ColBERT, SPLADE-Training Retrievers

🧠 Topic: ColBERT — Contextualized Late Interaction over BERT

🧩 1. Motivation — Why ColBERT?

Earlier dense retrievers like **DPR** used **a single vector** to represent an entire passage or document.

That's efficient but **too coarse** — it loses fine-grained token-level meaning.

Example:

> Passage: "Isaac Newton discovered gravity and developed calculus."
>
> Query 1: "Who discovered gravity?"
>
> Query 2: "Who developed calculus?"

A single embedding must represent *both ideas*, which causes ambiguity.

ColBERT was introduced to fix this by:

- Preserving **contextualized token-level embeddings** (not just one pooled vector).
- Enabling **fine-grained matching** between query tokens and document tokens.
- Maintaining **retrieval efficiency** by performing interaction **after encoding** (hence "Late Interaction").

⚙️ 2. Core Idea — "Late Interaction" Mechanism

Traditional BERT-based retrieval (like Cross-Encoder) encodes the **[query, passage] pair jointly**, which gives precise similarity but is **too slow** (needs reencoding for every query).

ColBERT introduces a compromise:

- Encode **queries and documents separately** using BERT (→ can precompute document embeddings).
- During retrieval, match them at the **token level** using a lightweight “late interaction” operation.

This gives **Cross-Encoder-level precision** with **bi-encoder-level efficiency**.

🧠 3. Architecture Overview

ColBERT has **three main components:**

🔷 a. Query Encoder

- Input: a query string
- Output: contextualized token embeddings

  $Q = [q_1, q_2, ..., q_m] \in \mathbb{R}^{m \times d}$

  (each (q_i) is a d-dimensional vector)

🔷 b. Document Encoder

- Input: a passage/document
- Output: contextualized token embeddings

  $D = [d_1, d_2, ..., d_n] \in \mathbb{R}^{n \times d}$

🔷 c. Late Interaction Layer

Instead of pooling into one vector, ColBERT computes token-level interactions:

$$\text{Sim}(Q, D) = \sum_{i=1}^{m} \max_{j=1,\ldots,n} q_i \cdot d_j$$

That means:

- For each **query token** (q_i), find the **most similar document token** (d_j).

- Take their **dot product** (semantic similarity).
- Then sum across all query tokens for a total passage score.

This preserves token-level semantics while keeping computation scalable.

🧮 4. Training Objective — Contrastive Learning

ColBERT uses a **softmax-based contrastive loss** (similar to DPR) but with the late interaction score as similarity.

For each query (q):

- (p^+): positive (relevant) passage
- (p^-): negative (irrelevant) passages

$$L = -\log \frac{\exp(\text{Sim}(q, p^+))}{\exp(\text{Sim}(q, p^+)) + \sum_{p^-} \exp(\text{Sim}(q, p^-))}$$

This encourages the model to produce higher similarity scores for correct query–document pairs.

🔍 5. How "Late Interaction" Improves Efficiency

- **Early Interaction (Cross-Encoder):** Query and document are concatenated → huge computational cost (cannot precompute docs).
- **Late Interaction (ColBERT):** Query and document are encoded independently → documents can be pre-embedded and stored.

During retrieval, only a **lightweight dot product** is computed between token embeddings — enabling **fast ANN search** with **near-Cross-Encoder accuracy**.

⚡ 6. Example — Token-Level Matching

Query: "Who discovered gravity?"

Passage: "Isaac Newton discovered gravity and developed calculus."

Step 1: Query tokens → [Who, discovered, gravity]

Step 2: Passage tokens → [Isaac, Newton, discovered, gravity, developed, calculus]

Each query token finds the *most similar* passage token:

- "Who" → "Isaac" or "Newton"
- "discovered" → "discovered"
- "gravity" → "gravity"

Then the max similarities are summed up:

$$\text{Sim} = (q_{who} \cdot d_{Newton}) + (q_{disc} \cdot d_{disc}) + (q_{grav} \cdot d_{grav})$$

✅ Result: The passage gets a high total similarity score.

🧩 7. ColBERT Workflow (Step-by-Step)

1. **Indexing:**
    - Encode all passages using the document encoder.
    - Store all their token embeddings (each doc has multiple vectors) in a **vector index** (e.g., FAISS).
2. **Query Encoding:**
    - Encode the input query to obtain query token embeddings.
3. **Retrieval:**
    - For each query token, retrieve the **nearest document token vectors** (ANN search).
    - Combine (max + sum) similarities to score documents.
    - Return top-k results.

🧩 8. Example Visualization (Text-Style)

```
Query: Who discovered gravity?
↓
```

```
[Who] ⟶ best match → [Newton]
[discovered] ⟶ best match → [discovered]
[gravity] ⟶ best match → [gravity]
------------------------------
Final Score = sum of max similarities
```

🧠 9. ColBERT-v2 (Improvement Highlights)

- Introduces **compressed embeddings** (from 128-dim → 32-dim)
- Uses **residual quantization** for smaller storage footprint
- Faster ANN search (up to 10× more efficient)
- Same or better retrieval accuracy

This makes ColBERT-v2 more suitable for **web-scale search** and **RAG systems**.

📘 10. Advantages

✅ **High accuracy:** Close to Cross-Encoder performance

✅ **Fast retrieval:** Scalable via precomputed embeddings

✅ **Fine-grained matching:** Token-level semantic understanding

✅ **Interpretability:** Shows which tokens drive retrieval

✅ **Widely adopted:** Used in FAISS, Pyserini, and RAG pipelines

⚠️ 11. Limitations

⚠️ **Storage heavy:** Each document has multiple embeddings (large vector index)

⚠️ **Retrieval latency:** More token comparisons

⚠️ **Complex setup:** Requires efficient vector compression and ANN indexing

⚠️ **Training data requirement:** Needs labeled (query, passage) pairs

💡 12. Applications

✅ **Open-Domain QA systems** (e.g., MS MARCO, Natural Questions)

✅ **RAG pipelines** — precise document retrieval for LLMs

✅ **Search engines** — hybrid sparse+dense retrieval

✅ **Domain-specific retrieval** — legal, academic, or biomedical search

---

🧾 13. Key Takeaways

- **ColBERT = Contextualized Late Interaction over BERT**
- Introduces **token-level semantic matching** with **late interaction**
- Balances **accuracy (like Cross-Encoder)** and **speed (like DPR)**
- Foundation for **multi-vector retrieval models** (ColBERT-v2, SPLADE, etc.)
- Core building block in **retrieval-augmented generation (RAG)** systems.

🧠 SPLADE — Sparse Lexical and Expansion Model for Retrieval

🧩 1. Motivation — Why SPLADE?

Before SPLADE, there were two main retrieval worlds:

Type	Example	Pros	Cons
Sparse retrieval	BM25, TF-IDF	Interpretable, fast, low memory	Relies on exact word overlap
Dense retrieval	DPR, ColBERT	Semantic understanding	Requires vector search, expensive storage

The problem:

> Dense retrieval captures meaning but loses interpretability and lexical control.
>
> Sparse retrieval is **interpretable** but misses **semantic matches**.

So researchers asked:

> 💡 Can we get the best of both — semantics of dense retrieval and explainability of sparse retrieval?

➡️ Enter **SPLADE** — a **hybrid model** that brings neural understanding into a **sparse lexical space**.

⚙️ 2. Core Idea — Sparse Expansion in Vocabulary Space

SPLADE stands for:

> SParse Lexical And DEnse model

But unlike DPR or ColBERT, SPLADE doesn't output continuous dense vectors.

Instead, it outputs **sparse high-dimensional vectors** aligned with the **vocabulary space** — just like TF-IDF or BM25.

How?

It uses a **transformer (like BERT)** to produce *contextualized embeddings for each token*,

then converts those embeddings into **vocabulary-level scores** using a **vocabulary projection** and **sparsifying function**.

This way:

- Each document or query is represented as a **sparse vector** over the vocabulary.
- Only a few tokens (the most meaningful ones) have **non-zero weights** — everything else is **zeroed out**.

Hence, "SPLADE" = *Sparse lexical expansion model*.

🧱 3. SPLADE Setup — Architecture and Workflow

Let's go through its setup step-by-step 👇

(a) Encoders

- Uses **a BERT encoder** (or any transformer) for both **queries** and **documents**.

- Each token embedding → transformed into **vocabulary logits** using a linear projection layer:

  $z_t = W \cdot h_t + b$

  where (h_t) = hidden representation of token t, and W projects to vocab size.

(b) Aggregation

- Each token predicts **weights** for many vocabulary words.
- To combine all token contributions, SPLADE uses:

  $\text{score}(w) = \max_{t \in T}(\log(1 + \text{ReLU}(z_{t,w})))$

  or sometimes a **log-sum-exp pooling** over tokens.

This creates a vector of length = vocabulary size (≈30,000 for BERT),

but **most entries are zero**, giving sparsity.

(c) Sparsity Control (Regularization)

To keep the representation sparse and interpretable, SPLADE adds an **L_1 regularization term**:

$L_{\text{sparse}} = \lambda \sum_w |\text{score}(w)|$

This encourages many weights to be zero → sparse representation.

(d) Similarity Calculation

Once both query and document are encoded into sparse vocab-weight vectors:

$\text{Sim}(q, d) = \sum_w q_w \cdot d_w$

— just a **dot product** in sparse space (like BM25 cosine similarity).

✅ Advantage: Retrieval can be done using **traditional inverted indexes** (like Lucene or Elasticsearch).

🧩 4. Training Objective — Contrastive Learning (Like DPR)

SPLADE is trained to bring relevant query-document pairs closer and irrelevant ones farther apart.

$$L = -\log \frac{\exp(\mathrm{Sim}(q, d^+))}{\exp(\mathrm{Sim}(q, d^+)) + \sum_{d^-} \exp(\mathrm{Sim}(q, d^-))}$$

and combined with sparsity loss:

$L_{\text{total}} = L_{\text{retrieval}} + \lambda L_{\text{sparse}}$

This balances:

- **Retrieval accuracy** (semantic matching)
- **Sparsity** (efficiency & interpretability)

📖 5. Intuition — Lexical Expansion

The "Expansion" part means SPLADE can **expand a document or query** with *semantically related words* even if they don't appear literally.

Example:

Query: "smartphone price"

Document: "The cost of the iPhone dropped last week."

A BM25 system would miss this because *price ≠ cost*.

But SPLADE learns to expand "price" → "cost", "value", "expense", etc.

So in the sparse vector:

```
Query vector non-zeros → [smartphone, price, cost]
Doc vector non-zeros   → [iphone, cost, price]
```

Now, "cost" overlaps → document gets retrieved ✅

This expansion is **data-driven and contextual**, unlike manually crafted synonym dictionaries.

⚙️ 6. SPLADE Setup (Full Pipeline Summary)

Step	Component	Description
1	Transformer Encoder	Contextualizes tokens
2	Vocab Projection	Projects token embeddings → vocab dimension

Step	Component	Description
3	Pooling	Combines token vocab scores (max or log-sum-exp)
4	L_1 Regularization	Enforces sparsity
5	Dot Product	Sparse similarity between query & doc
6	ANN/Index	Retrieval via inverted index (Lucene, Elasticsearch)

💬 7. Word Impacts in SPLADE — What Does It Mean?

"**Word Impacts**" refers to **which words in the vocabulary receive non-zero weights** in the SPLADE representation of a query or document.

- Each word's weight = its **impact** (importance score)
- Words with higher weights influence retrieval more
- This impact distribution shows *which words the model thinks are important for meaning*

🧮 Mathematically:

$$\text{Impact}(w) = \log(1 + \text{ReLU}(z_w))$$

and only the top few impacts are non-zero → sparse "lexical fingerprint."

🧠 Interpretation:

- In a **query**, high-impact words are the ones SPLADE deems essential for matching documents.
- In a **document**, high-impact words capture its main semantic themes.

Example:

Query: "What is the fastest animal on land?"

→ SPLADE might give:

```
word:   impact:
what     0
```

```
fastest  1.2
animal   1.0
land     0.8
speed    0.6 (expanded)
```

So the query representation now includes *speed*, even though it wasn't typed — this is **semantic lexical expansion**.

⚡ 8. Advantages

✅ Works with **existing inverted indexes** (no ANN needed).

✅ **Interpretable** — we can inspect word impacts.

✅ **Sparse but semantic** — efficient and meaningful.

✅ **Combines** neural understanding with lexical precision.

✅ High performance in **MS MARCO** and **BEIR** benchmarks.

⚠️ 9. Limitations

⚠️ Higher **training complexity** (regularization tuning).

⚠️ Still needs **powerful transformers** → costly to train.

⚠️ "Expansion" might introduce **noisy tokens** if not regularized well.

⚠️ Doesn't handle cross-modal inputs (text-only).

💡 10. Variants & Extensions

Model	Description
SPLADE (Base)	Original BERT-based sparse expansion
SPLADE-v2	Improved regularization and pooling (log-sum-exp)
DistilSPLADE	Distilled lightweight version for real-time search
SPLADE++	Hybrid fusion with dense retrievers (for RAG systems)

🧾 11. Key Takeaways

- **SPLADE = Sparse Lexical and Expansion model**
- Bridges **lexical** and **semantic** retrieval.
- Represents queries/docs as **sparse vectors** in **vocabulary space**.
- Learns **which words to emphasize or expand** through *word impacts*.
- Enables efficient, interpretable, semantic retrieval via **inverted indexes**.
- Core building block for **hybrid RAG systems** (Lexical + Dense).

🧠 Training SPLADE Retrievers — Ranking Loss and Non-Sparsity Loss

---

🧩 1. Goal of Training

SPLADE is trained to **score relevant documents higher** than irrelevant ones

while keeping its **representation sparse** (so that it can use inverted indexes efficiently).

So it needs to optimize **two competing objectives**:

1️⃣ Make good retrieval decisions → **Ranking Loss**

2️⃣ Keep the model efficient and interpretable → **Sparsity Regularization (Non-sparsity Loss)**

Let's break these down.

⚙️ 2. Ranking Loss (a.k.a. Retrieval Loss)

💬 Intuition

When a query (q) is given, the model should give:

$$\text{score}(q, d^+) > \text{score}(q, d^-)$$

for all negative documents (d^-).

This is the **core ranking principle** — relevant docs should score higher than irrelevant ones.

📐 Formulation

SPLADE uses **contrastive learning** (similar to DPR, ColBERT):

D

where:

- ($\text{sim}(q, d) = \sum_w q_w \cdot d_w$)

  (dot product of sparse vectors over vocab words)
- (d^+): relevant (positive) document
- (d^-): irrelevant (negative) documents

This is essentially a **softmax ranking loss** —

encouraging the positive document's similarity to dominate over negatives.

🧩 Alternative: Margin Ranking Loss

Sometimes SPLADE variants also use:

$$L_{\text{rank}} = \max(0, m - \text{sim}(q, d^+) + \text{sim}(q, d^-))$$

where (m) is a margin hyperparameter (say, 0.2).

This pushes the positive document to be at least *m* more similar than any negative.

💡 Intuitive Analogy

Think of **ranking loss** as a "teacher" telling the model:

> "For query q, doc A is good, doc B is bad — make sure A scores higher than B!"

The model learns which *vocabulary activations (word impacts)* help achieve this.

⚙️ 3. Non-Sparsity Loss (L_1 Regularization)

💬 Why Needed?

Without constraint, the model might assign **non-zero scores to every vocabulary word**.

That makes retrieval:

- slow (huge index)
- uninterpretable
- memory-heavy

So we add a **non-sparsity penalty** to force the model to “focus” only on a few key words.

📐 Formulation

$L_{\text{sparse}} = \lambda \sum_w \text{avg}_B \left| \text{impact}(w) \right|$

or more precisely:

$L_{\text{sparse}} = \lambda \cdot \mathbb{E}x \in \text{batch} \sum w \log(1 + \text{ReLU}(z_{x,w}))$

where:

- (λ) = regularization coefficient (controls sparsity strength)
- ($z_{x,w}$) = score of vocabulary word *w* for query/document *x*
- This penalty increases when too many vocab entries are non-zero (dense).

⚖️ 4. Combined Objective

SPLADE balances both goals:

$L_{\text{total}} = L_{\text{rank}} + \lambda L_{\text{sparse}}$

So during training:

- **Ranking loss** → pulls relevant docs closer
- **Sparsity loss** → pushes unimportant vocab activations toward zero

🧠 5. Intuitive Visualization

```
     ┌──────────────────────────────┐
     │      Transformer        │
     │    (query / document)     │
     └──────────────┬───────────────┘
                │
         Project to Vocabulary
                │
        [ word_1, word_2, ..., word_N ]
                │
     ┌────────────────┴────────────────┐
     │                 │
 ↑ L₁ Loss penalizes        ↑ Ranking Loss encourages
 non-zero weights          positive > negative
```

The L_1 loss "snips off" small weights → sparse representation.

Ranking loss tunes *which* weights survive → relevant features remain.

📊 6. Effect of λ (Sparsity Coefficient)

λ Value	Effect
Small λ	Model keeps many non-zero tokens → less sparse but higher recall
Large λ	More sparsity → efficient, but risk of losing important terms
Balanced λ	Optimal trade-off between speed and accuracy

Tuning λ is **critical** — too much sparsity = lost meaning, too little = wasted memory.

🧩 7. Measuring Sparsity

Researchers evaluate SPLADE's sparsity by metrics like:

- **Non-zero ratio** (fraction of active tokens in vocab)
- **Index size** (smaller = more efficient)
- **Query expansion interpretability** (are expansions meaningful?)

⚡ 8. Training Summary Table

Component	Purpose	Formula	Intuition
Ranking Loss	Make positives score higher than negatives	$(-\log \frac{e^{sim(q,d^+)}}{\sum e^{sim(q,d)}})$	Learn what "relevance" looks like
Non-Sparsity (L_1) Loss	Enforce sparsity	$\lambda \sum_w \mathrm{avg}_B \left\|\mathrm{impact}(w)\right\|$	pushes unimportant vocab activations toward zero
Total Loss	Combine both	$(L_{total} = L_{rank} + \lambda L_{sparse})$	Trade-off between accuracy & efficiency

Goal	What it does	Why it's important
Ranking loss	Makes relevant docs score higher than irrelevant ones	So retrieval works
Sparsity loss (L1 loss)	Encourages fewer words to be active	So the representation stays efficient

💬 9. Example: What Happens in Practice

Query: "cheap wireless earbuds"

During training:

- Ranking loss pushes the model to make the correct docs ("affordable Bluetooth earphones") score higher.
- The model learns word expansions like:

```
cheap → affordable
wireless → bluetooth
earbuds → earphones
```

- Non-sparsity loss forces the model to **drop unhelpful activations** like:

```
"cheap" → ["buy", "deal", "great", ...]
```

if they don't add to retrieval relevance.

End result:

→ A **sparse**, **semantically rich**, **interpretable** representation.

🧾 10. Final Key Takeaways

✅ **Ranking Loss** teaches *what is relevant*

✅ **Non-Sparsity Loss** teaches *what to ignore*

✅ Together, they balance **semantic accuracy** and **index efficiency**

✅ This dual-loss setup makes SPLADE's retriever both **powerful** and **practical**