Ref : https://jyoonjang.medium.com/re-ranker-%EB%AA%A8%EB%8D%B8-%ED%95%99%EC%8A%B5%EC%9D%84-%EC%9C%84%ED%95%9C-%EB%8B%A4%EC%96%91%ED%95%9C-loss%EB%A5%BC-%EC%95%8C%EC%95%84%EB%B3%B4%EC%9E%90-feat-learning-to-rank-listnetloss-lambdaloss-1a18b9697efb

Purpose

simply matching keywords is often insufficient. Users seek documents that are semantically relevant to their queries, even if specific keywords are not present. This system addresses that need by leveraging advanced neural networks for semantic understanding.

two-stage approach solves this by:

1. **Rapidly narrowing down** the search space using an efficient embedding-based similarity search.
2. **Precisely re-ranking** the top candidates using a more powerful, computationally intensive model.

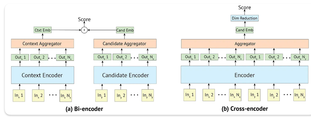
System Architecture

Stage 1: Fast Retrieval (Embedding Model & Vector Database)

- **Embedding Model:** A pre-trained transformer-based model (e.g., all-MiniLM-L6-v2) converts each document and query into a fixed-size numerical vector (embedding). These embeddings capture the semantic meaning of the text.
- **Document Indexing:** All documents in the corpus are pre-processed offline. Their embeddings are computed and then indexed in a specialized **Vector Database** (e.g., Faiss). This database is optimized for fast similarity searches between high-dimensional vectors.
- **Query Processing:** When a query arrives, its embedding is computed. The vector database then efficiently finds the top-K document embeddings that are most similar to the query embedding (typically using cosine similarity or dot product).

Stage 2: Precise Reranking (Cross-Encoder Reranker)

- **Reranker Model:** A cross-encoder model (e.g., cross-encoder/ms-marco-MiniLM-L-6-v2) takes a pair of texts (query and a retrieved document) as input. Unlike a bi-encoder (embedding model), it processes the query and document *together*, allowing for deeper contextual understanding and more nuanced relevance scoring.
- **Scoring:** For each of the top-K documents from Stage 1, the reranker computes a relevance score based on the query-document pair.
- **Final Ordering:** The top-K documents are then sorted by these reranker scores, and the highest-scoring N documents are presented as the final results.



An embedding model (e.g., BERT) takes text as input and outputs a single embedding for each input token. When generating the final embedding, various pooling methods (e.g., mean pooling, [CLS] token pooling) are applied to these tokens to produce a single, consolidated embedding.
In contrast, a reranker shares the same architecture as an embedding model, but it outputs a score by **applying a linear transformation to convert the final embedding into a single score.**

Implementtion Example

```
import torch
import numpy as np
import faiss
from sentence_transformers import SentenceTransformer
from transformers import AutoTokenizer, AutoModelForSequenceClassification

# --- Configuration ---
# Document corpus
DOCUMENTS = [
"The quick brown fox jumps over the lazy dog.",
"A red car drove by very fast.",
"The dog was sleeping soundly under the tree.",
"Foxes are known for their cunning nature.",
"Cats often chase mice.",
"Bears hibernate in winter.",
"The car was parked near the red building.",
"A speedy vehicle passed by.",
"The canine was resting under the shade.",
"Wildlife includes foxes and bears."
]

# Models
EMBEDDING_MODEL_NAME = 'all-MiniLM-L6-v2'
RERANKER_MODEL_NAME = 'cross-encoder/ms-marco-MiniLM-L-6-v2'

# Retrieval parameters
TOP_K_RETRIEVAL = 5  # Number of documents to retrieve in Stage 1
N_FINAL_RESULTS = 3  # Number of top results to show after reranking

# Device setup
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {DEVICE}")

# --- Stage 1: Fast Retrieval Setup (Offline/Indexing Phase) ---

print("\n--- Stage 1: Indexing Documents ---")
print(f"Loading Embedding Model: {EMBEDDING_MODEL_NAME}")
embedding_model = SentenceTransformer(EMBEDDING_MODEL_NAME)
embedding_model.to(DEVICE)
embedding_model.eval()

print("Generating document embeddings...")
# Encode documents in batches if your corpus is very large
document_embeddings = embedding_model.encode(DOCUMENTS, convert_to_tensor=True, show_progress_bar=True)
document_embeddings_np = document_embeddings.cpu().numpy()
embedding_dimension = document_embeddings_np.shape[1]

# Normalize embeddings for cosine similarity with Inner Product (IP) index
faiss.normalize_L2(document_embeddings_np)

print(f"Generated {len(document_embeddings)} embeddings, each of size {embedding_dimension}")

# Create and populate Faiss index
# IndexFlatIP is suitable for cosine similarity after L2 normalization
faiss_index = faiss.IndexFlatIP(embedding_dimension)
faiss_index.add(document_embeddings_np)
print(f"Faiss index created with {faiss_index.ntotal} documents.")

# --- Stage 2: Reranking Model Loading ---

print(f"\n--- Stage 2: Loading Reranker Model ---")
print(f"Loading Reranker Model: {RERANKER_MODEL_NAME}")
reranker_tokenizer = AutoTokenizer.from_pretrained(RERANKER_MODEL_NAME)
reranker_model = AutoModelForSequenceClassification.from_pretrained(RERANKER_MODEL_NAME)
reranker_model.to(DEVICE)
reranker_model.eval()

# --- Search Execution (Online/Query Phase) ---

def search_documents(query: str):
"""
Executes the two-stage retrieval process for a given query.
"""
print(f"\n--- Processing Query: '{query}' ---")

# Stage 1: Retrieval
print("  Stage 1: Performing fast retrieval...")
query_embedding = embedding_model.encode(query, convert_to_tensor=True).cpu().numpy()
faiss.normalize_L2(query_embedding) # Normalize query embedding
query_embedding = query_embedding.reshape(1, -1) # Reshape for Faiss

# Perform search
distances, indices = faiss_index.search(query_embedding, TOP_K_RETRIEVAL)

retrieved_doc_indices = indices[0].tolist()
retrieved_documents = [DOCUMENTS[i] for i in retrieved_doc_indices]

print(f"  Retrieved {len(retrieved_documents)} candidates from Stage 1:")
for i, doc_text in enumerate(retrieved_documents):
print(f"    {i+1}. '{doc_text}' (Original Index: {retrieved_doc_indices[i]})")

# Stage 2: Reranking
if not retrieved_documents:
print("  No documents retrieved in Stage 1. Skipping reranking.")
return []

print("  Stage 2: Performing precise reranking...")
reranker_pairs = [[query, doc] for doc in retrieved_documents]

with torch.no_grad():
inputs = reranker_tokenizer(reranker_pairs, padding=True, truncation=True, return_tensors='pt').to(DEVICE)
scores = reranker_model(**inputs).logits.squeeze().cpu().numpy()

# Combine results and sort
scored_results = []
for i, doc_text in enumerate(retrieved_documents):
original_idx = retrieved_doc_indices[i]
score = scores[i]
scored_results.append({'original_index': original_idx, 'text': doc_text, 'score': float(score)})

scored_results.sort(key=lambda x: x['score'], reverse=True)

print(f"\n  Final Reranked Results (Top {N_FINAL_RESULTS}):")
if not scored_results:
print("    No results found.")
else:
for i, doc_info in enumerate(scored_results[:N_FINAL_RESULTS]):
print(f"    {i+1}. Score: {doc_info['score']:.4f}, Document: '{doc_info['text']}' (Original Index: {doc_info['original_index']})")

return scored_results

# --- Example Queries ---
search_documents("fast animals")
search_documents("what do bears do in winter?")
search_documents("vehicles that drive quickly")
search_documents("a story about a clever animal")
```