\u00A0\u202F\r\n]+", "", s)
    # remove all punctuation completely
    s = re.sub(rf"[{re.escape(string.punctuation)}]", "", s)
    # remove anything non-alphanumeric (keep letters + digits only)
    s = re.sub(r"[^a-z0-9]", "", s)
    return s



def chunk_contains_all_spans(text: str, spans: List[str]) -> bool:
    t = normalize(text)
    for sp in spans:
        if normalize(sp) not in t:
            return False
    return True

def extract_gold_spans(row: pd.Series) -> List[str]:
    # Mirrors your evaluation: columns "A".."Z" plus "A1"
    gold_cols = ["A"] + [chr(c) for c in range(ord("B"), ord("Z")+1)] + ["A1"]
    spans = []
    for col in gold_cols:
        if col in row:
```

```
            val = row[col]
            if isinstance(val, str) and val.strip():
                spans.append(val.strip())
    return spans

def stable_softmax_1d(logits: np.ndarray) -> np.ndarray:
    mx = np.max(logits)
    exps = np.exp(logits - mx)
    s = np.sum(exps)
    if s == 0 or not np.isfinite(s):
        return np.ones_like(logits) / len(logits)
    return exps / s


# =========================
# == DATASET BUILDING =====
# =========================

def build_examples_from_csv(
    csv_path: str,
    table,
) -> Tuple[List[Tuple[str, np.ndarray, int]], int]:
    """
    For each question in csv:
      - retrieve top_m candidates from LanceDB using semantic search
      - locate the index of the 'positive' candidate: correct doc AND contains all gold
      ↪  spans
      - if not found, skip example
    Return: list of (question_text, candidate_vectors[N,D], positive_index) and embedding
    ↪  dim D
    """
    df = pd.read_csv(csv_path)
    examples = []
    D = None
    skipped = 0

    for i, row in df.iterrows():
        question = str(row.get("question", "")).strip()
        if not question:
            continue

        expected_doc = str(row.get("doc_id", "")).strip()
        gold_spans = extract_gold_spans(row)

        # Retrieve candidates
        cleaned = clean_query(question)
        q_emb = EMBEDDING_MODEL.encode(cleaned).tolist()
        num_chunks = len(table)
        # NOTE: search returns dicts with fields: doc_name, text, vector, type, ...
        candidates = table.search(q_emb).limit(num_chunks).to_list()
```

```
        if not candidates:
            skipped += 1
            continue

        # Find positive index
        pos_idx = None
        for idx, c in enumerate(candidates):
            if str(c.get("doc_name", "")).strip().lower() ==
            ↪  expected_doc.strip().lower():
                if chunk_contains_all_spans(c.get("text", ""), gold_spans):
                    pos_idx = idx
                    break

        if pos_idx is None:
            # no positive among top-m → skip
            skipped += 1
            print(f"[SKIP] No positive match for Q{ i }: '{question[:80]}' |
            ↪  doc_id={expected_doc} | spans={gold_spans}")
            continue

        # Build candidate embedding matrix
        vecs = []
        for c in candidates:
            v = np.array(c["vector"], dtype=np.float32)
            vecs.append(v)
        cand_matrix = np.stack(vecs, axis=0)   # (N, D)

        if D is None:
            D = cand_matrix.shape[1]
        else:
            if cand_matrix.shape[1] != D:
                raise RuntimeError(f"Inconsistent embedding dim: expected {D}, got
                ↪  {cand_matrix.shape[1]}")

        examples.append((question, cand_matrix, pos_idx))

    if len(examples) == 0:
        raise RuntimeError("No train/val examples built. Check TOP_M and gold spans
        ↪  coverage.")

    print(f"[build_examples_from_csv] {csv_path}: built {len(examples)} examples (skipped
    ↪  {skipped}); D={D}")
    return examples, D


class RerankDataset(Dataset):
    def __init__(self, examples: List[Tuple[str, np.ndarray, int]]):
        self.examples = examples
```

```
    def __len__(self):
        return len(self.examples)

    def __getitem__(self, idx):
        q_text, cand_vecs, pos_idx = self.examples[idx]
        q_emb = np.array(EMBEDDING_MODEL.encode(q_text), dtype=np.float32)
        return q_emb, cand_vecs.astype(np.float32), int(pos_idx)


def collate_fn(batch):
    q_list, cand_list, pos_list = zip(*batch)
    q_tensor = torch.tensor(np.stack(q_list, axis=0), dtype=torch.float32)          #
    ↪ (B,D)
    cand_tensor = torch.tensor(np.stack(cand_list, axis=0), dtype=torch.float32)    #
    ↪ (B,N,D)
    pos_tensor = torch.tensor(pos_list, dtype=torch.long)                           #
    ↪ (B,)
    return q_tensor, cand_tensor, pos_tensor


# ==========================
# ===== NSA MODEL =========
# ==========================

class NSARerankerModel(nn.Module):
    def __init__(self, dim: int, heads: int = 2, dim_head: int = None,
                 kv_heads: int = 1, selection_block_size=4,
                 compress_block_size=4, compress_block_stride=2,
                 num_selected_blocks=1):
        super().__init__()
        if dim_head is None:
            dim_head = max(16, dim // max(1, heads))

        self.dim = dim
        self.to_qkv = nn.Linear(dim, dim * 3)  # (optionally usable; SparseAttention
        ↪ handles QKV inside)
        self.attn = SparseAttention(
            dim=dim,
            dim_head=dim_head,
            heads=heads,
            kv_heads=kv_heads,
            causal=False,
            sliding_window_size=2,
            compress_block_size=compress_block_size,
            compress_block_sliding_stride=compress_block_stride,
            selection_block_size=selection_block_size,
            num_selected_blocks=num_selected_blocks
        )
        self.norm = nn.LayerNorm(dim)
```

```
    def forward(self, q_emb: torch.Tensor, cand_embs: torch.Tensor) -> torch.Tensor:
        """
        q_emb:     (B, D)
        cand_embs: (B, N, D)
        return:    (B, N) logits (higher should be 'more relevant')
        """
        tokens = torch.cat([q_emb.unsqueeze(1), cand_embs], dim=1)  # (B, N+1, D)
        out = self.attn(tokens)                                     # (B, N+1, D)
        out = self.norm(out)
        q_out = out[:, 0, :]      # (B, D)
        cand_outs = out[:, 1:, :] # (B, N, D)
        logits = torch.einsum("bd,bnd->bn", q_out, cand_outs)       # dot-product logits
        return logits


# ==========================
# ====== TRAIN/VAL ========
# ==========================

@torch.no_grad()
def evaluate_model(model, loader, device) -> Dict[str, float]:
    model.eval()
    ranks = []
    all_pos_probs = []
    all_neg_probs = []

    for q_t, cand_t, pos_t in loader:
        q_t = q_t.to(device)
        cand_t = cand_t.to(device)
        logits = model(q_t, cand_t)  # (B, N)
        # rank
        logits_np = logits.cpu().numpy()
        pos_np = pos_t.numpy()
        for i in range(logits_np.shape[0]):
            l = logits_np[i]
            r = np.argsort(l)[::-1]
            pos_idx = int(pos_np[i])
            rank = int(np.where(r == pos_idx)[0][0]) + 1
            ranks.append(rank)

            # collect probs for quick sanity histogram
            probs = stable_softmax_1d(np.clip(l, -50, 50))
            all_pos_probs.append(float(probs[pos_idx]))
            all_neg_probs.extend([float(probs[j]) for j in range(len(probs)) if j !=
            ↪  pos_idx])

    ranks = np.array(ranks)
    recall1 = float(np.mean(ranks <= 1))
    recall5 = float(np.mean(ranks <= 5))
    mrr = float(np.mean(1.0 / ranks))
```

```
    return {"recall@1": recall1, "recall@5": recall5, "mrr": mrr}, all_pos_probs,
    ↪  all_neg_probs


def train():
    set_seed(RANDOM_SEED)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"[INFO] Device: {device}")

    # Open table
    table = open_table(TABLE_NAME)

    # Build datasets from your splits
    train_examples, D1 = build_examples_from_csv(TRAIN_CSV, table)
    val_examples, D2   = build_examples_from_csv(VAL_CSV, table)
    if D1 != D2:
        raise RuntimeError(f"Embedding dim mismatch: train D={D1}, val D={D2}")
    D = D1

    train_ds = RerankDataset(train_examples)
    val_ds   = RerankDataset(val_examples)

    train_loader = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True,
                              collate_fn=collate_fn, num_workers=NUM_WORKERS)
    val_loader   = DataLoader(val_ds, batch_size=BATCH_SIZE, shuffle=False,
                              collate_fn=collate_fn, num_workers=NUM_WORKERS)

    # Model / optim
    model = NSARerankerModel(dim=D).to(device)
    optimizer = optim.Adam(model.parameters(), lr=LR, weight_decay=WEIGHT_DECAY)
    criterion = nn.CrossEntropyLoss()

    # Log config
    with open(CONFIG_JSON, "w", encoding="utf-8") as f:
        json.dump({
            "TABLE_NAME": TABLE_NAME,
            "BATCH_SIZE": BATCH_SIZE,
            "EPOCHS": EPOCHS,
            "LR": LR,
            "WEIGHT_DECAY": WEIGHT_DECAY,
            "RANDOM_SEED": RANDOM_SEED,
            "EMBED_DIM": D
        }, f, indent=2)

    # Training loop
    metrics_rows = []
    train_losses = []
    best_mrr = -1.0
    best_state = None
```

```
print("\n[TRAIN] Starting...")
for epoch in range(1, EPOCHS + 1):
    model.train()
    running_loss = 0.0
    steps = 0

    for q_t, cand_t, pos_t in train_loader:
        q_t = q_t.to(device)
        cand_t = cand_t.to(device)
        pos_t = pos_t.to(device)

        logits = model(q_t, cand_t)      # (B, N)
        loss = criterion(logits, pos_t)

        optimizer.zero_grad(set_to_none=True)
        loss.backward()
        if GRAD_CLIP_NORM is not None:
            nn.utils.clip_grad_norm_(model.parameters(), GRAD_CLIP_NORM)
        optimizer.step()

        running_loss += float(loss.item())
        steps += 1

    avg_loss = running_loss / max(1, steps)
    train_losses.append(avg_loss)

    # Validation
    val_metrics, pos_probs, neg_probs = evaluate_model(model, val_loader, device)

    # Save best by MRR
    if val_metrics["mrr"] > best_mrr:
        best_mrr = val_metrics["mrr"]
        best_state = {k: v.cpu().clone() if hasattr(v, "device") else v
                      for k, v in model.state_dict().items()}
        # torch.save(best_state, MODEL_PATH)

# --- Additionally: save snapshot every 10 epochs by top-5 recall ---
    if epoch % 10 == 0:
        model_snapshot_path = os.path.join(
            OUT_DIR, f"nsa_reranker_epoch{epoch}_r5{val_metrics['recall@5']:.3f}.pt"
        )
        snapshot_state = {k: v.cpu().clone() if hasattr(v, "device") else v
                          for k, v in model.state_dict().items()}
        torch.save(snapshot_state, model_snapshot_path)
        print(f"[SNAPSHOT] Saved model checkpoint at epoch {epoch}:
        ↪  {model_snapshot_path}")

    # Log row
    row = {
        "epoch": epoch,
```

```
            "train_loss": avg_loss,
            "val_recall@1": val_metrics["recall@1"],
            "val_recall@5": val_metrics["recall@5"],
            "val_mrr": val_metrics["mrr"]
        }
        metrics_rows.append(row)

        print(f"Epoch {epoch:02d} | loss {avg_loss:.5f} | "
              f"val r@1 {val_metrics['recall@1']:.3f} r@5 {val_metrics['recall@5']:.3f} "
              f"mrr {val_metrics['mrr']:.3f}")

        # Optional: update histogram each epoch (last epoch will overwrite)
        if len(pos_probs) > 0 and len(neg_probs) > 0:
            plt.figure(figsize=(7,4))
            plt.hist(pos_probs, bins=20, alpha=0.6, label="positive candidate prob")
            plt.hist(neg_probs, bins=20, alpha=0.6, label="negative candidate prob")
            plt.legend()
            plt.title("Validation predicted probabilities")
            plt.tight_layout()
            plt.savefig(VAL_HIST_PLOT)
            plt.close()

    # Save metrics CSV
    pd.DataFrame(metrics_rows).to_csv(METRICS_CSV, index=False)

    # Save loss curve
    plt.figure(figsize=(6,4))
    plt.plot(train_losses, marker="o")
    plt.title("Training loss")
    plt.xlabel("epoch")
    plt.ylabel("loss")
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(LOSS_PLOT)
    plt.close()

    print(f"\n[DONE] Best val MRR: {best_mrr:.4f}")
    print(f"Saved best model to: {MODEL_PATH}")
    print(f"Metrics CSV: {METRICS_CSV}")
    print(f"Loss plot: {LOSS_PLOT}")
    print(f"Val prob histogram: {VAL_HIST_PLOT}")


if __name__ == "__main__":
    train()
```

Appendix 3: Python code for NSA found online

```python
from __future__ import annotations

from copy import deepcopy
from math import ceil
from functools import partial

import torch
import torch.nn.functional as F
from torch import nn, arange, stack, cat, tensor, Tensor
from torch.nn import Module, ModuleList

from local_attention import LocalAttention

from rotary_embedding_torch import RotaryEmbedding

# einstein notation

import einx
from einops import einsum, repeat, rearrange, reduce, pack, unpack
from einops.layers.torch import Rearrange

# b - batch
# h - heads
# qh - grouped query heads
# n - sequence (token level or compressed)
# w - windows, for fine or compressed
# i, j - query / key sequence
# d - feature dimension
# s - strategies

# flex attention
# https://pytorch.org/blog/flexattention/

flex_attention = None

try:
    from torch.nn.attention.flex_attention import flex_attention, create_block_mask
    if torch.cuda.is_available():
        flex_attention = torch.compile(flex_attention)
except ImportError:
    pass

# flex attn sliding attention mask


def create_sliding_mask(seq_len, window_size, causal = True):
```

```
    def sliding_mask(_, __, q_idx, kv_idx):

        distance = q_idx - kv_idx
        backward_sliding_mask = distance <= window_size

        forward_distance = 0 if causal else -window_size
        forward_sliding_mask = distance >= forward_distance

        return backward_sliding_mask & forward_sliding_mask

    block_mask = create_block_mask(sliding_mask, B = None, H = None, Q_LEN = seq_len,
    ↪  KV_LEN = seq_len, _compile = True)
    return block_mask

def create_compress_mask(seq_len, kv_seq_len, compress_block_sliding_stride, mem_kv_len =
↪  0, causal = True):

    if not causal:
        return None

    # cannot be used as using attention logits for importance score
    # but just to show the immense potential of flex attention

    def compress_mask(_, __, q_idx, kv_idx):
        is_mem_kv = kv_idx < mem_kv_len

        kv_without_mem = kv_idx - mem_kv_len
        compress_kv_idx = (kv_without_mem * compress_block_sliding_stride) +
        ↪  (compress_block_sliding_stride - 1)

        causal_mask = q_idx > compress_kv_idx
        return causal_mask | is_mem_kv

    block_mask = create_block_mask(compress_mask, B = None, H = None, Q_LEN = seq_len,
    ↪  KV_LEN = kv_seq_len + mem_kv_len, _compile = True)
    return block_mask

def create_fine_mask(seq_len, fine_block_size, causal = True):

    def inner(selected_block_indices: Tensor, num_grouped_queries = 1):
        device = selected_block_indices.device
        batch, kv_heads = selected_block_indices.shape[:2]

        one_hot_selected_block_indices = torch.zeros((*selected_block_indices.shape[:-1],
        ↪  seq_len // fine_block_size), device = device, dtype = torch.bool)
        one_hot_selected_block_indices.scatter_(-1, selected_block_indices, True)

        def fine_mask(b_idx, h_idx, q_idx, kv_idx):
```

```
            compressed_q_idx = q_idx // fine_block_size
            compressed_kv_idx = kv_idx // fine_block_size
            kv_head_idx = h_idx // num_grouped_queries

            is_selected = one_hot_selected_block_indices[b_idx, kv_head_idx, q_idx,
            ↪  compressed_kv_idx]

            if not causal:
                return is_selected

            causal_mask = q_idx >= kv_idx
            block_diagonal = compressed_q_idx == compressed_kv_idx

            return (causal_mask & (block_diagonal | is_selected))

        block_mask = create_block_mask(fine_mask, B = batch, H = kv_heads *
        ↪  num_grouped_queries, Q_LEN = seq_len, KV_LEN = seq_len, _compile = True)
        return block_mask

    return inner

# helpers

def exists(v):
    return v is not None

def default(v, d):
    return v if exists(v) else d

def round_down_mult(n, mult):
    return n // mult * mult

def round_up_mult(n, mult):
    return ceil(n / mult) * mult

def divisible_by(num, den):
    return (num % den) == 0

def is_empty(t):
    return t.numel() == 0

def max_neg_value(t):
    return -torch.finfo(t.dtype).max

def pack_one_with_inverse(t, pattern):
    packed, ps = pack([t], pattern)
    def inverse(out):
        return unpack(out, ps, pattern)[0]

    return packed, inverse
```

```python
# tensor helpers

def pad_at_dim(t, pad, dim = -1, value = 0.):
    dims_from_right = (- dim - 1) if dim < 0 else (t.ndim - dim - 1)
    zeros = ((0, 0) * dims_from_right)
    return F.pad(t, (*zeros, *pad), value = value)

def straight_through(t, target):
    return t + (target - t).detach()

# attend function

def attend(
    q, k, v,
    mask = None,
    return_sim = False,
    scale = None
):
    scale = default(scale, q.shape[-1] ** -0.5)

    q_heads, k_heads = q.shape[1], k.shape[1]
    num_grouped_queries = q_heads // k_heads

    q = rearrange(q, 'b (h qh) ... -> b h qh ...', qh = num_grouped_queries)

    sim = einsum(q, k, 'b h qh i d, b h j d -> b h qh i j') * scale

    mask_value = max_neg_value(sim)

    if exists(mask):
        sim = sim.masked_fill(~mask, mask_value // 10)

    attn = sim.softmax(dim = -1)

    attn_out = einsum(attn, v, 'b h qh i j, b h j d -> b h qh i d')

    attn_out = rearrange(attn_out, 'b h qh ... -> b (h qh) ...')

    if not return_sim:
        return attn_out

    sim = rearrange(sim, 'b h qh ... -> b (h qh) ...')

    return attn_out, sim

# classes

class SparseAttention(Module):
    def __init__(
```

```
    self,
    dim,
    dim_head,
    heads,
    sliding_window_size,
    compress_block_size,
    compress_block_sliding_stride,
    selection_block_size,
    num_selected_blocks,
    kv_heads = None,
    num_compressed_mem_kv = 1,
    causal = False,
    norm = True,
    use_diff_topk = False,
    use_triton_kernel = False,
    query_heads_share_selected_kv = True, # if set to True, importance score is
    ↪ averaged across query heads to select top-n buckets of kv per kv head - but
    ↪ can be set to False for each query head within a group to look at different
    ↪ sets of kv buckets. will be more memory and compute of course
    compress_mlp: Module | None = None,
    compress_mlp_expand_factor = 1.,
    strategy_combine_mlp: Module | None = None
):
    super().__init__()

    # attention heads
    # handling gqa if `kv_heads` is set

    kv_heads = default(kv_heads, heads)
    assert kv_heads <= heads and divisible_by(heads, kv_heads)

    self.heads = heads
    self.dim_head = dim_head
    self.kv_heads = kv_heads
    self.num_grouped_queries = heads // kv_heads

    # scale

    self.scale = dim_head ** -0.5

    dim_inner = dim_head * heads
    dim_kv_inner = dim_head * kv_heads

    self.norm = nn.RMSNorm(dim) if norm else nn.Identity()

    # autoregressive or not - will extend this work for long context video / genomics
    ↪ use-cases

    self.causal = causal
```

```
        # rotary

        self.rotary_emb = RotaryEmbedding(dim_head)

        # qkv

        qkv_split = (dim_inner, dim_kv_inner, dim_kv_inner)

        self.to_qkv = nn.Linear(dim, sum(qkv_split), bias = False)

        self.qkv_split = qkv_split

        # sliding window strategy

        self.sliding_window = LocalAttention(
            dim = dim_head,
            window_size = sliding_window_size,
            causal = causal,
            exact_windowsize = True,
            autopad = True,
            use_rotary_pos_emb = False
        )

        self.sliding_window_size = sliding_window_size

        # compress strategy

        self.compress_block_size = compress_block_size
        self.compress_block_sliding_stride = compress_block_sliding_stride
        assert self.compress_block_size >= self.compress_block_sliding_stride,
        ↪  'compress_block_size must be >= compress_block_sliding_stride'
        assert self.compress_block_sliding_stride > 0, 'compress_block_sliding_stride
        ↪  must be greater than 0'
        assert divisible_by(selection_block_size, self.compress_block_sliding_stride),
        ↪  f'selection_block_size {selection_block_size} must be divisible by
        ↪  compress_block_sliding_stride {self.compress_block_sliding_stride}'

        # Compression window splitting
        self.split_compress_window = nn.Sequential(
            Rearrange('b h n d -> (b h) d 1 n'),
            nn.ZeroPad2d(((compress_block_size - compress_block_sliding_stride), 0, 0,
            ↪  0)),
            nn.Unfold(kernel_size=(1, self.compress_block_size), stride=(1,
            ↪  self.compress_block_sliding_stride)),
            Rearrange('(b h) (d n) w -> b h w n d', d=dim_head, h=kv_heads,
            ↪  n=self.compress_block_size)
        )

        assert num_compressed_mem_kv > 0
        self.num_mem_compress_kv = num_compressed_mem_kv
```

```
self.compress_mem_kv = nn.Parameter(torch.zeros(2, kv_heads,
↪  num_compressed_mem_kv, dim_head))

self.k_intrablock_positions = nn.Parameter(torch.zeros(kv_heads,
↪  self.compress_block_size, dim_head))
self.v_intrablock_positions = nn.Parameter(torch.zeros(kv_heads,
↪  self.compress_block_size, dim_head))

if not exists(compress_mlp):
    compress_dim = self.compress_block_size * dim_head
    compress_mlp_dim_hidden = int(compress_mlp_expand_factor * compress_dim)

    compress_mlp = nn.Sequential(
        Rearrange('b h w n d -> b h w (n d)'),
        nn.Linear(compress_dim, compress_mlp_dim_hidden),
        nn.ReLU(),
        nn.Linear(compress_mlp_dim_hidden, dim_head),
    )

self.k_compress = deepcopy(compress_mlp)
self.v_compress = deepcopy(compress_mlp)

# selection related

self.use_diff_topk = use_diff_topk
self.query_heads_share_selected_kv = query_heads_share_selected_kv
self.selection_block_size = selection_block_size

assert num_selected_blocks >= 0

if num_selected_blocks == 0:
    print(f'`num_selected_blocks` should be set greater than 0, unless if you are
    ↪  ablating it for experimental purposes')

self.num_selected_blocks = num_selected_blocks

self.use_triton_kernel = use_triton_kernel

# they combine the three sparse branches through a learned combine with sigmoid
↪  activation

if not exists(strategy_combine_mlp):
    strategy_combine_mlp = nn.Linear(dim, 3 * heads)

    # init to sliding windows first, as network tends to pick up on local
    ↪  patterns first before distant ones

    nn.init.zeros_(strategy_combine_mlp.weight)
    strategy_combine_mlp.bias.data.copy_(tensor([-2., -2., 2.] * heads))
```

```
        self.to_strategy_combine = nn.Sequential(
            strategy_combine_mlp,
            nn.Sigmoid(),
            Rearrange('b n (h s) -> b h n s', h = heads)
        )

        # split and merging heads

        self.split_heads = Rearrange('b n (h d) -> b h n d', d = dim_head)
        self.merge_heads = Rearrange('b h n d -> b n (h d)')

        # combining heads

        self.combine_heads = nn.Linear(dim_inner, dim, bias = False)

    def forward_inference(
        self,
        inp,
        cache,
        return_cache = True
    ):
        assert self.causal, 'inference only relevant for autoregressive'

        # destruct cache

        (
            (cache_k, cache_v),
            (
                (cache_ck, cache_cv),
                (run_k, run_v)
            )
         ) = cache

        # variables

        batch, scale, heads, device = inp.shape[0], self.scale, self.heads, inp.device
        cache_len = cache_k.shape[-2]
        seq_len = cache_len + 1

        sliding_window = self.sliding_window_size

        fine_divisible_seq_len = round_up_mult(seq_len, self.selection_block_size)
        num_fine_blocks = fine_divisible_seq_len // self.selection_block_size

        # maybe prenorm

        inp = self.norm(inp)

        # queries, keys, values
```

```
q, k, v = self.to_qkv(inp).split(self.qkv_split, dim = -1)

q, k, v = map(self.split_heads, (q, k, v))

# take care of running k and v for compression, which should NOT be rotated
↪  https://arxiv.org/abs/2501.18795

run_k = cat((run_k, k), dim = -2)
run_v = cat((run_v, v), dim = -2)

# rotate after updating the compression running k/v

rotated_q = self.rotary_emb.rotate_queries_or_keys(q, offset = cache_len)
k = self.rotary_emb.rotate_queries_or_keys(k, offset = cache_len)

# handle cache, which stores the rotated

k = cat((cache_k, k), dim = -2)
v = cat((cache_v, v), dim = -2)

if return_cache:
    cache_kv = (k, v)

# 1. compressed attn inference

cq = q
ck = cache_ck
cv = cache_cv

ck_for_attn = cache_ck
cv_for_attn = cache_cv

if not is_empty(ck):
    mem_ck, mem_cv = repeat(self.compress_mem_kv, 'kv ... -> kv b ...', b =
    ↪  batch)

    ck_for_attn = cat((mem_ck, ck_for_attn), dim = -2)
    cv_for_attn = cat((mem_cv, cv_for_attn), dim = -2)

repeated_ck = repeat(ck_for_attn, 'b h ... -> b (h gh) ...', gh =
↪  self.num_grouped_queries)
repeated_cv = repeat(cv_for_attn, 'b h ... -> b (h gh) ...', gh =
↪  self.num_grouped_queries)

csim = einsum(q, repeated_ck, 'b h i d, b h j d -> b h i j') * scale
cattn = csim.softmax(dim = -1)

compressed_attn_out = einsum(cattn, repeated_cv, 'b h i j, b h j d -> b h i d')

running_compress_seq_len = run_k.shape[-2]
```

```
if divisible_by(running_compress_seq_len, self.compress_block_size):
    k_compress_input = rearrange(run_k, 'b h n d -> b h 1 n d')
    v_compress_input = rearrange(run_v, 'b h n d -> b h 1 n d')

    k_compress_input = einx.add('b h w n d, h n d', k_compress_input,
    ↪  self.k_intrablock_positions)
    v_compress_input = einx.add('b h w n d, h n d', v_compress_input,
    ↪  self.v_intrablock_positions)

    next_ck = self.k_compress(k_compress_input)
    next_cv = self.v_compress(v_compress_input)

    compress_overlap_len = self.compress_block_size -
    ↪  self.compress_block_sliding_stride
    run_kv_slice = slice(-compress_overlap_len, None) if compress_overlap_len > 0
    ↪  else slice(0, 0)

    run_k = run_k[..., run_kv_slice, :]
    run_v = run_v[..., run_kv_slice, :]

    ck = cat((ck, next_ck), dim = -2)
    cv = cat((cv, next_cv), dim = -2)

if return_cache:
    cache_compressed_kv = ((ck, cv), (run_k, run_v))

# 2. fine attention inference

importance_scores = csim[..., self.num_mem_compress_kv:]

num_compress_blocks = importance_scores.shape[-1]
num_compress_per_fine = self.selection_block_size //
↪  self.compress_block_sliding_stride

if self.compress_block_sliding_stride != self.selection_block_size:
    compress_seq_len = round_down_mult(num_compress_blocks,
    ↪  num_compress_per_fine)
    importance_scores = importance_scores[..., :compress_seq_len]
    importance_scores = reduce(importance_scores, '... (j num_compress_per_fine)
    ↪  -> ... j', 'mean', num_compress_per_fine = num_compress_per_fine)

num_fine_blocks = importance_scores.shape[-1]
num_selected = min(self.num_selected_blocks, num_fine_blocks)
has_selected_kv_for_fine_attn = num_selected > 0

# block causal diagonal

fine_sliding_window = ((seq_len - 1) % self.selection_block_size) + 1
fk = k[..., -fine_sliding_window:, :]
```

```
fv = v[..., -fine_sliding_window:, :]

# select out the sparse kv segments as defined by compressed attention map as
↪  importance score

fmask = None

if has_selected_kv_for_fine_attn:
    if self.query_heads_share_selected_kv:
        importance_scores = reduce(importance_scores, 'b (h grouped_queries) ...
        ↪  -> b h ...', 'mean', grouped_queries = self.num_grouped_queries)

    importance_scores = F.pad(importance_scores, (1, 0), value = -1e3)
    importance_scores = importance_scores.softmax(dim = -1)
    importance_scores = importance_scores[..., 1:]

    sel_scores, sel_indices = importance_scores.topk(num_selected, dim = -1)

    fine_divisible_seq_len = round_up_mult(seq_len, self.selection_block_size)
    remainder = fine_divisible_seq_len - k.shape[-2]

    sel_fk = pad_at_dim(k, (0, remainder), dim = -2)
    sel_fv = pad_at_dim(v, (0, remainder), dim = -2)

    sel_fk = rearrange(sel_fk, 'b h (w j) d -> b h w j d', j =
    ↪  self.selection_block_size)
    sel_fv = rearrange(sel_fv, 'b h (w j) d -> b h w j d', j =
    ↪  self.selection_block_size)

    # get_at('b h [w] j d, b h 1 sel -> b h (sel j) d'

    sel_indices = repeat(sel_indices, 'b h 1 sel -> b h sel j d', j =
    ↪  self.selection_block_size, d = sel_fk.shape[-1])

    sel_fk = sel_fk.gather(2, sel_indices)
    sel_fv = sel_fv.gather(2, sel_indices)

    sel_fk, sel_fv = tuple(rearrange(t, 'b h sel j d -> b h (sel j) d') for t in
    ↪  (sel_fk, sel_fv))

    fmask = sel_scores > 1e-10

    fmask = repeat(fmask, 'b h i sel -> b h i (sel j)', j =
    ↪  self.selection_block_size)

    fk = cat((sel_fk, fk), dim = -2)
    fv = cat((sel_fv, fv), dim = -2)

    fmask = F.pad(fmask, (0, fk.shape[-2] - fmask.shape[-1]), value = True)
```

```
        # remove later

        fq = rearrange(rotated_q, 'b (h gh) ... -> b h gh ...', gh =
        ↪  self.num_grouped_queries)

        fsim = einsum(fq, fk, 'b h gh i d, b h j d -> b h gh i j') * scale

        if exists(fmask):
            fsim = einx.where('b h i j, b h gh i j, -> b h gh i j', fmask, fsim,
            ↪  max_neg_value(fsim))

        fattn = fsim.softmax(dim = -1)

        fine_attn_out = einsum(fattn, fv, 'b h gh i j, b h j d -> b h gh i d')
        fine_attn_out = rearrange(fine_attn_out, 'b h gh ... -> b (h gh) ...')

        # 3. sliding window

        k = repeat(k, 'b h ... -> b (h gh) ...', gh = self.num_grouped_queries)
        v = repeat(v, 'b h ... -> b (h gh) ...', gh = self.num_grouped_queries)

        sliding_slice = (Ellipsis, slice(-(sliding_window + 1), None), slice(None))

        k, v  = k[sliding_slice], v[sliding_slice]

        sim = einsum(rotated_q, k, 'b h i d, b h j d -> b h i j') * scale
        attn = sim.softmax(dim = -1)
        sliding_window_attn_out = einsum(attn, v, 'b h i j, b h j d -> b h i d')

        # combine strategies

        strategy_weighted_combine = self.to_strategy_combine(inp)

        out = einsum(strategy_weighted_combine, stack([compressed_attn_out,
        ↪  fine_attn_out, sliding_window_attn_out]), 'b h n s, s b h n d -> b h n d')

        # merge heads and combine them

        out = self.merge_heads(out)

        out = self.combine_heads(out)

        if not return_cache:
            return out

        return out, (cache_kv, cache_compressed_kv)

    def forward(
        self,
        inp,
```

```
    cache = None,
    disable_triton_kernel = False,
    sliding_window_flex_mask = None,
    fine_selection_flex_mask = None,
    return_cache = False
):
    is_inferencing = exists(cache)

    if is_inferencing:
        assert inp.shape[1] == 1, 'input must be single tokens if inferencing with
        ↪  cache key values'
        return self.forward_inference(inp, cache, return_cache = return_cache)

    assert not (not self.causal and return_cache)

    batch, seq_len, scale, heads, kv_heads, device = *inp.shape[:2], self.scale,
    ↪  self.heads, self.kv_heads, inp.device

    compress_divisible_seq_len = round_down_mult(seq_len,
    ↪  self.compress_block_sliding_stride)
    num_compress_blocks = compress_divisible_seq_len //
    ↪  self.compress_block_sliding_stride

    compress_overlap_len = self.compress_block_size -
    ↪  self.compress_block_sliding_stride
    has_compress_overlap = compress_overlap_len > 0

    fine_divisible_seq_len = round_up_mult(seq_len, self.selection_block_size)
    num_fine_blocks = fine_divisible_seq_len // self.selection_block_size

    # maybe prenorm

    inp = self.norm(inp)

    # queries, keys, values

    q, k, v = self.to_qkv(inp).split(self.qkv_split, dim = -1)

    q, k, v = map(self.split_heads, (q, k, v))

    # compressed key / values - variables prepended with `c` stands for compressed

    k_compress_input, v_compress_input = k[..., :compress_divisible_seq_len, :],
    ↪  v[..., :compress_divisible_seq_len, :]

    if not is_empty(k_compress_input):
        k_compress_input = self.split_compress_window(k_compress_input)
        v_compress_input = self.split_compress_window(v_compress_input)
    else:
```

```
        k_compress_input, v_compress_input = tuple(t.reshape(batch, kv_heads, 0,
        ↪  self.compress_block_size, self.dim_head) for t in (k_compress_input,
        ↪  v_compress_input))

    # add the intra block positions

    if not is_empty(k_compress_input):
        k_compress_input = einx.add('b h w n d, h n d', k_compress_input,
        ↪  self.k_intrablock_positions)
        v_compress_input = einx.add('b h w n d, h n d', v_compress_input,
        ↪  self.v_intrablock_positions)

    run_k, run_v = k, v

    if return_cache and has_compress_overlap:
        run_k = pad_at_dim(run_k, (compress_overlap_len, 0), value = 0., dim = -2)
        run_v = pad_at_dim(run_v, (compress_overlap_len, 0), value = 0., dim = -2)

    run_k = run_k[..., compress_divisible_seq_len:, :]
    run_v = run_v[..., compress_divisible_seq_len:, :]

    cq = q
    ck = self.k_compress(k_compress_input)   # Equation (7) of the Native Sparse
    ↪  Attention paper
    cv = self.v_compress(v_compress_input)

    if return_cache:
        cache_compressed_kv = ((ck, cv), (run_k, run_v))

    # 1. coarse attention over compressed

    mem_ck, mem_cv = repeat(self.compress_mem_kv, 'kv ... -> kv b ...', b = batch)

    num_mem_compress_kv = mem_ck.shape[-2]

    ck = cat((mem_ck, ck), dim = -2)
    cv = cat((mem_cv, cv), dim = -2)

    # compressed masking

    cmask = None

    if self.causal:
        cq_seq = arange(seq_len, device = device)
        ck_seq = ((arange(num_compress_blocks, device = device) + 1) *
        ↪  self.compress_block_sliding_stride) - 1
        ck_seq = F.pad(ck_seq, (num_mem_compress_kv, 0), value = -1)

        cmask = einx.less('j, i -> i j', ck_seq, cq_seq)
```

```
compressed_attn_out, csim = attend(cq, ck, cv, mask = cmask, return_sim = True)

# for 2. and 3., will give them relative positions with rotary - compressed needs
↪ to be handled separately (even if they already have intra block absolute
↪ positions)

q, k = self.rotary_emb.rotate_queries_with_cached_keys(q, k)

# handle cache

if return_cache:
    cache_kv = (k, v)

# 2. fine attention over selected based on compressed attention logits -
↪ variables prepended with `f` stands for the fine attention pathway

importance_scores = csim[..., num_mem_compress_kv:]

num_selected = min(self.num_selected_blocks, num_compress_blocks)
has_selected_kv_for_fine_attn = num_selected > 0

# maybe average the compressed attention across each grouped queries (per key /
↪ values)

if self.query_heads_share_selected_kv:
    importance_scores = reduce(importance_scores, 'b (h grouped_queries) ... -> b
    ↪ h ...', 'mean', grouped_queries = self.num_grouped_queries)

    fine_num_grouped_queries = self.num_grouped_queries
else:
    fine_num_grouped_queries = 1

# handle if compress block size does not equal to the fine block size
# cannot parse their equation, so will just improvise
# first we expand all the compressed scores to the full sequence length, then
↪ average within each fine / selection block size - pad on the right to 0s,
↪ which should be fine as sliding window convers the local anyways

if has_selected_kv_for_fine_attn:

    if self.compress_block_sliding_stride != self.selection_block_size:

        num_compress_per_fine = self.selection_block_size //
        ↪ self.compress_block_sliding_stride

        round_down_score_len = round_down_mult(importance_scores.shape[-1],
        ↪ num_compress_per_fine)
        importance_scores = importance_scores[..., :round_down_score_len]
```

```
        if not is_empty(importance_scores):
            importance_scores = reduce(importance_scores, '... (j
            ↪ num_compress_per_fine) -> ... j', 'mean', num_compress_per_fine =
            ↪ num_compress_per_fine)

            i, j = importance_scores.shape[-2:]

            # mask out block diagonal

            q_seq = arange(i, device = device) // self.selection_block_size
            k_seq = arange(j, device = device)

            block_diagonal_mask = einx.equal('i, j -> i j', q_seq, k_seq)

            importance_scores =
            ↪ importance_scores.masked_fill(block_diagonal_mask,
            ↪ max_neg_value(csim))

    importance_scores = F.pad(importance_scores, (1, 0), value = -1e3)
    importance_scores = importance_scores.softmax(dim = -1)
    importance_scores = importance_scores[..., 1:]

# handle if number of total blocks is less than number to select for fine
↪ attention

fq = q
fk = k
fv = v

num_selected = min(num_selected, importance_scores.shape[-1])
has_selected_kv_for_fine_attn = num_selected > 0

remainder = fine_divisible_seq_len - seq_len
pad_to_multiple = partial(pad_at_dim, pad = (0, remainder), dim = -2)

if has_selected_kv_for_fine_attn:

    # get the top-n kv segments for fine attention

    selected_importance_values, selected_block_indices =
    ↪ importance_scores.topk(num_selected, dim = -1)

    gates = straight_through(selected_importance_values, 1.) if
    ↪ self.use_diff_topk else None

    if self.use_triton_kernel and not disable_triton_kernel:

        from native_sparse_attention_pytorch.triton_native_sparse_attention
        ↪ import native_sparse_attend
```

```
        fmask = selected_importance_values > 1e-10

        fine_attn_out = native_sparse_attend(
            fq, fk, fv,
            self.selection_block_size,
            selected_block_indices,
            fmask,
            sel_scale = gates,
            include_block_causal = self.causal
        )

    elif exists(fine_selection_flex_mask):
        assert not self.use_diff_topk, 'differential topk is not available for
        ↪  flex attention'

        # flex attention for the selection for fine attention

        fine_block_mask = fine_selection_flex_mask(selected_block_indices,
        ↪  num_grouped_queries = fine_num_grouped_queries)

        fine_attn_out = flex_attention(fq, fk, fv, block_mask = fine_block_mask,
        ↪  enable_gqa = True)

    else:
        fmask = selected_importance_values > 1e-10

        if seq_len < fine_divisible_seq_len:
            fk, fv, fq = map(pad_to_multiple, (fk, fv, fq))

            fmask = pad_at_dim(fmask, (0, remainder), value = False, dim = -2)

            selected_block_indices = pad_at_dim(selected_block_indices, (0,
            ↪  remainder), value = 0, dim = -2)

            if exists(gates):
                gates = pad_at_dim(gates, (0, remainder), value = 0, dim = -2)

        if self.causal:
            # handle block causal diagonal in the diagram, but run experiments
            ↪  without to see

            fine_window_seq = arange(fine_divisible_seq_len, device = device) //
            ↪  self.selection_block_size
            fine_window_seq = repeat(fine_window_seq, 'n -> b h n 1', b = batch,
            ↪  h = selected_block_indices.shape[1])
            selected_block_indices = cat((selected_block_indices,
            ↪  fine_window_seq), dim = -1) # for the block causal diagonal in
            ↪  fig2
```

```
        fmask = repeat(fmask, 'b h i w -> b h i w j', j =
        ↪  self.selection_block_size)

        causal_mask = torch.ones((self.selection_block_size,) * 2, device =
        ↪  device, dtype = torch.bool).tril()
        causal_mask = repeat(causal_mask, 'i j -> b h (w i) 1 j', w =
        ↪  num_fine_blocks, b = batch, h = fmask.shape[1])

        fmask = cat((fmask, causal_mask), dim = -2)
        fmask = rearrange(fmask, 'b h i w j -> b h 1 i (w j)')

    else:
        fmask = repeat(fmask, 'b h i w -> b h 1 i (w j)', j =
        ↪  self.selection_block_size)

    # select out the spatial crops of keys / values for fine attention

    fk = rearrange(fk, 'b h (w n) d -> b h w n d', w = num_fine_blocks)
    fv = rearrange(fv, 'b h (w n) d -> b h w n d', w = num_fine_blocks)

    # get_at("b h [w] j d, b h i selected -> b h i selected j d", fkv,
    ↪  selected_block_indices)

    if self.query_heads_share_selected_kv:
        fk = repeat(fk, 'b h w j d -> b h i w j d', i =
        ↪  selected_block_indices.shape[2])
        fv = repeat(fv, 'b h w j d -> b h i w j d', i =
        ↪  selected_block_indices.shape[2])
    else:
        fk = repeat(fk, 'b h w j d -> b (h qh) i w j d', i =
        ↪  selected_block_indices.shape[2], qh = self.num_grouped_queries)
        fv = repeat(fv, 'b h w j d -> b (h qh) i w j d', i =
        ↪  selected_block_indices.shape[2], qh = self.num_grouped_queries)

    selected_block_indices = repeat(selected_block_indices, 'b h i sel -> b h
    ↪  i sel j d', j = fk.shape[-2], d = fk.shape[-1])

    fk = fk.gather(3, selected_block_indices)
    fv = fv.gather(3, selected_block_indices)

    # differential topk gating

    if self.use_diff_topk:
        if self.causal:
            gates = F.pad(gates, (0, 1), value = 1.)

        fk = einx.multiply('b h i sel, b h i sel j d -> b h i sel j d',
        ↪  gates, fk)
```

```
        # merge selected key values

        fk, fv = tuple(rearrange(t, 'b h i w j d -> b h i (w j) d') for t in (fk,
        ↪  fv))

        # fine attention

        fq = rearrange(fq, 'b (h qh) ... -> b h qh ...', qh =
        ↪  fine_num_grouped_queries)

        fsim = einsum(fq, fk, 'b h qh i d, b h i j d -> b h qh i j') * self.scale

        mask_value = max_neg_value(fsim)

        fsim = fsim.masked_fill(~fmask, mask_value)

        fattn = fsim.softmax(dim = -1)

        fine_attn_out = einsum(fattn, fv, 'b h qh i j, b h i j d -> b h qh i d')

        fine_attn_out = rearrange(fine_attn_out, 'b h qh ... -> b (h qh) ...')

        fine_attn_out = fine_attn_out[..., :seq_len, :]

else:
    # if only first block, just do a simple block causal

    seq_len = fk.shape[-2]
    fmask = None

    fk, fv, fq = map(pad_to_multiple, (fk, fv, fq))

    fq, fk, fv = tuple(rearrange(t, 'b h (w n) d -> (b w) h n d', n =
    ↪  self.selection_block_size) for t in (fq, fk, fv))

    if self.causal:
        fmask = causal_mask = torch.ones((self.selection_block_size,
        ↪  self.selection_block_size), device = device, dtype =
        ↪  torch.bool).tril()

    fine_attn_out = attend(fq, fk, fv, mask = fmask)

    fine_attn_out = rearrange(fine_attn_out, '(b w) h n d -> b h (w n) d', b =
    ↪  batch)
    fine_attn_out = fine_attn_out[..., :seq_len, :]

# 3. overlapping sliding window, this is unsurprising and expected - `s` for
↪  sliding
```

```
sq = q
sk = k
sv = v

if exists(sliding_window_flex_mask):
    sliding_window_attn_out = flex_attention(sq, sk, sv, block_mask =
    ↪  sliding_window_flex_mask, enable_gqa = True)
else:
    sk, sv = tuple(repeat(t, 'b h ... -> b (h num_grouped_queries) ...',
    ↪  num_grouped_queries = self.num_grouped_queries) for t in (sk, sv))

    sliding_window_attn_out = self.sliding_window(sq, sk, sv)

# combine strategies

strategy_weighted_combine = self.to_strategy_combine(inp)

out = einsum(strategy_weighted_combine, stack([compressed_attn_out,
↪  fine_attn_out, sliding_window_attn_out]), 'b h n s, s b h n d -> b h n d')

# merge heads and combine them

out = self.merge_heads(out)

out = self.combine_heads(out)

if not return_cache:
    return out

return out, (cache_kv, cache_compressed_kv)
```