Gemini based Advanced RAG App

Abhinav Deshpande IMT2022580, Valmik Belgaonkar IMT2022020, Aditya Joshi IMT2022092
Advances in NLP
IIITB

I. Technical Documentation: BERT Encoder with MoE and Bayesian Optimization (5-Fold CV)

Configuration and Hyperparameter Settings

Key settings and fixed parameters:

- **Model Size (Fixed):** `HIDDEN_SIZE` $= 768$.
- **BO Iterations:** `BO_ITERATIONS` $= \mathbf{10}$.
- **Epochs per Fold:** `TRIAL_EPOCHS` $= \mathbf{3}$.
- **Cross-Validation Folds:** `K_FOLDS` $= \mathbf{5}$ (used within the objective function).

Model Architecture and Embedding

Input Representation: The final token representation $\mathbf{x}$ passed to the first Transformer layer is the element-wise sum of three learned embeddings, followed by Layer Normalization and Dropout:

$$\mathbf{x} = \text{Dropout}\left(\text{LayerNorm}\left(E_{\text{token}} + E_{\text{position}} + E_{\text{segment}}\right)\right)$$

- $\mathbf{E}_{\text{token}}$: Vector embeddings, initialized using `Word2Vec` (CBOW mode, `sg` $= 0$).
- $\mathbf{E}_{\text{position}}$: Learned positional encodings based on token index.
- $\mathbf{E}_{\text{segment}}$: Learned segment embeddings (0 for Sentence A, 1 for Sentence B).

Mixture-of-Experts (MoE) Module: The standard dense FFN in each Transformer layer is replaced by the MoE module (`MoE`).

- **Top-K Routing and Noise:** The MoE module computes expert logits $\mathbf{l} \in \mathbb{R}^E$ via a linear router. During training (`self.training` is true), Gaussian noise is added to promote expert diversity:

$$\mathbf{l}_{\text{noisy}} = \mathbf{l} + \mathbf{n} \cdot \texttt{noise\_std},$$

$$\text{where } \mathbf{n} \sim \mathcal{N}(0,1) \text{ and } \texttt{noise\_std} = 1.0$$

The top K expert indices $\mathbf{i}_k$ and corresponding logits $\mathbf{v}_k$ are selected from $\mathbf{l}_{\text{noisy}}$. The final gating weights $\mathbf{G}_k$ are obtained via softmax over the top-K values:

$$\mathbf{G}_k = \text{Softmax}(\mathbf{v}_k)$$

The module output $\mathbf{y}$ is the weighted sum of the outputs of the K selected experts:

$$\mathbf{y}_{b,s} = \sum_{e \in \mathbf{i}_k[b,s]} \mathbf{G}_k[b,s,e] \cdot \text{Expert}_e(\mathbf{x}_{b,s})$$

- **Auxiliary Load-Balancing Loss($\mathcal{L}_{\text{aux}}$):** This loss encourages equal utilization of all E experts, calculated using the full softmax over the noiseless logits $\mathbf{p} = \text{Softmax}(\mathbf{l})$:

$$\mathcal{L}_{\text{aux}} = E \cdot \sum_{e=1}^{E} \left(\frac{\sum_{b,s} \mathbf{p}_{b,s,e}}{B \cdot S} \right)^2$$

B is batch size, S is sequence length. This term is added to the total loss as a regularization term.

Pre-training Objectives and Total Loss

The model is pre-trained using Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). The combined loss is:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{MLM}} + \mathcal{L}_{\text{NSP}} + \alpha \cdot \mathcal{L}_{\text{aux}}$$

The auxiliary loss coefficient α (`aux_coeff`) is fixed at $\mathbf{0.01}$.

Masked Language Modeling Loss ($\mathcal{L}_{MLM}$):

- **Task:** Predict original tokens that were masked (MLM masking procedure uses 80/10/10 split on the 15% masked tokens).
- **Loss:** Standard Cross-Entropy Loss (`nn.CrossEntropyLoss`), ignoring padding and unmasked positions (label = -100).

Next Sentence Prediction Loss ($\mathcal{L}_{NSP}$):

- **Task:** Binary classification (IsNext: 1, NotNext: 0) based on the representation of the `[CLS]` token.
- **Loss:** Standard Cross-Entropy Loss (`nn.CrossEntropyLoss`).
- **Conditional Use:** Only calculated for 'C' (Chunk) type batches where sentence pairs are provided.

Bayesian Optimization with K-Fold Cross-Validation

The script uses a sophisticated BO scheme where the evaluation of each hyperparameter configuration relies on 5-Fold Cross-Validation.

Hyperparameters and Search Spaces: The BO tunes 9 hyperparameters (8 architecture/training, plus batch size), all mapped to a continuous vector via `hyperparams_to_vector`:

The K-Fold Cross-Validation Process (Evaluation of one BO trial): The `run_bayesian_optimization_with_heldout_test_cv5` function first performs a single split of the entire dataset into **Train**, **Validation**, and **Test** sets. The **Test** set is held-out and never used in BO. The $\mathbf{Train + Validation}$

Hyperparameter	Code Key	Search Space (Tuning Range)	Type
Learning Rate	learning_rate	$[10^{-6}, 5\times 10^{-5}]$	Log-Uniform
MoE Experts	moe_experts	$\{3, 4, 5, 6\}$	Discrete Set
MoE Top-K	moe_k	$\{1, 2\}$	Discrete Set
FFN Inner Dimension	ffn_dim	$\{1024, 1280, \ldots, 4096\}$ (mult. of 256)	Discrete Set
Number of Layers	num_layers	$\{6, 8, 12\}$	Discrete Set
Number of Heads	num_heads	$\{6, 8, 12\}$	Discrete Set
Word2Vec Window Size	word2vec_window	$\{3, 5, 7\}$	Discrete Set
MLM Mask Probability	mlm_mask_prob	$[0.10, 0.20]$	Uniform
Batch Size	batch_size	$\{4, 8, 16\}$	Discrete Set

TABLE I: Hyperparameter Search Space

sets are combined into a pool (`combined_indices`) for CV.

The **objective_function_with_kfold** performs the 5-Fold CV on this pool:

1) The combined pool is split into $K = 5$ non-overlapping folds using `sklearn.model_selection.KFold`.
2) For $k = 1$ to 5:
   - A model is initialized from scratch.
   - The model is trained on $K - 1 = 4$ folds for `TRIAL_EPOCHS` $= 3$.
   - The total loss ($\mathcal{L}_{\text{total}}$) is computed on the single held-out validation fold.
3) The final score reported to the BO is the **average validation total loss across all 5 folds**.

This process provides a statistically more robust objective score than a single train/val split.

Bayesian Optimization (GP-EI):

- **Surrogate Model:** Gaussian Process Regressor (`GaussianProcessRegressor`), using a **Matern** kernel ($\nu = 2.5$) for flexibility and robustness, combined with a **WhiteKernel** for observation noise.
- **Acquisition Function:** Expected Improvement (EI), which is maximized to select the next hyperparameter vector $\mathbf{x}_{\text{next}}$ to test.

$$EI(\mathbf{x}) = (f_{\text{best}} - \mu - \xi)\Phi(Z) + \sigma\phi(Z)$$

Where f_{best} is the best recorded loss so far, μ and σ are the GP's predicted mean and standard deviation, and $\xi = 0.01$ controls the trade-off between exploitation and exploration.

Final Evaluation: After **10** BO iterations, the best hyperparameter set is chosen (minimizing the average CV loss). The final model is retrained on the entire **Train + Validation** pool (using the best parameters) for 3 epochs, and then its final performance metrics (MLM Accuracy, NSP Accuracy, and Loss) are calculated and reported on the completely **held-out Test set**.

Masking and Pooling Strategies and used:

Masking strategy:: First we select 15% of the tokens to be mask candidates. Then out of these 80% are replaced with [MASK], 10% are replaced with a random token and 10% are left unchanged.

Pooling strategy:: The pooling strategy used at the end to represent N * d matrix into a 1 * d vector embedding is mask aware mean pooling. CLS token pooling may be used but I searched online: For downstream RAG tasks, mask aware mean pooling is generally considered better than CLS token pooling.

II. Technical Documentation: LoRA Fine-Tuning of BERT Encoder for Contrastive Retrieval

Overview and Objective

This script implements **Low-Rank Adaptation (LoRA)** to efficiently fine-tune a pre-trained BERT Encoder model for a **Contrastive Learning** task, specifically intended for improving retrieval performance (Query-Chunk matching). The core objective is to move query embeddings closer to their corresponding positive chunk embeddings and farther from sampled negative chunk embeddings in the embedding space.

Key Components:

- **Base Model:** A custom implementation of the BERT Encoder architecture.
- **Fine-tuning Method:** LoRA, applied only to the attention mechanism's linear projection layers (Q, K, V).
- **Data Source:** A `ContrastiveDataset` loading (query, positive_ID) pairs from a CSV, and retrieving chunk documents from a persistent `ChromaDB` vector store.
- **Loss Function:** A custom Contrastive Loss based on cosine similarity and softmax normalization.

BERT Encoder Architecture

The model (`BertEncoderModel`) is a standard Transformer encoder:

$$\mathbf{x} = \text{Dropout}\left(\text{LayerNorm}\left(E_{\text{token}} + E_{\text{position}} + E_{\text{segment}}\right)\right)$$

The `encode` method returns the final embedding of the [**CLS**] token, which serves as the fixed-length representation (embedding vector) for the entire input sequence (Query or Chunk).

Low-Rank Adaptation (LoRA)

LoRA is a parameter-efficient technique used here to update the model weights without modifying the pre-trained weights.

LoRA Module (`LoRALinear`): The `LoRALinear` module replaces a base linear layer W_0 with a combination of the frozen base weights and a low-rank decomposition matrix (BA), where $A \in \mathbb{R}^{d\times r}$ and $B \in \mathbb{R}^{r\times d}$.

Let $W_0 \in \mathbb{R}^{d\times d}$ be the base weight matrix (e.g., the Query projection matrix). The forward pass is defined as:

$$h = W_0 x + \Delta W x = W_0 x + BAx$$

In the code, the forward computation for input x is:

$$\text{Output} = \text{base}(x) + \text{Dropout}(\text{lora\_B}(\text{lora\_A}(x))) \cdot \frac{\alpha}{r}$$

- **Rank (r):** `r` $= 8$. This defines the size of the bottleneck layer (the rank of ΔW).
- **Scaling (α):** `alpha` $= 8$. The output of the low-rank path is scaled by $\frac{\alpha}{r}$. In this case, $\frac{8}{8} = 1$.
- **Training:** The base layer weights W_0 are frozen (`requires_grad=False`), and only the low-rank matrices `lora_A` and `lora_B` are trained.

Application to BERT: The `apply_lora` function wraps the linear layers responsible for the Query (**Q**), Key (**K**), and Value (**V**) projections within the `nn.MultiheadAttention` module in every Transformer layer. This targets the most critical components for adapting semantic understanding.

Contrastive Dataset and Negative Sampling

Data Source and Ratios: The `ContrastiveDataset` ensures that for every query, there is exactly one positive chunk and multiple negative chunks.

- **Positive Pair:** A query and its corresponding correct chunk document (1 positive chunk per query).
- **Negative Sampling (K):** The `sample_negatives` function samples $\mathbf{k = 5}$ negative chunk IDs for each query.
- **Ratio:** The evaluation uses $\mathbf{1:5}$ ratio (1 positive document for 5 negative documents) per query in the loss calculation.

Data Loading: The `collate` function processes the batch:

- It tokenizes the query, positive chunk, and all 5 negative chunks using the simple whitespace `tokenize` function.
- All inputs are padded to the maximum sequence length found in the current batch.
- The final tensor shape for the negative chunks is (B, K, L), where B is batch size, $K = 5$ is the number of negatives, and L is sequence length. This is flattened to $(B \cdot K, L)$ before being passed to the `model.encode`.

Contrastive Loss Function

The `contrastive_loss` function implements a softmax-normalized loss (often equivalent to InfoNCE loss) which maximizes the similarity between the query and the positive chunk relative to the similarities with all negative chunks.

Similarity Calculation: The similarity metric used is the **Cosine Similarity**. Let $\mathbf{q}$ be the query embedding, $\mathbf{p}$ be the positive chunk embedding, and $\mathbf{n}_i$ be the i-th negative chunk embedding. All are vectors in $\mathbb{R}^H$.

- **Positive Similarity (Scalar):** $s_{\text{pos}} = \cos(\mathbf{q}, \mathbf{p})$
- **Negative Similarities (Vector):** $s_{\text{neg},i} = \cos(\mathbf{q}, \mathbf{n}_i)$

The code concatenates these into a similarity vector $\mathbf{S} = [s_{\text{pos}}, s_{\text{neg},1}, \ldots, s_{\text{neg},5}]$, resulting in a shape of $(B, 6)$.

Softmax and Loss (InfoNCE): The loss assumes that the positive pair (s_{pos}) should have the highest similarity. The probability that the positive chunk is the correct match is calculated using softmax over all similarities:

$$P(\text{pos}) = \frac{e^{s_{\text{pos}}}}{\sum_{j=0}^{5} e^{s_j}} \quad (\text{where } s_0 = s_{\text{pos}}, s_{j>0} = s_{\text{neg},j})$$

The objective is to maximize $P(\text{pos})$, which is equivalent to minimizing the negative log-probability:

$$\mathcal{L}_{\text{contrastive}} = -\log(P(\text{pos}))$$

The final loss is the mean of this quantity across the batch:

$$\mathcal{L} = -\frac{1}{B}\sum_{b=1}^{B} \log\left(\frac{e^{\cos(\mathbf{q}_b, \mathbf{p}_b)}}{\sum_{j=1}^{5} e^{\cos(\mathbf{q}_b, \mathbf{n}_{b,j})} + e^{\cos(\mathbf{q}_b, \mathbf{p}_b)}}\right)$$

This is implemented in the code using `-torch.log(probs[:, 0]).mean()`, where `probs[:, 0]` corresponds to $P(\text{pos})$.

III. Technical Documentation: LoRA Fine Tuning of Cross-Encoder Reranker with Grid Search and 10-Fold CV

Overview and Objective

This script implements a parameter-efficient fine-tuning approach for a **Cross-Encoder Reranker** using **Low-Rank Adaptation (LoRA)**. The fine-tuning hyperparameters are optimized using a comprehensive **Grid Search** evaluated via **10-Fold Cross-Validation (CV)**.

Cross-Encoder Model Architecture: The `CrossEncoderLoRA` wraps a HuggingFace `AutoModel` (the base encoder) and attaches a custom classification head.

- **Input:** Concatenated query and chunk (`[CLS]` Query `[SEP]` Chunk `[SEP]`).
- **Output Logit (z):** The `[CLS]` token embedding (h_{CLS}) from the last layer is used:

$$z = \text{Classifier}(\text{Dropout}(h_{\text{CLS}})) \in \mathbb{R}^1$$

Loss Function and Metric:

- **Training Loss:** `nn.BCEWithLogitsLoss` (Binary Cross-Entropy with Logits), which stabilizes classification training by combining the sigmoid and BCE loss:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} [y_i \log(\sigma(z_i)) + (1 - y_i) \log(1 - \sigma(z_i))]$$

- **Primary Evaluation Metric (CV):** Area Under the Receiver Operating Characteristic Curve (**AUC**), used to select the best hyperparameter configuration.

LoRA Fine-Tuning Mathematical Details

The **Low-Rank Adaptation (LoRA)** method replaces the update to a pretrained weight matrix W_0 with a low-rank matrix decomposition $\Delta W = BA$.

LoRA Module (`LoRALinear`): The `LoRALinear` class replaces any `nn.Linear` layer W_0 in the encoder.

- **Effective Weight Matrix:**

$$W_{\text{eff}} = W_{\text{base}} + \Delta W, \quad \text{where } \Delta W = BA \cdot \frac{\alpha}{r}$$

- **Parameters:**
  - W_{base}: Original weight (`self.weight`), **frozen** (`requires_grad=False`).
  - A: Down-projection matrix $(\mathbf{r}, d_{\text{in}})$, **trainable** (`self.A`).
  - B: Up-projection matrix $(d_{\text{out}}, \mathbf{r})$, **trainable** (`self.B`).
- **Scaling Factor:** The result of the low-rank path is scaled by $\frac{\alpha}{\mathbf{r}}$ (`self.scaling`), where $\mathbf{r}$ is the LoRA rank (`lora_rank`) and α is the scaling factor (`lora_alpha`).

Forward Pass Computation: For an input x, the output is the sum of the base path (frozen) and the low-rank path (trainable and scaled):

$$\text{Output} = \underbrace{F.\text{linear}(x, W_{\text{base}}, b_{\text{base}})}_{\text{Base Path (Frozen)}} + \underbrace{\left(x_{\text{dropped}} \cdot A^T \cdot B^T\right) \cdot \frac{\alpha}{r}}_{\text{LoRA Path (Trainable and Scaled)}}$$

where x_{dropped} is the input x after optional `lora_dropout`.

Trainable Parameters: The `inject_lora` function ensures that only the following parameters are updated by the optimizer:

1) The low-rank matrices $\mathbf{A}$ and $\mathbf{B}$ in all injected `LoRALinear` modules.
2) The weights and bias of the final linear `classifier` head.

All original pretrained encoder weights and biases are frozen.

Data Splitting and Hyperparameter Tuning

Data Split Strategy: The entire dataset is divided into two parts using `train_test_split` with `test_size=0.1` and `stratify=df["label"]`:

1) **Train + Validation Pool (90%):** Used entirely for the Grid Search and Cross-Validation (CV).
2) **Held-out Test Set (10%):** Never seen during training or tuning. Used only once for final, unbiased evaluation.

Grid Search and K-Fold CV Logic: The script performs a Grid Search over specific hyperparameters, with each configuration evaluated via **10**-Fold CV ($K = 10$ in `KFold`).

- Tuned Hyperparameters (Grid)

TABLE II: Hyperparameter Search Space

Hyperparameter	Search Values
Learning Rate (`lr`)	$\{2 \times 10^{-5}, 3 \times 10^{-5}, 5 \times 10^{-5}\}$
Classifier Dropout (`dropout`)	$\{0.1, 0.2\}$

- **Cross-Validation Procedure** For each configuration in the `ParameterGrid`:
  1) **Model Initialization:** A fresh `CrossEncoderLoRA` model is created, `inject_lora` is called, and the model is configured with the current `lr` and `dropout` values.
  2) **Fold Training:** The Train+Validation pool is split into 10 folds. The model is trained on 9 folds for **2** epochs (`args.epochs`).
  3) **Fold Evaluation:** The trained model is evaluated on the remaining 1 validation fold, yielding an **AUC score**.
  4) **Mean Score:** The 10 AUC scores from all folds are averaged (`mean_auc`).

  The hyperparameter configuration that achieves the highest `mean_auc` is selected as the best set (`best_params`).

Final Evaluation: The final model trained using the best parameters (from the last fold of the best configuration) is used to calculate metrics (`loss`, `acc`, `auc`) on the entirely independent **Held-out Test Set**.

IV. Semantic Chunking: A Condensed Overview

Semantic chunking divides a token sequence into contiguous segments that are internally coherent and useful as independent units for reasoning or retrieval. Desired properties include (i) *coherence* of content within each segment, (ii) *compactness* through practical size limits, (iii) *coverage* of the full input with minimal redundancy, and (iv) *task-awareness* when downstream objectives are involved.

A. Basic Formalism

Definition IV.1 (Document)**.** A document is a token sequence

$$D = (t_1, \ldots, t_N),$$

and segments are defined as its contiguous subsequences.

Definition IV.2 (Chunking)**.** A chunking is a partition of D into

$$D = S_1 \circ \cdots \circ S_K,$$

where $S_k = (t_{a_k}, \ldots, t_{b_k})$ and the segments are non-overlapping and cover the entire document.

B. Embedding-Based Representation

Let $\mathcal{E} : \mathcal{X} \to \mathbb{R}^d$ map a segment to a semantic vector. For a segment S we write

$$\mathbf{v}(S) = \mathcal{E}(S).$$

This representation may be obtained from sentence encoders, transformer pooling, or multimodal models.

C. Objectives for Chunking

a) Intra-chunk coherence.: Given sub-units $u_1, \ldots, u_m \subset S$,

$$\text{coh}(S) = -\frac{1}{m} \sum_{i=1}^{m} \|\mathcal{E}(u_i) - \bar{\mathbf{v}}_S\|^2, \qquad \bar{\mathbf{v}}_S = \tfrac{1}{m} \sum_i \mathcal{E}(u_i).$$

b) Inter-chunk distinctness.:

$$\text{sep} = \frac{2}{K(K-1)} \sum_{i<j} \left(1 - \cos(\mathbf{v}(S_i), \mathbf{v}(S_j))\right).$$

c) Local reconstruction.:

$$\mathcal{L}_{\text{rec}} = \sum_{k} \ell\big(S_k, f(\mathbf{v}(S_k))\big),$$

measuring how well segments can be approximated from their embeddings.

d) Downstream-task awareness.:

$$\min_{\{S_k\}} \mathcal{L}_{\text{task}}(\{\mathbf{v}(S_k)\}) + \lambda\, \mathcal{R}(\{S_k\}),$$

where $\mathcal{R}$ regularizes chunk size, count, and redundancy.

D. Optimization Strategies

1) **Greedy segmentation:** introduce boundaries when coherence falls below a threshold.
2) **Dynamic programming:** optimize additive scoring functions using

$$F(i) = \max_{1 \le j \le i} \{F(j-1) + \text{score}(t_j, \ldots, t_i)\}.$$

3) **Clustering or change-point detection:** identify boundaries using transitions in sliding-window embeddings.
4) **Learned models:** predict boundary probabilities using sequence models and decode via thresholding or constrained DP.

E. Probabilistic Interpretation

A simple model assumes a slowly varying latent topic sequence. Let

$$z_{i+1} \sim \text{Cat}((1-\rho)\delta_{z_i} + \rho\nu), \qquad \mathbf{x}_i | z_i \sim \mathcal{N}(\mu_{z_i}, \Sigma).$$

Segment boundaries correspond to topic changes $z_i \neq z_{i+1}$, and MAP segmentation can be recovered via Viterbi decoding.

F. Properties and Trade-offs

- Smaller segments increase coherence but may inflate redundancy; larger segments risk mixing topics.
- Coherence alone may not preserve answer spans for retrieval, motivating joint optimization with task loss.
- Multimodal settings naturally extend the same principles to joint embeddings.

G. Example DP Scoring Function

For $S = (t_j, \ldots, t_i)$,

$$\text{score}(S) = \alpha\, \text{sim}(\bar{\mathbf{v}}_S, \mathcal{E}(\text{label})) - \beta\, \text{len}(S).$$

V. Architectural Decision: Selection of Vector Store

To ensure the retrieval component of our RAG pipeline is robust, scalable, and capable of handling metadata efficiently, we conducted a comparative analysis between using a standalone Vector Library and a full-fledged Vector Database.

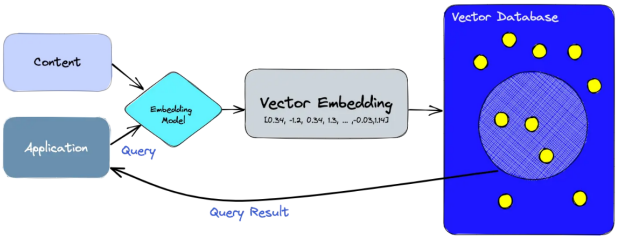

Fig. 1: VectorDB flow

A. Comparative Analysis: Vector Libraries vs. Databases

1) Vector Libraries (e.g., FAISS): Vector libraries, such as FAISS (Facebook AI Similarity Search), are optimized strictly for dense vector similarity search and clustering.

- **Limitations:** While computationally efficient, vector libraries do not natively store the associated metadata (e.g., the actual text content of the chunk, document IDs) alongside the embeddings. This necessitates maintaining a secondary key-value store to map retrieved vector indices back to text.
- **Static Indexing:** Libraries often require building the index from a static dataset. Updating the index (adding/removing documents) is non-trivial and often requires a full rebuild, which is inefficient for dynamic RAG applications.

2) Vector Databases (e.g., Pinecone, Milvus, Chroma): Vector databases wrap the indexing capabilities of libraries with a management layer.

- **Data Persistence:** They handle the storage of both the high-dimensional vectors and the associated metadata (images, text) in a single system.
- **CRUD Operations:** Unlike raw libraries, vector DBs support querying during data import and allow for real-time updates and deletions, offering an "enterprise-ready" solution for long-running applications.

B. Why choose ChromaDB

For this project, we selected **ChromaDB** as our vector store. This decision was driven by the following factors derived from our requirements analysis:

1) **Metadata Filtering:** Our RAG approach requires not just similarity search but also the retrieval of the raw text chunk for the generation step. ChromaDB allows storing the text directly with the embedding, eliminating the need for a separate database for document storage.
2) **"Batteries Included" Architecture:** Unlike Weaviate (which often requires Kubernetes) or FAISS (which requires manual index management), Chroma offers a lightweight, developer-friendly setup that functions as "ChatGPT for data." It provides built-in embedding functions and persistent storage with minimal configuration.
3) **Serverless/Local Nature:** The project workload involves linear scaling (adding documents incrementally).

Chroma's architecture is highly efficient for this usage pattern, avoiding the overhead and cost of provisioning traditional server-based instances like traditional SQL or distributed vector stores.

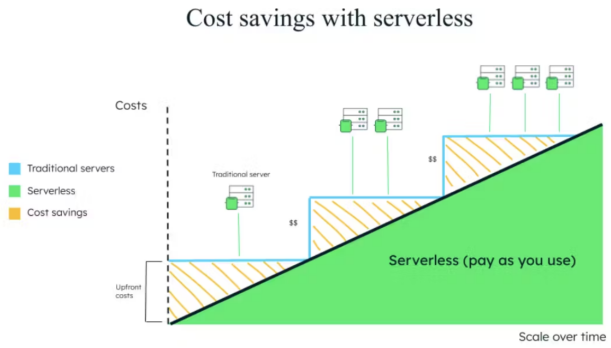

Fig. 2: Server vs Serverless Architecture

VI. End-to-End RAG Pipeline With Encoder Retrieval, Cross-Encoder Reranking, LLaMA Generation, and RAGAS Evaluation

A. Introduction

This report describes a complete Retrieval-Augmented Generation (RAG) pipeline implemented entirely in Python. The system supports end-to-end ingestion, retrieval, ranking, generation, and evaluation. The pipeline begins with *semantic chunking* of raw documents and proceeds through a sequence of retrieval and reasoning stages that collectively enable grounded answer generation.

The full workflow includes:

1) **Semantic Chunking and Indexing:** Raw documents are split into meaningful, semantically coherent segments rather than fixed-size windows. Each chunk is embedded and stored in a persistent ChromaDB collection for vector-based retrieval.
2) **Dense Embedding Retrieval:** A fine-tuned encoder model converts the user query q into a dense vector representation $\mathbf{e}_q$.
3) **Top-K Candidate Retrieval (ChromaDB):** Vector similarity search retrieves the K most relevant chunks from the Chroma index.
4) **Cross-Encoder Re-Ranking:** A LoRA-enhanced BERT cross-encoder re-scores each (q, c_i) pair to refine the ranking and produce a more precise Top-M set of chunks.
5) **Gemini-Based Answer Generation:** The selected M chunks are fed into a Gemini model to generate a grounded answer $\hat{a}$.
6) **RAGAS Evaluation:** The generated answer and evidence are assessed using RAGAS metrics:
   - Answer Relevancy
   - Faithfulness
   - Contextual Precision
   - Contextual Recall
   - Contextual Relevancy

This document provides mathematical descriptions and formal definitions for each stage of the pipeline.

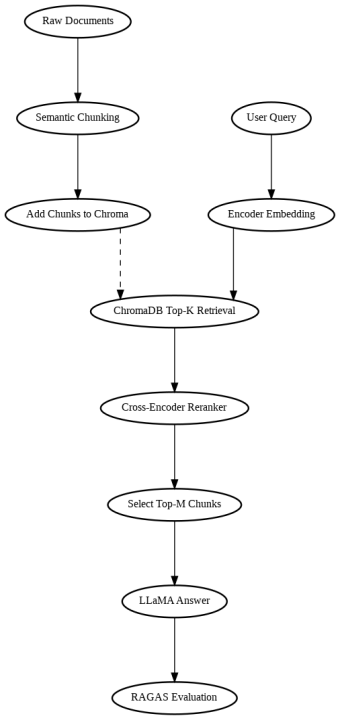

Fig. 3: Complete RAG pipeline

B. Encoder Embeddings

The encoder produces embeddings using mean pooling over token representations.

1) Tokenization and Hidden States: Given a query text q, the encoder tokenizes:

$$q \to (t_1, t_2, \ldots, t_L)$$

The model outputs hidden states:

$$H = [\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_L] \in \mathbb{R}^{L \times d}$$

2) Masked Aware Mean Pooling: Let $m_i \in \{0, 1\}$ denote whether token i is not padding. Then the embedding is computed as:

$$\mathbf{e}_q = \frac{\sum_{i=1}^{L} m_i \mathbf{h}_i}{\sum_{i=1}^{L} m_i}$$

This embedding is used for similarity search.

3) Retrieval: For a query embedding $\mathbf{e}_q$ and document embeddings $\mathbf{e}_i$, retrieval typically uses cosine similarity:

$$\text{sim}(\mathbf{e}_q, \mathbf{e}_i) = \frac{\mathbf{e}_q \cdot \mathbf{e}_i}{\|\mathbf{e}_q\| \, \|\mathbf{e}_i\|}$$

ChromaDB returns the top-K documents with highest similarity.

C. Cross-Encoder Reranking

1) Input Encoding: For each candidate chunk c_i, the cross-encoder receives a concatenated pair:

$$\text{Input} = [\text{CLS}]\, q\, [\text{SEP}]\, c_i\, [\text{SEP}]$$

2) Transformer Forward Pass: The cross-encoder transformer produces token embeddings H_i including:

$$\mathbf{h}_{\text{CLS}}^{(i)} \in \mathbb{R}^d$$

3) Classifier Head: The model includes a single-neuron classifier:

$$\ell_i = \mathbf{w}^\top \mathbf{h}_{\text{CLS}}^{(i)} + b$$

These logits are used as ranking scores.

4) Softmax-based Interpretation (Optional): For binary relevance, one can interpret:

$$p_i = \sigma(\ell_i) = \frac{1}{1 + e^{-\ell_i}}$$

The system sorts candidates by score and selects:

$$\text{Top-}M = \text{argsort}(\{\ell_i\})[:M]$$

D. LoRA Mathematics

LoRA (Low-Rank Adaptation) injects learnable rank-r updates into pretrained weights.

Given a frozen weight matrix $W_0 \in \mathbb{R}^{d\times k}$, LoRA adds:

$$W = W_0 + \Delta W$$

with:

$$\Delta W = BA$$

Where:

$$A \in \mathbb{R}^{r\times k}, \quad B \in \mathbb{R}^{d\times r}$$

Initialization:

$$A \sim \mathcal{N}(0, 0.01), \qquad B = 0$$

During training:

$$W_0 \text{ is frozen}, \quad A, B \text{ are learned}$$

The effective forward computation for input $\mathbf{x}$ is:

$$W\mathbf{x} = W_0\mathbf{x} + B(A\mathbf{x})$$

E. Gemini Answer Generation

1) Prompt Construction: Chunks $c_1, \ldots, c_M$ form the context:

$$C = \bigcup_{i=1}^{M} c_i$$

Prompt:

$$P = \text{Instructions} + C + \text{Question}(q)$$

```python
def _build_prompt(self, query: str, contexts: List[str]) -> str:
    ctx = "\n\n---\n\n".join(contexts)

    return (
        "You are a hybrid RAG assistant.\n"
        "Primary goal: use and prioritize the information provided in the context.\n"
        "Secondary goal: if the context is incomplete, unclear, or missing key details, "
        "you may use your own general knowledge — BUT keep it factual and reasonable.\n\n"

        "GUIDELINES:\n"
        "1. Prefer context when relevant.\n"
        "2. If context is insufficient, answer using your own knowledge.\n"
        "3. Clearly separate what comes from context vs your own reasoning.\n"
        "4. Do NOT hallucinate details that contradict the context.\n"
        "5. Keep the answer concise and helpful.\n\n"

        "======== CONTEXT START ========\n"
        f"{ctx}\n"
        "========= CONTEXT END =========\n\n"

        f"QUESTION:\n{query}\n\n"

        "FINAL ANSWER FORMAT:\n"
        "- Answer: <your answer>\n"
        "- Context usage: <explain briefly whether context was used or not>\n\n"

        "Now produce the final answer.\n"
        "ANSWER:"
    )
```

Fig. 4: LLM system instruction

2) Autoregressive Generation: Gemini generates tokens:

$$p(x_t | x_{<t}, P)$$

The model uses greedy decoding:

$$x_t = \arg\max_{v\in V} p(v | x_{<t}, P)$$

F. RAGAS Metrics:

RAGAS (Retrieval-Augmented Generation Assessment Suite) provides a set of quantitative metrics to evaluate the quality of Retrieval-Augmented Generation (RAG) systems. Each metric measures a different aspect of the system: the usefulness of retrieved context, the alignment of the generated answer with the reference answer, and the faithfulness of the generated content to the provided context.

We denote:

$$q = \text{query}, \quad C = \{c_1, \ldots, c_M\} = \text{retrieved context},$$

$$R = \text{reference (ground-truth) answer}, \quad \hat{a} = \text{generated answer}$$

Token sets:

$$T_C = \text{tokens}(C), \quad T_R = \text{tokens}(R), \quad T_{\hat{a}} = \text{tokens}(\hat{a})$$

We now define each metric mathematically.

1) 1. Answer Relevancy: Answer relevancy measures whether the generated answer $\hat{a}$ is relevant to the query q. This is computed using a large language model (LLM) as a judge.

Let $f_{\text{rel}}(q, \hat{a})$ be a scoring function that returns either:

$$f_{\text{rel}} \in \{0, 1\}$$

or a continuous score in $[0, 1]$ if probabilistic LLM scoring is used.

Thus:

$$\text{AnswerRelevancy}(q, \hat{a}) = f_{\text{rel}}(q, \hat{a})$$

For a dataset of N samples:

$$\text{AnswerRelevancy} = \frac{1}{N} \sum_{i=1}^{N} f_{\text{rel}}(q_i, \hat{a}_i)$$

Where f_{rel} is typically implemented by prompting the LLM: *"Does the answer directly address the question?"*

2) 2. Faithfulness: Faithfulness measures whether the generated answer $\hat{a}$ is supported by the retrieved context C.

Define:

$$f_{\text{faith}}(\hat{a}, C) \in \{0, 1\}$$

A value of 1 means every factual claim in $\hat{a}$ is grounded in C.

Let the set of claims extracted from $\hat{a}$ be:

$$\mathcal{K}(\hat{a}) = \{k_1, \ldots, k_L\}$$

For each claim k_j we ask the LLM:

$$g(k_j, C) = \begin{cases} 1 & \text{if claim } k_j \text{ is supported by } C \\ 0 & \text{otherwise} \end{cases}$$

Then faithfulness is:

$$f_{\text{faith}}(\hat{a}, C) = \frac{1}{|\mathcal{K}(\hat{a})|} \sum_{j=1}^{|\mathcal{K}(\hat{a})|} g(k_j, C)$$

Dataset-level faithfulness:

$$\text{Faithfulness} = \frac{1}{N} \sum_{i=1}^{N} f_{\text{faith}}(\hat{a}_i, C_i)$$

3) 3. Contextual Precision: Contextual precision measures how much of the generated answer $\hat{a}$ is actually grounded in the retrieved context C.

We treat context grounding as an overlap between token sets:

$$\text{ContextualPrecision} = \frac{|T_{\hat{a}} \cap T_C|}{|T_{\hat{a}}|}$$

Interpretation:

- High precision: the answer mostly uses tokens or information found in context.
- Low precision: the answer contains hallucinated or irrelevant content.

a) Semantic Enhancement: If token overlap is replaced with semantic grounding using embedding similarity $\text{sim}(\cdot, \cdot)$, we instead compute:

$$\text{ContextualPrecision}_{\text{semantic}} = \frac{\sum_{t \in T_{\hat{a}}} \max_{c \in T_C} \mathbf{1}(\text{sim}(t, c) > \tau)}{|T_{\hat{a}}|}$$

Where τ is a semantic similarity threshold.

4) 4. Contextual Recall: Contextual recall measures how much of the reference answer R is recoverable from the generated answer $\hat{a}$.

$$\text{ContextualRecall} = \frac{|T_{\hat{a}} \cap T_R|}{|T_R|}$$

Interpretation:

- High recall: the model covers most of the important content.
- Low recall: the model misses essential information.

a) Semantic Variant:

$$\text{ContextualRecall}_{\text{semantic}} = \frac{\sum_{t \in T_R} \max_{a \in T_{\hat{a}}} \mathbf{1}(\text{sim}(t, a) > \tau)}{|T_R|}$$

5) 5. Contextual Relevancy: Contextual relevancy measures whether the retrieved context C is actually useful for answering the query.

It evaluates whether the answer $\hat{a}$ uses the provided context. Mathematically:

$$\text{ContextualRelevancy} = \frac{|T_{\hat{a}} \cap T_C|}{|T_C|}$$

Interpretation:

- High contextual relevancy: retrieved chunks are aligned with the question.
- Low contextual relevancy: retrieval is poor; wrong documents retrieved.

a) Semantic Variant:

$$\text{ContextualRelevancy}_{\text{semantic}} = \frac{\sum_{c \in T_C} \max_{a \in T_{\hat{a}}} \mathbf{1}(\text{sim}(c, a) > \tau)}{|T_C|}$$

6) Summary of All Metrics:

$$\text{AnswerRelevancy} = f_{\text{rel}}(q, \hat{a})$$

$$\text{Faithfulness} = f_{\text{faith}}(\hat{a}, C)$$

$$\text{ContextualPrecision} = \frac{|T_{\hat{a}} \cap T_C|}{|T_{\hat{a}}|}$$

$$\text{ContextualRecall} = \frac{|T_{\hat{a}} \cap T_R|}{|T_R|}$$

$$\text{ContextualRelevancy} = \frac{|T_{\hat{a}} \cap T_C|}{|T_C|}$$

Each captures a different part of the RAG pipeline:

- **Answer Relevancy**: correctness w.r.t. the question.
- **Faithfulness**: non-hallucination w.r.t. retrieved chunks.
- **Contextual Precision**: how precisely the model used the context.
- **Contextual Recall**: how much of the reference answer was reproduced.
- **Contextual Relevancy**: how good retrieval was.

VII. Results and Discussion

This section reports the outcomes of all major stages in the pipeline, including: (1) BERT pre-training (MLM and NSP), (2) BERT fine-tuning on positive/negative query–chunk pairs, and (3) Cross-encoder re-ranking performance using LoRA adapters.

A. BERT Pre-Training Performance

The custom BERT model was pre-trained using two standard self-supervised objectives: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). Although the exact logs were not captured, the final averaged metrics during pre-training were recorded as follows:

Objective	Performance
MLM Accuracy	0.2331
NSP Accuracy	0.9143

TABLE III: BERT Pre-Training Results

Interpretation. The MLM accuracy is relatively low (0.2), which is expected given that the model was trained on a limited corpus and with significantly fewer steps than typical large-scale pre-training. MLM generally requires substantial amounts of text to build strong bidirectional contextual representations; therefore, modest performance at this stage is not surprising.

The NSP accuracy is high (0.9), indicating that the model easily distinguished whether sentence pairs were contiguous or random. NSP is a simpler binary task and tends to converge faster than MLM even with limited data. Overall, the pre-training results are consistent with expectations for a small-scale, resource-constrained BERT variant.

B. BERT Fine-Tuning on Positive/Negative Pairs

The model was fine-tuned on labeled query–chunk relevance pairs (+ and −) as part of the retrieval re-ranking pipeline. The fine-tuned classifier achieved the following performance:

Task	Accuracy
Query–Chunk Relevance Classification	0.60932

TABLE IV: BERT Fine-Tuning Results

Interpretation. The accuracy is reasonable for a BERT model trained from scratch using a small corpus. Since the encoder does not benefit from the massive-scale pre-training that standard BERT models receive, it lacks the deep linguistic priors that typically boost performance. Despite these limitations, the model still learns useful discriminative patterns that allow it to outperform chance substantially.

C. Cross-Encoder Re-Ranking with LoRA Adapters

The LoRA-enhanced cross-encoder was trained for 5 epochs on the pairwise relevance dataset. The training and test logs are summarized below (taken from Figure **??**):

- Training AUC: 0.97 – 0.99
- Test AUC: 0.97+
- Test Accuracy: 0.96+
- Test Loss: ≈ 0.13

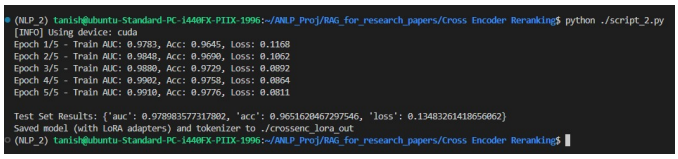

Fig. 5: Cross-Encoder LoRA Training and Evaluation Output

Interpretation. The cross-encoder achieves significantly stronger performance than the base BERT encoder. This is expected: unlike the dual-encoder (which embeds query and chunk independently), the cross-encoder processes both inputs jointly, enabling deeper token-level interactions. The strong AUC and accuracy values indicate that the model is highly effective at pairwise ranking even with LoRA adapters and no full-model fine-tuning. **Hybrid Knowledge Utilization:** Given

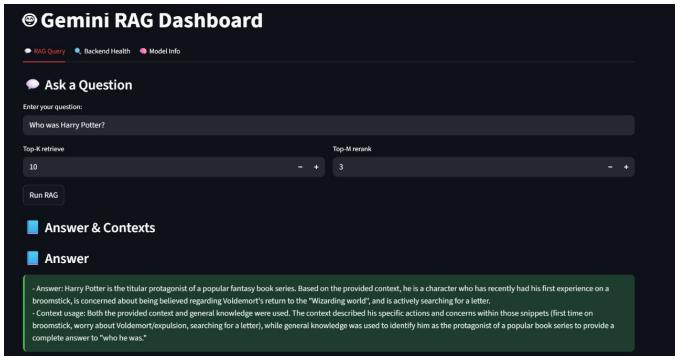

Fig. 6: RAG App Example

the constraints of our custom-trained small-scale retrieval components, the system is designed to function as a hybrid engine. When the retrieved context (non-parametric knowledge) is sparse or lacks sufficient semantic density, the generation model is explicitly permitted to bridge these information gaps using its pre-trained internal world knowledge (parametric knowledge). This ensures response completeness even when the retrieval stage yields suboptimal candidates.

D. RAG assessment using RAGAS - Performance Metrics:

Listing 1: Example JSON

```
{
    "faithfulness": 0.0400,
    "answer_relevancy": 0.0000,
    "context_precision": 0.0145,
    "context_recall": 0.0435,
    "nv_context_relevance"': 0.0700
}
```

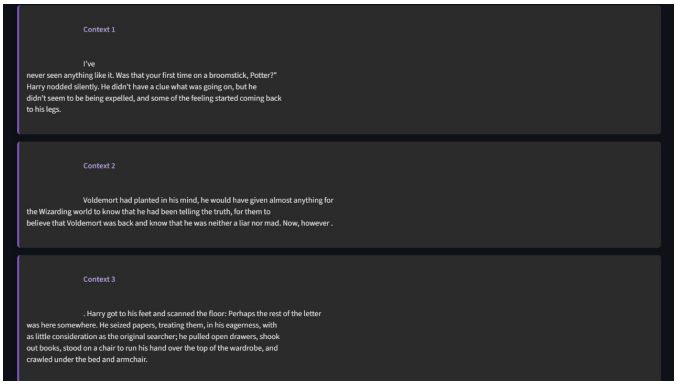

Fig. 7: RAG App Example

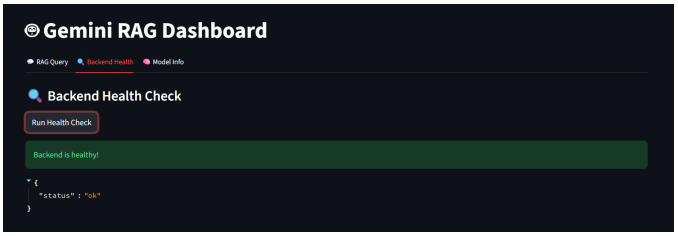

Fig. 8: RAG App backend health check endpoint

E. Conclusion

This report described the complete mathematical and algorithmic structure of the implemented RAG pipeline:

- Dense retrieval using encoder embeddings.
- Cross-encoder reranking using LoRA-injected BERT.
- LLaMA generation using retrieved context.
- RAGAS-inspired evaluation combining LLM judgment and token overlap metrics.

The pipeline provides an interpretable and modular structure suitable for large-scale RAG evaluation using locally hosted models and local vector databases.

VIII. Disclaimer

We were not able to tune the hyperparameters of our BERT large model (that has Mixture of Experts strategy in the FFN layer) using Bayesian optimization cross validation/Grid Search CV/Random Search CV because it demands a lot of compute power which is provided by a GPU, which we did not get till the last day of our work on this project. These hyperparamter tuning methods require a lot of time especially when we are working on pretraining our own model. Thus, platforms like Google Colab and Kaggle were not suitable for us, in the long term. The incompetency of our RAGAS metrics is a consequence of a) lack of hyperparameter tuning, b) size of dataset and c) number of epochs. All of these features could have been improved upon had we got access to a GPU. Please do not consider this as a complaint from our side, we just wanted to let you know that we have tried to put in as much efforts as we could in designing and improving our project. We hope that we are assessed based on these grounds.

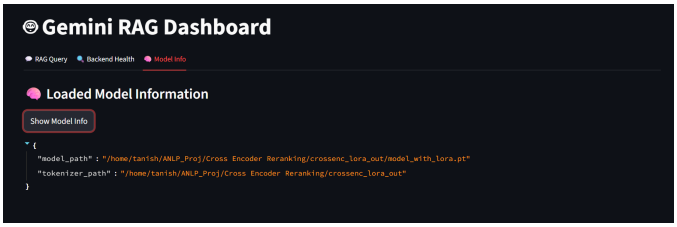

Fig. 9: RAG App model info

IX. GitHub link

Link to Github Repository: Click here

X. Acknowledgements

Portions of the coding workflow, document structuring, and report refinement were assisted by AI tools, including large language models, which were used for tasks such as code debugging, documentation formatting, and summarization. All modelling decisions, experimental setups, and analyses, however, were designed, implemented, and critically evaluated by the authors.

We also thank the maintainers of open-source libraries such as PyTorch, HuggingFace Transformers, ChromaDB, and scikit-learn, without which the development of our pipeline would not have been possible.

XI. References

1) Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.* NAACL, 2019.
2) Reimers, N. & Gurevych, I. *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks.* EMNLP, 2019. (Cross-Encoder and Bi-Encoder Reranking Framework)
3) Johnson, J., Douze, M., & Jégou, H. *Billion-Scale Similarity Search with GPUs.* IEEE Transactions on Big Data, 2019. (FAISS: Facebook AI Similarity Search)
4) ChromaDB Documentation. *https://docs.trychroma.com* (Accessed 2025)
5) Anaconda Software Distribution. *Anaconda Documentation: Package, Environment, and Dependency Management.* Available at: https://docs.anaconda.com
6) OpenAI. *GPT-4 and ChatGPT Technical Reports.* Available at: https://openai.com/research (ChatGPT usage during RAG development)