sentence-transformers

sentence-transformers turns sentences into embeddings for semantic search, covered here through similarity, retrieval, and reranking for RAG.

Load a Model

One line pulls a pretrained embedding model from the Hub.

PURPOSE	COMMAND	RESULT
Load a small, fast default model.	`model = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")`	
Check the embedding dimension.	`model.get_sentence_embedding_dimension()`	
See the input length limit.	`model.max_seq_length`	
Pick the device (CPU vs GPU).	`SentenceTransformer("all-mpnet-base-v2", device="cuda")`	
Pick a stronger model (slower).	`SentenceTransformer("all-mpnet-base-v2")`	

SentenceTransformer("name") loads a Hub model; all-MiniLM-L6-v2 (384 dims) is the fast default, all-mpnet-base-v2 (768) is stronger

Encode Text to Vectors

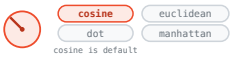
model.encode turns strings into dense embedding vectors.

PURPOSE	COMMAND	RESULT
Encode a list of sentences.	`emb = model.encode(["a cat...","a dog..."])`	
Get a torch tensor instead.	`model.encode(texts, convert_to_tensor=True)`	
Normalize to unit length.	`model.encode(texts, normalize_embeddings=True)`	
Encode a query vs a document.	`model.encode_query(q)` `model.encode_document(doc)`	
Show progress on long jobs.	`model.encode(texts, show_progress_bar=True)`	
Encode a single string too.	`model.encode("just one sentence")`	

model.encode(texts) returns an (n, dim) float32 ndarray; convert_to_tensor=True for torch, normalize_embeddings=True for cosine = dot

Cosine Similarity

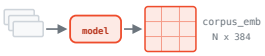
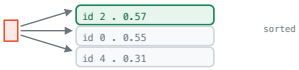
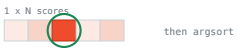
Compare embedding vectors; cosine in [-1, 1], higher means closer.

PURPOSE	COMMAND	RESULT
Score every pair (matrix).	`sim = model.similarity(emb, emb)`	
Read one pair's similarity.	`float(sim[0, 1])`	
Pairwise (row i vs row i only).	`model.similarity_pairwise(a, b)`	
Cosine without a model handle.	`cos_sim(emb[0], emb[1])`	
Switch the similarity metric.	`SentenceTransformer(name, similarity_fn_name="dot")`	

model.similarity(a, b) gives the full pairwise cosine matrix; similarity_pairwise compares rows one-to-one; cosine is the default metric

Semantic Search

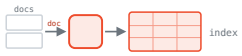
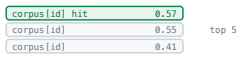
Rank a whole corpus against a query by vector similarity.

PURPOSE	COMMAND	RESULT
Embed the corpus once.	`corpus_emb = model.encode(corpus, convert_to_tensor=True)`	
Embed the query.	`q_emb = model.encode(query, convert_to_tensor=True)`	
Top-k search in one call.	`hits = semantic_search(q_emb, corpus_emb, top_k=3)`	
Score the matrix by hand.	`cos_sim(q_emb, corpus_emb)`	
Use dot product for speed.	`semantic_search(q_emb, corpus_emb, score_function=dot_score)`	

embed the corpus once, embed each query, then semantic_search returns the top-k {corpus_id, score} hits sorted best-first

Build & Query an Index

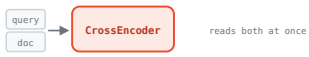
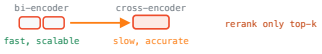
Encode the corpus once, reuse it for every query.

PURPOSE	COMMAND	RESULT
Build the index (asymmetric).	`corpus_emb = model.encode_document( corpus, convert_to_tensor=True)`	
Save the index to disk.	`torch.save(corpus_emb, "corpus.pt")`	
Load it back later.	`corpus_emb = torch.load("corpus.pt")`	
Embed the query side.	`q_emb = model.encode_query( "how to reset password")`	
Query the index, read hits.	`for h in semantic_search(q_emb, corpus_emb, top_k=5)[0]:`	
Keep adding documents.	`corpus_emb = torch.cat([corpus_emb, model.encode_document(new)])`	

encode_document builds the index once; torch.save / torch.load persist it; encode_query each incoming query and semantic_search the saved tensor

Rerank with a CrossEncoder

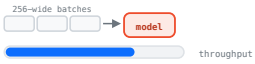
A cross-encoder reads (query, doc) together for sharper scores.

PURPOSE	COMMAND	RESULT
Load a reranker model.	`ce = CrossEncoder( "cross-encoder/ms-marco-MiniLM-L6-v2")`	
Score query-document pairs.	`scores = ce.predict( [(query, d) for d in docs])`	
Rank candidates directly.	`ranks = ce.rank(query, docs, top_k=3)`	
Get 0..1 probabilities.	`ce.predict(pairs, activation_fn=Sigmoid())`	
Bi-encoder vs cross-encoder.	`# two vectors then compare vs one joined pass`	

a CrossEncoder reads (query, doc) jointly for a sharper score; too slow for a whole corpus, so run it only on the top-k from retrieval

Batch Encode at Scale

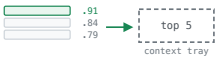
Tune batch size, precision, and parallelism for big corpora.

PURPOSE	COMMAND	RESULT
Encode in tuned batches.	`model.encode(texts, batch_size=256, show_progress_bar=True)`	
Half precision for GPU speed.	`model.half()` then `model.encode(texts)`	
Quantize embeddings (smaller).	`model.encode(texts, precision="int8")`	
Truncate dims (Matryoshka).	`model.encode(texts, truncate_dim=128)`	
Parallel multi-process pool.	`pool = model.start_multi_process_pool()` `model.encode(texts, pool=pool)`	
Quantize plus normalize on disk.	`model.encode(texts, precision="int8", normalize_embeddings=True)`	

raise batch_size and show_progress_bar; half()/int8/truncate_dim shrink and speed it up; pass pool= for CPU parallelism

The retrieve-and-rerank (RAG) flow

Bi-encoder retrieves top-k, cross-encoder reranks, then you answer.

PURPOSE	COMMAND	RESULT
1. Retrieve top-k fast.	`hits = semantic_search(` `model.encode_query(q), corpus_emb, top_k=50)`	
2. Rerank the candidates.	`ranks = ce.rank(q,` `[corpus[h['corpus_id']] for h in hits[0]])`	
3. Take the best passages.	`top = [corpus[r['corpus_id']]` `for r in ranks[:5]]`	
4. Build the prompt context.	`context = "\n\n".join(top)`	
Why two stages.	`# recall (millions) then precision`	
Build the two-stage retriever.	`bi = SentenceTransformer(name)` `ce = CrossEncoder(reranker_name)`	

retrieve-and-rerank: a cheap bi-encoder finds many candidates (recall), a cross-encoder reranks the few (precision), the top passages become LLM context