**Getting started with vector search**

This guide shows you how to use your own vectors in OpenSearch. You’ll learn to create a vector index, add location data, and run a vector search to find the nearest hotels on a coordinate plane. While this example uses two-dimensional vectors for simplicity, the same approach applies to higher-dimensional vectors used in semantic search and recommendation systems.

**Prerequisite: Install OpenSearch**

 If you don't have OpenSearch installed, follow these steps to create a cluster.

**Step 1: Create a vector index**

First, create an index that will store sample hotel data. To signal to OpenSearch that this is a vector index, set index.knn to true. You’ll store the vectors in a vector field named location. The vectors you’ll ingest will be two-dimensional, and the distance between vectors will be calculated using the [Euclidean l2 similarity metric](https://docs.opensearch.org/docs/latest/vector-search/getting-started/vector-search-basics/" \l "calculating-similarity):

PUT /hotels-index

{

"settings": {

"index.knn": true

},

"mappings": {

"properties": {

"location": {

"type": "knn\_vector",

"dimension": 2,

"space\_type": "l2"

}

}

}

}

CopyCopy as cURL

Vector queries usually have a size > 0, so by default they don’t enter the request cache. In OpenSearch 2.19 or later, if your workload mostly consists of vector queries, consider increasing the dynamic indices.requests.cache.maximum\_cacheable\_sizecluster setting to a larger value, such as 256. This allows queries with a size of up to 256 to enter the request cache, improving performance. For more information, see [Request cache](https://docs.opensearch.org/docs/latest/search-plugins/caching/request-cache).

**Step 2: Add data to your index**

Next, add data to your index. Each document represents a hotel. The location field in each document contains a two-dimensional vector specifying the hotel’s location:

POST /\_bulk

{ "index": { "\_index": "hotels-index", "\_id": "1" } }

{ "location": [5.2, 4.4] }

{ "index": { "\_index": "hotels-index", "\_id": "2" } }

{ "location": [5.2, 3.9] }

{ "index": { "\_index": "hotels-index", "\_id": "3" } }

{ "location": [4.9, 3.4] }

{ "index": { "\_index": "hotels-index", "\_id": "4" } }

{ "location": [4.2, 4.6] }

{ "index": { "\_index": "hotels-index", "\_id": "5" } }

{ "location": [3.3, 4.5] }

CopyCopy as cURL

**Step 3: Search your data**

Now search for hotels closest to the pin location [5, 4]. To search for the top three closest hotels, set k to 3:

POST /hotels-index/\_search

{

"size": 3,

"query": {

"knn": {

"location": {

"vector": [5, 4],

"k": 3

}

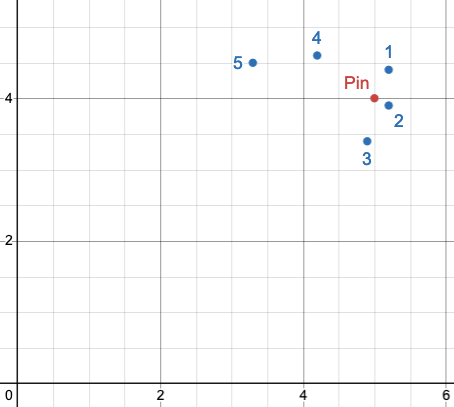
}

}

}

CopyCopy as cURL

The following image shows the hotels on the coordinate plane. The query point is labeled Pin, and each hotel is labeled with its document number.



The response contains the hotels closest to the specified pin location:

{

"took": 1093,

"timed\_out": false,

"\_shards": {

"total": 1,

"successful": 1,

"skipped": 0,

"failed": 0

},

"hits": {

"total": {

"value": 3,

"relation": "eq"

},

"max\_score": 0.952381,

"hits": [

{

"\_index": "hotels-index",

"\_id": "2",

"\_score": 0.952381,

"\_source": {

"location": [

5.2,

3.9

]

}

},

{

"\_index": "hotels-index",

"\_id": "1",

"\_score": 0.8333333,

"\_source": {

"location": [

5.2,

4.4

]

}

},

{

"\_index": "hotels-index",

"\_id": "3",

"\_score": 0.72992706,

"\_source": {

"location": [

4.9,

3.4

]

}

}

]

}

}

**Generating vector embeddings automatically**

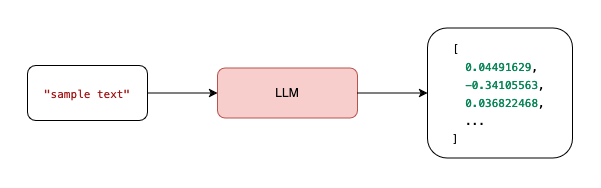
If your data isn’t already in vector format, you can generate vector embeddings directly within OpenSearch. This allows you to transform text or images into their numerical representations for similarity search. For more information, see [Generating vector embeddings automatically](https://docs.opensearch.org/docs/latest/vector-search/getting-started/auto-generated-embeddings/).

**Vector search basics**

*Vector search*, also known as *similarity search* or *nearest neighbor search*, is a powerful technique for finding items that are most similar to a given input. Use cases include semantic search to understand user intent, recommendations (for example, an “other songs you might like” feature in a music application), image recognition, and fraud detection. For more background information about vector search, see [Nearest neighbor search](https://en.wikipedia.org/wiki/Nearest_neighbor_search).

**Vector embeddings**

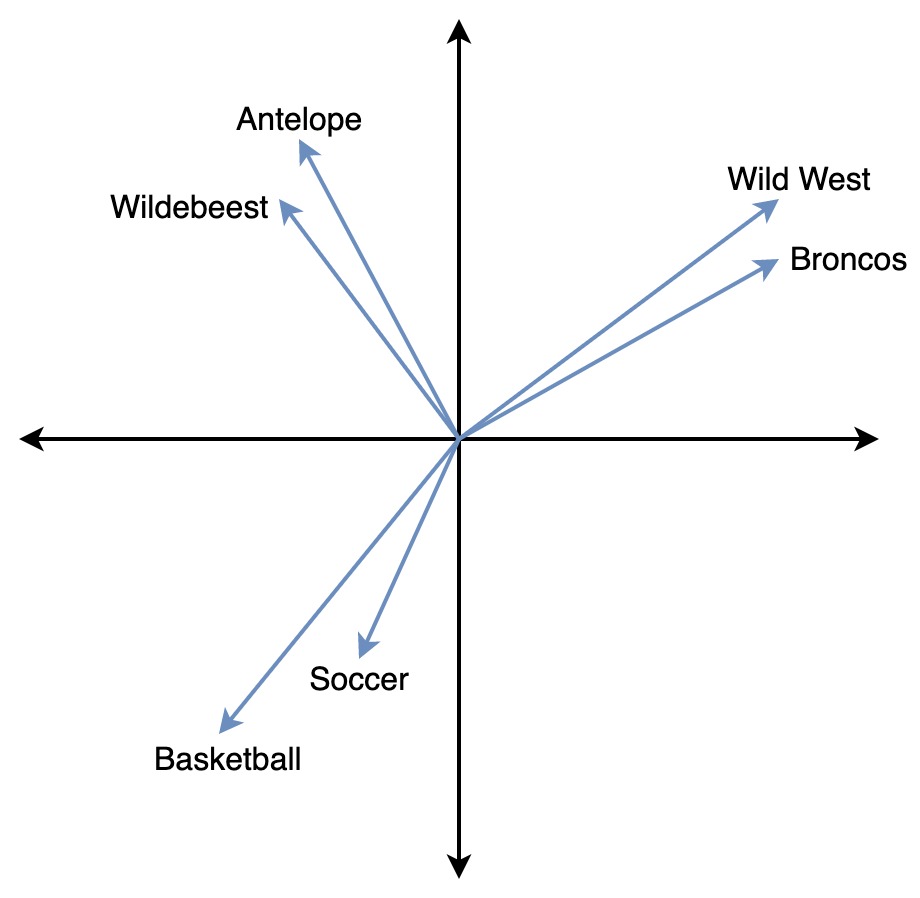
Unlike traditional search methods that rely on exact keyword matches, vector search uses *vector embeddings*—numerical representations of data such as text, images, or audio. These embeddings are stored as multi-dimensional vectors, capturing deeper patterns and similarities in meaning, context, or structure. For example, a large language model (LLM) can create vector embeddings from input text, as shown in the following image.



**Similarity search**

A vector embedding is a vector in a high-dimensional space. Its position and orientation capture meaningful relationships between objects. Vector search finds the most similar results by comparing a query vector to stored vectors and returning the closest matches. OpenSearch uses the [k-nearest neighbors (k-NN) algorithm](https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm) to efficiently identify the most similar vectors. Unlike keyword search, which relies on exact word matches, vector search measures similarity based on distance in this high-dimensional space.

In the following image, the vectors for Wild West and Broncos are closer to each other, while both are far from Basketball, reflecting their semantic differences.



To learn more about the types of vector search that OpenSearch supports, see [Vector search techniques](https://docs.opensearch.org/docs/latest/vector-search/vector-search-techniques/).

**Calculating similarity**

Vector similarity measures how close two vectors are in a multi-dimensional space, facilitating tasks like nearest neighbor search and ranking results by relevance. OpenSearch supports multiple distance metrics (*spaces*) for calculating vector similarity:

* **L1 (Manhattan distance):** Sums the absolute differences between vector components.
* **L2 (Euclidean distance):** Calculates the square root of the sum of squared differences, making it sensitive to magnitude.
* **L∞ (Chebyshev distance):** Considers only the maximum absolute difference between corresponding vector elements.
* **Cosine similarity:** Measures the angle between vectors, focusing on direction rather than magnitude.
* **Inner product:** Determines similarity based on vector dot products, which can be useful for ranking.
* **Hamming distance:** Counts differing elements in binary vectors.
* **Hamming bit:** Applies the same principle as Hamming distance but is optimized for binary-encoded data.

To learn more about the distance metrics, see [Spaces](https://docs.opensearch.org/docs/latest/field-types/supported-field-types/knn-spaces/).

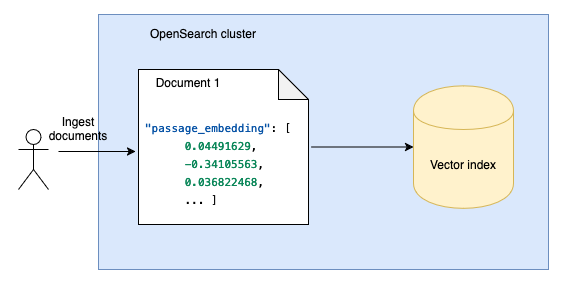
# Preparing vectors

In OpenSearch, you can either bring your own vectors or let OpenSearch generate them automatically from your data. Letting OpenSearch automatically generate your embeddings reduces data preprocessing effort at ingestion and search time.

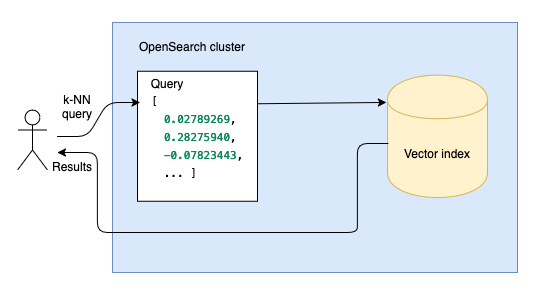
### Option 1: Bring your own raw vectors or generated embeddings

You already have pre-computed embeddings or raw vectors from external tools or services.

* **Ingestion**: Ingest pregenerated embeddings directly into OpenSearch.



* **Search**: Perform vector search to find the vectors that are closest to a query vector.



 Steps

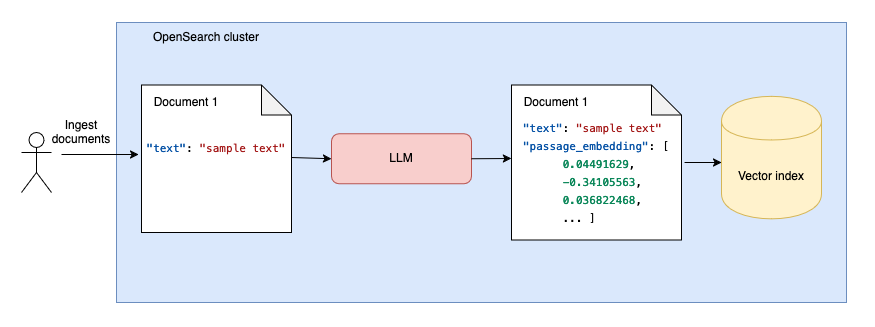
# **Getting started with vector search**

Use raw vectors or embeddings generated outside of OpenSearch

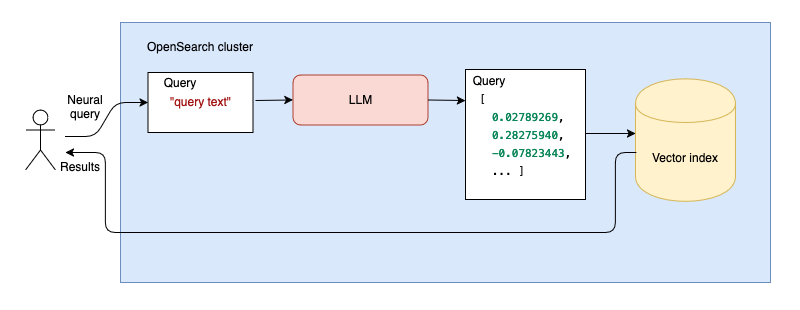
### Option 2: Generate embeddings within OpenSearch

Use this option to let OpenSearch automatically generate vector embeddings from your data using a machine learning (ML) model.

* **Ingestion**: You ingest plain data, and OpenSearch uses an ML model to generate embeddings dynamically.



* **Search**: At query time, OpenSearch uses the same ML model to convert your input data to embeddings, and these embeddings are used for vector search.



# Generating embeddings automatically

You can generate embeddings dynamically during ingestion within OpenSearch. This method provides a simplified workflow by converting data to vectors automatically.

OpenSearch can automatically generate embeddings from your text data using two approaches:

* [**Manual setup**](https://docs.opensearch.org/docs/latest/vector-search/getting-started/auto-generated-embeddings/#manual-setup) (Recommended for custom configurations): Configure each component individually for full control over the implementation.
* [**Automated workflow**](https://docs.opensearch.org/docs/latest/vector-search/getting-started/auto-generated-embeddings/#using-automated-workflows) (Recommended for quick setup): Use defaults and workflows for quick implementation with minimal configuration.

## Prerequisites

For this simple setup, you’ll use an OpenSearch-provided machine learning (ML) model and a cluster with no dedicated ML nodes. To ensure that this basic local setup works, send the following request to update ML-related cluster settings:

PUT \_cluster/settings

{

"persistent": {

"plugins.ml\_commons.only\_run\_on\_ml\_node": "false",

"plugins.ml\_commons.model\_access\_control\_enabled": "true",

"plugins.ml\_commons.native\_memory\_threshold": "99"

}

}

CopyCopy as cURL

### Choose an ML model

Generating embeddings automatically requires configuring a language model that will convert text to embeddings both at ingestion time and query time.

When selecting a model, you have the following options:

* Use a pretrained model provided by OpenSearch. For more information, see [OpenSearch-provided pretrained models](https://docs.opensearch.org/docs/latest/ml-commons-plugin/pretrained-models/).
* Upload your own model to OpenSearch. For more information, see [Custom local models](https://docs.opensearch.org/docs/latest/ml-commons-plugin/custom-local-models/).
* Connect to a foundation model hosted on an external platform. For more information, see [Connecting to remote models](https://docs.opensearch.org/docs/latest/ml-commons-plugin/remote-models/index/).

In this example, you’ll use the [DistilBERT](https://huggingface.co/docs/transformers/model_doc/distilbert) model from Hugging Face, which is one of the [pretrained models](https://docs.opensearch.org/docs/latest/ml-commons-plugin/pretrained-models/" \l "sentence-transformers) available in OpenSearch. For more information, see [Integrating ML models](https://docs.opensearch.org/docs/latest/ml-commons-plugin/integrating-ml-models/).

Take note of the dimensionality of the model because you’ll need it when you set up a vector index.

## Manual setup

For more control over the configuration, you can set up each component manually using the following steps.

### Step 1: Register and deploy the model

To register and deploy the model, send the following request:

POST /\_plugins/\_ml/models/\_register?deploy=true

{

"name": "huggingface/sentence-transformers/msmarco-distilbert-base-tas-b",

"version": "1.0.1",

"model\_format": "TORCH\_SCRIPT"

}

CopyCopy as cURL

Registering a model is an asynchronous task. OpenSearch returns a task ID for this task:

{

"task\_id": "aFeif4oB5Vm0Tdw8yoN7",

"status": "CREATED"

}

You can check the status of the task by using the Tasks API:

GET /\_plugins/\_ml/tasks/aFeif4oB5Vm0Tdw8yoN7

CopyCopy as cURL

Once the task is complete, the task state will change to COMPLETED and the Tasks API response will contain a model ID for the registered model:

{

"model\_id": "aVeif4oB5Vm0Tdw8zYO2",

"task\_type": "REGISTER\_MODEL",

"function\_name": "TEXT\_EMBEDDING",

"state": "COMPLETED",

"worker\_node": [

"4p6FVOmJRtu3wehDD74hzQ"

],

"create\_time": 1694358489722,

"last\_update\_time": 1694358499139,

"is\_async": true

}

You’ll need the model ID in order to use this model for several of the following steps.

### Step 2: Create an ingest pipeline

First, you need to create an [ingest pipeline](https://docs.opensearch.org/docs/latest/api-reference/ingest-apis/index/) that contains one processor: a task that transforms document fields before documents are ingested into an index. You’ll set up a text\_embedding processor that creates vector embeddings from text. You’ll need the model\_id of the model you set up in the previous section and a field\_map, which specifies the name of the field from which to take the text (text) and the name of the field in which to record embeddings (passage\_embedding):

PUT /\_ingest/pipeline/nlp-ingest-pipeline

{

"description": "An NLP ingest pipeline",

"processors": [

{

"text\_embedding": {

"model\_id": "aVeif4oB5Vm0Tdw8zYO2",

"field\_map": {

"text": "passage\_embedding"

}

}

}

]

}

CopyCopy as cURL

### Step 3: Create a vector index

Now you’ll create a vector index by setting index.knn to true. In the index, the field named text contains an image description, and a [knn\_vector](https://docs.opensearch.org/docs/latest/field-types/supported-field-types/knn-vector/) field named passage\_embedding contains the vector embedding of the text. The vector field dimension must match the dimensionality of the model you configured in Step 2. Additionally, set the default ingest pipeline to the nlp-ingest-pipeline you created in the previous step:

PUT /my-nlp-index

{

"settings": {

"index.knn": true,

"default\_pipeline": "nlp-ingest-pipeline"

},

"mappings": {

"properties": {

"passage\_embedding": {

"type": "knn\_vector",

"dimension": 768,

"space\_type": "l2"

},

"text": {

"type": "text"

}

}

}

}

CopyCopy as cURL

Setting up a vector index allows you to later perform a vector search on the passage\_embedding field.

### Step 4: Ingest documents into the index

In this step, you’ll ingest several sample documents into the index. The sample data is taken from the [Flickr image dataset](https://www.kaggle.com/datasets/hsankesara/flickr-image-dataset). Each document contains a text field corresponding to the image description and an id field corresponding to the image ID:

PUT /my-nlp-index/\_doc/1

{

"text": "A man who is riding a wild horse in the rodeo is very near to falling off ."

}

CopyCopy as cURL

PUT /my-nlp-index/\_doc/2

{

"text": "A rodeo cowboy , wearing a cowboy hat , is being thrown off of a wild white horse ."

}

CopyCopy as cURL

PUT /my-nlp-index/\_doc/3

{

"text": "People line the stands which advertise Freemont 's orthopedics , a cowboy rides a light brown bucking bronco ."

}

CopyCopy as cURL

### Step 5: Search the data

Now you’ll search the index using semantic search. To automatically generate vector embeddings from query text, use a neural query and provide the model ID of the model you set up earlier so that vector embeddings for the query text are generated with the model used at ingestion time:

GET /my-nlp-index/\_search

{

"\_source": {

"excludes": [

"passage\_embedding"

]

},

"query": {

"neural": {

"passage\_embedding": {

"query\_text": "wild west",

"model\_id": "aVeif4oB5Vm0Tdw8zYO2",

"k": 3

}

}

}

}

CopyCopy as cURL

The response contains the matching documents:

{

"took": 127,

"timed\_out": false,

"\_shards": {

"total": 1,

"successful": 1,

"skipped": 0,

"failed": 0

},

"hits": {

"total": {

"value": 3,

"relation": "eq"

},

"max\_score": 0.015851952,

"hits": [

{

"\_index": "my-nlp-index",

"\_id": "1",

"\_score": 0.015851952,

"\_source": {

"text": "A man who is riding a wild horse in the rodeo is very near to falling off ."

}

},

{

"\_index": "my-nlp-index",

"\_id": "2",

"\_score": 0.015177963,

"\_source": {

"text": "A rodeo cowboy , wearing a cowboy hat , is being thrown off of a wild white horse ."

}

},

{

"\_index": "my-nlp-index",

"\_id": "3",

"\_score": 0.011347729,

"\_source": {

"text": "People line the stands which advertise Freemont 's orthopedics , a cowboy rides a light brown bucking bronco ."

}

}

]

}

}

## Using automated workflows

You can quickly set up automatic embedding generation using [automated workflows](https://docs.opensearch.org/docs/latest/automating-configurations/). This approach automatically creates and provisions all necessary resources. For more information, see [Workflow templates](https://docs.opensearch.org/docs/latest/automating-configurations/workflow-templates/).

You can use automated workflows to create and deploy externally hosted models and create resources for various AI search types. In this example, you’ll create the same search you’ve already created following manual steps.

### Step 1: Register and deploy the model

To register and deploy a model, select the built-in workflow template for the model provider. For more information, see [Supported workflow templates](https://docs.opensearch.org/docs/latest/automating-configurations/workflow-templates/" \l "supported-workflow-templates). Alternatively, to configure a custom model, use [Step 1 of the manual setup](https://docs.opensearch.org/docs/latest/vector-search/getting-started/auto-generated-embeddings/" \l "step-1-register-and-deploy-the-model).

### Step 2: Configure a workflow

Create and provision a semantic search workflow. You must provide the model ID for the configured model. Review your selected workflow template [defaults](https://github.com/opensearch-project/flow-framework/blob/2.13/src/main/resources/defaults/semantic-search-defaults.json) to determine whether you need to update any of the parameters. For example, if the model dimensionality is different from the default (1024), specify the dimensionality of your model in the output\_dimension parameter. Change the workflow template default text field from passage\_text to text in order to match the manual example:

POST /\_plugins/\_flow\_framework/workflow?use\_case=semantic\_search&provision=true

{

"create\_ingest\_pipeline.model\_id" : "mBGzipQB2gmRjlv\_dOoB",

"text\_embedding.field\_map.output.dimension": "768",

"text\_embedding.field\_map.input": "text"

}

CopyCopy as cURL

OpenSearch responds with a workflow ID for the created workflow:

{

"workflow\_id" : "U\_nMXJUBq\_4FYQzMOS4B"

}

To check the workflow status, send the following request:

GET /\_plugins/\_flow\_framework/workflow/U\_nMXJUBq\_4FYQzMOS4B/\_status

CopyCopy as cURL

Once the workflow completes, the state changes to COMPLETED. The workflow has created an ingest pipeline and an index called my-nlp-index:

{

"workflow\_id": "U\_nMXJUBq\_4FYQzMOS4B",

"state": "COMPLETED",

"resources\_created": [

{

"workflow\_step\_id": "create\_ingest\_pipeline",

"workflow\_step\_name": "create\_ingest\_pipeline",

"resource\_id": "nlp-ingest-pipeline",

"resource\_type": "pipeline\_id"

},

{

"workflow\_step\_name": "create\_index",

"workflow\_step\_id": "create\_index",

"resource\_id": "my-nlp-index",

"resource\_type": "index\_name"

}

]

}

You can now continue with [steps 4 and 5](https://docs.opensearch.org/docs/latest/vector-search/getting-started/auto-generated-embeddings/#step-4-ingest-documents-into-the-index) to ingest documents into the index and search the index.

# Concepts

This page defines key terms and techniques related to vector search in OpenSearch.

## Vector representations

* [**Vector embeddings**](https://docs.opensearch.org/docs/latest/vector-search/getting-started/vector-search-basics/#vector-embeddings) are numerical representations of data—such as text, images, or audio—that encode meaning or features into a high-dimensional space. These embeddings enable similarity-based comparisons for search and machine learning (ML) tasks.
* **Dense vectors** are high-dimensional numerical representations where most elements have nonzero values. They are typically produced by deep learning models and are used in semantic search and ML applications.
* **Sparse vectors** contain mostly zero values and are often used in techniques like neural sparse search to efficiently represent and retrieve information.

## Vector search fundamentals

* [**Vector search**](https://docs.opensearch.org/docs/latest/vector-search/getting-started/vector-search-basics/), also known as similarity search or nearest neighbor search, is a technique for finding items that are most similar to a given input vector. It is widely used in applications such as recommendation systems, image retrieval, and natural language processing.
* A [**space**](https://docs.opensearch.org/docs/latest/vector-search/getting-started/vector-search-basics/#calculating-similarity) defines how similarity or distance between two vectors is measured. Different spaces use different distance metrics, such as Euclidean distance or cosine similarity, to determine how closely vectors resemble each other.
* A [**method**](https://docs.opensearch.org/docs/latest/field-types/supported-field-types/knn-methods-engines/) refers to the algorithm used to organize vector data during indexing and retrieve relevant results during search in approximate k-NN search. Different methods balance trade-offs between accuracy, speed, and memory usage.
* An [**engine**](https://docs.opensearch.org/docs/latest/field-types/supported-field-types/knn-methods-engines/) is the underlying library that implements vector search methods. It determines how vectors are indexed, stored, and retrieved during similarity search operations.

## k-NN search

* **k-nearest neighbors (k-NN) search** finds the k most similar vectors to a given query vector in an index. The similarity is determined based on a specified distance metric.
* [**Exact k-NN search**](https://docs.opensearch.org/docs/latest/vector-search/vector-search-techniques/knn-score-script/) performs a brute-force comparison between a query vector and all vectors in an index, computing the exact nearest neighbors. This approach provides high accuracy but can be computationally expensive for large datasets.
* [**Approximate k-NN search**](https://docs.opensearch.org/docs/latest/vector-search/vector-search-techniques/approximate-knn/) reduces computational complexity by using indexing techniques that speed up search operations while maintaining high accuracy. These methods restructure the index or reduce the dimensionality of vectors to improve performance.

## Query types

* A [**k-NN query**](https://docs.opensearch.org/docs/latest/query-dsl/specialized/k-nn/) searches vector fields using a query vector.
* A [**neural query**](https://docs.opensearch.org/docs/latest/query-dsl/specialized/neural/) searches vector fields using text or image data.
* A [**neural sparse query**](https://docs.opensearch.org/docs/latest/query-dsl/specialized/neural-sparse/) searches vector fields using raw text or sparse vector tokens.

## Search techniques

* [**Semantic search**](https://docs.opensearch.org/docs/latest/vector-search/ai-search/semantic-search/) interprets the intent and contextual meaning of a query rather than relying solely on exact keyword matches. This approach improves the relevance of search results, especially for natural language queries.
* [**Hybrid search**](https://docs.opensearch.org/docs/latest/vector-search/ai-search/hybrid-search/) combines lexical (keyword-based) search with semantic (vector-based) search to improve search relevance. This approach ensures that results include both exact keyword matches and conceptually similar content.
* [**Multimodal search**](https://docs.opensearch.org/docs/latest/vector-search/ai-search/multimodal-search/) enables you to search across multiple types of data, such as text and images. It allows queries in one format (for example, text) to retrieve results in another (for example, images).
* [**Radial search**](https://docs.opensearch.org/docs/latest/vector-search/specialized-operations/radial-search-knn/) retrieves all vectors within a specified distance or similarity threshold from a query vector. It is useful for tasks that require finding all relevant matches within a given range rather than retrieving a fixed number of nearest neighbors.
* [**Neural sparse search**](https://docs.opensearch.org/docs/latest/vector-search/ai-search/neural-sparse-search/) uses an inverted index, similar to BM25, to efficiently retrieve relevant documents based on sparse vector representations. This approach maintains the efficiency of traditional lexical search while incorporating semantic understanding.
* [**Conversational search**](https://docs.opensearch.org/docs/latest/vector-search/ai-search/conversational-search/) allows you to interact with a search system using natural language queries and refine results through follow-up questions. This approach enhances the user experience by making search more intuitive and interactive.
* [**Retrieval-augmented generation (RAG)**](https://docs.opensearch.org/docs/latest/vector-search/ai-search/conversational-search/#rag) enhances large language models (LLMs) by retrieving relevant information from an index and incorporating it into the model’s response. This approach improves the accuracy and relevance of generated text.

## Indexing and storage techniques

* [**Text chunking**](https://docs.opensearch.org/docs/latest/vector-search/ingesting-data/text-chunking/) involves splitting long documents or text passages into smaller segments to improve search retrieval and relevance. Chunking helps vector search models process large amounts of text more effectively.
* [**Vector quantization**](https://docs.opensearch.org/docs/latest/vector-search/optimizing-storage/knn-vector-quantization/) is a technique for reducing the storage size of vector embeddings by approximating them using a smaller set of representative vectors. This process enables efficient storage and retrieval in large-scale vector search applications.
* **Scalar quantization (SQ)** reduces vector precision by mapping floating-point values to a limited set of discrete values, decreasing memory requirements while preserving search accuracy.
* **Product quantization (PQ)** divides high-dimensional vectors into smaller subspaces and quantizes each subspace separately, enabling efficient approximate nearest neighbor search with reduced memory usage.
* **Binary quantization** compresses vector representations by converting numerical values to binary formats. This technique reduces storage requirements and accelerates similarity computations.
* [**Disk-based vector search**](https://docs.opensearch.org/docs/latest/vector-search/optimizing-storage/disk-based-vector-search/) stores vector embeddings on disk rather than in memory, using binary quantization to reduce memory consumption while maintaining search efficiency.

**Vector search techniques**

OpenSearch implements vector search as *k-nearest neighbors*, or *k-NN*, search. k-NN search finds the k neighbors closest to a query point across an index of vectors. To determine the neighbors, you can specify the space (the distance function) you want to use to measure the distance between points.

OpenSearch supports three different methods for obtaining the k-nearest neighbors from an index of vectors:

* [Approximate search](https://docs.opensearch.org/docs/latest/search-plugins/knn/approximate-knn/) (approximate k-NN, or ANN): Returns approximate nearest neighbors to the query vector. Usually, approximate search algorithms sacrifice indexing speed and search accuracy in exchange for performance benefits such as lower latency, smaller memory footprints, and more scalable search. For most use cases, approximate search is the best option.
* Exact search: A brute-force, exact k-NN search of vector fields. OpenSearch supports the following types of exact search:
  + [Exact search with a scoring script](https://docs.opensearch.org/docs/latest/search-plugins/knn/knn-score-script/): Using a scoring script, you can apply a filter to an index before executing the nearest neighbor search.
  + [Painless extensions](https://docs.opensearch.org/docs/latest/search-plugins/knn/painless-functions/): Adds the distance functions as Painless extensions that you can use in more complex combinations. You can use this method to perform a brute-force, exact vector search of an index, which also supports pre-filtering.

In general, you should choose the ANN method for larger datasets because it scales significantly better. For smaller datasets, where you may want to apply a filter, you should choose the custom scoring approach. If you have a more complex use case in which you need to use a distance function as part of the scoring method, you should use the Painless scripting approach.

**Approximate search**

OpenSearch supports multiple backend algorithms (*methods*) and libraries for implementing these algorithms (*engines*). It automatically selects the optimal configuration based on the chosen mode and available memory. For more information, see [Methods and engines](https://docs.opensearch.org/docs/latest/field-types/supported-field-types/knn-methods-engines/).

**Using sparse vectors**

*Neural sparse search* offers an efficient alternative to dense vector search by using sparse embedding models and inverted indexes, providing performance similar to BM25. Unlike dense vector methods that require significant memory and CPU resources, sparse search creates a list of token-weight pairs and stores them in a rank features index. This approach combines the efficiency of traditional search with the semantic understanding of neural networks. OpenSearch supports both automatic embedding generation through ingest pipelines and direct sparse vector ingestion. For more information, see [Neural sparse search](https://docs.opensearch.org/docs/latest/vector-search/ai-search/neural-sparse-search/).

**Combining multiple search techniques**

*Hybrid search* enhances search relevance by combining multiple search techniques in OpenSearch. It integrates traditional keyword search with vector-based semantic search. Through a configurable search pipeline, hybrid search normalizes and combines scores from different search methods to provide unified, relevant results. This approach is particularly effective for complex queries where both semantic understanding and exact matching are important. The search pipeline can be further customized with post-filtering operations and aggregations to meet specific search requirements. For more information, see [Hybrid search](https://docs.opensearch.org/docs/latest/vector-search/ai-search/hybrid-search/).

**Approximate k-NN search**

Standard k-nearest neighbors (k-NN) search methods compute similarity using a brute-force approach that measures the nearest distance between a query and a number of points, which produces exact results. This works well in many applications. However, in the case of extremely large datasets with high dimensionality, this creates a scaling problem that reduces the efficiency of the search. Approximate k-NN search methods can overcome this by employing tools that restructure indexes more efficiently and reduce the dimensionality of searchable vectors. Using this approach requires a sacrifice in accuracy but increases search processing speeds appreciably.

The approximate k-NN search methods in OpenSearch use approximate nearest neighbor (ANN) algorithms from the [NMSLIB](https://github.com/nmslib/nmslib), [Faiss](https://github.com/facebookresearch/faiss), and [Lucene](https://lucene.apache.org/) libraries to power k-NN search. These search methods employ ANN to improve search latency for large datasets. Of the three search methods OpenSearch provides, this method offers the best search scalability for large datasets. This approach is the preferred method when a dataset reaches hundreds of thousands of vectors.

For information about the algorithms OpenSearch supports, see [Methods and engines](https://docs.opensearch.org/docs/latest/field-types/supported-field-types/knn-methods-engines/).

OpenSearch builds a native library index of the vectors for each knn-vectorfield/Lucene segment pair during indexing, which can be used to efficiently find the k-nearest neighbors to a query vector during search. To learn more about Lucene segments, see the [Apache Lucene documentation](https://lucene.apache.org/core/8_9_0/core/org/apache/lucene/codecs/lucene87/package-summary.html#package.description). These native library indexes are loaded into native memory during search and managed by a cache. To learn more about preloading native library indexes into memory, see [Warmup API](https://docs.opensearch.org/docs/latest/vector-search/api/knn#warmup-operation). Additionally, you can see which native library indexes are already loaded into memory using the [Stats API](https://docs.opensearch.org/docs/latest/vector-search/api/knn#stats).

Because the native library indexes are constructed during indexing, it is not possible to apply a filter on an index and then use this search method. All filters are applied to the results produced by the ANN search.

**Get started with approximate k-NN**

To use the approximate search functionality, you must first create a vector index with index.knn set to true. This setting tells OpenSearch to create native library indexes for the index.

Next, you must add one or more fields of the knn\_vector data type. The following example creates an index with two knn\_vector fields using the faiss engine:

PUT my-knn-index-1

{

"settings": {

"index": {

"knn": true,

"knn.algo\_param.ef\_search": 100

}

},

"mappings": {

"properties": {

"my\_vector1": {

"type": "knn\_vector",

"dimension": 2,

"space\_type": "l2",

"method": {

"name": "hnsw",

"engine": "faiss",

"parameters": {

"ef\_construction": 128,

"m": 24

}

}

},

"my\_vector2": {

"type": "knn\_vector",

"dimension": 4,

"space\_type": "innerproduct",

"method": {

"name": "hnsw",

"engine": "faiss",

"parameters": {

"ef\_construction": 256,

"m": 48

}

}

}

}

}

}

CopyCopy as cURL

In the preceding example, both knn\_vector fields are configured using method definitions. Additionally, knn\_vector fields can be configured using models. For more information, see [k-NN vector](https://docs.opensearch.org/docs/latest/field-types/supported-field-types/knn-vector/).

The knn\_vector data type supports a vector of floats that can have a dimension count of up to 16,000 for the NMSLIB, Faiss, and Lucene engines, as set by the dimensionmapping parameter.

In OpenSearch, codecs handle the storage and retrieval of indexes. OpenSearch uses a custom codec to write vector data to native library indexes so that the underlying k-NN search library can read it.

After you create the index, you can add some data to it:

POST \_bulk

{ "index": { "\_index": "my-knn-index-1", "\_id": "1" } }

{ "my\_vector1": [1.5, 2.5], "price": 12.2 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "2" } }

{ "my\_vector1": [2.5, 3.5], "price": 7.1 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "3" } }

{ "my\_vector1": [3.5, 4.5], "price": 12.9 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "4" } }

{ "my\_vector1": [5.5, 6.5], "price": 1.2 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "5" } }

{ "my\_vector1": [4.5, 5.5], "price": 3.7 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "6" } }

{ "my\_vector2": [1.5, 5.5, 4.5, 6.4], "price": 10.3 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "7" } }

{ "my\_vector2": [2.5, 3.5, 5.6, 6.7], "price": 5.5 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "8" } }

{ "my\_vector2": [4.5, 5.5, 6.7, 3.7], "price": 4.4 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "9" } }

{ "my\_vector2": [1.5, 5.5, 4.5, 6.4], "price": 8.9 }

CopyCopy as cURL

Then you can run an ANN search on the data using the knn query type:

GET my-knn-index-1/\_search

{

"size": 2,

"query": {

"knn": {

"my\_vector2": {

"vector": [2, 3, 5, 6],

"k": 2

}

}

}

}

CopyCopy as cURL

**The number of returned results**

In the preceding query, k represents the number of neighbors returned by the search of each graph. You must also include the size parameter, indicating the final number of results that you want the query to return.

For the NMSLIB and Faiss engines, k represents the maximum number of documents returned for all segments of a shard. For the Lucene engine, k represents the number of documents returned for a shard. The maximum value of k is 10,000.

For any engine, each shard returns size results to the coordinator node. Thus, the total number of results that the coordinator node receives is size \* number of shards. After the coordinator node consolidates the results received from all nodes, the query returns the top size results.

The following table provides examples of the number of results returned by various engines in several scenarios. For these examples, assume that the number of documents contained in the segments and shards is sufficient to return the number of results specified in the table.

| size | k | **Number of primary shards** | **Number of segments per shard** | **Number of returned results, Faiss/NMSLIB** | **Number of returned results, Lucene** |
| --- | --- | --- | --- | --- | --- |
| 10 | 1 | 1 | 4 | 4 | 1 |
| 10 | 10 | 1 | 4 | 10 | 10 |
| 10 | 1 | 2 | 4 | 8 | 2 |

The number of results returned by Faiss/NMSLIB differs from the number of results returned by Lucene only when k is smaller than size. If k and size are equal, all engines return the same number of results.

Starting in OpenSearch 2.14, you can use k, min\_score, or max\_distance for [radial search](https://docs.opensearch.org/docs/latest/search-plugins/knn/radial-search-knn/).

**Building a vector index from a model**

For some of the algorithms that OpenSearch supports, the native library index needs to be trained before it can be used. It would be expensive to train every newly created segment, so, instead, OpenSearch features the concept of a *model* that initializes the native library index during segment creation. You can create a model by calling the [Train API](https://docs.opensearch.org/docs/latest/vector-search/api/knn#train-a-model) and passing in the source of the training data and the method definition of the model. Once training is complete, the model is serialized to a k-NN model system index. Then, during indexing, the model is pulled from that index to initialize the segments.

To train a model, you first need an OpenSearch index containing training data. Training data can come from any knn\_vector field that has a dimension matching the dimension of the model you want to create. Training data can be the same as the data you plan to index or come from a separate dataset. To create a training index, send the following request:

PUT /train-index

{

"settings": {

"number\_of\_shards": 3,

"number\_of\_replicas": 0

},

"mappings": {

"properties": {

"train-field": {

"type": "knn\_vector",

"dimension": 4

}

}

}

}

CopyCopy as cURL

Notice that index.knn is not set in the index settings. This ensures that you do not create native library indexes for this index.

You can now add some data to the index:

POST \_bulk

{ "index": { "\_index": "train-index", "\_id": "1" } }

{ "train-field": [1.5, 5.5, 4.5, 6.4]}

{ "index": { "\_index": "train-index", "\_id": "2" } }

{ "train-field": [2.5, 3.5, 5.6, 6.7]}

{ "index": { "\_index": "train-index", "\_id": "3" } }

{ "train-field": [4.5, 5.5, 6.7, 3.7]}

{ "index": { "\_index": "train-index", "\_id": "4" } }

{ "train-field": [1.5, 5.5, 4.5, 6.4]}

CopyCopy as cURL

After completing indexing into the training index, you can call the Train API:

POST /\_plugins/\_knn/models/my-model/\_train

{

"training\_index": "train-index",

"training\_field": "train-field",

"dimension": 4,

"description": "My model description",

"method": {

"name": "ivf",

"engine": "faiss",

"parameters": {

"encoder": {

"name": "pq",

"parameters": {

"code\_size": 2,

"m": 2

}

}

}

}

}

CopyCopy as cURL

For more information about the method parameters, see [IVF training requirements](https://docs.opensearch.org/docs/latest/field-types/supported-field-types/knn-methods-engines/#ivf-training-requirements).

The Train API returns as soon as the training job is started. To check the job status, use the Get Model API:

GET /\_plugins/\_knn/models/my-model?filter\_path=state&pretty

{

"state": "training"

}

CopyCopy as cURL

Once the model enters the created state, you can create an index that will use this model to initialize its native library indexes:

PUT /target-index

{

"settings": {

"number\_of\_shards": 3,

"number\_of\_replicas": 1,

"index.knn": true

},

"mappings": {

"properties": {

"target-field": {

"type": "knn\_vector",

"model\_id": "my-model"

}

}

}

}

CopyCopy as cURL

Lastly, you can add the documents you want to be searched to the index:

POST \_bulk

{ "index": { "\_index": "target-index", "\_id": "1" } }

{ "target-field": [1.5, 5.5, 4.5, 6.4]}

{ "index": { "\_index": "target-index", "\_id": "2" } }

{ "target-field": [2.5, 3.5, 5.6, 6.7]}

{ "index": { "\_index": "target-index", "\_id": "3" } }

{ "target-field": [4.5, 5.5, 6.7, 3.7]}

{ "index": { "\_index": "target-index", "\_id": "4" } }

{ "target-field": [1.5, 5.5, 4.5, 6.4]}

CopyCopy as cURL

After data is ingested, it can be searched in the same way as any other knn\_vectorfield.

**Exact k-NN search with a scoring script**

You can use exact k-nearest neighbors (k-NN) search with a scoring script to find the exact k-nearest neighbors to a given query point. Using the k-NN scoring script, you can apply a filter on an index before executing the nearest neighbor search. This is useful for dynamic search use cases, where the index body may vary based on other conditions.

Because the scoring script approach executes a brute force search, it doesn’t scale as efficiently as the [approximate approach](https://docs.opensearch.org/docs/latest/search-plugins/knn/approximate-knn/). In some cases, it might be better to consider refactoring your workflow or index structure to use the approximate approach instead of the scoring script approach.

**Getting started with the scoring script for vectors**

Similarly to approximate nearest neighbor (ANN) search, in order to use the scoring script on a body of vectors, you must first create an index with one or more knn\_vector fields.

If you intend to only use the scoring script approach (and not the approximate approach), you can set index.knn to false and not set index.knn.space\_type. You can choose the space type during search. For the spaces that the k-NN scoring script supports, see [Spaces](https://docs.opensearch.org/docs/latest/field-types/supported-field-types/knn-spaces/).

This example creates an index with two knn\_vector fields:

PUT my-knn-index-1

{

"mappings": {

"properties": {

"my\_vector1": {

"type": "knn\_vector",

"dimension": 2

},

"my\_vector2": {

"type": "knn\_vector",

"dimension": 4

}

}

}

}

CopyCopy as cURL

If you want to *only* use the scoring script, you can omit "index.knn": true. This approach leads to faster indexing speed and lower memory usage, but you lose the ability to run standard k-NN queries on the index.

After you create the index, you can add some data to it:

POST \_bulk

{ "index": { "\_index": "my-knn-index-1", "\_id": "1" } }

{ "my\_vector1": [1.5, 2.5], "price": 12.2 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "2" } }

{ "my\_vector1": [2.5, 3.5], "price": 7.1 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "3" } }

{ "my\_vector1": [3.5, 4.5], "price": 12.9 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "4" } }

{ "my\_vector1": [5.5, 6.5], "price": 1.2 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "5" } }

{ "my\_vector1": [4.5, 5.5], "price": 3.7 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "6" } }

{ "my\_vector2": [1.5, 5.5, 4.5, 6.4], "price": 10.3 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "7" } }

{ "my\_vector2": [2.5, 3.5, 5.6, 6.7], "price": 5.5 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "8" } }

{ "my\_vector2": [4.5, 5.5, 6.7, 3.7], "price": 4.4 }

{ "index": { "\_index": "my-knn-index-1", "\_id": "9" } }

{ "my\_vector2": [1.5, 5.5, 4.5, 6.4], "price": 8.9 }

CopyCopy as cURL

Finally, you can run an exact nearest neighbor search on the data using the knn script:

GET my-knn-index-1/\_search

{

"size": 4,

"query": {

"script\_score": {

"query": {

"match\_all": {}

},

"script": {

"source": "knn\_score",

"lang": "knn",

"params": {

"field": "my\_vector2",

"query\_value": [2.0, 3.0, 5.0, 6.0],

"space\_type": "cosinesimil"

}

}

}

}

}

CopyCopy as cURL

All parameters are required.

* lang is the script type. This value is usually painless, but here you must specify knn.
* source is the name of the script, knn\_score.

This script is part of the k-NN plugin and isn’t available at the standard \_scriptspath. A GET request to \_cluster/state/metadata doesn’t return it, either.

* field is the field that contains your vector data.
* query\_value is the point you want to find the nearest neighbors for. For the Euclidean and cosine similarity spaces, the value must be an array of floats that matches the dimension set in the field’s mapping. For Hamming bit distance, this value can be either of type signed long or a base64-encoded string (for the long and binary field types, respectively).
* space\_type corresponds to the distance function. For more information, see [Spaces](https://docs.opensearch.org/docs/latest/field-types/supported-field-types/knn-spaces/).

The [post filter example in the approximate approach](https://docs.opensearch.org/docs/latest/vector-search/filter-search-knn/) shows a search that returns fewer than k results. If you want to avoid this, the scoring script method lets you essentially invert the order of events. In other words, you can filter the set of documents on which to execute the k-NN search.

This example shows a pre-filter approach to k-NN search with the scoring script approach. First, create the index:

PUT my-knn-index-2

{

"mappings": {

"properties": {

"my\_vector": {

"type": "knn\_vector",

"dimension": 2

},

"color": {

"type": "keyword"

}

}

}

}

CopyCopy as cURL

Then add some documents:

POST \_bulk

{ "index": { "\_index": "my-knn-index-2", "\_id": "1" } }

{ "my\_vector": [1, 1], "color" : "RED" }

{ "index": { "\_index": "my-knn-index-2", "\_id": "2" } }

{ "my\_vector": [2, 2], "color" : "RED" }

{ "index": { "\_index": "my-knn-index-2", "\_id": "3" } }

{ "my\_vector": [3, 3], "color" : "RED" }

{ "index": { "\_index": "my-knn-index-2", "\_id": "4" } }

{ "my\_vector": [10, 10], "color" : "BLUE" }

{ "index": { "\_index": "my-knn-index-2", "\_id": "5" } }

{ "my\_vector": [20, 20], "color" : "BLUE" }

{ "index": { "\_index": "my-knn-index-2", "\_id": "6" } }

{ "my\_vector": [30, 30], "color" : "BLUE" }

CopyCopy as cURL

Finally, use the script\_score query to pre-filter your documents before identifying nearest neighbors:

GET my-knn-index-2/\_search

{

"size": 2,

"query": {

"script\_score": {

"query": {

"bool": {

"filter": {

"term": {

"color": "BLUE"

}

}

}

},

"script": {

"lang": "knn",

"source": "knn\_score",

"params": {

"field": "my\_vector",

"query\_value": [9.9, 9.9],

"space\_type": "l2"

}

}

}

}

}

CopyCopy as cURL

**Getting started with the scoring script for binary data**

The k-NN scoring script also allows you to run k-NN search on your binary data with the Hamming distance space. In order to use Hamming distance, the field of interest must have either a binary or long field type. If you’re using binary type, the data must be a base64-encoded string.

This example shows how to use the Hamming distance space with a binary field type:

PUT my-index

{

"mappings": {

"properties": {

"my\_binary": {

"type": "binary",

"doc\_values": true

},

"color": {

"type": "keyword"

}

}

}

}

CopyCopy as cURL

Then add some documents:

POST \_bulk

{ "index": { "\_index": "my-index", "\_id": "1" } }

{ "my\_binary": "SGVsbG8gV29ybGQh", "color" : "RED" }

{ "index": { "\_index": "my-index", "\_id": "2" } }

{ "my\_binary": "ay1OTiBjdXN0b20gc2NvcmluZyE=", "color" : "RED" }

{ "index": { "\_index": "my-index", "\_id": "3" } }

{ "my\_binary": "V2VsY29tZSB0byBrLU5O", "color" : "RED" }

{ "index": { "\_index": "my-index", "\_id": "4" } }

{ "my\_binary": "SSBob3BlIHRoaXMgaXMgaGVscGZ1bA==", "color" : "BLUE" }

{ "index": { "\_index": "my-index", "\_id": "5" } }

{ "my\_binary": "QSBjb3VwbGUgbW9yZSBkb2NzLi4u", "color" : "BLUE" }

{ "index": { "\_index": "my-index", "\_id": "6" } }

{ "my\_binary": "TGFzdCBvbmUh", "color" : "BLUE" }

CopyCopy as cURL

Finally, use the script\_score query to pre-filter your documents before identifying nearest neighbors:

GET my-index/\_search

{

"size": 2,

"query": {

"script\_score": {

"query": {

"bool": {

"filter": {

"term": {

"color": "BLUE"

}

}

}

},

"script": {

"lang": "knn",

"source": "knn\_score",

"params": {

"field": "my\_binary",

"query\_value": "U29tZXRoaW5nIEltIGxvb2tpbmcgZm9y",

"space\_type": "hammingbit"

}

}

}

}

}

CopyCopy as cURL

Similarly, you can encode your data with the long field and run a search:

GET my-long-index/\_search

{

"size": 2,

"query": {

"script\_score": {

"query": {

"bool": {

"filter": {

"term": {

"color": "BLUE"

}

}

}

},

"script": {

"lang": "knn",

"source": "knn\_score",

"params": {

"field": "my\_long",

"query\_value": 23,

"space\_type": "hammingbit"

}

}

}

}

}

CopyCopy as cURL

**Painless scripting extensions**

With Painless scripting extensions, you can use k-nearest neighbors (k-NN) distance functions directly in your Painless scripts to perform operations on knn\_vector fields. Painless has a strict list of allowed functions and classes per context to ensure that its scripts are secure. OpenSearch adds Painless scripting extensions to a few of the distance functions used in [k-NN scoring script](https://docs.opensearch.org/docs/latest/search-plugins/knn/knn-score-script/), so you can use them to customize your k-NN workload.

**Get started with k-NN Painless scripting functions**

To use k-NN Painless scripting functions, first create an index with knn\_vector fields, as described in [Getting started with the scoring script for vectors](https://docs.opensearch.org/docs/latest/search-plugins/knn/knn-score-script#getting-started-with-the-scoring-script-for-vectors). Once you have created the index and ingested some data, you can use Painless extensions:

GET my-knn-index-2/\_search

{

"size": 2,

"query": {

"script\_score": {

"query": {

"bool": {

"filter": {

"term": {

"color": "BLUE"

}

}

}

},

"script": {

"source": "1.0 + cosineSimilarity(params.query\_value, doc[params.field])",

"params": {

"field": "my\_vector",

"query\_value": [9.9, 9.9]

}

}

}

}

}

CopyCopy as cURL

field needs to map to a knn\_vector field, and query\_value must be a floating-point array with the same dimension as field.

**Function types**

The following table describes the Painless functions OpenSearch provides.

| **Function name** | **Function signature** | **Description** |
| --- | --- | --- |
| l2Squared | float l2Squared (float[] queryVector, doc['vector field']) | This function calculates the square of the L2 distance (Euclidean distance) between a given query vector and document vectors. A shorter distance indicates a more relevant document, so this example inverts the return value of the l2Squared function. If the document vector matches the query vector, the result is 0, so this example also adds 1 to the distance to avoid divide-by-zero errors. |
| l1Norm | float l1Norm (float[] queryVector, doc['vector field']) | This function calculates the L1 norm distance (Manhattan distance) between a given query vector and document vectors. |
| cosineSimilarity | float cosineSimilarity (float[] queryVector, doc['vector field']) | Cosine similarity is an inner product of the query vector and document vector normalized to both have a length of 1. If the magnitude of the query vector doesn’t change throughout the query, you can pass the magnitude of the query vector to improve performance instead of repeatedly calculating the magnitude for every filtered document: float cosineSimilarity (float[] queryVector, doc['vector field'], float normQueryVector)  In general, the range of cosine similarity is [-1, 1]. However, in the case of information retrieval, the cosine similarity of two documents ranges from 0 to 1 because the tf-idf statistic can’t be negative. Therefore, OpenSearch adds 1.0 in order to always yield a positive cosine similarity score. |
| hamming | float hamming (float[] queryVector, doc['vector field']) | This function calculates the Hamming distance between a given query vector and document vectors. The Hamming distance is the number of positions at which the corresponding elements are different. A shorter distance indicates a more relevant document, so this example inverts the return value of the Hamming distance. |

The hamming space type is supported for binary vectors in OpenSearch version 2.16 and later. For more information, see [Binary k-NN vectors](https://docs.opensearch.org/docs/latest/field-types/supported-field-types/knn-memory-optimized#binary-vectors).

**Constraints**

1. If a document’s knn\_vector field has different dimensions than the query, the function throws an IllegalArgumentException.
2. If a vector field doesn’t have a value, the function throws an IllegalStateException.

You can avoid this by first checking whether a document contains a value in its field:

"source": "doc[params.field].size() == 0 ? 0 : 1 / (1 + l2Squared(params.query\_value, doc[params.field]))",

Because scores can only be positive, this script ranks documents with vector fields higher than those without vector fields.

When using cosine similarity, it is not valid to pass a zero vector ([0, 0, ...]) as input. This is because the magnitude of such a vector is 0, which raises a divide by 0 exception in the corresponding formula. Requests containing the zero vector will be rejected, and a corresponding exception will be thrown.

**Creating a vector index**

Creating a vector index in OpenSearch involves a common core process with some variations depending on the type of vector search. This guide outlines the key elements shared across all vector indexes and the differences specific to supported use cases.

Before you start, review the options for generating embeddings to help you decide on the option suitable for your use case. For more information, see [Preparing vectors](https://docs.opensearch.org/docs/latest/vector-search/getting-started/vector-search-options/).

**Overview**

To create a vector index, set the index.knn parameter to truein the settings:

PUT /test-index

{

"settings": {

"index.knn": true

},

"mappings": {

"properties": {

"my\_vector": {

"type": "knn\_vector",

"dimension": 3,

"space\_type": "l2",

"mode": "on\_disk",

"method": {

"name": "hnsw"

}

}

}

}

}

CopyCopy as cURL

Creating a vector index involves the following key steps:

1. **Enable k-nearest neighbors (k-NN) search**: Set index.knn to true in the index settings to enable k-NN search functionality.
2. **Define a vector field**: Specify the field that will store the vector data. When defining a knn\_vector field in OpenSearch, you can select from different data types to balance storage requirements and performance. By default, k-NN vectors are float vectors, but you can also choose byte or binary vectors for more efficient storage. For more information, see [k-NN vector](https://docs.opensearch.org/docs/latest/field-types/supported-field-types/knn-vector/).
3. **Specify the dimension**: Set the dimension property to match the size of the vectors used.
4. (Optional) **Choose a space type**: Select a distance metric for similarity comparisons, such as l2 (Euclidean distance) or cosinesimil. For more information, see [Spaces](https://docs.opensearch.org/docs/latest/field-types/supported-field-types/knn-spaces/).
5. (Optional) **Select a workload mode and/or compression level**: Select a workload mode and/or compression level in order to optimize vector storage. For more information, see [Optimizing vector storage](https://docs.opensearch.org/docs/latest/vector-search/optimizing-storage/).
6. (Optional, advanced) **Select a method**: Configure the indexing method, such as HNSW or IVF, used to optimize vector search performance. For more information, see [Methods and engines](https://docs.opensearch.org/docs/latest/field-types/supported-field-types/knn-methods-engines/).

**Implementation options**

Based on your vector generation approach, choose one of the following implementation options:

* [Store raw vectors or embeddings generated outside of OpenSearch](https://docs.opensearch.org/docs/latest/vector-search/creating-vector-index/#storing-raw-vectors-or-embeddings-generated-outside-of-opensearch): Ingest pregenerated embeddings or raw vectors into your index for raw vector search.
* [Convert data to embeddings during ingestion](https://docs.opensearch.org/docs/latest/vector-search/creating-vector-index/#converting-data-to-embeddings-during-ingestion): Ingest text that will be converted into vector embeddings in OpenSearch in order to perform semantic search using machine learning (ML) models.

The following table summarizes key index configuration differences for the supported use cases.

| **Feature** | **Vector field type** | **Ingest pipeline** | **Transformation** | **Use case** |
| --- | --- | --- | --- | --- |
| **Store raw vectors or embeddings generated outside of OpenSearch** | [knn\_vector](https://docs.opensearch.org/docs/latest/field-types/supported-field-types/knn-vector/) | Not required | Direct ingestion | Raw vector search |
| **Convert data to embeddings during ingestion** | [knn\_vector](https://docs.opensearch.org/docs/latest/field-types/supported-field-types/knn-vector/) | Required | Auto-generated vectors | AI search   Automating embedding generation reduces data preprocessing and provides a more managed vector search experience. |

**Storing raw vectors or embeddings generated outside of OpenSearch**

To ingest raw vectors into an index, configure a vector field (in this request, my\_vector) and specify its dimension:

PUT /my-raw-vector-index

{

"settings": {

"index.knn": true

},

"mappings": {

"properties": {

"my\_vector": {

"type": "knn\_vector",

"dimension": 3

}

}

}

}

CopyCopy as cURL

**Converting data to embeddings during ingestion**

To automatically generate embeddings during ingestion, configure an [ingest pipeline](https://docs.opensearch.org/docs/latest/api-reference/ingest-apis/index/)with the model ID of the embedding model. For more information about configuring a model, see [Integrating ML models](https://docs.opensearch.org/docs/latest/ml-commons-plugin/integrating-ml-models/).

Specify the field\_map to define the source field for input text and the target field for storing embeddings. In this example, text from the text field is converted into embeddings and stored in passage\_embedding:

PUT /\_ingest/pipeline/auto-embed-pipeline

{

"description": "AI search ingest pipeline that automatically converts text to embeddings",

"processors": [

{

"text\_embedding": {

"model\_id": "mBGzipQB2gmRjlv\_dOoB",

"field\_map": {

"input\_text": "output\_embedding"

}

}

}

]

}

CopyCopy as cURL

For more information, see [Text embedding processor](https://docs.opensearch.org/docs/latest/api-reference/ingest-apis/processors/text-embedding/).

When creating an index, specify the pipeline as the default\_pipeline. Ensure that dimension matches the dimensionality of the model configured in the pipeline:

PUT /my-ai-search-index

{

"settings": {

"index.knn": true,

"default\_pipeline": "auto-embed-pipeline"

},

"mappings": {

"properties": {

"input\_text": {

"type": "text"

},

"output\_embedding": {

"type": "knn\_vector",

"dimension": 768

}

}

}

}

CopyCopy as cURL

**ngesting data into a vector index**

After creating a vector index, you need to either ingest raw vector data or convert data to embeddings while ingesting it.

**Comparison of ingestion methods**

The following table compares the two ingestion methods.

| **Feature** | **Data format** | **Ingest pipeline** | **Vector generation** | **Additional fields** |
| --- | --- | --- | --- | --- |
| **Raw vector ingestion** | Pre-generated vectors | Not required | External | Optional metadata |
| **Converting data to embeddings during ingestion** | Text or image data | Required | Internal (during ingestion) | Original data + embeddings |

**Raw vector ingestion**

When working with raw vectors or embeddings generated outside of OpenSearch, you directly ingest vector data into the knn\_vector field. No pipeline is required because the vectors are already generated:

PUT /my-raw-vector-index/\_doc/1

{

"my\_vector": [0.1, 0.2, 0.3],

"metadata": "Optional additional information"

}

CopyCopy as cURL

You can also use the [Bulk API](https://docs.opensearch.org/docs/latest/api-reference/document-apis/bulk/) to ingest multiple vectors efficiently:

PUT /\_bulk

{"index": {"\_index": "my-raw-vector-index", "\_id": 1}}

{"my\_vector": [0.1, 0.2, 0.3], "metadata": "First item"}

{"index": {"\_index": "my-raw-vector-index", "\_id": 2}}

{"my\_vector": [0.2, 0.3, 0.4], "metadata": "Second item"}

CopyCopy as cURL

**Converting data to embeddings during ingestion**

After you have [configured an ingest pipeline](https://docs.opensearch.org/docs/latest/vector-search/creating-vector-index/#converting-data-to-embeddings-during-ingestion) that automatically generates embeddings, you can ingest text data directly into your index:

PUT /my-ai-search-index/\_doc/1

{

"input\_text": "Example: AI search description"

}

CopyCopy as cURL

The pipeline automatically generates and stores the embeddings in the output\_embedding field.

You can also use the [Bulk API](https://docs.opensearch.org/docs/latest/api-reference/document-apis/bulk/) to ingest multiple documents efficiently:

PUT /\_bulk

{"index": {"\_index": "my-ai-search-index", "\_id": 1}}

{"input\_text": "Example AI search description"}

{"index": {"\_index": "my-ai-search-index", "\_id": 2}}

{"input\_text": "Bulk API operation description"}

CopyCopy as cURL

**Working with sparse vectors**

OpenSearch also supports sparse vectors. For more information, see [Neural sparse search](https://docs.opensearch.org/docs/latest/vector-search/ai-search/neural-sparse-search/).

**Text chunking**

For information about splitting large documents into smaller passages before generating embeddings during dense or sparse AI search, see [Text chunking](https://docs.opensearch.org/docs/latest/vector-search/ingesting-data/text-chunking/).

**Text chunking**

INTRODUCED 2.13

When working with large text documents in [AI search](https://docs.opensearch.org/docs/latest/vector-search/ai-search/), it’s often necessary to split them into smaller passages because most embedding models have token length limitations. This process, called *text chunking*, helps maintain the quality and relevance of vector search results by ensuring that each embedding represents a focused piece of content that fits within model constraints.

To split long text into passages, you can use a text\_chunking processor as a preprocessing step for a text\_embedding or sparse\_encoding processor in order to obtain embeddings for each chunked passage. For more information about the processor parameters, see [Text chunking processor](https://docs.opensearch.org/docs/latest/ingest-pipelines/processors/text-chunking/). Before you start, follow the steps outlined in the [pretrained model documentation](https://docs.opensearch.org/docs/latest/ml-commons-plugin/pretrained-models/) to register an embedding model. The following example preprocesses text by splitting it into passages and then produces embeddings using the text\_embedding processor.

**Step 1: Create a pipeline**

The following example request creates an ingest pipeline that converts the text in the passage\_text field into chunked passages, which will be stored in the passage\_chunkfield. The text in the passage\_chunk field is then converted into text embeddings, and the embeddings are stored in the passage\_embedding field:

PUT \_ingest/pipeline/text-chunking-embedding-ingest-pipeline

{

"description": "A text chunking and embedding ingest pipeline",

"processors": [

{

"text\_chunking": {

"algorithm": {

"fixed\_token\_length": {

"token\_limit": 10,

"overlap\_rate": 0.2,

"tokenizer": "standard"

}

},

"field\_map": {

"passage\_text": "passage\_chunk"

}

}

},

{

"text\_embedding": {

"model\_id": "LMLPWY4BROvhdbtgETaI",

"field\_map": {

"passage\_chunk": "passage\_chunk\_embedding"

}

}

}

]

}

CopyCopy as cURL

**Step 2: Create an index for ingestion**

In order to use the ingest pipeline, you need to create a vector index. The passage\_chunk\_embedding field must be of the nested type. The knn.dimension field must contain the number of dimensions for your model:

PUT testindex

{

"settings": {

"index": {

"knn": true

}

},

"mappings": {

"properties": {

"text": {

"type": "text"

},

"passage\_chunk\_embedding": {

"type": "nested",

"properties": {

"knn": {

"type": "knn\_vector",

"dimension": 768

}

}

}

}

}

}

CopyCopy as cURL

**Step 3: Ingest documents into the index**

To ingest a document into the index created in the previous step, send the following request:

POST testindex/\_doc?pipeline=text-chunking-embedding-ingest-pipeline

{

"passage\_text": "This is an example document to be chunked. The document contains a single paragraph, two sentences and 24 tokens by standard tokenizer in OpenSearch."

}

CopyCopy as cURL

**Step 4: Search the index**

You can use a nested query to perform vector search on your index. We recommend setting score\_mode to max, where the document score is set to the highest score out of all passage embeddings:

GET testindex/\_search

{

"query": {

"nested": {

"score\_mode": "max",

"path": "passage\_chunk\_embedding",

"query": {

"neural": {

"passage\_chunk\_embedding.knn": {

"query\_text": "document",

"model\_id": "-tHZeI4BdQKclr136Wl7"

}

}

}

}

}

}

CopyCopy as cURL

**Searching vector data**

OpenSearch supports various methods for searching vector data, tailored to how the vectors were created and indexed. This guide explains the query syntax and options for raw vector search and auto-generated embedding search.

**Search type comparison**

The following table compares the search syntax and typical use cases for each vector search method.

| **Feature** | **Query type** | **Input format** | **Model required** | **Use case** |
| --- | --- | --- | --- | --- |
| **Raw vectors** | [knn](https://docs.opensearch.org/docs/latest/query-dsl/specialized/k-nn/) | Vector array | No | Raw vector search |
| **Auto-generated embeddings** | [neural](https://docs.opensearch.org/docs/latest/query-dsl/specialized/neural/) | Text or image data | Yes | [AI search](https://docs.opensearch.org/docs/latest/vector-search/ai-search/) |

**Searching raw vectors**

To search raw vectors, use the knn query type, provide the vector array as input, and specify the number of returned results k:

GET /my-raw-vector-index/\_search

{

"query": {

"knn": {

"my\_vector": {

"vector": [0.1, 0.2, 0.3],

"k": 2

}

}

}

}

CopyCopy as cURL

**Searching auto-generated embeddings**

OpenSearch supports [AI-powered search methods](https://docs.opensearch.org/docs/latest/vector-search/ai-search/), including semantic, hybrid, multimodal, and conversational search with retrieval-augmented generation (RAG). These methods automatically generate embeddings from query input.

To run an AI-powered search, use the neural query type. Specify the query\_text input, the model ID of the embedding model you [configured in the ingest pipeline](https://docs.opensearch.org/docs/latest/vector-search/creating-vector-index/#converting-data-to-embeddings-during-ingestion), and the number of returned results k. To exclude embeddings from being returned in search results, specify the embedding field in the \_source.excludes parameter:

GET /my-ai-search-index/\_search

{

"\_source": {

"excludes": [

"output\_embedding"

]

},

"query": {

"neural": {

"output\_embedding": {

"query\_text": "What is AI search?",

"model\_id": "mBGzipQB2gmRjlv\_dOoB",

"k": 2

}

}

}

}

CopyCopy as cURL