Day 4 Exercises: Vector Search, Semantic Search & Hybrid Search

Stack: `elk-ml` (8GB+ RAM) | **Duration:** ~85 minutes total | **Kibana Dev Tools:** `http://localhost:5601`

Prerequisites

- `elk-ml` stack running: `docker compose -f docker/elk-ml/docker-compose.yml --env-file .env up -d`
- Movies embeddings dataset loaded: `python data/load_data.py --dataset movies --with-embeddings`

> If RAM is limited, you can do Part A with `elk-single` (kNN works without ML nodes). Parts B-C require `elk-ml` for ELSER.

Part A: Vector Search Basics (5 tasks, ~20 min)

Task 1: 3D Vector Demo — Create Index (Basic)

Create a vector demo index and explore how similarity works with simple 3D vectors.

Step 1: Create the index:

```
PUT /vector-demo
{
  "mappings": {
    "properties": {
      "title": { "type": "text" },
      "description": { "type": "text" },
      "embedding": {
        "type": "dense_vector",
        "dims": 3,
        "index": true,
        "similarity": "cosine"
      }
    }
  }
}
```

Task 1: 3D Vector Demo — Index Documents (Basic)

Step 2: Index these documents:

```
POST /vector-demo/_bulk
{"index": {"_id": "1"}}
{"title": "Action Movie", "description": "Explosions and car chases", "embedding": [1.0, 0.2, 0.1]}
{"index": {"_id": "2"}}
{"title": "Romantic Comedy", "description": "Love and laughter", "embedding": [0.1, 1.0, 0.2]}
{"index": {"_id": "3"}}
{"title": "Sci-Fi Thriller", "description": "Space and suspense", "embedding": [0.8, 0.3, 0.9]}
{"index": {"_id": "4"}}
{"title": "Action Comedy", "description": "Funny action scenes", "embedding": [0.7, 0.8, 0.2]}
{"index": {"_id": "5"}}
{"title": "Horror Film", "description": "Scary and suspenseful", "embedding": [0.2, 0.1, 0.9]}
```

Task 1: 3D Vector Demo — Query & Questions (Basic)

Step 3: Run a kNN query with query vector `[0.9, 0.3, 0.2]`, k=3, num_candidates=10.

Questions:

1. What are the top 3 results? Why do they match?
2. Change the query vector to `[0.1, 0.9, 0.1]`. How do results change?

> **Hint:** The vector `[0.9, 0.3, 0.2]` is closest to `[1.0, 0.2, 0.1]` (Action Movie) by cosine similarity — both have a high first dimension. The vector `[0.1, 0.9, 0.1]` is closest to `[0.1, 1.0, 0.2]` (Romantic Comedy) — both have a high second dimension.

Task 2: Real Movie Embeddings kNN — Setup (Basic)

Using the `movies-embeddings` index (500 movies with 384-dim vectors):

1. Get the embedding from movie ID 1 (The Shawshank Redemption):

   ```
   GET /movies-embeddings/_doc/1
   ```

2. Copy the `overview_embedding` array from the response

Task 2: Real Movie Embeddings kNN — Query (Basic)

3. Use it as a kNN query vector to find the 5 most similar movies:

```
GET /movies-embeddings/_search
{
  "knn": {
    "field": "overview_embedding",
    "query_vector": [/* paste embedding here */],
    "k": 5,
    "num_candidates": 50
  },
  "_source": ["title", "overview", "genres", "vote_average"]
}
```

Task 2: Real Movie Embeddings kNN — Questions (Basic)

Questions:

1. Do the results make thematic sense? Are they similar to Shawshank?
2. Try with movie ID 8 (Interstellar). Are the neighbors space-related?

> **Hint:** The first result will be Shawshank itself (distance 0). The other results should be thematically similar — dramas about hardship, redemption, or prison. For Interstellar, expect sci-fi/space movies.

Task 3: Filtered kNN (Intermediate)

Using movie ID 1's embedding from Task 2, find similar movies but:

1. First, search **without** any filter (k=10)
2. Then, add a filter to only include `"Drama"` genre
3. Then, add a filter for `vote_average >= 7.5` AND genre `"Drama"`

Compare the results across all three queries.

Task 3: Filtered kNN (continued)

Hint: Add a `filter` block inside the `knn` object. For multiple conditions, use `bool.must` with an array of filters:

```
"filter": {
  "bool": {
    "must": [
      { "term": { "genres": "Drama" } },
      { "range": { "vote_average": { "gte": 7.5 } } }
    ]
  }
}
```

Task 4: num_candidates Effect (Intermediate)

Using the same query from Task 2, run the kNN search with different `num_candidates` values:

1. `num_candidates: 5`
2. `num_candidates: 50`
3. `num_candidates: 200`

Compare the results and the `took` time. At what point do results stabilize?

> **Hint:** With only 500 documents, differences will be small. In production with millions of docs, `num_candidates` has a significant impact. Lower values are faster but may miss relevant results. Higher values are more accurate but slower.

Task 5: Similarity Metrics Comparison (Bonus)

Create three indices with different similarity metrics (`cosine`, `l2_norm`, `dot_product`), each with 3 dimensions. Index the same 5 documents from Task 1 into all three.

Run the same kNN query on all three. Do the result rankings differ?

Questions:

1. When would you choose `l2_norm` over `cosine`?
2. What happens with `dot_product` if vectors aren't normalized?

> **Hint:** For text embeddings, `cosine` and `dot_product` (with normalized vectors) give identical rankings. `l2_norm` considers magnitude — two vectors pointing the same direction but with different lengths would be "far" in L2 but "identical" in cosine. Use `l2_norm` when magnitude carries meaning (e.g., image feature vectors).

Part B: ELSER & Semantic Search (4 tasks, ~20 min)

> These tasks require the `elk-ml` stack with ELSER deployed. If you can't deploy ELSER (RAM limitations), read through the tasks and study the expected outputs.

Task 6: Deploy ELSER — Setup (Basic)

Deploy the ELSER model and create an inference endpoint.

Option A: Via Kibana UI

1. Go to Machine Learning → Trained Models
2. Find `.elser_model_2_linux-x86_64`
3. Download and Deploy

Option B: Via API

```
PUT /_inference/sparse_embedding/my-elser-endpoint
{
  "service": "elser",
  "service_settings": {
    "num_allocations": 1,
    "num_threads": 1
  }
}
```

Task 6: Deploy ELSER — Verify (Basic)

Verify it works:

```
POST /_inference/sparse_embedding/my-elser-endpoint
{
  "input": "Two men escape from a maximum security prison"
}
```

Questions:

1. What does the output look like? (Sparse tokens with weights)
2. How many tokens were generated? Are they all words from the input?

Hint: ELSER generates **expanded** tokens — you'll see tokens that weren't in the original text but are semantically related. For "escape from prison", you might see tokens like "jail", "convict", "flee", "inmate" with various weights. This token expansion is how ELSER captures meaning.

Task 7: Create Semantic Index (Basic)

Create the `movies-semantic` index with a `semantic_text` field and index a few movies:

```
PUT /movies-semantic
{
  "mappings": {
    "properties": {
      "title": { "type": "text" },
      "overview": { "type": "text" },
      "overview_semantic": {
        "type": "semantic_text",
        "inference_id": "my-elser-endpoint"
      },
      "genres": { "type": "keyword" },
      "vote_average": { "type": "float" }
    }
  }
}
```

Task 7: Create Semantic Index (continued)

Index at least 5 movies (copy the overview text to both `overview` and `overview_semantic` fields). Use movies with diverse themes — e.g., Shawshank, Godfather, Inception, Interstellar, The Matrix.

> **Hint:** Use `POST /movies-semantic/_bulk` with both fields containing the same text. Indexing may take 5-10 seconds per document as ELSER generates embeddings. Check progress with `GET /movies-semantic/_count`.

Task 8: Keyword vs Semantic Comparison (Intermediate)

Run the same search two ways and compare:

Search term: `"movies about escaping from prison"`

1. Keyword search: `match` on `overview`
2. Semantic search: `semantic` on `overview_semantic`

Then try:

- `"films about family and power"`
- `"virtual reality simulations"`
- `"space exploration and survival"`

Task 8: Keyword vs Semantic (continued)

Questions:

1. Which approach finds "The Shawshank Redemption" for "escaping prison"? Why?
2. For which queries does keyword search work better? Semantic?

> **Hint:** Keyword search fails for "escaping from prison" because Shawshank's overview uses "imprisoned" and "redemption" (not "escaping" or "prison"). Semantic search understands these are related concepts. Keyword search works better for exact terms or proper nouns.

Task 9: Semantic with Filters (Intermediate)

Write a query that combines semantic search with traditional filters:

1. Semantic query: `"science fiction technology"`
2. Filter: `vote_average >= 8.0`
3. Filter: genre must be one of `["Action", "Science Fiction"]`

Use a `bool` query with `must` for semantic and `filter` for the constraints.

Task 9: Semantic with Filters (continued)

Hint:

```
{
  "query": {
    "bool": {
      "must": {
        "semantic": {
          "field": "overview_semantic",
          "query": "science fiction technology"
        }
      },
      "filter": [
        { "range": { "vote_average": { "gte": 8.0 } } },
        { "terms": { "genres": ["Action", "Science Fiction"] } }
      ]
    }
  }
}
```

Part C: Hybrid Search with RRF (5 tasks, ~25 min)

Task 10: BM25 vs kNN Side by Side (Basic)

Using the `movies-embeddings` index, run two separate searches for the concept of "space travel and survival":

1. **BM25 only:** `multi_match` on title and overview
2. **kNN only:** Use an embedding from a space movie (e.g., movie ID 8 — Interstellar)

Compare the top 5 results from each. Which movies appear in both lists? Which are unique to each?

> **Hint:** Get Interstellar's embedding with `GET /movies-embeddings/_doc/8`, then use it as `query_vector`. For BM25, use `multi_match` with `"fields": ["title^2", "overview"]`. Look for movies that appear in both result sets — these are likely the most relevant.

Task 11: Basic Hybrid with RRF — Query (Basic)

Combine BM25 + kNN into a single hybrid query using the Retriever API:

```
GET /movies-embeddings/_search
{
  "retriever": {
    "rrf": {
      "retrievers": [
        { "standard": { "query": {
              "multi_match": {
                "query": "space travel survival",
                "fields": ["title^2", "overview"]
              }
        }}},
        { "knn": {
            "field": "overview_embedding",
            "query_vector": [/* Interstellar's embedding */],
            "k": 10, "num_candidates": 50
        }}
      ],
      "rank_window_size": 100, "rank_constant": 60
    }
  },
  "_source": ["title", "overview", "genres"], "size": 10
}
```

Task 11: Basic Hybrid with RRF — Questions (Basic)

Questions:

1. How do the hybrid results compare to BM25-only and kNN-only?
2. Are there movies in the hybrid results that weren't in either individual result?

> **Hint:** Hybrid results typically surface documents that rank well in BOTH retrievers. A movie that's #3 in BM25 and #4 in kNN will score higher than one that's #1 in BM25 but #50 in kNN. This is the power of RRF.

Task 12: Filtered Hybrid Search (Intermediate)

Create a hybrid query for "crime family power" that:

1. Uses BM25 (`multi_match` on title and overview)
2. Uses kNN (get embedding from movie ID 2 — The Godfather)
3. Filters both retrievers to: `vote_average >= 7.0` AND genres contains `"Crime"` or `"Drama"`

Remember: filters must be applied to **each retriever separately**.

> **Hint:** For the standard retriever, wrap the multi_match in a `bool` query with `filter`. For the kNN retriever, add a `filter` block directly inside the kNN object. Both filters should have the same conditions.

Task 13: Tuning RRF Parameters (Intermediate)

Using the hybrid query from Task 11, experiment with different parameter combinations:

Experiment	rank_constant	rank_window_size
A	10	50
B	60	100
C	100	200

For each combination, note:

1. The top 5 results and their order
2. How the result ordering changes

Task 13: Tuning RRF Parameters (continued)

Questions:

1. What effect does lowering `rank_constant` have?
2. Does increasing `rank_window_size` change results with 500 documents?

> **Hint:** With `rank_constant=10`, the top-ranked results from each retriever get disproportionately high RRF scores — it emphasizes the #1 result much more than #10. With `rank_constant=100`, the difference between ranks is smaller, making the fusion more "democratic." With only 500 documents, `rank_window_size` changes may be subtle.

Task 14: Three-way Comparison Report (Bonus)

Pick a search query that's interesting to you (e.g., "underdog overcomes impossible odds").

Run three separate searches on `movies-embeddings`:

1. BM25 only (standard retriever)
2. kNN only (use a thematically related movie's embedding)
3. Hybrid (RRF combining both)

Create a comparison table of the top 5 results from each method. Write a brief analysis: which method gave the best results for your query and why?

> **Hint:** For "underdog overcomes impossible odds", BM25 might find movies with those exact words, kNN might find thematically similar movies (Rocky, Rudy, etc.), and hybrid should combine both. If kNN finds results that BM25 misses (because different words are used), that demonstrates the value of semantic search.

Part D: Advanced Techniques (4 tasks, ~20 min)

> These are bonus tasks for students who finish early or want to explore further.

Task 15: Quantized Index — Create Mapping (Intermediate)

Create a quantized version of the embeddings index:

```
PUT /movies-quantized
{
  "mappings": {
    "properties": {
      "title": { "type": "text" },
      "overview": { "type": "text" },
      "overview_embedding": {
        "type": "dense_vector",
        "dims": 384,
        "index": true,
        "similarity": "cosine",
        "index_options": { "type": "int8_hnsw", "m": 16, "ef_construction": 100 }
      },
      "genres": { "type": "keyword" },
      "vote_average": { "type": "float" }
    }
  }
}
```

Task 15: Quantized Index — Reindex & Compare (Intermediate)

Reindex from `movies-embeddings` into `movies-quantized`. Run the same kNN query on both indices and compare:

1. Are the results identical?
2. Compare index sizes: `GET /_cat/indices/movies-embeddings,movies-quantized?v`

> **Hint:** Use `POST /_reindex` to copy data. With only 500 documents, size and accuracy differences will be minimal. In production with millions of vectors, int8_hnsw saves ~75% memory with typically <5% accuracy loss.

Task 16: Approximate vs Exact kNN — Approximate (Intermediate)

Run the same similarity search two ways:

Approximate (HNSW):

```
GET /movies-embeddings/_search
{
  "knn": {
    "field": "overview_embedding",
    "query_vector": [/* movie 1 embedding */],
    "k": 10,
    "num_candidates": 100
  }
}
```

Task 16: Approximate vs Exact kNN — Exact (Intermediate)

Exact (script_score):

```
GET /movies-embeddings/_search
{
  "query": {
    "script_score": {
      "query": { "match_all": {} },
      "script": {
        "source": "cosineSimilarity(params.qv, 'overview_embedding') + 1.0",
        "params": {
          "qv": [/* same embedding */]
        }
      }
    }
  },
  "size": 10
}
```

Task 16: Approximate vs Exact kNN (continued)

Compare results and `took` time. Are the top 10 identical?

> **Hint:** With 500 documents, approximate kNN should return the exact same results as brute-force (the HNSW graph is small enough to explore thoroughly). The difference becomes significant at scale (millions of documents) where approximate kNN is orders of magnitude faster but may miss some true neighbors.

Task 17: Profile kNN Performance (Intermediate)

Add `"profile": true` to your kNN query to see execution details:

```
GET /movies-embeddings/_search
{
  "profile": true,
  "knn": {
    "field": "overview_embedding",
    "query_vector": [/* embedding */],
    "k": 10,
    "num_candidates": 100
  }
}
```

Task 17: Profile kNN Performance (continued)

Examine the profile output:

1. How long did the kNN search take?
2. What percentage of time was spent on the vector search vs other operations?

> **Hint:** The profile output shows detailed timing for each shard and query phase. Look for the `knn` section showing vector search time. With a small dataset, most time is overhead rather than actual vector comparison.

Task 18: Chunking Schema Design (Bonus)

Design (don't implement) an index mapping for a document search system where:

- Documents can be up to 50,000 words
- Each document belongs to a department and has a security level
- Users should be able to search semantically and filter by department/security
- Results should return the document (not individual chunks)

Write out:

1. The index mapping (with fields for document metadata + chunk data)
2. A sample kNN query with filtering and `collapse` for deduplication
3. How many chunks a 50,000-word document would produce (with 500-word chunks and 50-word overlap)

Task 18: Chunking Schema Design (continued)

> **Hint:** Mapping needs: `doc_id` (keyword), `doc_title` (text), `department` (keyword), `security_level` (keyword), `chunk_id` (integer), `chunk_text` (text), `chunk_embedding` (dense_vector). Use `collapse: {"field": "doc_id"}` in the search to get unique documents. A 50,000-word doc with 500-word chunks and 50-word overlap: ceil(50000 / 450) = ~112 chunks.

Cleanup

```
DELETE /vector-demo
DELETE /movies-semantic
DELETE /movies-quantized
DELETE /movies-hybrid

# Keep movies-embeddings if you want to continue experimenting
```

To stop the elk-ml stack:

```
docker compose -f docker/elk-ml/docker-compose.yml --env-file .env down -v
```