TABLE OF CONTENTS (LIST OF DOCUMENTS)

        - **What is the Jaccard Similarity?**
        - **What is the Tanimoto Similarity?**
    - **Basic Info**
        - **What are Vector Embeddings?**
        - **What is a Vector Database?**
    - **Use cases**
        - **Semantic Search**
        - **Generative QA with OpenAI**
        - **Ecommerce Hybrid Search**
        - **Image Search**
        - **Product Recommender**
    - **Pricing**
    - **Workflow**
        - **Manage indexes**
        - **Manage data**
        - **Insert data**
        - **Query data**
        - **Metadata filtering**
        - **Scale indexes**
    - **Limits**
    - **Concepts**
        - **Collections**
            - **Back up indexes**
    - **Indexes**
    - **Sparse-dense embeddings**
        - **sparse-dense vectors allow keyword-aware semantic search combining with keyword search results can improve relevance**
        - **Sparse-dense embeddings - Keyword search algorithms like the BM25 algorithm compute the relevance of text documents based on the number of keyword matches, their frequency, and other factors.**
    - **Organizations**
    - **Projects**
    - **Multitenancy**
    - **Guides**
        - **Using namespaces**
        - **Monitoring**
        - **Performance tuning**
        - **Troubleshooting**
        - **Moving to production**
        - **Manage Projects - Create a project , Add users to projects and organizations, Change project pod limit, Rename a project**
        - **Manage billing**
            - **Understanding cost**
            - **Managing cost**

- **Manage datasets**
  - **Pinecone public datasets**
  - **Using public Pinecone datasets**
  - **Creating and loading private datasets**
- **Integrations**
  - **OpenAI**
  - **Cohere**
  - **Haystack**
  - **Hugging Face Inference Endpoints**
  - **Elasticsearch**
  - **Databricks**
  - **LangChain**
- **Support**
  - **Libraries**
  - **FAQ**
    - **Node.JS Troubleshooting**
    - **Parallel Queries**
    - **Managing cardinality in an index**
    - **CORS Issues**
    - **How do I keep my customer data separate in Pinecone?**
    - **Using namespaces vs. metadata filtering**
    - **Metadata string value returned as a datetime object**
    - **TypeError when running pinecone.list_indexes or pinecone.create_index**
    - **Handshake read failed" error when connecting to Pinecone server**
    - **How much is Pinecone?**
    - **Guide for customers to follow when debugging model vs. Pinecone recall issues**
    - **Differences between Lexical and Semantic Search regarding relevancy**
    - **How does billing work?**
    - **How to wait for index creation to be complete**
    - **Pinecone client libraries, connectors, and SDKs**
  - **Release notes**

Support - Guide for customers to follow when debugging model vs. Pinecone recall issues

Step 1: Establish the evaluation framework
Before starting, establish an evaluation framework for your model and Pinecone recall issues. You will need to query a dataset of at least 10 samples and a source dataset of 100k samples, and choose an evaluation metric that is appropriate for your use case. Pinecone recommends Evaluation Measures in Information Retrieval as a guide for choosing an evaluation metric. Label the "right answers" in the source dataset for each query.
Step 2: Generate embeddings for queries + source dataset with the model
Use your model to generate embeddings for your queries and the source dataset. Run the model on the source dataset to create the vector dataset and query vectors.
Step 3: Calculate brute force vector distance to evaluate model quality
Run a brute force search using query vectors over the vector dataset via FAISS or numpy and record the record IDs for each query. Evaluate the returned list using your evaluation metric and the set of "right answers" labeled in step 1. If this metric is unacceptable, it indicates a model issue.
Step 4: Upload vector dataset to Pinecone + query
Upload the vector dataset to Pinecone and query it using your queries. Record the vector IDs returned for each query.
Step 5: Calculate Pinecone recall
For each query, compare the % of vector IDs that Pinecone recalled compared to the brute force search. This will be the % recall for each query. You can then average across all queries to get average recall. Typically, average recall should be close to 0.99 for s1/p1 indexes.
Step 6: If recall is too low, reach out to Pinecone Support (reproducible dataset and queries)
If the recall metric is too low for your use case, reach out to Pinecone product and engineering with the query and vector dataset that reproduces the issue for further investigation. Pinecone's team will investigate.

Support - How to wait for index creation to be complete

Pinecone index creation involves several different subsystems, including one which accepts the job of creating the index and one that actually performs the action. The Python client and the REST API are designed to interact with the first system during index creation but not the second.

This means that when a request call to create_index() is made, what's actually happening is that the job is being submitted to the queue to be completed. We do it this way for several reasons, including enforcing separation between the control and data planes.
If you need your application to wait for the index to be created before continuing to its next step, there is a way to ensure this happens, though. describe_index() returns data about the state of the index, including whether it is ready to accept data. You simply call that method until it returns a 200 status code and the status object reports that the index is ready. Because the return is a tuple, we just have to access the slice containing the status object and check the boolean state of the ready variable. This is one possible method of doing so using the Python client:

```
import pinecone
from time import sleep

def wait_on_index(index: str):
"""
Takes the name of the index to wait for and blocks until it's available and ready.
"""
ready = False
while not ready:
try:
desc = pinecone.describe_index(index)
if desc[7]['ready']:
return True
except pinecone.core.client.exceptions.NotFoundException:
# NotFoundException means the index is created yet.
pass
sleep(5)
```

Calling wait_on_index() would then allow your application to only continue to upsert data once the index is fully online and available to accept data, avoiding potential 403 or 404 errors.

Support - How does billing work?

Overview
Pinecone offers three different pod types: s1, p1, and p2. Each pod type offers different levels of performance and cost that users can take advantage of. Pinecone is SOC2 Type II compliant and users accept the End-User License Agreement when creating an account.
Billing
Pinecone bills users for pod-hours used in 15-minute increments. Pre-commitment discounts and Enterprise Dedicated pricing are available.
Managing Your Subscription
Users can manage their subscription directly in the console and contact Pinecone for help with sizing and testing. ***If you already have an index with a pod running it will start charging for***

that pod after you upgrade to the standard plan or enterprise plan. The reason it is no longer free is it is subject to higher level SLAs and support.

Support - Differences between Lexical and Semantic Search regarding relevancy

When it comes to searching for information in a large corpus of text, there are two main approaches that search engines use: keyword or lexical search and vector semantic similarity search. While both methods aim to retrieve relevant documents, they use different techniques to do so.

Keyword or lexical search relies on matching exact words or phrases that appear in a query with those in the documents. This approach is relatively simple and fast, but it has limitations. For example, it may not be able to handle misspellings, synonyms, or polysemy (when a word has multiple meanings). In addition, it does not take into account the context or meaning of the words, which can lead to irrelevant results.

On the other hand, vector semantic similarity search uses natural language processing (NLP) techniques to analyze the meaning of words and their relationships. It represents words as vectors in a high-dimensional space, where the distance between vectors indicates their semantic similarity. This approach can handle misspellings, synonyms, and polysemy, and it can also capture more subtle relationships between words, such as antonyms, hypernyms, and meronyms. As a result, it can produce more accurate and relevant results.

However, there is a caveat to using vector semantic similarity search. It requires a large amount of data to train the NLP models, which can be computationally expensive and time-consuming. As a result, it may not be as effective for short documents or queries that do not contain enough context to determine the meaning of the words. In such cases, a simple keyword or lexical search may be more suitable and effective.

In fact, in some cases, a short document may actually show higher in a vector space for a given query, even if it is not as relevant as a longer document. This is because short documents typically have fewer words, which means that their word vectors are more likely to be closer to the query vector in the high-dimensional space. As a result, they may have a higher cosine similarity score than longer documents, even if they do not contain as much information or context. This phenomenon is known as the "curse of dimensionality" and it can affect the performance of vector semantic similarity search in certain scenarios.

In conclusion, both keyword or lexical search and vector semantic similarity search have their strengths and weaknesses. Depending on the nature of the corpus, the type of queries, and the computational resources available, one approach may be more appropriate than the other. It is important to understand the differences between the two methods and use them judiciously to achieve the best results.

Support - How much is Pinecone?

Pinecone charges on a per pod, per hour basis. For the latest pricing details, please see our pricing page. Pricing is determined by a combination of factors:

- The pod type. p1 and s1 pods have different per-hour pricing than p2 pods.
- The number of pods. Each shard adds a pod, and each replica increases the total number of pods by a factor of the replicas.
- Which cloud provider you are using for your project.
- If you are using the Standard or Enterprise plan.

For example, if your index uses a single p1.x1 pod and runs continuously for a month in a project hosted in GCP, on the Standard plan, it would cost (using pricing as of March 16, 2023) about $70 per month. On the Enterprise plan, it would cost about $105 per month.

Support - Handshake read failed" error when connecting to Pinecone server

Problem: When trying to connect to Pinecone server, some users may receive an error message that says "Handshake read failed" and their connection attempt fails. This error can prevent them from running queries against their Pinecone indexes.
Solution: If you encounter this error message, it means that your computer is not properly connecting with the Pinecone server. The error is often due to a misconfiguration of your Pinecone client or API key. Here is a recommended solution:

1. Make sure your firewall is not blocking any traffic and your internet connection is working fine. If you are unsure about how to do this, please consult your IT team.
2. Check that you have set up the Pinecone client and API key correctly. Double-check that you have followed the instructions in our documentation (**https://www.pinecone.io/docs/quickstart/**) correctly.
3. If you are still having issues, try creating a new index on Pinecone and populating it with data by running another script on your computer. This will verify that your computer can access the Pinecone servers for some tasks.
4. If the error persists, you may need to check your code for any misconfigurations. Make sure you are setting up your Pinecone client correctly and passing the right parameters when running queries against your indexes.
5. If you are still unable to resolve the issue, you can reach out to Pinecone support for assistance. They will be able to help you diagnose and resolve the issue.

In one instance, a customer encountered the same "Handshake read failed" error but was able to resolve it by putting the API key and environment variables inside pinecone.init(). This

resolved the issue, and they were able to successfully run queries against their Pinecone indexes.
Conclusion: If you encounter the "Handshake read failed" error when trying to connect to Pinecone server, there are several steps you can take to resolve the issue. First, double-check that you have set up the Pinecone client and API key correctly. Then, check for any misconfigurations in your code. If the error persists, reach out to Pinecone support for assistance.

Support - TypeError when running pinecone.list_indexes or pinecone.create_index

Problem: Running pinecone.list_indexes() or pinecone.create_index() in the Python client results in a TypeError.
TypeError: expected string or bytes-like object
If this happens, it means you still need to initialize your connection to Pinecone. Ensure you first run pinecone.init() before issuing any commands.

Support - Using namespaces vs. metadata filtering

This article will discuss the advantages and disadvantages of using namespaces and metadata filtering in your application. Performance is the same whether you use namespaces or metadata filtering. The most significant difference is how you query your index.

Namespaces
Namespaces are a way to segment data into distinct areas within your index. The intent is to have the ability for an index to serve multiple purposes. For example, you could have a single index containing customers, catalog items, and articles. These can be queried and treated effectively as separate entities in your index. However, there is one strong consideration. You can only query one namespace (or none) at a time. This means you cannot choose to query the entire corpus of data in the future. If you see the need to query across namespaces, then use metadata filtering instead. If you never need to cross namespaces with queries, then using namespaces is fine.
See also namespaces

Metadata Filtering
Metadata fields or you could call them key:value pairs, are a way to add information to individual vectors to give them more meaning. By adding metadata to your vectors, you can filter by those fields at query time. This is similar to namespaces, except you are not limited to a single filter. You can use a variety of filter patterns and conditions to search subsets of your data without requiring namespaces or multiple queries. This is a popular alternative to namespaces, and many customers use this method instead. This gives the same performance and more flexibility in the future if you want to search across the entire index.
See also metadata filtering
Switching from namespaces to metadata filtering:1.) Clear your index or create a new index2.) Re-ingest your data and use a metadata field instead. Leave out the namespace parameter.3.) Now you can run queries using these methods: https://www.pinecone.io/docs/metadata-filtering/

Support - Metadata string value returned as a datetime object

When using the Pinecone Python client you may encounter a bug where strings in metadata are interpreted as datetime objects. This is because of a bug in the OpenAPI code used for the REST interface in the client. The OpenAPI spec does not clearly define all accepted data types. While we fix this, we recommend using GRPCIndex to instantiate your index connections. Because of differences between the REST and GRPC connections this bug does not affect GRPCIndex objects.

Support - How do I keep my customer data separate in Pinecone?

Some customer use cases require vectors to be segmented by their customers either physically or logically. The table below describes three techniques to accomplish this and the pros and cons of considering each.

Technique	Pros	Cons

Separate Indexes Each customer would have a separate index	• Customer data is truly separated physically by indexes	• You cannot query across customers if you wish • Cost and maintenance of several indexes
Namespaces You can isolate data within a single index using namespaces	• You can only query one namespace at a time, which would keep customer data separate logically • Cheaper than #1 potentially by making more efficient use of index storage	• You cannot query across namespaces • Customer data is only separated logically
Metadata Filtering All data is kept in a single index and logically separated by filtering at query time	• Most versatile if you wish to query across customers • As with namespaces cheaper than #1 potentially	• Customer data is separated only by filtering at query time

---

Support - CORS Issues

Cross-Origin Resource Sharing (CORS) is an HTTP-header based security feature that allows a server to indicate which domains, schemes or ports a browser should accept content from. When a browser-based app, by default, only loads content from the same origin as the original request, CORS errors can appear if the responses come from a different origin. Pinecone's current implementation of CORS can cause this mismatch and display the error 'No 'Access-Control-Allow-Origin' header is present on the requested resource'. To address this, customers should use a server that is not hosted locally.

About Localhost (running a web server locally)

Localhost is not inherently a problem. However, when running a web server on a local machine (e.g., laptop or desktop computer), using "localhost" as the hostname can cause issues with cross-origin resource sharing (CORS).
The reason for this is that the Same-Origin Policy (SOP) enforced by web browsers treats "localhost" as a different origin than the actual IP address of the machine. For example, if a web application running on "localhost" makes a cross-origin request to a server running on the actual IP address of the machine, the browser will treat it as a cross-origin request and enforce the SOP.
To allow cross-origin requests between "localhost" and the actual IP address of the machine, the server needs to explicitly allow them by including the appropriate CORS headers in its response. However, as mentioned earlier, running a web server on a local machine can present security risks and is generally not recommended for production use.
Therefore, while "localhost" itself is not a problem, using it as the hostname for a web server can cause CORS issues that need to be properly addressed. Additionally, running a web server on a local machine should be done with caution and only for development or testing purposes, rather than for production use.

Support - Release notes

This document contains details about Pinecone releases. For information about using specific features, see our API reference.

April 26, 2023

Indexes in the starter plannow support approximately 100,000 1536-dimensional embeddings with metadata. Capacity is proportional for other dimensionalities.

April 3, 2023

Pinecone now supports new US and EU cloud regions.

March 21, 2023

Pinecone now supports enterprise SSO. Contact us at support@pinecone.ioto set up your integration.

March 1, 2023

Pinecone now supports 40kb of metadata per vector.

February 22, 2023

Sparse-dense embeddings are now in Public Preview.
Pinecone now supports vectors with sparse and dense values. To use sparse-dense embeddings in Python, upgrade to Python client version 2.2.0.

Pinecone Python client version 2.2.0 is available
Python client version 2.2.0 with support for sparse-dense embeddings is now available on GitHuband PYPI.

February 15, 2023

New Node.js client is now available in public preview
You can now try out our new Node.js client for Pinecone.

February 14, 2023

New usage reports in the Pinecone console
You can now monitor your current and projected Pinecone usage with the Usagedashboard.

January 31, 2023

Pinecone is now available in AWS Marketplace
You can now sign up for Pinecone billing through Amazon Web Services Marketplace.

January 3, 2023

Pinecone Python client version 2.1.0 is now available on GitHub.
The latest release of the Python clientmakes the following changes:

- Fixes "Connection Reset by peer" error after long idle periods
- Adds typing and explicit names for arguments in all client operations

- Adds docstrings to all client operations
- Adds Support for batch upserts by passing batch_sizeto the upsert method
- Improves gRPC query results parsing performance

December 22, 2022

Pinecone is now available in GCP Marketplace
You can now sign up for Pinecone billing through Google Cloud Platform Marketplace.

December 6, 2022

Organizations are generally available
Pinecone now features organizations, which allow one or more users to control billing and project settings across multiple projects owned by the same organization.

p2 pod type is generally available
The p2 pod typeis now generally available and ready for production workloads. p2 pods are now available in the Starter plan and support the dotproductdistance metric.

Performance improvements

- Bulk vector_deletesare now up to 10x faster in many circumstances.
- Creating collectionsis now faster.

October 31, 2022

Hybrid search (Early access)
Pinecone now supports keyword-aware semantic search with the new hybrid searchindexes and endpoints. Hybrid search enables improved relevance for semantic search results by combining them with keyword search.

This is an early accessfeature and is available only by signing up.

October 17, 2022

Status page
The new Pinecone Status Pagedisplays information about the status of the Pinecone service, including the status of individual cloud regions and a log of recent incidents.

September 16, 2022

Public collections

You can now create indexes from public collections, which are collections containing public data from real-world data sources. Currently, public collections include the Glue - SSTB collection, the TREC Question classification collection, and the SQuAD collection.

August 16, 2022

Collections (Public Preview)("Beta")

You can now make static copies of your indexusing collections. After you create a collection from an index, you can create a new index from that collection. The new index can use any pod type and any number of pods. Collections only consume storage.

This is a public previewfeature and is not appropriate for production workloads.

Vertical scaling

You can now change the size of the podsfor a live index to accommodate more vectors or queries without interrupting reads or writes. The p1 and s1 pod types are now available in 4 different sizes: 1x, 2x, 4x, and 8x. Capacity and compute per pod double with each size increment.

p2 pod type (Public Preview)("Beta")

The new p2 pod typeprovides search speeds of around 5ms and throughput of 200 queries per second per replica, or approximately 10x faster speeds and higher throughput than the p1 pod type, depending on your data and network conditions.

This is a public previewfeature and is not appropriate for production workloads.

Improved p1 and s1 performance

The s1and p1pod types now offer approximately 50% higher query throughput and 50% lower latency, depending on your workload.

July 26, 2022

You can now specify a metadata filterto get results for a subset of the vectors in your index by calling describe_index_statswith a filterobject.

The describe_index_statsoperation now uses the POSTHTTP request type. The filterparameter is only accepted by describe_index_statscalls using the POSTrequest type. Calls to describe_index_statsusing the GETrequest typeare now deprecated.

July 12, 2022

Pinecone Console Guided Tour

You can now choose to follow a guided tour in the Pinecone Console. This interactive tutorial walks you through creating your first index, upserting vectors, and querying your data. The purpose of the tour is to show you all the steps you need to start your first project in Pinecone.

June 24, 2022

Updated response codes

The create_index, delete_index, and scale_indexoperations now use more specific HTTP response codes that describe the type of operation that succeeded.

June 7, 2022

Selective metadata indexing

You can now store more metadata and more unique metadata values! Select which metadata fields you want to index for filteringand which fields you only wish to store and retrieve. When you index metadata fields, you can filter vector search queries using those fields. When you store metadata fields without indexing them, you keep memory utilization low, especially when you have many unique metadata values, and therefore can fit more vectors per pod.

Single-vector queries

You can now specify a single query vector using the vectorinput. We now encourage all users to query using a single vector rather than a batch of vectors, because batching queries can lead to long response messages and query times, and single queries execute just as fast on the server side.

Query by ID

You can now query your Pinecone index using only the ID for another vector. This is useful when you want to search for the nearest neighbors of a vector that is already stored in Pinecone.

Improved index fullness accuracy

The index fullness metric in describe_index_stats()results is now more accurate.

April 25, 2022

Partial updates (Public Preview)

You can now perform a partial updateby ID and individual value pairs. This allows you to update individual metadata fields without having to upsert a matching vector or update all metadata fields at once.

New metrics

Users on all plans can now see metrics for the past one (1) week in the Pinecone console. Users on the Enterprise and Enterprise Dedicated plan now have access to the following metrics via the Prometheus metrics endpoint:

- pinecone_vector_count
- pinecone_request_count_total
- pinecone_request_error_count_total
- pinecone_request_latency_seconds
- pinecone_index_fullness(Public Preview)

Note:The accuracy of the pinecone_index_fullnessmetric is improved. This may result in changes from historic reported values. This metric is in public preview.

Spark Connector

Spark users who want to manage parallel upserts into Pinecone can now use the official Spark connector for Pineconeto upsert their data from a Spark dataframe.

Support for Boolean and float metadata in Pinecone indexes

You can now add Booleanand float64values to metadata JSON objects associated with a Pinecone index.

New state field in describe_index results

The describe_indexoperation results now contain a value for state, which describes the state of the index. The possible values for stateare Initializing, ScalingUp, ScalingDown, Terminating, and Ready.

Delete by metadata filter

The Deleteoperation now supports filtering my metadata.

Support - Node.JS Troubleshooting

Thank you for reaching out to Pinecone. Our Pinecone Node.js client is in Public beta with limited support. This case appears to be a configuration issue and not a bug or defect within the product. There could be several reasons why a Node.js application works in development mode but not in deployment. Here are a few possibilities:

1. Dependency version mismatch: Sometimes, different environments have different versions of dependencies installed. If the application was developed using a specific version of a dependency, and that version is not installed on the deployment environment, the application may not work as expected.
2. Environment configuration: The development environment may have different configurations from the deployment environment. For example, the development environment may have different environment variables set or different network settings. If the application relies on specific configuration settings that are not present in the deployment environment, it may not work.
3. Permissions: The application may require permissions to access certain resources that are only granted in the development environment. For example, if the application needs to write to a specific directory, the permissions to write to that directory may only be granted in the development environment.
4. Database connection: If the application relies on a database connection, it's possible that the connection settings are different in the deployment environment. For example, the database may have a different hostname or port number.
5. Code optimization: During development, the application may have been running on a development server that did not optimize the code. However, when deployed, the application may be running on a production server that is optimized for performance. If there are code issues or performance bottlenecks, they may only appear when the application is deployed.
6. It may be necessary to install the fetch python library for compatibility with node.js

In order to troubleshoot the issue, it's important to identify where the application is failing and compare the development and deployment environments to see what differences exist. It's also important to review any error messages or logs that are generated to help identify the issue.

You may also reach out to our community of Pinecone users for help at https://community.pinecone.io/

Support - Parallel Queries

There are many approaches to accomplish performing parallel queries in your application from using the python client, to making REST calls. Here is one example of an approach. You can use this as a guide to develop your solution that fits your use-case.

Parallel_queries.py

```
import time
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import TimeoutError as FutureTimeoutError
from typing import List, Optional, Collection, Any, Dict

import pinecone
from sentry_sdk import start_span


class PineconeProvider(SearchProvider):
    QUERY_BATCH_SIZE = 20
    MAX_CONCURRENT_QUERIES = 4

    def __init__(self, api_key: str, index_name: str):
        self._api_key: Optional[str] = None
        self._client: Optional[pinecone.GRPCIndex] = None
        self._executor = ThreadPoolExecutor(self.QUERY_BATCH_SIZE *
self.MAX_CONCURRENT_QUERIES)

        self.index_name = index_name

        if api_key is not None:
```

```
        self.set_api_key(api_key)

    def set_api_key(self, api_key: str):
        self._api_key = api_key
        if self._api_key:
            pinecone.init(api_key=self._api_key)
            self._client = pinecone.GRPCIndex(self.index_name)
        else:
            self._client = None

    @instrument
    def _query(
            self,
            queries: List[List[float]],
            product_sets: Collection[str],
            num_neighbors: int,
            return_metadata: bool,
            return_features: bool
    ) -> List[List[SearchMatch]]:
        if self._client is None:
            raise RuntimeError('API key is not set')

        with start_span(op='pinecone.query') as span:
            span.set_tag('pinecone.queries', len(queries))
            span.set_tag('pinecone.top_k', num_neighbors)
            span.set_tag('pinecone.include_metadata', return_metadata)
            span.set_tag('pinecone.include_values', return_features)

            response = self._client.query(
                queries=queries,
                top_k=num_neighbors,
                include_metadata=return_metadata,
                include_values=return_features
            )

        transformed_response = []

        for result in response.results:
            matches = []
            for match in result.matches:
                result = SearchMatch(
                    crop_id=match.id,
                    score=match.score
                )
```

```
            if return_metadata:
                result.metadata = match.metadata

            if return_features and match.values:
                result.features = match.values

            matches.append(result)

        transformed_response.append(matches)

    return transformed_response

@instrument
def query(
        self,
        queries: List[List[float]],
        product_sets: Collection[str],
        num_neighbors: int,
        return_metadata: bool,
        return_features: bool
):
    response = []

    futures = {
        i: self._executor.submit(
            self._query,
            [query_vector],
            num_neighbors=num_neighbors,
            return_metadata=return_metadata,
            return_features=return_features
        )
        for i, query_vector in enumerate(queries)
    }

    chunk_responses = {}

    while len(chunk_responses) != len(queries):
        for i in range(len(queries)):
            if i in chunk_responses:
                continue

            future = futures[i]
            try:
```

```
                chunk_responses[i] = future.result(timeout=1)[0]
            except FutureTimeoutError:
                continue

        time.sleep(1)

    for i in range(len(queries)):
        response.append(chunk_responses[i])

    return response
```

Support - Managing cardinality in an index

Cardinality is an important factor to consider when managing an index, as high cardinality can lead to performance issues, pod fullness, and a reduction in the number of possible vectors that can fit per pod.
Fortunately, there is a solution to this problem - Selective Metadata Indexing. Selective Metadata Indexing allows you to specify which fields need to be indexed and which do not, helping to reduce the overall cardinality of the metadata index while still ensuring that the necessary fields are able to be filtered.
In order to set up Selective Metadata Indexing, you will need to use the Pinecone API. With the API, you can create a metadata index and specify which fields need to be indexed. This will allow you to reduce the cardinality of the index while still ensuring that all of the necessary fields are indexed.
By using Selective Metadata Indexing, you can manage the cardinality of your index and ensure that your queries are fast and reliable. It is a simple and effective way to ensure that your index is optimized for performance and that your pod does not become full.

Integrations - LangChain

LangChain

Suggest Edits
Welcome to the integration guide for Pinecone and LangChain. This documentation covers the steps to integrate Pinecone, a high-performance vector database, with LangChain, a framework for building applications powered by large language models (LLMs).

Pinecone enables developers to build scalable, real-time recommendation and search systems based on vector similarity search. LangChain, on the other hand, provides modules for managing and optimizing the use of language models in applications. Its core philosophy is to facilitate data-aware applications where the language model interacts with other data sources and its environment.

By integrating Pinecone with LangChain, you can develop sophisticated applications that leverage both the platforms' strengths. Allowing us to add *"long-term memory"*to LLMs, greatly enhancing the capabilities of autonomous agents, chatbots, and question answering systems, among others.

There are naturally many ways to use these two tools together. We have covered the process in detail across our many examples and learning material, including:

- LangChain AI Handbook
- Retrieval Augmentation for LLMs
- Retrieval Augmented Conversational Agent

The remainder of this guide will walk you through a simple retrieval augmentation example using Pinecone and LangChain.

Retrieval Augmentation in LangChain

LLMs have a data freshness problem. The most powerful LLMs in the world, like GPT-4, have no idea about recent world events.

The world of LLMs is frozen in time. Their world exists as a static snapshot of the world as it was within their training data.

A solution to this problem is *retrieval augmentation*. The idea behind this is that we retrieve relevant information from an external knowledge base and give that information to our LLM. In this notebook we will learn how to do that.

To begin, we must install the prerequisite libraries that we will be using in this notebook.

Python
```
!pip install -qU langchain==0.0.162 openai tiktoken pinecone-client[grpc] datasets
apache_beam mwparserfromhell
```

zsh:1: no matches found: pinecone-client[grpc]

Building the Knowledge Base

Python
from datasets import load_datasetdata = load_dataset("wikipedia", "20220301.simple", split='train[:10000]')data

Found cached dataset wikipedia
(/Users/jamesbriggs/.cache/huggingface/datasets/wikipedia/20220301.simple/2.0.0/aa542ed919df55cc5d3347f42dd4521d05ca68751f50dbc32bae2a7f1e167559)

Dataset({
features: ['id', 'url', 'title', 'text'],
num_rows: 10000
})

Python
data[6]

{'id': '13',
'url': 'https://simple.wikipedia.org/wiki/Alan%20Turing',
'title': 'Alan Turing',
'text': 'Alan Mathison Turing OBE FRS (London, 23 June 1912 – Wilmslow, Cheshire, 7 June 1954) was an English mathematician and computer scientist. He was born in Maida Vale, London.\n\nEarly life and family \nAlan Turing was born in Maida Vale, London on 23 June 1912. His father was part of a family of merchants from Scotland. His mother, Ethel Sara, was the daughter of an engineer.\n\nEducation \nTuring went to St. Michael\'s, a school at 20 Charles Road, St Leonards-on-sea, when he was five years old.\n"This is only a foretaste of what is to come, and only the shadow of what is going to be.” – Alan Turing.\n\nThe Stoney family were once prominent landlords, here in North Tipperary. His mother Ethel Sara Stoney (1881–1976) was daughter of Edward Waller Stoney (Borrisokane, North Tipperary) and Sarah Crawford (Cartron Abbey, Co. Longford); Protestant Anglo-Irish gentry.\n\nEducated in Dublin at Alexandra School and College; on October 1st 1907 she married Julius Mathison Turing, latter son of Reverend John Robert Turing and Fanny Boyd, in Dublin. Born on June 23rd 1912, Alan Turing would go on to be regarded as one of the greatest figures of the twentieth century.\n\nA brilliant mathematician and cryptographer Alan was to become the founder of modern-day computer science and artificial intelligence; designing a machine at Bletchley Park to break secret Enigma encrypted messages used by the Nazi German war machine to protect sensitive commercial, diplomatic and military communications during World War 2. Thus, Turing made the single biggest contribution to the Allied victory in the war against Nazi Germany, possibly saving the lives of an estimated 2 million people, through his effort in shortening World War II.\n\nIn

2013, almost 60 years later, Turing received a posthumous Royal Pardon from Queen Elizabeth II. Today, the “Turing law” grants an automatic pardon to men who died before the law came into force, making it possible for living convicted gay men to seek pardons for offences now no longer on the statute book.\n\nAlas, Turing accidentally or otherwise lost his life in 1954, having been subjected by a British court to chemical castration, thus avoiding a custodial sentence. He is known to have ended his life at the age of 41 years, by eating an apple laced with cyanide.\n\nCareer \nTuring was one of the people who worked on the first computers. He created the theoretical Turing machine in 1936. The machine was imaginary, but it included the idea of a computer program.\n\nTuring was interested in artificial intelligence. He proposed the Turing test, to say when a machine could be called "intelligent". A computer could be said to "think" if a human talking with it could not tell it was a machine.\n\nDuring World War II, Turing worked with others to break German ciphers (secret messages). He worked for the Government Code and Cypher School (GC&CS) at Bletchley Park, Britain\'s codebreaking centre that produced Ultra intelligence.\nUsing cryptanalysis, he helped to break the codes of the Enigma machine. After that, he worked on other German codes.\n\nFrom 1945 to 1947, Turing worked on the design of the ACE (Automatic Computing Engine) at the National Physical Laboratory. He presented a paper on 19 February 1946. That paper was "the first detailed design of a stored-program computer". Although it was possible to build ACE, there were delays in starting the project. In late 1947 he returned to Cambridge for a sabbatical year. While he was at Cambridge, the Pilot ACE was built without him. It ran its first program on 10\xa0May 1950.\n\nPrivate life \nTuring was a homosexual man. In 1952, he admitted having had sex with a man in England. At that time, homosexual acts were illegal. Turing was convicted. He had to choose between going to jail and taking hormones to lower his sex drive. He decided to take the hormones. After his punishment, he became impotent. He also grew breasts.\n\nIn May 2012, a private member\'s bill was put before the House of Lords to grant Turing a statutory pardon. In July 2013, the government supported it. A royal pardon was granted on 24 December 2013.\n\nDeath \nIn 1954, Turing died from cyanide poisoning. The cyanide came from either an apple which was poisoned with cyanide, or from water that had cyanide in it. The reason for the confusion is that the police never tested the apple for cyanide. It is also suspected that he committed suicide.\n\nThe treatment forced on him is now believed to be very wrong. It is against medical ethics and international laws of human rights. In August 2009, a petition asking the British Government to apologise to Turing for punishing him for being a homosexual was started. The petition received thousands of signatures. Prime Minister Gordon Brown acknowledged the petition. He called Turing\'s treatment "appalling".\n\nReferences\n\nOther websites \nJack Copeland 2012. Alan Turing: The codebreaker who saved \'millions of lives\'. BBC News / Technology \n\nEnglish computer scientists\nEnglish LGBT people\nEnglish mathematicians\nGay men\nLGBT scientists\nScientists from London\nSuicides by poison\nSuicides in the United Kingdom\n1912 births\n1954 deaths\nOfficers of the Order of the British Empire'}

Every record contains *a lot*of text. Our first task is therefore to identify a good preprocessing methodology for chunking these articles into more "concise" chunks to later be embedding and stored in our Pinecone vector database.

For this we use LangChain's RecursiveCharacterTextSplitterto split our text into chunks of a specified max length.

Python
import tiktokentiktoken.encoding_for_model('gpt-3.5-turbo')

<Encoding 'cl100k_base'>

Python
import tiktokentokenizer = tiktoken.get_encoding('cl100k_base')# create the length functiondef tiktoken_len(text): tokens = tokenizer.encode(text, disallowed_special=()) return len(tokens)tiktoken_len("hello I am a chunk of text and using the tiktoken_len function " "we can find the length of this chunk of text in tokens")

26

Python
from langchain.text_splitter import RecursiveCharacterTextSplittertext_splitter = RecursiveCharacterTextSplitter(chunk_size=400, chunk_overlap=20, length_function=tiktoken_len, separators=["\n\n", "\n", " ", ""])
Python
chunks = text_splitter.split_text(data[6]['text'])[:3]chunks

['Alan Mathison Turing OBE FRS (London, 23 June 1912 – Wilmslow, Cheshire, 7 June 1954) was an English mathematician and computer scientist. He was born in Maida Vale, London.\n\nEarly life and family \nAlan Turing was born in Maida Vale, London on 23 June 1912. His father was part of a family of merchants from Scotland. His mother, Ethel Sara, was the daughter of an engineer.\n\nEducation \nTuring went to St. Michael\'s, a school at 20 Charles Road, St Leonards-on-sea, when he was five years old.\n"This is only a foretaste of what is to come, and only the shadow of what is going to be.” – Alan Turing.\n\nThe Stoney family were once prominent landlords, here in North Tipperary. His mother Ethel Sara Stoney (1881–1976) was daughter of Edward Waller Stoney (Borrisokane, North Tipperary) and Sarah Crawford (Cartron Abbey, Co. Longford); Protestant Anglo-Irish gentry.\n\nEducated in Dublin at Alexandra School and College; on October 1st 1907 she married Julius Mathison Turing, latter son of Reverend John Robert Turing and Fanny Boyd, in Dublin. Born on June 23rd 1912, Alan Turing would go on to be regarded as one of the greatest figures of the twentieth century.',
'A brilliant mathematician and cryptographer Alan was to become the founder of modern-day computer science and artificial intelligence; designing a machine at Bletchley Park to break secret Enigma encrypted messages used by the Nazi German war machine to protect sensitive commercial, diplomatic and military communications during World War 2. Thus, Turing made the single biggest contribution to the Allied victory in the war against Nazi Germany, possibly saving the lives of an estimated 2 million people, through his effort in shortening World War II.\n\nIn 2013, almost 60 years later, Turing received a posthumous Royal Pardon from Queen Elizabeth II. Today, the “Turing law” grants an automatic pardon to men who died before the law came into

force, making it possible for living convicted gay men to seek pardons for offences now no longer on the statute book.\n\nAlas, Turing accidentally or otherwise lost his life in 1954, having been subjected by a British court to chemical castration, thus avoiding a custodial sentence. He is known to have ended his life at the age of 41 years, by eating an apple laced with cyanide.\n\nCareer \nTuring was one of the people who worked on the first computers. He created the theoretical Turing machine in 1936. The machine was imaginary, but it included the idea of a computer program.\n\nTuring was interested in artificial intelligence. He proposed the Turing test, to say when a machine could be called "intelligent". A computer could be said to "think" if a human talking with it could not tell it was a machine.',
'During World War II, Turing worked with others to break German ciphers (secret messages). He worked for the Government Code and Cypher School (GC&CS) at Bletchley Park, Britain\'s codebreaking centre that produced Ultra intelligence.\nUsing cryptanalysis, he helped to break the codes of the Enigma machine. After that, he worked on other German codes.\n\nFrom 1945 to 1947, Turing worked on the design of the ACE (Automatic Computing Engine) at the National Physical Laboratory. He presented a paper on 19 February 1946. That paper was "the first detailed design of a stored-program computer". Although it was possible to build ACE, there were delays in starting the project. In late 1947 he returned to Cambridge for a sabbatical year. While he was at Cambridge, the Pilot ACE was built without him. It ran its first program on 10\xa0May 1950.\n\nPrivate life \nTuring was a homosexual man. In 1952, he admitted having had sex with a man in England. At that time, homosexual acts were illegal. Turing was convicted. He had to choose between going to jail and taking hormones to lower his sex drive. He decided to take the hormones. After his punishment, he became impotent. He also grew breasts.\n\nIn May 2012, a private member\'s bill was put before the House of Lords to grant Turing a statutory pardon. In July 2013, the government supported it. A royal pardon was granted on 24 December 2013.\n\nDeath \nIn 1954, Turing died from cyanide poisoning. The cyanide came from either an apple which was poisoned with cyanide, or from water that had cyanide in it. The reason for the confusion is that the police never tested the apple for cyanide. It is also suspected that he committed suicide.']

Python
tiktoken_len(chunks[0]), tiktoken_len(chunks[1]), tiktoken_len(chunks[2])

(299, 323, 382)

Using the text_splitterwe get much better sized chunks of text. We'll use this functionality during the indexing process later. Now let's take a look at embedding.

Creating Embeddings

Building embeddings using LangChain's OpenAI embedding support is fairly straightforward. We first need to add our OpenAI api keyby running the next cell:

Python

from getpass import getpassOPENAI_API_KEY = getpass("OpenAI API Key: ") # platform.openai.com
(Note that OpenAI is a paid service and so running the remainder of this notebook may incur some small cost)

After initializing the API key we can initialize our text-embedding-ada-002embedding model like so:

Python
from langchain.embeddings.openai import OpenAIEmbeddingsmodel_name = 'text-embedding-ada-002'embed = OpenAIEmbeddings(model=model_name, openai_api_key=OPENAI_API_KEY)
Now we embed some text like so:

Python
texts = ['this is the first chunk of text', 'then another second chunk of text is here']res = embed.embed_documents(texts)len(res), len(res[0])

(2, 1536)

From this we get *two*(aligning to our two chunks of text) 1536-dimensional embeddings.

Now we move on to initializing our Pinecone vector database.

Vector Database

To create our vector database we first need a free API key from Pinecone. Then we initialize like so:

Python
import pinecone# find API key in console at app.pinecone.ioYOUR_API_KEY = getpass("Pinecone API Key: ")# find ENV (cloud region) next to API key in consoleYOUR_ENV = input("Pinecone environment: ")index_name = 'langchain-retrieval-augmentation'pinecone.init(api_key=YOUR_API_KEY, environment=YOUR_ENV)if index_name not in pinecone.list_indexes(): # we create a new index pinecone.create_index(name=index_name, metric='dotproduct', dimension=len(res[0]) # 1536 dim of text-embedding-ada-002)
Then we connect to the new index:

Python
index = pinecone.GRPCIndex(index_name)index.describe_index_stats()

{'dimension': 1536,
'index_fullness': 0.1,

'namespaces': {'': {'vector_count': 27437}},
'total_vector_count': 27437}

We should see that the new Pinecone index has a total_vector_countof 0, as we haven't added any vectors yet.

Indexing

We can perform the indexing task using the LangChain vector store object. But for now it is much faster to do it via the Pinecone python client directly. We will do this in batches of 100or more.

Python
from tqdm.auto import tqdmfrom uuid import uuid4batch_limit = 100texts = []metadatas = []for i, record in enumerate(tqdm(data)): # first get metadata fields for this record metadata = { 'wiki-id': str(record['id']), 'source': record['url'], 'title': record['title'] } # now we create chunks from the record text record_texts = text_splitter.split_text(record['text']) # create individual metadata dicts for each chunk record_metadatas = [{ "chunk": j, "text": text, **metadata } for j, text in enumerate(record_texts)] # append these to current batches texts.extend(record_texts) metadatas.extend(record_metadatas) # if we have reached the batch_limit we can add texts if len(texts) >= batch_limit: ids = [str(uuid4()) for _ in range(len(texts))] embeds = embed.embed_documents(texts) index.upsert(vectors=zip(ids, embeds, metadatas)) texts = [] metadatas = []if len(texts) > 0: ids = [str(uuid4()) for _ in range(len(texts))] embeds = embed.embed_documents(texts) index.upsert(vectors=zip(ids, embeds, metadatas))

0%| | 0/10000 [00:00<?, ?it/s]

We've now indexed everything. We can check the number of vectors in our index like so:

Python
index.describe_index_stats()

{'dimension': 1536,
'index_fullness': 0.1,
'namespaces': {'': {'vector_count': 27437}},
'total_vector_count': 27437}

Creating a Vector Store and Querying

Now that we've build our index we can switch back over to LangChain. We start by initializing a vector store using the same index we just built. We do that like so:

Python
```
from langchain.vectorstores import Pineconetext_field = "text"# switch back to normal index for langchainindex = pinecone.Index(index_name)vectorstore = Pinecone( index, embed.embed_query, text_field)
```
Python
```
query = "who was Benito Mussolini?"vectorstore.similarity_search( query, # our search query k=3 # return 3 most relevant docs)
```

[Document(page_content='Benito Amilcare Andrea Mussolini KSMOM GCTE (29 July 1883 – 28 April 1945) was an Italian politician and journalist. He was also the Prime Minister of Italy from 1922 until 1943. He was the leader of the National Fascist Party.\n\nBiography\n\nEarly life\nBenito Mussolini was named after Benito Juarez, a Mexican opponent of the political power of the Roman Catholic Church, by his anticlerical (a person who opposes the political interference of the Roman Catholic Church in secular affairs) father. Mussolini\'s father was a blacksmith. Before being involved in politics, Mussolini was a newspaper editor (where he learned all his propaganda skills) and elementary school teacher.\n\nAt first, Mussolini was a socialist, but when he wanted Italy to join the First World War, he was thrown out of the socialist party. He \'invented\' a new ideology, Fascism, much out of Nationalist\xa0and Conservative views.\n\nRise to power and becoming dictator\nIn 1922, he took power by having a large group of men, "Black Shirts," march on Rome and threaten to take over the government. King Vittorio Emanuele III gave in, allowed him to form a government, and made him prime minister. In the following five years, he gained power, and in 1927 created the OVRA, his personal secret police force. Using the agency to arrest, scare, or murder people against his regime, Mussolini was dictator\xa0of Italy by the end of 1927. Only the King and his own Fascist party could challenge his power.', lookup_str='', metadata={'chunk': 0.0, 'source': 'https://simple.wikipedia.org/wiki/Benito%20Mussolini', 'title': 'Benito Mussolini', 'wiki-id': '6754'}, lookup_index=0),
Document(page_content='Fascism as practiced by Mussolini\nMussolini\'s form of Fascism, "Italian Fascism"- unlike Nazism, the racist ideology that Adolf Hitler followed- was different and less destructive than Hitler\'s. Although a believer in the superiority of the Italian nation and national unity, Mussolini, unlike Hitler, is quoted "Race? It is a feeling, not a reality. Nothing will ever make me believe that biologically pure races can be shown to exist today".\n\nMussolini wanted Italy to become a new Roman Empire. In 1923, he attacked the island of Corfu, and in 1924, he occupied the city state of Fiume. In 1935, he attacked the African country Abyssinia (now called Ethiopia). His forces occupied it in 1936. Italy was thrown out of the League of Nations because of this aggression. In 1939, he occupied the country Albania. In 1936, Mussolini signed an alliance with Adolf Hitler, the dictator of Germany.\n\nFall from power and death\nIn 1940, he sent Italy into the Second World War on the side of the Axis countries. Mussolini attacked Greece, but he failed to conquer it. In 1943, the Allies landed in Southern Italy. The Fascist party and King Vittorio Emanuel III deposed Mussolini and put him in jail, but he was set free by the Germans, who made him ruler of the Italian Social Republic puppet state which was in a small part of Central Italy. When the war was almost over, Mussolini tried to escape to Switzerland with his mistress, Clara Petacci, but they were both captured and shot by partisans. Mussolini\'s dead body was hanged upside-down, together with his mistress and

some of Mussolini\'s helpers, on a pole at a gas station in the village of Millan, which is near the border between Italy and Switzerland.', lookup_str='', metadata={'chunk': 1.0, 'source': 'https://simple.wikipedia.org/wiki/Benito%20Mussolini', 'title': 'Benito Mussolini', 'wiki-id': '6754'}, lookup_index=0),
Document(page_content='Fascist Italy \nIn 1922, a new Italian government started. It was ruled by Benito Mussolini, the leader of Fascism in Italy. He became head of government and dictator, calling himself "Il Duce" (which means "leader" in Italian). He became friends with German dictator Adolf Hitler. Germany, Japan, and Italy became the Axis Powers. In 1940, they entered World War II together against France, Great Britain, and later the Soviet Union. During the war, Italy controlled most of the Mediterranean Sea.\n\nOn July 25, 1943, Mussolini was removed by the Great Council of Fascism. On September 8, 1943, Badoglio said that the war as an ally of Germany was ended. Italy started fighting as an ally of France and the UK, but Italian soldiers did not know whom to shoot. In Northern Italy, a movement called Resistenza started to fight against the German invaders. On April 25, 1945, much of Italy became free, while Mussolini tried to make a small Northern Italian fascist state called the Republic of Salò. The fascist state failed and Mussolini tried to flee to Switzerland and escape to Francoist Spain, but he was captured by Italian partisans. On 28 April 1945 Mussolini was executed by a partisan.\n\nAfter World War Two \n\nThe state became a republic on June 2, 1946. For the first time, women were able to vote. Italian people ended the Savoia dynasty and adopted a republic government.\n\nIn February 1947, Italy signed a peace treaty with the Allies. They lost all the colonies and some territorial areas (Istria and parts of Dalmatia).\n\nSince then Italy has joined NATO and the European Community (as a founding member). It is one of the seven biggest industrial economies in the world.\n\nTransportation \n\nThe railway network in Italy totals . It is the 17th longest in the world. High speed trains include ETR-class trains which travel at .', lookup_str='', metadata={'chunk': 5.0, 'source': 'https://simple.wikipedia.org/wiki/Italy', 'title': 'Italy', 'wiki-id': '363'}, lookup_index=0)]

All of these are good, relevant results. But what can we do with this? There are many tasks, one of the most interesting (and well supported by LangChain) is called *"Generative Question-Answering"*or GQA.

Generative Question-Answering

In GQA we take the query as a question that is to be answered by a LLM, but the LLM must answer the question based on the information it is seeing being returned from the vectorstore.

To do this we initialize a RetrievalQAobject like so:

Python
```
from langchain.chat_models import ChatOpenAIfrom langchain.chains import RetrievalQA# completion llmllm = ChatOpenAI( openai_api_key=OPENAI_API_KEY, model_name='gpt-3.5-turbo', temperature=0.0)qa = RetrievalQA.from_chain_type( llm=llm, chain_type="stuff", retriever=vectorstore.as_retriever())
```

Python
qa.run(query)

'Benito Mussolini was an Italian politician and journalist who served as the Prime Minister of Italy from 1922 until 1943. He was the leader of the National Fascist Party and invented the ideology of Fascism. Mussolini was a dictator of Italy by the end of 1927, and his form of Fascism, "Italian Fascism," was different and less destructive than Hitler\'s Nazism. Mussolini wanted Italy to become a new Roman Empire and attacked several countries, including Abyssinia (now called Ethiopia) and Greece. He was removed from power in 1943 and was executed by Italian partisans in 1945.'

We can also include the sources of information that the LLM is using to answer our question. We can do this using a slightly different version of RetrievalQAcalled RetrievalQAWithSourcesChain:

Python
from langchain.chains import RetrievalQAWithSourcesChainqa_with_sources = RetrievalQAWithSourcesChain.from_chain_type(llm=llm, chain_type="stuff", retriever=vectorstore.as_retriever())
Python
qa_with_sources(query)

{'question': 'who was Benito Mussolini?',
'answer': 'Benito Mussolini was an Italian politician and journalist who was the Prime Minister of Italy from 1922 until 1943. He was the leader of the National Fascist Party and invented the ideology of Fascism. He became dictator of Italy by the end of 1927 and was friends with German dictator Adolf Hitler. Mussolini attacked Greece and failed to conquer it. He was removed by the Great Council of Fascism in 1943 and was executed by a partisan on April 28, 1945. After the war, several Neo-Fascist movements have had success in Italy, the most important being the Movimento Sociale Italiano. His granddaughter Alessandra Mussolini has outspoken views similar to Fascism. \n',
'sources': 'https://simple.wikipedia.org/wiki/Benito%20Mussolini, https://simple.wikipedia.org/wiki/Fascism'}

Now we answer the question being asked, *and*return the source of this information being used by the LLM.

Integrations - Databricks

Databricks

Suggest Edits

Using Databricks and Pinecone to create and index vector embeddings at scale

Databricks, built on top of Apache Spark, is a powerful platform for data processing and analytics, known for its ability to efficiently handle large datasets. In this guide, we will show you how to use Spark (with Databricks) to create vector embeddingsand load them into Pinecone.

First, let's discuss why using Databricks and Pinecone is necessary in this context. When you process less than a million records, using a single machine might be sufficient. But when you work with hundreds of millions of records, you have to start thinking about how the operation scales. We need to consider two things:

1. How efficiently can we generate the embeddings at scale?
2. How efficiently would we be able to ingest and update these embeddings, at scale?

Databricks is a great tool for creating embeddings at scale: it allows us to parallelize the process over multiple machines and leverage GPUs to accelerate the process.

Pinecone lets us efficiently ingest, update and query hundreds of millions or even billions of embeddings. As a managed service, Pinecone can guarantee a very high degree of reliability and performance when it comes to datasets of this size.

Pinecone provides a specialized connector for Databricks that is optimized to ingest data from Databricks and into Pinecone. That allows the ingestion process to be completed much faster than it would have if we were to use Pinecone's REST or gRPC APIs on a large-scale dataset.

Together, Pinecone and Databricks make a great combination for managing the entire lifecycle of vector embeddings at scale.

Why Databricks?

Databricks is a Unified Analytics Platform on top of Apache Spark. The primary advantage of using Spark is its ability to distribute the workload across a cluster of machines, allowing it to process large amounts of data quickly and efficiently. By adding more machines or increasing the number of cores on each machine, it is easy to horizontally scale the cluster as needed to handle larger workloads.

At the core of Spark is the map-reduce pattern, where data is divided into partitions and a series of transformations is applied to each partition in parallel. The results from each partition are then automatically collected and aggregated into the final result. This approach makes Spark both fast and fault-tolerant, as it can retry failed tasks without requiring the entire workload to be reprocessed.

In addition to its parallel processing capabilities, Spark allows developers to write code in popular languages like Python and Scala, which are then optimized for parallel execution under the covers. This makes it easier for developers to focus on the data processing itself, rather than worrying about the details of distributed computing.

Vector embedding is a computationally intensive task, where parallelization can save many hours of precious computation time and resources. Leveraging GPUs with Spark can produce even better results — enjoying the benefits of the fast computation of a GPU combined with parallelization will ensure optimal performance.

Databricks makes it easier to work with Apache Spark: it provides easy set-up and tear-down of clusters, dependency management, compute allocation, storage solution integrations, and more.

Why Pinecone?

Pinecone is a vector database that makes it easy to build high-performance vector search applications. It offers a number of key benefits for dealing with vector embeddings at scale, including ultra-low query latency at any scale, live index updates when you add, edit, or delete data, and the ability to combine vector search with metadata filtering or keyword searchfor more relevant results. As mentioned before, Pinecone can easily handle very large scales of hundreds of millions and even billions of vector embeddings. Additionally, Pinecone is fully managed, so it's easy to use and scale.

With Pinecone, you can easily index and search through vector embeddings. It is ideal for a variety of use cases such as semantic text search, question-answering, visual search, recommendation systems, and more.

In this example, we'll create embeddings based on the sentence-transformers/all-MiniLM-L6-v2model from Hugging Face. We'll then use a dataset with a large volume of documents to produce the embeddings and upsert them into Pinecone. Note that the actual model and dataset we'll use are immaterial for this example. This method should work on any embeddings you may want to create, with whatever dataset you may choose.

In order to create embeddings at scale, we need to do four things:

1. Set up a Spark cluster
2. Load the dataset into partitions
3. Apply an embedding model on each entry to produce the embedding
4. Save the results

Let's get started!

Setting up a Spark Cluster

Using Databricks makes it easy to speed up the creation of our embedding even more by using GPUs instead of CPUs in our cluster. To do this, navigate to the "Compute" section in your Databricks console, and select the following options:

Next, we'll add the Pinecone Spark connector to our cluster. Navigate to the "Libraries" tab and click "Instal" new".

Select "DBF"/S3" and paste the following S3 URI:

s3://pinecone-jars/spark-pinecone-uberjar.jar

To complete the installation, click "Install". To use the new cluster, create a new notebook and attach it to the newly created cluster.

Environment Setup

We'll start by installing some dependencies:

%pip install datasets transformers pinecone-client torch

Next, we'll set up the connection to Pinecone. You'll have to retrieve the following information from your Pinecone console:

1. API Key: navigate to your project and click the "API Keys" button on the sidebar
2. Environment: check the browser url to fetch the environment. https://app.pinecone.io/organizations/[org-id]/projects/[environment]:[project_name]/indexes

Your index name will be the same index name used when we initialized the index (in this case, news).

Python
```
import pinecone

api_key = # <YOUR_PINECONE_API_KEY>
environment = 'us-west1-gcp'
pinecone.init(api_key=api_key, environment=environment)
```

Next, we'll create a new index in Pinecone, where our vector embeddings will be saved:

Python
```
index_name = 'news'

if index_name in pinecone.list_indexes():
pinecone.delete_index(index_name)
pinecone.create_index(name=index_name, dimension=384)
index = pinecone.Index(index_name=index_name)
```

Load the dataset into partitions

In this example, we'll use a collection of news articles as our example dataset. We'll use Hugging Face's datasets library and load the data into our environment:

Python
```
from datasets import list_datasets, load_dataset

dataset_name = "allenai/multinews_sparse_max"
dataset = load_dataset(dataset_name, split="train")
```
Next, we'll convert the dataset from the Hugging Face format and repartition it:

Python
```
dataset.to_parquet('/dbfs/tmp/dataset_parquet.pq')
num_workers = 10
dataset_df = spark.read.parquet('/tmp/dataset_parquet.pq').repartition(num_workers)
```
Once the repartition is complete, we get back a DataFrame, which is a distributed collection of the data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. As mentioned above, each partition in the dataframe has an equal amount of the original data.

The dataset doesn't have identifiers associated with each document, so let's add them:

Python
```
from pyspark.sql.types import StringType
from pyspark.sql.functions import monotonically_increasing_id

dataset_df = dataset_df.withColumn('id', monotonically_increasing_id().cast(StringType()))
```
As its name suggests, withColumn adds a column to the dataframe, containing a simple increasing identifier that we cast to a string. Great! Now we have identifiers for each document. Let's move on to creating the embeddings for each document.

Create a function for transforming text into embeddings

In this example, we will create a UDF (User Defined Function) to create the embeddings, using the AutoTokenizer and AutoModel classes from the Hugging Face transformers library. The UDF will be applied to each partition in a dataframe. When applied to a partition, a UDF is executed on each row in the partition. The UDF will tokenize the document using AutoTokenzier and then pass the result to the model (in this case we're using sentence-transformers/all-MiniLM-L6-v2). Finally, we'll produce the embeddings themselves by extracting the last hidden layer from the result.

Once the UDF is created, it can be applied to a dataframe to transform the data in the specified column. The Python UDF will be sent to the Spark workers, where it will be used to transform the data. After the transformation is complete, the results will be sent back to the driver program and stored in a new column.

Python
```
from transformers import AutoTokenizer, AutoModel

def create_embeddings(partitionData):
tokenizer = AutoTokenizer.from_pretrained("sentence-transformers/all-MiniLM-L6-v2")
model = AutoModel.from_pretrained("sentence-transformers/all-MiniLM-L6-v2")

for row in partitionData:
document = str(row.document)
inputs = tokenizer(document, padding=True, truncation=True, return_tensors="pt",
max_length=512)
result = model(**inputs)
embeddings = result.last_hidden_state[:, 0, :].cpu().detach().numpy()
lst = embeddings.flatten().tolist()
yield [row.id, lst, '', '{}']
```

Applying the UDF to the data

A dataframe in Spark is a higher-level abstraction built on top of a more fundamental building block called an RDD - or Resilient Distributed Dataset. We're going to use the mapPartitionsfunction that gives us finer control over the execution of our UDF, by explicitly applying it to each partition of the RDD.

Python
```
embeddings = dataset_df.rdd.mapPartitions(create_embeddings)
```
Next, we'll convert the resulting RDD back into a dataframe with the schema required by Pinecone:

Python

```
from pyspark.sql.types import StructType,StructField, ArrayType, FloatType

schema = StructType([
StructField("id",StringType(),True),
StructField("vector",ArrayType(FloatType()),True),
StructField("namespace",StringType(),True),
StructField("metadata", StringType(), True),
])

embeddings_df = spark.createDataFrame(data=embeddings,schema=schema)
```

Upserting the embeddings

Lastly, we'll use the Pinecone Spark connector to save the embeddings to our index.

Python

```
(
df.write
.option("pinecone.apiKey", api_key)
.option("pinecone.environment", environment)
.option("pinecone.projectName", pinecone.whoami().projectname)
.option("pinecone.indexName", index_name)
.format("io.pinecone.spark.pinecone.Pinecone")
.mode("append")
.save()
)
```

The process of writing the embeddings to Pinecone should take approximately 15 seconds. When it completes, you'll see the following:

```
spark: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@41638051


pineconeOptions: scala.collection.immutable.Map[String,String] = Map(pinecone.apiKey
-><YOUR API KEY>, pinecone.environment -> us-west1-gcp, pinecone.projectName -><YOUR
PROJECT NAME>, pinecone.indexName -> "news")
```

This means the process was completed successfully and the embeddings have been stored in Pinecone.

Summary

Creating vector embeddings for large datasets can be challenging, but Databricks a great tool to accomplish the task. Databricks makes it easy to set up a GPU cluster and handle the required dependencies, allowing for efficient creation of embeddings at scale.

Databricks and Pinecone are the perfect combination for working with very large vector datasets. Pinecone provides a way to efficiently store and retrieve the vectors created by Databricks, making it easy and performant to work with a huge number of vectors. Overall, the combination of Databricks and Pinecone provides a powerful and effective solution for creating embeddings for very large datasets. By parallelizing the embedding generation and the data ingestion processes, we can create a fast and resilient pipeline that will be able to index and update large volumes of vectors.

Integrations - Elasticsearch

Elasticsearchis a powerful open-source search engine and analytics platform that is widely used as a document store for keyword-based text search.

Pinecone is a vector databasewidely used for production applications — such as semantic search, recommenders, and threat detection — that require fast and fresh vector search at the scale of tens or hundreds of millions (or even billions) of embeddings. Although Pinecone offers hybrid searchfor keyword-aware semantic search, Pinecone is not a document store and does not replace Elasticsearch for keyword-only retrieval.

If you already use Elasticsearch and want to add Pinecone's low-latency and large-scale vector search to your applications, this guide will show you how. You will see how to:

- Add an embedding model to Elasticsearch
- Transform text data into vector embeddings within Elasticsearch
- Load those vector embeddings into Pinecone, with corresponding IDs and metadata.

Uploading the embedding model

We first need to upload the embedding model to our Elastic instance. To do so, we'll use the [eland](https://github.com/elastic/eland)Elastic client. We'll have to clone the "eland" repository and build the docker image before running it:

Bash
git clone git@github.com:elastic/eland.git
cd eland
docker build -t elastic/eland .

In this example, we'll use the [sentence-transformers/msmarco-MiniLM-L-12-v3](https://huggingface.co/sentence-transformers/msmarco-MiniLM-L-12-v3)model from Hugging Face— although you could use any model you'd like. To upload the model to your Elasticsearch deployment, run the following command:

Bash
```
docker run -it --rm elastic/eland \
eland_import_hub_model \
--url https://<user>:<password>@<host>:<port>/ \
--hub-model-id sentence-transformers/msmarco-MiniLM-L-12-v3 \
--task-type text_embedding \
--start
```
Note that you'll have to replace the placeholders with your Elasticsearch instance user, password, host, and port. If you set up your own Elasticsearch instance, you would have already set the username and password when initially setting up the instance. If you're using the hosted Elastic Stack, you can find the username and password in the "Security" section of the Elastic Stack console.

We can quickly test the uploaded model by running the following command in the Elasticsearch developer console:

```
POST /_ml/trained_models/sentence-transformers__msmarco-minilm-l-12-v3/deployment/_infer
{
"docs": {
"text_field": "Hello World!"
}
}
```

We should get the following result:

JSON
```
{
"predicted_value": [
-0.06176435202360153,
-0.008180409669876099,
0.3309500813484192,
0.38672536611557007,
...
]
}
```
This is the vector embedding for our query. We're now ready to upload our dataset and apply the model to produce the vector embeddings.

Uploading the dataset

Next, upload a dataset of documents to Elasticsearch. In this example, we'll use a subset of the MSMacro dataset. You can download the fileor run the following command:

Bash
curl -O
https://msmarco.blob.core.windows.net/msmarcoranking/msmarco-passagetest2019-top1000.tsv.gz
gunzip msmarco-passagetest2019-top1000.tsv
In this example, we'll be using the hosted Elastic Stack, which makes it easier to use various integrations. We'll use the "Upload" integration to load the data into an Elasticsearch index.

We'll drag the unzipped TSV file. The Upload integration will sample the data for us and show the following:

We'll click the "Import" button and continue to name the index:

Once the import is complete, you'll see the following:

Clicking "View index in Discover" will reveal the index view where we can look at the uploaded data:

Creating the embeddings

We've now created an index for our data. Next, we'll create a pipeline to produce a vector embedding for each document. We'll head to the Elasticsearch developer console and issue the following command to create the pipeline:

```
PUT _ingest/pipeline/produce-embeddings
{
"description": "Vector embedding pipeline",
"processors": [
{
"inference": {
"model_id": "sentence-transformers__msmarco-minilm-l-12-v3",
"target_field": "text_embedding",
"field_map": {
"text": "text_field"
}
}
}
],
```

```
"on_failure": [
{
"set": {
"description": "Index document to 'failed-<index>'",
"field": "_index",
"value": "failed-{{{_index}}}"
}
},
{
"set": {
"description": "Set error message",
"field": "ingest.failure",
"value": "{{_ingest.on_failure_message}}"
}
}
]
}
```

The "processor" definition tells Elasticsearch which model to use and which field to read from. The "on_failure" definition defines the failure behavior that Elasticsearch will apply — specifically, which error message to write and which file to write them into.

Once the embedding pipeline is created, we'll re-index our "msmacro-raw" index, applying the embedding pipeline to produce the new embeddings. In the developer console, execute the following command:

```
POST _reindex?wait_for_completion=false
{
"source": {
"index": "msmacro-raw"
},
"dest": {
"index": "msmacro-with-embeddings",
"pipeline": "text-embeddings"
}
}
```

This will kick off the embedding pipeline. We'll get a task id which we can track with the following command:

```
GET _tasks/<task_id>
```

Looking at the index, we can see that the embeddings have been created in an object called "text_embeddings" under the field "predicted_value".

To make the loading process a bit easier, we're going to pluck the "predicted_value" field and add it as its own column:

```
POST _reindex?wait_for_completion=false
{
"source": {
"index": "msmacro-with-embeddings"
},
"dest": {
"index": "msmacro-with-embeddings-flat"
},
"script": {
"source": "ctx._source.predicted_value = ctx._source.text_embedding.predicted_value"
}
}
```

Next, we'll load the embeddings into Pinecone. Since the index size is considerable, we'll use Apache Spark to parallelize the process.

Moving the Elasticsearch index to Pinecone

In this example, we'll be using Databricks to handle the process of loading Elasticsearch index to Pinecone. We'll add the Elasticsearch Spark from Maven by navigating to the "Libraries" tab in the cluster settings view, and clicking "Install new":

Use the following Maven coordinates:

org.elasticsearch:elasticsearch-spark-30_2.12:8.5.2

We'll add the Pinecone Databricks connectors from S3:

s3://pinecone-jars/spark-pinecone-uberjar.jar

Restart the cluster if needed. Next, we'll create a new notebook, attach it to the cluster and import the required dependencies:

Scala

```
import org.apache.spark.SparkConf
import org.apache.spark.sql.SparkSession
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.elasticsearch.spark._
```

We'll initialize the Spark context:

Scala
```
val spark = SparkSession.builder.appName("elasticSpark").master("local[*]").getOrCreate()
```
Next, we'll read the index from Elasticsearch:

Scala
```
val df = (spark.read
.format( "org.elasticsearch.spark.sql" )
.option( "es.nodes", "<ELASTIC_URL>" )
.option( "es.net.http.auth.user", "<ELASTIC_USER>" )
.option( "es.net.http.auth.pass", "<ELASTIC_PASSWORD>" )
.option( "es.port", 443 )
.option( "es.nodes.wan.only", "true" )
.option("es.net.ssl", "true")
.option("es.read.field.as.array.include","predicted_value:1")
.load( "msmacro-with-embeddings")
)
```
Note that to ensure the index is read correctly into the dataframe, we must specify that the "predicted_value" field is an array with a depth of 1, as shown below:

Scala
```
.option("es.read.field.as.array.include","predicted_value:1")
```
Next, we'll use the Pinecone Spark connector to load this dataframe into a Pinecone index. We'll start by creating an index in the Pinecone console. Log in to the console and click "Create Index". Then, name your index, and configure it to use 384 dimensions.

When you're done configuring the index, click "Create Index".

We have to do some prep work to get the dataframe ready for indexing. In order to index the original document with the embeddings we've created, we'll create the following UDF which will encode the original document as a Base64 string. This will ensure the metadata object will remain a valid JSON object regardless of the content of the document.

Scala
```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions.udf
import java.util.Base64

val text_to_metadata = udf((text: String) => "{ \"document\" : \"" +
Base64.getEncoder.encodeToString(text.getBytes("UTF-8")) + "\" }")
```
We'll apply the UDF and get rid of some unnecessary columns:

Scala

```
val clean_df = df.drop("text_embedding").withColumnRenamed("predicted_value",
"vector").withColumn("metadata", text_to_metadata(col("text_field"))).withColumn("namespace",
lit("")).drop("text_field")
```

Next, we'll use the Pinecone Spark connector:

Scala

```
val pineconeOptions = Map(
"pinecone.apiKey" -> "<PINECONE_API_KEY>",
"pinecone.environment" -> "us-west1-gcp",
"pinecone.projectName" -> "<PROJECT_IDENTIFIER>",
"pinecone.indexName" -> "elastic-index"
)

clean_df.write
.options(pineconeOptions)
.format("io.pinecone.spark.pinecone.Pinecone")
.mode(SaveMode.Append)
.save()
```

Our vectors have been added to our Pinecone index!

To query the index, we'll need to generate a vector embedding for our query first, using the sentence-transformers/msmarco-MiniLM-L-12-v3model. Then, we'll use the Pinecone client to issue the query. We'll do this in a Python notebook.

We'll start by installing the required dependencies:

```
!pip install -qU pinecone-client sentence-transformers pandas
```

Next, we'll set up the client:

Python

```
import pinecone

# connect to pinecone environment
pinecone.init(
api_key="<PINECONE API KEY>",
environment="us-west1-gcp"
)
```

We'll set up the index:

Python

```
index_name = "elastic-index"
index = pinecone.Index(index_name)
```

We'll create a helper function that will decode the encoded documents we get:

Python
```
def decode_entries(entries):
return list(map(lambda entry: {
"id": entry["id"],
"score": entry["score"],
"document": base64.b64decode(entry["metadata"]["document"]).decode("UTF-8"),
}, entries))
```
Next, we'll create a function that will encode our query, query the index and convert the display the data using Pandas:

Python
```
def queryIndex(query, num_results):
vector = model.encode(query).tolist()
result = index.query(vector, top_k=num_results, include_metadata=True)
return pd.DataFrame(decode_entries(result.matches))
```
Finally, we'll test our index:

Python
```
display(queryIndex("star trek", 10))
```
Should yield the results:

Summary

In conclusion, by following the steps outlined in this post, you can easily upload an embedding model to Elasticsearch, ingest raw textual data, create the embeddings, and load them into Pinecone. With this approach, you can take advantage of the benefits of integrating Elasticsearch and Pinecone. As mentioned, while Elasticsearch is optimized for indexing documents, Pinecone provides vector storage and search capabilities that can handle hundreds of millions and even billions of vectors.

Integrations - Hugging Face Inference Endpoints

Hugging Face Inference Endpoints

Suggest Edits

Hugging Face Inference Endpoints allows access to straightforward model inference. Coupled with Pinecone we can generate and index high-quality vector embeddings with ease.

Let's get started by initializing an Inference Endpoint for generating vector embeddings.

Endpoints

We start by heading over to the Hugging Face Inference Endpoints homepageand signing up for an account if needed. After, we should find ourselves on this page:

We click on Create new endpoint, choose a model repository (eg name of the model), endpoint name (this can be anything), and select a cloud environment. Before moving on it is *very important*that we set the Taskto Sentence Embeddings(found within the *Advanced configuration*settings).

Other important options include the *Instance Type*, by default this uses CPU which is cheaper but also slower. For faster processing we need a GPU instance. And finally, we set our privacy setting near the end of the page.

After setting our options we can click Create Endpointat the bottom of the page. This action should take use to the next page where we will see the current status of our endpoint.

Once the status has moved from Buildingto Running(this can take some time), we're ready to begin creating embeddings with it.

Creating Embeddings

Each endpoint is given an Endpoint URL, it can be found on the endpoint Overviewpage. We need to assign this endpoint URL to the endpoint_urlvariable.

Python
endpoint = "ENDPOINT_URL"
We will also need the organization API token, we find this via the organization settings on Hugging Face (https://huggingface.co/organizations/<ORG_NAME>/settings/profile). This is assigned to the api_orgvariable.

Python
api_org = "API_ORG_TOKEN"
Now we're ready to create embeddings via Inference Endpoints. Let's start with a toy example.

Python
import requests

```
# add the api org token to the headers
headers = {
'Authorization': f'Bearer {api_org}'
}
# we add sentences to embed like so
json_data = {"inputs": ["a happy dog", "a sad dog"]}
# make the request
res = requests.post(
endpoint,
headers=headers,
json=json_data
)
```

We should see a 200response.

Python

```
res
```

```
<Response [200]>
```

Inside the response we should find two embeddings...

Python

```
len(res.json()['embeddings'])
```

```
2
```

We can also see the dimensionality of our embeddings like so:

Python

```
dim = len(res.json()['embeddings'][0])
dim
```

```
768
```

We will need more than two items to search through, so let's download a larger dataset. For this we will use Hugging Face datasets.

Python

```
from datasets import load_dataset

snli = load_dataset("snli", split='train')
snli
```

Downloading: 100%|████████████| 1.93k/1.93k [00:00<00:00, 992kB/s]
Downloading: 100%|████████████| 1.26M/1.26M [00:00<00:00, 31.2MB/s]
Downloading: 100%|████████████| 65.9M/65.9M [00:01<00:00, 57.9MB/s]
Downloading: 100%|████████████| 1.26M/1.26M [00:00<00:00, 43.6MB/s]

Dataset({
features: ['premise', 'hypothesis', 'label'],
num_rows: 550152
})

SNLI contains 550K sentence pairs, many of these include duplicate items so we will take just one set of these (the *hypothesis*) and deduplicate them.

Python
passages = list(set(snli['hypothesis']))
len(passages)

480042

We will drop to 50K sentences so that the example is quick to run, if you have time, feel free to keep the full 480K.

Python
passages = passages[:50_000]

Vector DB

With our endpoint and dataset ready, all that we're missing is a vector database. For this, we need to initialize our connection to Pinecone, this requires a free API key.

Python
import pinecone

# initialize connection to pinecone (get API key at app.pinecone.io)
pinecone.init(api_key="YOUR_API_KEY", environment="YOUR_ENVIRONMENT")

Now we create a new index called 'hf-endpoints', the name isn't important *but*the dimensionmust align to our endpoint model output dimensionality (we found this in dimabove) and the model metric (typically cosineis okay, but not for all models).

Python
index_name = 'hf-endpoints'

```
# check if the hf-endpoints index exists
if index_name not in pinecone.list_indexes():
# create the index if it does not exist
pinecone.create_index(
index_name,
dimension=dim,
metric="cosine"
)

# connect to hf-endpoints index we created
index = pinecone.Index(index_name)
```

Create and Index Embeddings

Now we have all of our components ready; endpoints, dataset, and Pinecone. Let's go ahead and create our dataset embeddings and index them within Pinecone.

Python

```
from tqdm.auto import tqdm

# we will use batches of 64
batch_size = 64

for i in tqdm(range(0, len(passages), batch_size)):
# find end of batch
i_end = min(i+batch_size, len(passages))
# extract batch
batch = passages[i:i_end]
# generate embeddings for batch via endpoints
res = requests.post(
endpoint,
headers=headers,
json={"inputs": batch}
)
emb = res.json()['embeddings']
# get metadata (just the original text)
meta = [{'text': text} for text in batch]
# create IDs
ids = [str(x) for x in range(i, i_end)]
# add all to upsert list
to_upsert = list(zip(ids, emb, meta))
# upsert/insert these records to pinecone
_ = index.upsert(vectors=to_upsert)
```

```
# check that we have all vectors in index
index.describe_index_stats()
```

100%|██████████| 782/782 [11:02<00:00, 1.18it/s]

```
{'dimension': 768,
'index_fullness': 0.1,
'namespaces': {'': {'vector_count': 50000}},
'total_vector_count': 50000}
```

With everything indexed we can begin querying. We will take a few examples from the *premise*column of the dataset.

Python
```
query = snli['premise'][0]
print(f"Query: {query}")
# encode with HF endpoints
res = requests.post(endpoint, headers=headers, json={"inputs": query})
xq = res.json()['embeddings']
# query and return top 5
xc = index.query(xq, top_k=5, include_metadata=True)
# iterate through results and print text
print("Answers:")
for match in xc['matches']:
print(match['metadata']['text'])
```

```
Query: A person on a horse jumps over a broken down airplane.
Answers:
The horse jumps over a toy airplane.
a lady rides a horse over a plane shaped obstacle
A person getting onto a horse.
person rides horse
A woman riding a horse jumps over a bar.
```

These look good, let's try a couple more examples.

Python
```
query = snli['premise'][100]
print(f"Query: {query}")
# encode with HF endpoints
res = requests.post(endpoint, headers=headers, json={"inputs": query})
```

```
xq = res.json()['embeddings']
# query and return top 5
xc = index.query(xq, top_k=5, include_metadata=True)
# iterate through results and print text
print("Answers:")
for match in xc['matches']:
print(match['metadata']['text'])
```

Query: A woman is walking across the street eating a banana, while a man is following with his briefcase.
Answers:
A woman eats a banana and walks across a street, and there is a man trailing behind her.
A woman eats a banana split.
A woman is carrying two small watermelons and a purse while walking down the street.
The woman walked across the street.
A woman walking on the street with a monkey on her back.

And one more...

Python
```
query = snli['premise'][200]
print(f"Query: {query}")
# encode with HF endpoints
res = requests.post(endpoint, headers=headers, json={"inputs": query})
xq = res.json()['embeddings']
# query and return top 5
xc = index.query(xq, top_k=5, include_metadata=True)
# iterate through results and print text
print("Answers:")
for match in xc['matches']:
print(match['metadata']['text'])
```

Query: People on bicycles waiting at an intersection.
Answers:
A pair of people on bikes are waiting at a stoplight.
Bike riders wait to cross the street.
people on bicycles
Group of bike riders stopped in the street.
There are bicycles outside.

All of these results look excellent. If you are not planning on running your endpoint and vector DB beyond this tutorial, you can shut down both.

Once the index is deleted, you cannot use it again.

Shut down the endpoint by navigating to the Inference Endpoints Overviewpage and selecting Delete endpoint. Delete the Pinecone index with:

Python
pinecone.delete_index(index_name)

Integrations - Haystack

In this guide we will see how to integrate Pinecone and the popular Haystack libraryfor *Question-Answering*.

Installing Haystack

We start by installing the latest version of Haystack with all dependencies required for the PineconeDocumentStore.

Python
!pip install -U farm-haystack>=1.3.0 pinecone-client datasets

Initializing the PineconeDocumentStore

We initialize a PineconeDocumentStoreby providing an API key and environment name. Create an accountto get your free API key.

Python
from haystack.document_stores import PineconeDocumentStore

document_store = PineconeDocumentStore(
api_key='YOUR_API_KEY',
index='haystack-extractive-qa',
similarity="cosine",
embedding_dim=384
)

INFO - haystack.document_stores.pinecone - Index statistics: name: haystack-extractive-qa, embedding dimensions: 384, record count: 0

Data Preparation

Before adding data to the document store, we must download and convert data into the Document format that Haystack uses.

We will use the SQuAD dataset available from Hugging Face Datasets.

Python
```
from datasets import load_dataset

# load the squad dataset
data = load_dataset("squad", split="train")
```
Next, we remove duplicates and unecessary columns.

Python
```
# convert to a pandas dataframe
df = data.to_pandas()
# select only title and context column
df = df[["title", "context"]]
# drop rows containing duplicate context passages
df = df.drop_duplicates(subset="context")
df.head()
```

	title	context
0	University_of_Notre_Dame	Architecturally, the school has a Catholic cha...
5	University_of_Notre_Dame	As at most other universities, Notre Dame's st...
10	University_of_Notre_Dame	The university is the major seat of the Congre...
15	University_of_Notre_Dame	The College of Engineering was established in ...
20	University_of_Notre_Dame	All of Notre Dame's undergraduate students are...

Then convert these records into the Document format.

Python
```
from haystack import Document

docs = []
for d in df.iterrows():
d = d[1]
# create haystack document object with text content and doc metadata
doc = Document(
content=d["context"],
```

```
meta={
"title": d["title"],
'context': d['context']
}
)
docs.append(doc)
```
This Documentformat contains two fields; *'content'*for the text content or paragraphs, and *'meta'*where we can place any additional information that can later be used to apply metadata filtering in our search.

Now we upsert the documents to Pinecone.

Python
```
# upsert the data document to pinecone index
document_store.write_documents(docs)
```

Initialize Retriever

The next step is to create embeddings from these documents. We will use Haystacks EmbeddingRetrieverwith a SentenceTransformer model (multi-qa-MiniLM-L6-cos-v1) which has been designed for question-answering.

Python
```
from haystack.retriever.dense import EmbeddingRetriever

retriever = EmbeddingRetriever(
document_store=document_store,
embedding_model="multi-qa-MiniLM-L6-cos-v1",
model_format="sentence_transformers"
)
```
Then we run the PineconeDocumentStore.update_embeddingsmethod with the retrieverprovided as an argument. GPU acceleration can greatly reduce the time required for this step.

Python
```
document_store.update_embeddings(
retriever,
batch_size=16
)
```

Inspect Documents and Embeddings

We can get documents by their ID with the PineconeDocumentStore.get_documents_by_idmethod.

Python
d = document_store.get_documents_by_id(ids=['49091c797d2236e73fab510b1e9c7f6b'], return_embedding=True)[0]
From here we return can view document content with d.contentand the document embedding with d.embedding.

Initializing an Extractive QA Pipeline

An ExtractiveQAPipelinecontains three key components by default:

- a document store (PineconeDocumentStore)
- a retriever model
- a reader model

We use the deepset/electra-base-squad2model from the HuggingFace model hub as our reader model.

Python
from haystack.nodes import FARMReader

reader = FARMReader(
model_name_or_path='deepset/electra-base-squad2',
use_gpu=True
)
We are now ready to initialize the ExtractiveQAPipeline.

Python
from haystack.pipelines import ExtractiveQAPipeline

pipe = ExtractiveQAPipeline(reader, retriever)

Asking Questions

Using our QA pipeline we can begin querying with pipe.run.

Python
from haystack.utils import print_answers

query = "What was Albert Einstein famous for?"
# get the answer
answer = pipe.run(

```
query=query,
params={
"Retriever": {"top_k": 1},
}
)
# print the answer(s)
print_answers(answer)
```

```
Inferencing Samples: 100%|██████████| 1/1 [00:00<00:00, 3.53 Batches/s]
```

```
Query: What was Albert Einstein famous for?
Answers:
[ <Answer {
'answer': 'his theories of special relativity and general relativity', 'type': 'extractive', 'score':
0.993550717830658,
'context': 'Albert Einstein is known for his theories of special relativity and general relativity. He
also made important contributions to statistical mechanics,',
'offsets_in_document': [{'start': 29, 'end': 86}],
'offsets_in_context': [{'start': 29, 'end': 86}],
'document_id': '23357c05e3e46bacea556705de1ea6a5',
'meta': {
'context': 'Albert Einstein is known for his theories of special relativity and general relativity. He
also made important contributions to statistical mechanics, especially his mathematical
treatment of Brownian motion, his resolution of the paradox of specific heats, and his connection
of fluctuations and dissipation. Despite his reservations about its interpretation, Einstein also
made contributions to quantum mechanics and, indirectly, quantum field theory, primarily
through his theoretical studies of the photon.', 'title': 'Modern_history'
}
}>]
```

Python

```
query = "How much oil is Egypt producing in a day?"
# get the answer
answer = pipe.run(
query=query,
params={
"Retriever": {"top_k": 1},
}
)
# print the answer(s)
print_answers(answer)
```

Inferencing Samples: 100%|██████████| 1/1 [00:00<00:00, 3.81 Batches/s]

Query: How much oil is Egypt producing in a day?
Answers:
[<Answer {
'answer': '691,000 bbl/d', 'type': 'extractive', 'score': 0.9999906420707703,
'context': 'Egypt was producing 691,000 bbl/d of oil and 2,141.05 Tcf of natural gas (in 2013), which makes Egypt as the largest oil producer not member of the Or',
'offsets_in_document': [{'start': 20, 'end': 33}],
'offsets_in_context': [{'start': 20, 'end': 33}],
'document_id': '57ed9720050a17237e323da5e3969a9b',
'meta': {
'context': 'Egypt was producing 691,000 bbl/d of oil and 2,141.05 Tcf of natural gas (in 2013), which makes Egypt as the largest oil producer not member of the Organization of the Petroleum Exporting Countries (OPEC) and the second-largest dry natural gas producer in Africa. In 2013, Egypt was the largest consumer of oil and natural gas in Africa, as more than 20% of total oil consumption and more than 40% of total dry natural gas consumption in Africa. Also, Egypt possesses the largest oil refinery capacity in Africa 726,000 bbl/d (in 2012). Egypt is currently planning to build its first nuclear power plant in El Dabaa city, northern Egypt.', 'title': 'Egypt'
}
}>]

Python
```
query = "What are the first names of the youtube founders?"
# get the answer
answer = pipe.run(
query=query,
params={
"Retriever": {"top_k": 1},
}
)
# print the answer(s)
print_answers(answer)
```

Inferencing Samples: 100%|██████████| 1/1 [00:00<00:00, 3.83 Batches/s]

Query: What are the first names of the youtube founders?

Answers:
[<Answer {
'answer': 'Hurley and Chen', 'type': 'extractive', 'score': 0.9998972713947296,
'context': 'According to a story that has often been repeated in the media, Hurley and Chen developed the idea for YouTube during the early months of 2005, after ',
'offsets_in_document': [{'start': 64, 'end': 79}],
'offsets_in_context': [{'start': 64, 'end': 79}],
'document_id': 'bd1cbd61ab617d840c5f295e21e80092',
'meta': {
'context': 'According to a story that has often been repeated in the media, Hurley and Chen developed the idea for YouTube during the early months of 2005, after they had experienced difficulty sharing videos that had been shot at a dinner party at Chen\'s apartment in San Francisco. Karim did not attend the party and denied that it had occurred, but Chen commented that the idea that YouTube was founded after a dinner party "was probably very strengthened by marketing ideas around creating a story that was very digestible".', 'title': 'YouTube'
}
}>]

We can return multiple answers by setting the top_kparameter.

Python
```
query = "Who was the first person to step foot on the moon?"
# get the answer
answer = pipe.run(
query=query,
params={
"Retriever": {"top_k": 3},
}
)
# print the answer(s)
print_answers(answer)
```

Inferencing Samples: 100%|██████████| 1/1 [00:00<00:00, 3.71 Batches/s]
Inferencing Samples: 100%|██████████| 1/1 [00:00<00:00, 3.78 Batches/s]
Inferencing Samples: 100%|██████████| 1/1 [00:00<00:00, 3.88 Batches/s]

Query: Who was the first person to step foot on the moon?
Answers:
[<Answer {
'answer': 'Armstrong', 'type': 'extractive', 'score': 0.9998227059841156,

'context': 'The trip to the Moon took just over three days. After achieving orbit, Armstrong and Aldrin transferred into the Lunar Module, named Eagle, and after ',
'offsets_in_document': [{'start': 71, 'end': 80}],
'offsets_in_context': [{'start': 71, 'end': 80}],
'document_id': 'f74e1bf667e68d72e45437a7895df921',
'meta': {
'context': 'The trip to the Moon took just over three days. After achieving orbit, Armstrong and Aldrin transferred into the Lunar Module, named Eagle, and after a landing gear inspection by Collins remaining in the Command/Service Module Columbia, began their descent. After overcoming several computer overload alarms caused by an antenna switch left in the wrong position, and a slight downrange error, Armstrong took over manual flight control at about 180 meters (590 ft), and guided the Lunar Module to a safe landing spot at 20:18:04 UTC, July 20, 1969 (3:17:04 pm CDT). The first humans on the Moon would wait another six hours before they ventured out of their craft. At 02:56 UTC, July 21 (9:56 pm CDT July 20), Armstrong became the first human to set foot on the Moon.', 'title': 'Space_Race'
}
}>, <Answer {
'answer': 'Frank Borman', 'type': 'extractive', 'score': 0.7770257890224457,
'context': 'On December 21, 1968, Frank Borman, James Lovell, and William Anders became the first humans to ride the Saturn V rocket into space on Apollo 8. They ',
'offsets_in_document': [{'start': 22, 'end': 34}],
'offsets_in_context': [{'start': 22, 'end': 34}],
'document_id': '2bc046ba90d94fe201ccde9d20552200',
'meta': {
'context': "On December 21, 1968, Frank Borman, James Lovell, and William Anders became the first humans to ride the Saturn V rocket into space on Apollo 8. They also became the first to leave low-Earth orbit and go to another celestial body, and entered lunar orbit on December 24. They made ten orbits in twenty hours, and transmitted one of the most watched TV broadcasts in history, with their Christmas Eve program from lunar orbit, that concluded with a reading from the biblical Book of Genesis. Two and a half hours after the broadcast, they fired their engine to perform the first trans-Earth injection to leave lunar orbit and return to the Earth. Apollo 8 safely landed in the Pacific ocean on December 27, in NASA's first dawn splashdown and recovery.",
'title': 'Space_Race'
}
}>, <Answer {
'answer': 'Aldrin', 'type': 'extractive', 'score': 0.6680101901292801,
'context': ' were, "That\'s one small step for [a] man, one giant leap for mankind." Aldrin joined him on the surface almost 20 minutes later. Altogether, they spe',
'offsets_in_document': [{'start': 240, 'end': 246}],
'offsets_in_context': [{'start': 72, 'end': 78}],
'document_id': 'ae1c366b1eaf5fc9d32a8d81f76bd795',
'meta': {
'context': 'The first step was witnessed by at least one-fifth of the population of Earth, or about 723 million people. His first words when he stepped off the LM\'s landing footpad were, "That\'s

one small step for [a] man, one giant leap for mankind." Aldrin joined him on the surface almost 20 minutes later. Altogether, they spent just under two and one-quarter hours outside their craft. The next day, they performed the first launch from another celestial body, and rendezvoused back with Columbia.', 'title': 'Space_Race'
}
}>
]

Integrations - Cohere

Github Source: https://github.com/pinecone-io/examples/tree/master/integrations/cohere

Cohere

Suggest Edits
View Source

Open in Colab
In this guide you will learn how to use the Cohere Embed API endpointto generate language embeddings, and then index those embeddings in the Pinecone vector databasefor fast and scalable vector search.

This is a powerful and common combination for building semantic search, question-answering, threat-detection, and other applications that rely on NLP and search over a large corpus of text data.

The basic workflow looks like this:

- Embed and index
    - Use the Cohere Embed API endpoint to generate vector embeddings of your documents (or any text data).
    - Upload those vector embeddings into Pinecone, which can store and index millions/billions of these vector embeddings, and search through them at ultra-low latencies.
- Search
    - Pass your query text or document through the Cohere Embed API endpoint again.
    - Take the resulting vector embedding and send it as a queryto Pinecone.

- Get back semantically similar documents, even if they don't share any keywords with the query.

Let's get started...

https://files.readme.io/fd0ba7b-pinecone-cohere-overview.png

Environment Setup

We start by installing the Cohere and Pinecone clients, we will also need HuggingFace *Datasets*for downloading the TREC dataset that we will use in this guide.

Bash
pip install -U cohere pinecone-client datasets

Creating Embeddings

To create embeddings we must first initialize our connection to Cohere, we sign up for an API key at Cohere.

Python
import cohere

co = cohere.Client("YOUR_API_KEY")
We will load the Text REtrieval Conference (TREC) question classification dataset which contains 5.5K labeled questions. We will take the first 1K samples for this walkthrough, but this can be scaled to millions or even billions of samples.

Python
from datasets import load_dataset

# load the first 1K rows of the TREC dataset
trec = load_dataset('trec', split='train[:1000]')
Each sample in treccontains two label features and the *text*feature, which we will be using. We can pass the questions from the *text*feature to Cohere to create embeddings.

Python
embeds = co.embed(
texts=trec['text'],
model='small',
truncate='LEFT'
).embeddings

We can check the dimensionality of the returned vectors, for this we will convert it from a list of lists to a Numpy array. We will need to save the embedding dimensionality from this to be used when initializing our Pinecone index later.

Python
```
import numpy as np

shape = np.array(embeds).shape
print(shape)
```

[Out]:
```
(1000, 1024)
```

Here we can see the 1024embedding dimensionality produced by Cohere's small model, and the 1000samples we built embeddings for.

Storing the Embeddings

Now that we have our embeddings we can move on to indexing them in the Pinecone vector database. For this we need a Pinecone API key, sign up for one here.

We first initialize our connection to Pinecone, and then create a new index for storing the embeddings (we will call it "cohere-pinecone-trec"). When creating the index we specify that we would like to use the cosine similarity metric to align with Cohere's embeddings, and also pass the embedding dimensionality of 1024.

Python
```
import pinecone

# initialize connection to pinecone (get API key at app.pinecone.io)
pinecone.init(api_key="YOUR_API_KEY", environment="YOUR_ENVIRONMENT")

index_name = 'cohere-pinecone-trec'

# if the index does not exist, we create it
if index_name not in pinecone.list_indexes():
pinecone.create_index(
index_name,
dimension=shape[1],
metric='cosine'
)

# connect to index
```

index = pinecone.Index(index_name)
Now we can begin populating the index with our embeddings. Pinecone expects us to provide a list of tuples in the format *(id, vector, metadata)*, where the *metadata*field is an optional extra field where we can store anything we want in a dictionary format. For this example, we will store the original text of the embeddings.

⚠️Warning

High-cardinality metadata values (like the unique text values we use here)
can reduce the number of vectors that fit on a single pod. See
Limitsfor more.
While uploading our data, we will batch everything to avoid pushing too much data in one go.

Python
batch_size = 128

ids = [str(i) for i in range(shape[0])]
# create list of metadata dictionaries
meta = [{'text': text} for text in trec['text']]

# create list of (id, vector, metadata) tuples to be upserted
to_upsert = list(zip(ids, embeds, meta))

for i in range(0, shape[0], batch_size):
i_end = min(i+batch_size, shape[0])
index.upsert(vectors=to_upsert[i:i_end])

# let's view the index statistics
print(index.describe_index_stats())

[Out]:
{'dimension': 1024,
'index_fullness': 0.0,
'namespaces': {'': {'vector_count': 1000}}}

We can see from index.describe_index_statsthat we have a *1024-dimensionality*index populated with *1000*embeddings. The indexFullnessmetric tells us how full our index is, at the moment it is empty. Using the default value of one *p1*pod we can fit around 750K embeddings before the indexFullnessreaches capacity. The Usage Estimatorcan be used to identify the number of pods required for a given number of *n*-dimensional embeddings.

Semantic Search

Now that we have our indexed vectors we can perform a few search queries. When searching we will first embed our query using Cohere, and then search using the returned vector in Pinecone.

Python
```
query = "What caused the 1929 Great Depression?"

# create the query embedding
xq = co.embed(
texts=[query],
model='small',
truncate='LEFT'
).embeddings

print(np.array(xq).shape)

# query, returning the top 5 most similar results
res = index.query(xq, top_k=5, include_metadata=True)
```
The response from Pinecone includes our original text in the metadatafield, let's print out the top_kmost similar questions and their respective similarity scores.

Python
```
for match in res['matches']:
print(f"{match['score']:.2f}: {match['metadata']['text']}")
```

[Out]:
```
0.83: Why did the world enter a global depression in 1929 ?
0.75: When was `` the Great Depression " ?
0.50: What crop failure caused the Irish Famine ?
0.34: What war did the Wanna-Go-Home Riots occur after ?
0.34: What were popular songs and types of songs in the 1920s ?
```

Looks good, let's make it harder and replace *"depression"*with the incorrect term *"recession"*.

Python
```
query = "What was the cause of the major recession in the early 20th century?"

# create the query embedding
xq = co.embed(
texts=[query],
model='small',
truncate='LEFT'
).embeddings
```

```
# query, returning the top 5 most similar results
res = index.query(xq, top_k=5, include_metadata=True)

for match in res['matches']:
print(f"{match['score']:.2f}: {match['metadata']['text']}")

[Out]:
0.66: Why did the world enter a global depression in 1929 ?
0.61: When was `` the Great Depression " ?
0.43: What are some of the significant historical events of the 1990s ?
0.43: What crop failure caused the Irish Famine ?
0.37: What were popular songs and types of songs in the 1920s ?
```

Let's perform one final search using the definition of depression rather than the word or related words.

```
Python
query = "Why was there a long-term economic downturn in the early 20th century?"

# create the query embedding
xq = co.embed(
texts=[query],
model='small',
truncate='LEFT'
).embeddings

# query, returning the top 10 most similar results
res = index.query(xq, top_k=10, include_metadata=True)

for match in res['matches']:
print(f"{match['score']:.2f}: {match['metadata']['text']}")

[Out]:
0.71: Why did the world enter a global depression in 1929 ?
0.62: When was `` the Great Depression " ?
0.40: What crop failure caused the Irish Famine ?
0.38: What are some of the significant historical events of the 1990s ?
0.38: When did the Dow first reach ?
```

It's clear from this example that the semantic search pipeline is clearly able to identify the meaning between each of our queries. Using these embeddings with Pinecone allows us to return the most semantically similar questions from the already indexed TREC dataset.

Integrations - OpenAI

Github Source: https://github.com/pinecone-io/examples/tree/master/integrations/openai

This guide covers the integration of OpenAI's Large Language Models (LLMs) with Pinecone (referred to as the OP stack), enhancing semantic search or 'long-term memory' for LLMs. This combo utilizes LLMs' embedding and completion (or generation) endpoints, alongside Pinecone's vector search capabilities, for nuanced information retrieval.

LLMs like OpenAI's text-embedding-ada-002generate vector embeddings, numerical representations of text semantics. These embeddings facilitate semantic-based rather than literal textual matches. Additionally, LLMs like gpt-4or gpt-3.5-turbopredict text completions based on previous context.

Pinecone is a vector database designed for storing and querying high-dimensional vectors. It provides fast, efficient semantic search over these vector embeddings.

By integrating OpenAI's LLMs with Pinecone, we combine deep learning capabilities for embedding generation with efficient vector storage and retrieval. This approach surpasses traditional keyword-based search, offering contextually-aware, precise results.

There are many ways of integrating these two tools and we have several guides focusing on specific use-cases. If you already know what you'd like to do you can jump to these specific materials:

- Retrieval Augmentation with GPT-4
- ChatGPT Plugins Walkthrough
- Ask Lex ChatGPT Plugin
- Generative Question-Answering
- Retrieval Augmentation using LangChain

Introduction to Embeddings

At the core of the OP stack we have embeddings which are supported via the OpenAI Embedding API. We index those embeddings in the Pinecone vector databasefor fast and scalable retrieval augmentation of our LLMs or other information retrieval use-cases.

This example demonstrates the core OP stack. It is the simplest workflow and is present in each of the other workflows, but is not the only way to use the stack. Please refer to the links above for more advanced usage.

The OP stack is built for semantic search, question-answering, threat-detection, and other applications that rely on language models and a large corpus of text data.

The basic workflow looks like this:

- Embed and index
    - Use the OpenAI Embedding API to generate vector embeddings of your documents (or any text data).
    - Upload those vector embeddings into Pinecone, which can store and index millions/billions of these vector embeddings, and search through them at ultra-low latencies.
- Search
    - Pass your query text or document through the OpenAI Embedding API again.
    - Take the resulting vector embedding and send it as a queryto Pinecone.
    - Get back semantically similar documents, even if they don't share any keywords with the query.

Let's get started...

Environment Setup

We start by installing the OpenAI and Pinecone clients, we will also need HuggingFace *Datasets*for downloading the TREC dataset that we will use in this guide.

Bash
pip install -U openai pinecone-client datasets

Creating Embeddings

To create embeddings we must first initialize our connection to OpenAI Embeddings, we sign up for an API key at OpenAI.

Python
import openai

openai.api_key = "YOUR_API_KEY"
# get API key from top-right dropdown on OpenAI website

openai.Engine.list() # check we have authenticated
The openai.Engine.list()function should return a list of models that we can use. We will use OpenAI's Ada 002 model.

Python
MODEL = "text-embedding-ada-002"

res = openai.Embedding.create(

```
input=[
"Sample document text goes here",
"there will be several phrases in each batch"
], engine=MODEL
)
```

In reswe should find a JSON-like object containing two 1536-dimensional embeddings, these are the vector representations of the two inputs provided above. To access the embeddings directly we can write:

Python

```
# extract embeddings to a list
embeds = [record['embedding'] for record in res['data']]
```

We will use this logic when creating our embeddings for the Text REtrieval Conference (TREC) question classification dataset later.

Initializing a Pinecone Index

Next, we initialize an index to store the vector embeddings. For this we need a Pinecone API key, [sign up for one here](#).

Python

```
import pinecone

# initialize connection to pinecone (get API key at app.pinecone.io)
pinecone.init(
api_key="YOUR_API_KEY",
environment="YOUR_ENV" # find next to API key in console
)

# check if 'openai' index already exists (only create index if not)
if 'openai' not in pinecone.list_indexes():
pinecone.create_index('openai', dimension=len(embeds[0]))
# connect to index
index = pinecone.Index('openai')
```

Populating the Index

With both OpenAI and Pinecone connections initialized, we can move onto populating the index. For this, we need the TREC dataset.

Python

```
from datasets import load_dataset

# load the first 1K rows of the TREC dataset
```

trec = load_dataset('trec', split='train[:1000]')
Then we create a vector embedding for each question using OpenAI (as demonstrated earlier), and upsertthe ID, vector embedding, and original text for each phrase to Pinecone.

⚠️Warning

High-cardinality metadata values (like the unique text values we use here) can reduce the number of vectors that fit on a single pod. See Limitsfor more.

Python
```
from tqdm.auto import tqdm # this is our progress bar

batch_size = 32 # process everything in batches of 32
for i in tqdm(range(0, len(trec['text']), batch_size)):
# set end position of batch
i_end = min(i+batch_size, len(trec['text']))
# get batch of lines and IDs
lines_batch = trec['text'][i: i+batch_size]
ids_batch = [str(n) for n in range(i, i_end)]
# create embeddings
res = openai.Embedding.create(input=lines_batch, engine=MODEL)
embeds = [record['embedding'] for record in res['data']]
# prep metadata and upsert batch
meta = [{'text': line} for line in lines_batch]
to_upsert = zip(ids_batch, embeds, meta)
# upsert to Pinecone
index.upsert(vectors=list(to_upsert))
```

Querying

With our data indexed, we're now ready to move onto performing searches. This follows a similar process to indexing. We start with a text query, that we would like to use to find similar sentences. As before we encode this with OpenAI's text similarity Babbage model to create a *query vector*xq. We then use xqto query the Pinecone index.

Python
```
query = "What caused the 1929 Great Depression?"

xq = openai.Embedding.create(input=query, engine=MODEL)['data'][0]['embedding']
```
Now we query.

Python
```
res = index.query([xq], top_k=5, include_metadata=True)
```

The response from Pinecone includes our original text in the metadatafield, let's print out the top_kmost similar questions and their respective similarity scores.

Python
```
for match in res['matches']:
print(f"{match['score']:.2f}: {match['metadata']['text']}")
```

[Out]:
```
0.95: Why did the world enter a global depression in 1929 ?
0.87: When was `` the Great Depression '' ?
0.86: What crop failure caused the Irish Famine ?
0.82: What caused the Lynmouth floods ?
0.79: What caused Harry Houdini 's death ?
```

Looks good, let's make it harder and replace *"depression"*with the incorrect term *"recession"*.

Python
```
query = "What was the cause of the major recession in the early 20th century?"

# create the query embedding
xq = openai.Embedding.create(input=query, engine=MODEL)['data'][0]['embedding']

# query, returning the top 5 most similar results
res = index.query([xq], top_k=5, include_metadata=True)

for match in res['matches']:
print(f"{match['score']:.2f}: {match['metadata']['text']}")
```

[Out]:
```
0.92: Why did the world enter a global depression in 1929 ?
0.85: What crop failure caused the Irish Famine ?
0.83: When was `` the Great Depression '' ?
0.82: What are some of the significant historical events of the 1990s ?
0.82: What is considered the costliest disaster the insurance industry has ever faced ?
```

Let's perform one final search using the definition of depression rather than the word or related words.

Python
```
query = "Why was there a long-term economic downturn in the early 20th century?"

# create the query embedding
xq = openai.Embedding.create(input=query, engine=MODEL)['data'][0]['embedding']
```

```
# query, returning the top 5 most similar results
res = index.query([xq], top_k=5, include_metadata=True)

for match in res['matches']:
print(f"{match['score']:.2f}: {match['metadata']['text']}")

[Out]:
0.93: Why did the world enter a global depression in 1929 ?
0.83: What crop failure caused the Irish Famine ?
0.82: When was `` the Great Depression '' ?
0.82: How did serfdom develop in and then leave Russia ?
0.80: Why were people recruited for the Vietnam War ?
```

It's clear from this example that the semantic search pipeline is clearly able to identify the meaning between each of our queries. Using these embeddings with Pinecone allows us to return the most semantically similar questions from the already indexed TREC dataset.

Manage datasets - Creating and loading private datasets

Overview

This document explains how to create, upload, and list your dataset for use by other Pinecone users. This guide shows how to create your own dataset using your own storage; you cannot upload your own dataset to the Pinecone dataset directory.

To learn about using existing Pinecone datasets, see Working with datasets.

Creating a dataset

Dependencies

The Pinecone datasets project uses poetryfor dependency management and supports python versions 3.8+.

To install poetry, run the following command from the project root directory:

Shell
```
poetry install --with dev
```

Dataset metadata

To create a public dataset, you may need to generate dataset metadata.

Example

The following example creates a metadata object metacontaining metadata for a dataset test_dataset.

Python
```
from pinecone_datasets.catalog import DatasetMetadata

meta = DatasetMetadata(
name="test_dataset",
created_at="2023-02-17 14:17:01.481785",
documents=2,
queries=2,
source="manual",
bucket="LOCAL",
task="unittests",
dense_model={"name": "bert", "dimension": 3},
sparse_model={"name": "bm25"},
)
```
If you intend to list your dataset, you can save the dataset metadata using the following command. Write permission to location is needed.

Python
```
dataset._save_metadata("non-listed-dataset", meta)
```

Viewing dataset schema

To see the complete schema, run the following command:

Python
```
meta.schema()
```

Running tests

To run tests locally, run the following command:

Shell
```
poetry run pytest --cov pinecone_datasets
```

Uploading and listing a dataset

Pinecone datasets can load a dataset from any storage bucket where it has access using the default access controls for s3, gcs or local permissions.

Pinecone datasets expects data to be uploaded with the following directory structure:

Figure 1: Expected directory structure for Pinecone datasets

```
├── base_path # path to where all datasets
│  ├── dataset_id # name of dataset
│  │  ├── metadata.json # dataset metadata (optional, only for listed)
│  │  ├── documents # datasets documents
│  │  │  ├── file1.parquet
│  │  │  └── file2.parquet
│  │  ├── queries # dataset queries
│  │  │  ├── file1.parquet
│  │  │  └── file2.parquet
└── ...
```

Pinecone datasets scans storage and lists every dataset with metadata file.

Example

The following shows the format of an example s3 bucket address for a datasets metadata file:

s3://my-bucket/my-dataset/metadata.json

Using your own dataset

By default, the Pinecone client uses Pinecone's public datasets bucket on GCS. You can use your own bucket by setting the PINECONE_DATASETS_ENDPOINTenvironment variable.

Example

The following export command changes the default dataset storage endpoint to gs://my-bucket. Calling list_datasetsor load_datasetnow scans that bucket and list all datasets.

Python
export PINECONE_DATASETS_ENDPOINT="gs://my-bucket"
You can also use s3://as a prefix to your bucket to access an s3 bucket.

Authenticating to your own storage bucket

Pinecone Datasets supports GCS and S3 storage buckets, using default authentication as provided by the fsspec implementation: gcsfs for GCS and s3fs for AWS.

To authenticate to an AWS s3 bucket using the key/secret method, follow these steps:

1. Set a new endpoint by setting the environment variable PINECONE_DATASETS_ENDPOINTto the s3 address for your bucket.

Example

Shell
export PINECONE_DATASETS_ENDPOINT="s3://my-bucket"

1. Use the key and secret parameters to pass your credentials to the list_datasetsand load_datasetfunctions.

Example

Python
st = list_datasets(
key=os.environ.get("S3_ACCESS_KEY"),
secret=os.environ.get("S3_SECRET"),
)

ds = load_dataset(
"test_dataset",
key=os.environ.get("S3_ACCESS_KEY"),
secret=os.environ.get("S3_SECRET"),
)

Accessing a non-listed dataset

To access a non-listed dataset, load it directly using the Datasetconstructor.

Example

The following loads the dataset non-listed-dataset.

Python
from pinecone_datasets import Dataset

dataset = Dataset("non-listed-dataset")

Manage datasets - Using public Pinecone datasets

Overview

This document explains how to use existing Pinecone datasets.

To learn about creating and listing datasets, see Creating datasets.

Datasets contain vectors and metadata

Pinecone datasets contain rows of dense and sparse vector values and metadata. Pinecone's Python client supports upserting vectors from a dataset. You can also use datasets to iterate over vectors to automate queries.

Listing public datasets

To list available public Pinecone datasets, use the list_datasets()method.

Example

The following example retrieves an object containing information about public Pinecone datasets.

Python
```
from pinecone_datasets import list_datasets

list_datasets()
```
The example above returns an object like the following:

Shell
```
['ANN_DEEP1B_d96_angular', 'ANN_Fashion-MNIST_d784_euclidean',
'ANN_GIST_d960_euclidean', 'ANN_GloVe_d100_angular', 'ANN_GloVe_d200_angular',
'ANN_GloVe_d25_angular', 'ANN_GloVe_d50_angular', 'ANN_LastFM_d64_angular',
'ANN_MNIST_d784_euclidean', 'ANN_NYTimes_d256_angular',
'ANN_SIFT1M_d128_euclidean', 'quora_all-MiniLM-L6-bm25', 'quora_all-MiniLM-L6-v2_Splade']
```

Loading datasets

To load a dataset into memory, use the load_datasetmethod. You can use load a Pinecone public dataset or your own dataset.

Example

The following example loads the quora_al-MiniLM-L6-bm25Pinecone public dataset.

Python
```
from pinecone_datasets import list_datasets, load_dataset

list_datasets()
# ["quora_all-MiniLM-L6-bm25", ... ]

dataset = load_dataset("quora_all-MiniLM-L6-bm25")

dataset.head()
```
The example above prints the following output:

Shell
```
┌─────────┬──────────────────────────────┬──────────────────────────────┬──────────┬──────┐
│ id      ┆ values                       ┆ sparse_values                ┆ metadata ┆ blob │
│         ┆                              ┆                              ┆          ┆      │
│ str     ┆ list[f32]                    ┆ struct[2]                    ┆ struct[3] ┆      │
╞═════════╪══════════════════════════════╪══════════════════════════════╪══════════╪══════╡
│ 0       ┆ [0.118014, -0.069717, ...    ┆ {[470065541, 52922727, ... 22364... ┆ {2017,12,"other"} ┆ .... │
│         ┆ 0.0060...                    ┆                              ┆          ┆      │
└─────────┴──────────────────────────────┴──────────────────────────────┴──────────┴──────┘
```

Iterating over datasets

You can iterate over vector data in a dataset using the iter_documentsmethod. You can use this method to upsert or update vectors, to automate benchmarking, or other tasks.

Example

The following example loads the quora_all-MiniLM-L6-bm25dataset, then iterates over the documents in the dataset in batches of 100 and upserts the vector data to a Pinecone index named my-index.

Python

```
import pinecone
from pinecone_datasets import list_datasets, load_dataset

dataset = load_dataset("quora_all-MiniLM-L6-bm25")

pinecone.init(api_key="API_KEY", environment="us-west1-gcp")

pinecone.create_index(name="my-index", dimension=384, pod_type='s1')

index = pinecone.Index("my-index")
```

Iterate over documents in batches and upsert to an index.

Python

```
for batch in dataset.iter_documents(batch_size=100):
index.upsert(vectors=batch)
```

The following example upserts the dataset as dataframe.

Python

```
import pinecone

from pinecone_datasets import list_datasets, load_dataset

dataset = load_dataset("quora_all-MiniLM-L6-bm25")

pinecone.init(api_key="API_KEY", environment="us-west1-gcp")

pinecone.create_index(name="my-index", dimension=384, pod_type='s1')

index = pinecone.Index("my-index")
```

Upsert the dataset as a dataframe.

Python

```
index.upsert_from_dataframe(dataset.drop(columns=["blob"]))
```

Manage datasets - Pinecone public datasets

Overview

This document explains and describes Pinecone datasets.

To learn about using public Pinecone datasets, see Using public datasets.

To learn about creating and listing datasets, see Creating datasets.

Datasets contain vectors and metadata

Pinecone datasets contain rows of dense and sparse vector values and metadata. Pinecone's Python client supports upserting vectors from a dataset. You can also use datasets to iterate over vectors to automate queries.

Available public datasets

The following table lists information about public Pinecone datasets that are currently available:

name	documents	source	bucket	task	dense model	sparse model
ANN_DEEP1B_d96_angular	9,990,000	https://github.com/erikbern/ann-benchmarks	gs://pinecone-datasets-dev/ANN_DEEP1B_d96_angular	ANN	ANN benchmark	None
ANN_Fashion-MNIST_d784_euclidean	60,000	https://github.com/erikbern/ann-benchmarks	gs://pinecone-datasets-dev/ANN_Fashion-MNIST_d784_euclidean	ANN	ANN benchmark	None
ANN_GloVe_d200_angular	1,183,514	https://github.com/erikbern/ann-benchmarks	gs://pinecone-datasets-dev/ANN_GloVe_d200_angular	ANN	ANN benchmark	None
ANN_GloVe_d50_angular	1,183,514	https://github.com/erikbern/ann-benchmarks	gs://pinecone-datasets-dev/ANN_GloVe_d50_angular	ANN	ANN benchmark	None
ANN_GloVe_d64_angular	292,385	https://github.com/erikbern/ann-benchmarks	gs://pinecone-datasets-dev/ANN_GloVe_d64_angular	ANN	ANN benchmark	None

ANN_MNIST_d784_euclidean	60,000	https://github.com/erikbern/ann-benchmarks	gs://pinecone-datasets-dev/ANN_MNIST_d784_euclidean	ANN	ANN benchmark	None
ANN_NYTimes_d256_angular	290,000	https://github.com/erikbern/ann-benchmarks	gs://pinecone-datasets-dev/ANN_NYTimes_d256_angular	ANN	ANN benchmark	None
ANN_SIFT1M_d128_euclidean	1,000,000	https://github.com/erikbern/ann-benchmarks	gs://pinecone-datasets-dev/ANN_SIFT1M_d128_euclidean	ANN	ANN benchmark	None
quora_all-MiniLM-L6-bm25	522,931	https://quoradata.quora.com/First-Quora-Dataset-Release-Question-Pairs	gs://pinecone-datasets-dev/quora_all-MiniLM-L6-bm25	similar questions	sentence-transformers/msmarco-MiniLM-L6-cos-v5	naver/splade-cocondenser-ensembledistil
quora_all-MiniLM-L6-v2_Splade	522,931	https://quoradata.quora.com/First-Quora-Dataset-Release-Question-Pairs	gs://pinecone-datasets-dev/quora_all-MiniLM-L6-v2_Splade	similar questions	sentence-transformers/msmarco-MiniLM-L6-cos-v5	naver/splade-cocondenser-ensembledistil

---

Manage billing - Managing cost

Overview

This topic provides guidance on managing the cost of Pinecone. For the latest pricing details, see our pricing page. For help estimating total cost, see Understanding total cost. To see a calculation of your current usage and costs, see the usage dashboardin the Pinecone console.

The total cost of Pinecone usagederives from pod type, the number of pods in use, pod size, the total time each pod is running, and the billing plan. This topic describes several ways you can manage your overall Pinecone cost by adjusting these variables.

Use the Starter Plan for small projects or prototypes

The Starter Planincurs no costs, and supports roughly 100,000 vectors with 1536 dimensions. If this meets the needs of your project, you can use Pinecone for free; if you decide to scale your indexor move it to production, you can upgrade your billing plan later.

Choose the right pod size for your application

Different Pinecone pod sizes are designed for different applications, and some are more expensive than others. By choosing the appropriate pod type and size, you can pay for the resources you need. For example, the s1pod type provides large storage capacity and lower overall costs with slightly higher query latencies than p1pods. By switching to a different pod type, you may be able to reduce costs while still getting the performance your application needs.

Back up inactive indexes

When a specific index is not in use, back it up using collectionsand delete the inactive index. When you're ready to use these vectors again, you can create a new index from the collection. This new index can also use a different index type or size. Because it's relatively cheap to store collections, you can reduce costs by only running an index when it's in use.

Use namespaces for multitenancy

If your application requires you to separate users into groups, consider using namespaces to isolate segments of vectors within a single index. Depending on your application requirements, this may allow you to reduce the total number of active indexes.

Commit to annual spend

Users who commit to an annual contract may qualify for discounted rates. To learn more, contact Pinecone sales.

Manage billing - Understanding cost

Overview

This topic describes the calculation of total cost for Pinecone, including an example. All prices are examples; for the latest pricing details, please see our pricing page. While our pricing page lists rates on an hourly basis for ease of comparison, this topic lists prices per minute, as this is how Pinecone calculates billing.

How does Pinecone calculate costs?

For each index, billing is determined by the per-minute price per pod and the number of pods the index uses, regardless of index activity. The per-minute price varies by pod type, pod size, account plan, and cloud region.

Total cost depends on a combination of factors:

- Pod type.Each pod typehas different per-minute pricing.
- Number of pods.This includes replicas, which duplicate pods.
- Pod size.Larger pod sizes have proportionally higher costs per minute.
- Total pod-minutes.This includes the total time each pod is running, starting at pod creation and rounded up to 15-minute increments.
- Cloud provider.The cost per pod-type and pod-minute varies depending on the cloud provider you choose for your project.
- Collection storage.Collections incur costs per GB of data per minute in storage, rounded up to 15-minute increments.
- Plan.The free plan incurs no costs; the Standard or Enterprise plans incur different costs per pod-type, pod-minute, cloud provider, and collection storage.

The following equation calculates the total costs accrued over time:

(Number of pods) * (pod size) * (number of replicas) * (minutes pod exists) * (pod price per minute)

- (collection storage in GB) * (collection storage time in minutes) * (collection storage price per GB per minute)

To see a calculation of your current usage and costs, see the usage dashboardin the Pinecone console.

Example total cost calculation

An example application has the following requirements:

- 1,000,000 vectors with 1536 dimensions
- 150 queries per second with top_k= 10
- Deployment in an EU region
- Ability to store 1GB of inactive vectors

Based on these requirements, the organization chooses to configure the project to use the Standard billing plan to host one p1.x2pod with two replicas and a collection containing 1 GB of data. This project runs continuously for the month of January on the Standard plan. The components of the total cost for this example are given in Table 1 below:

Table 1: Example billing components

Billing component	Value
Number of pods	1
Number of replicas	3
Pod size	x2
Total pod count	6
Minutes in January	44,640
Pod-minutes (pods * minutes)	267,840
Pod price per minute	$0.0012
Collection storage	1 GB
Collection storage minutes	44,640
Price per storage minute	$0.00000056

The invoice for this example is given in Table 2 below:

Table 2: Example invoice

Product	Quantity	Price per unit	Charge
Collections	44,640	$0.00000056	$0.025

P2 Pods (AWS)	0		$0.00
P2 Pods (GCP)	0		$0.00
S1 Pods	0		$0.00
P1 Pods	267,840	$0.0012	$514.29

Amount due $514.54

Cost controls

Pinecone offers tools to help you understand and control your costs.

- Monitoring usage.Using the usage dashboardin the Pinecone console, you can monitor your Pinecone usage and costs as these accrue.
- Pod limits.Pinecone project owners can set limits for the total number of podsacross all indexes in the project. The default pod limit is 5.

Manage Projects - Create a project , Add users to projects and organizations, Change project pod limit, Rename a project

Create a project

Suggest Edits

Overview

ℹ️Info

Starter (free) users can only have 1 owned project. To create a new project, Starter users must upgrade to the Standard or Enterprise plan or delete their default project.
Follow these steps to create a new project:

1. Access the Pinecone Console.
2. Click Organizationsin the left menu.
3. In the Organizationsview, click the PROJECTStab.
4. Click the +CREATE PROJECTbutton.
5. Enter the Project Name.
6. Select a cloud provider and region.
7. Enter the project pod limit.
8. Click CREATE PROJECT.

Add users to projects and organizations

Suggest Edits

Overview

If you are a projector organizationowner, follow these steps to add users to organizations and projects.

Add users to projects and organizations

1. Access the Pinecone Console.
2. Click Settingsin the left menu.
3. In the Settingsview, click the USERStab.
4. Click +INVITE USER.
5. (Organization owner only) Select an organization role.
6. Select one or more projects.
7. Select a project role.
8. Enter the user's email address.
9. Click +INVITE USER.

When you invite another user to join your organization or project, Pinecone sends them an email containing a link that enables them to gain access to the organization or project. If they already have a Pinecone account, they still receive an email, but they can also immediately view the project.

Change project pod limit

Suggest Edits

Overview

If you are a project owner, follow these steps to change the maximum total number of podsin your project.

Change project pod limit in console

1. Access the Pinecone Console.
2. Click Settingsin the left menu.
3. In the Settingsview, click the PROJECTStab.
4. Next to the project you want to update, click
   .
5. Under Pod Limit, enter the new number of pods.
6. Click SAVE CHANGES.

Rename a project

Suggest Edits

Overview

If you are a project owner, follow these steps to change the name of your project.

1. Access the Pinecone Console.
2. Click Settingsin the left menu.
3. In the Settingsview, click the PROJECTStab.
4. Next to the project you want to update, click
   .
5. Under Project Name, enter the new project name.
6. Click SAVE CHANGES.

Guides - Moving to production

Introduction

The goal of this document is to prepare users to begin using their Pinecone indexes in production by anticipating production issues and identifying best practices for production indexes. Because these issues are highly workload-specific, the recommendations here are general.

Overview

Once you have become familiar with Pinecone and experimented with creating indexes and queries that reflect your intended workload, you may be planning to use your indexes to serve production queries. Before you do, there are several steps you can take that can prepare your project for production workloads, anticipate production issues, and enable reliability and growth.

Consider the following areas before moving your indexes to production:

Prepare your project structure

One of the first steps towards a production-ready Pinecone index is configuring your project correctly. Consider creating a separate projectfor your development and production indexes, to allow for testing changes to your index before deploying them to production. Ensure that you have properly configured user accessto your production environment so that only those users who need to access the production index can do so. Consider how best to manage the API key associated with your production project.

Test your query results

Before you move your index to production, make sure that your index is returning accurate results in the context of your application. Consider identifying the appropriate metricsfor evaluating your results.

Estimate the appropriate number and size of pods and replicas

Depending on your data and the types of workloads you intend to run, your project may require a different number and size of podsand replicas. Factors to consider include the number of vectors, the dimensions per vector, the amount and cardinality of metadata, and the acceptable queries per second (QPS). Use the index fullness metricto identify how much of your current resources your indexes are using. You can use collections to create indexeswith different pod types and sizes to experiment.

Load test your indexes

Before moving your project to production, consider determining whether your index configuration can serve the load of queries you anticipate from your application. You can write load tests in Python from scratch or using a load testing framework like Locust.

Back up your indexes

In order to enable long-term retention, compliance archiving, and deployment of new indexes, consider backing up your production indexes by creating collections.

Tune for performance

Before serving production workloads, identify ways to improve latencyby making changes to your deployment, project configuration, or client.

Configure monitoring

Prepare to observe production performance and availability by configuring monitoringwith Prometheus or OpenMetrics on your production indexes.

Plan for scaling

Before going to production, consider planning ahead for how you might scale your indexes when the need arises. Identify metrics that may indicate the need to scale, such as index fullnessand average request latency. Plan for increasing the number of pods, changing to a more performant pod type, vertically scalingthe size of your pods, increasing the number of replicas, or increasing storage capacity with a storage-optimized pod type.

Know how to get support

If you need help, visit support.pinecone.io, or talk to the Pinecone community. Ensure that your plan tiermatches the support and availability SLAs you need. This may require you to upgrade to Enterprise.

Concepts - Multitenancy

This document describes concepts related to multitenancy in Pinecone indexes. This includes information on different approaches to keeping sets of vectors separate within a Pinecone index. To learn how to create or modify an index, see Manage indexes.

You may need to segment vectors by customer or otherwise either physically or logically. This document describes different techniques to accomplish this and the advantages and advantages of each approach.

Namespaces

One approach to multitenancy is to use namespacesto isolate segments of vectors within a single index. This is a 'pool' model that shares most resources between tenants while keeping them logically separate.

Advantages

- Namespaces reduce the need for additional indexes, reducing maintenance effort.

Disadvantages

- Customer data is not isolated in its own pod. This means tenants share compute and storage resources.
- Deleting tenant data is more complicated and takes more time.

Metadata filtering

This approach to multitenancy stores all segments of vectors in a single index and filters by metadataat query time. This is another 'pool' model; here, you separate tenants on the query level.

Advantages

- Metadata filtering allows you to query across multiple tenants.
- Metadata filtering reduces the need for additional indexes, reducing maintenance effort.

Disadvantages

- Customer data is not isolated in its own pod. This means tenants share compute and storage resources.

- There is no way to track tenant-specific costs.
- You cannot provision tenants with different dimensions or pod type characteristics, which are set at the index level.

Separate indexes

Another approach to multitenancy is to create a separate index for each segment. This is a 'silo' model that provides dedicated resources to each tenant. For example, if you need to separate vectors for each customer, you can create a separate index for each customer.

Advantages

- This model separates tenants into different pods, preventing queries across tenants and allocating compute and storage resources to each tenant.

Disadvantages

- Creating and maintaining multiple indexes requires additional effort and cost.
- Because this model doesn't share resources between tenants, you must assign each tenant at minimum one pod. This is extremely inefficient if you have tenants with small number of vectors.
- Creating a new index takes more time than creating a namespace.

Guides - Monitoring

This document describes how to configure monitoring for your Pinecone index using Prometheus or compatible tools.

Overview

You can ingest performance metrics from Pinecone indexes into your own Prometheus instances, or into Prometheus- and OpenMetrics-compatible monitoring tools. The Prometheus metric endpoint is for users who want to monitor and store system health metrics using their own Prometheus metrics logger.

⚠️Warning

This feature is in public preview and is only available to Enterprise or Enterprise Dedicated users.

Connect

Metrics are available at a URL like the following:

https://metrics.YOUR_ENVIRONMENT.pinecone.io/metrics

Your API key must be passed via the Authorization header as a bearer token like the following:

Authorization: Bearer \<api-key\>

Only the metrics for the project associated with the API key are available at this URL.

For Prometheus, configure prometheus.yml as follows:

YAML
scrape_configs:
- job_name: pinecone-job-1
authorization:
credentials: <api-key-here>
scheme: https
static_configs:
- targets: ['metrics.YOUR_ENVIRONMENT.pinecone.io']
See Prometheus docsfor more configuration details.

Available Metrics

The metrics available are as follows:

Name	Type	Description	Labels
pinecone_vector_count	gauge	pinecone_vector_count gives the number of items per pod in the index. *Labels:* *- pid: Process identifier* *- index_name: Name of the index* *- project_name: Pinecone project name*	

pinecone_request_count_total	counter	pinecone_request_count_total gives the number of data plane calls made by clients. *Labels:* *- pid: Process identifier* *- index_name: Name of the index* *- project_name: Pinecone project name* *- request_type: One of upsert, delete, fetch, query, describe_index_stats*
pinecone_request_error_count_total	counter	pinecone_request_error_count_total gives the number of data plane calls made by clients that resulted in errors. *Labels:* *- pid: Process identifier* *- index_name: Name of the index* *- project_name: Pinecone project name* *- request_type: One of upsert, delete, fetch, query, describe_index_stats*
pinecone_request_latency_seconds	histogram	pinecone_request_latency_seconds gives the distribution of server-side processing latency for pinecone data plane calls. *Labels:* *- pid: Process identifier* *- index_name: Name of the index* *- project_name: Pinecone project name* *- request_type: One of upsert, delete, fetch, query, describe_index_stats*
pinecone_index_fullness	gauge	pinecone_index_fullness gives the fullness of the index on a scale of 0 to 1. *Labels:* *- pid: Process identifier* *- index_name: Name of the index* *- project_name: Pinecone project name*

Example queries

The following Prometheus queries gather information about your Pinecone index.

Average request latency

The following query returns the average latency in seconds for all requests against the Pinecone index example-index.

avg by (request_type) (pinecone_request_latency_seconds{index_name="example-index"})

The following query returns the vector count for the Pinecone index example-index.

sum ((avg by (app) (pinecone_vector_count{index_name="example-index"})))

The following query returns the total number of requests against the Pinecone index example-indexover one minute.

sum by
(request_type)(increase(pinecone_request_count_total{index_name="example-index"}[60s]))

The following query returns the total number of upsert requests against the Pinecone index example-indexover one minute.

sum by (request_type)(increase(pinecone_request_count_total{index_name="example-index", request_type="upsert"}[60s]))

The following query returns the total errors returned by the Pinecone index example-indexover one minute.

sum by (request_type) (increase(pinecone_request_error_count{
index_name="example-index"}[60s]))

The following query returns the index fullness metric for the Pinecone index example-index.

round(max (pinecone_index_fullness{index_name="example-index"} * 100))

Guides - Troubleshooting

This section describes common issues and how to solve them. Need help? Ask your question in our support forum.Standard, Enterprise, and Dedicated customers can also contact supportfor help.

Unable to pip install

Version 3 of Python uses pip3. Use the following commands at the command line (the terminal):

pip3 install -U pinecone-client

Index is missing after inactivity

In general, indexes on the Starter (free) plan are archived as collections and deleted after 7 days of inactivity; for indexes created by certain open source projects such as AutoGPT, indexes are archived and deleted after 1 day of inactivity. To prevent this, you can send any API request to Pinecone and the counter will reset.

Slow uploads or high latencies

To minimize latency when accessing Pinecone:

- Switch to a cloud environment. For example: EC2, GCE, Google Colab, GCP AI Platform Notebook, or SageMaker Notebook. If you experience slow uploads or high query latencies, it might be because you are accessing Pinecone from your home network.
- Consider deploying your application in the same environment as your Pinecone service. For users on the Starter (free) plan, the environment is GCP US-West (Oregon).
- See performance tuningfor more tips.

High query latencies with batching

If you're batching queries, try reducing the number of queries per call to 1 query vector. You can make these calls in parallel and expect roughly the same performance as with batching.

Upsert throttling when using the gRPC client

It's possible to get write-throttled sooner when upserting using the gRPC index. If you see this often, then we recommend using a backoff algorithm while upserting.

Pods are full

There is a limit to how much vector data a single pod can hold. Create an index with more pods to hold more data. Estimate the right index configurationand scale your indexto increase capacity.

If your metadata has high cardinality, such as having a unique value for every vector in a large index, the index will take up more memory than estimated. This could result in the pods being full sooner than you expected. Consider only indexing metadata to be used for filtering, and storing the rest in a separate key-value store.

See the Manage Indexes documentationfor information on how to specify the number of pods for your index.

Security concerns

We work hard to earn and maintain trust by treating security and reliability as a cornerstone of our company and product. Pinecone is SOC 2 Type II compliant and GDPR-ready. See the Trust & Security pagefor more information. Contact usto report any security concerns.

CORS errors

When sending requests to Pinecone, you may receive the following error:

console
No 'Access-Control-Allow-Origin' header is present on the requested resource.
This error occurs in response to cross-origin requests. Most commonly, it occurs when a user is running a local web server with the hostname 'localhost', which Pinecone's Same Origin Policy (SOP) treats as distinct from the IP address of the local machine.

To resolve this issue, host your web server on an external server with a public IP address and DNS name entry.

Guides - Performance tuning

This section provides some tips for getting the best performance out of Pinecone.

Basic performance checklist

- Switch to a cloud environment.For example: EC2, GCE, Google Colab, GCP AI Platform Notebook, or SageMaker Notebook. If you experience slow uploads or high query latencies, it might be because you are accessing Pinecone from your home network.
- Deploy your application and your Pinecone service in the same region.For users on the Free plan, Pinecone runs in GCP US-West (Oregon). Contact usif you need a dedicated deployment.
- Reuse connections.We recommend you reuse the same pinecone.Index()instance when you are upserting and querying the same index.
- Operate within known limits.

How to increase throughput

To increase throughput (QPS), increase the number of replicas for your index.

Example

The following example increases the number of replicas for example-indexto 4.

PythonJavaScriptcurl
pinecone.configure_index("example-index", replicas=4)
See the configure_index API referencefor more details.

Using the gRPC client to get higher upsert speeds

Pinecone has a gRPC flavor of the standard client (installation) that can provide higher upsert speeds for multi-pod indexes.

To connect to an index via the gRPC client:

Python

index = pinecone.GRPCIndex("index-name")

The syntax for upsert, query, fetch, and delete with the gRPC client remain the same as the standard client.

We recommend you use parallel upserts to get the best performance.

Python

index = pinecone.GRPCIndex('example-index')
def chunker(seq, batch_size):
return (seq[pos:pos + batch_size] for pos in range(0, len(seq), batch_size))
async_results = [
index.upsert(vectors=chunk, async_req=True)
for chunk in chunker(data, batch_size=100)
]
Wait for and retrieve responses (in case of error)
[async_result.result() for async_result in async_results]

We recommend you use the gRPC client for multi-pod indexes only. The performance of the standard and gRPC clients are similar in a single-pod index.

It's possible to get write throttled faster when upserting using the gRPC index. If you see this often, we recommend you use a backoff algorithm while upserting.

Pinecone is thread-safe, so you can launch multiple read requests and multiple write requests in parallel. Launching multiple requests can help with improving your throughput. However, reads and writes can't be performed in parallel, therefore writing in large batches might affect query latency and vice versa.

Guides - Using namespaces

Pinecone allows you to partition the vectors in an index into namespaces. Queries and other operations are then limited to one namespace, so different requests can search different subsets of your index.

For example, you might want to define a namespace for indexing articles by content, and another for indexing articles by title. For a complete example, see: Semantic Text Search (Example).

Every index is made up of one or more namespaces. Every vector exists in exactly one namespace.

Namespaces are uniquely identified by a namespace name, which almost all operations accept as a parameter to limit their work to the specified namespace. When you don't specify a namespace name for an operation, Pinecone uses the default namespace name of ""(the empty string).

Creating a namespace

A destination namespace can be specified when vectors are upserted. If the namespace doesn't exist, it is created implicitly.

The example below will create a "my-first-namespace" namespace if it doesn't already exist:

PythonJavaScriptcurl
```
# Upsert vectors while creating a new namespace
index.upsert(vectors=[('id-1', [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1])],
namespace='my-first-namespace')
```
Then you can submit queries and other operations specifying that namespace as a parameter. For example, to query the vectors in namespace "my-first-namespace":

PythonJavaScriptcurl

```
index.query(vector=[0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1],
top_k=1,
namespace='my-first-namespace')
```

Creating more than one namespace

You can create more than one namespace. For example, insert data into separate namespaces:

Pythoncurl

```
import numpy as np

# Create three sets of 8-dimensional vectors
vectors_a = np.random.rand(15, 8).tolist()
vectors_b = np.random.rand(20, 8).tolist()
vectors_c = np.random.rand(30, 8).tolist()

# Create ids
ids_a = map(str, np.arange(15).tolist())
ids_b = map(str, np.arange(20).tolist())
ids_c = map(str, np.arange(30).tolist())

# Insert into separate namespaces
index.upsert(vectors=zip(ids_a,vectors_a),namespace='namespace_a')
index.upsert(vectors=zip(ids_b,vectors_b),namespace='namespace_b')

# if no namespaces are specified, the index uses the default namespace
index.upsert(vectors=zip(ids_c,vectors_c))

# At this point, index.describe_index_stats() returns:
# {'dimension': 8,
# 'namespaces': {'': {'vector_count': 30},
# 'namespace_a': {'vector_count': 15},
# 'namespace_b': {'vector_count': 20}}}
```

Operations across all namespaces

All vector operations apply to a single namespace, with one exception:

The DescribeIndexStatisticsoperation returns per-namespace statistics about the contents of allnamespaces in an index. More details

---

Workflow - Scale indexes

n this topic, we explain how you can scale your indexes horizontally and vertically.

Vertical vs. horizontal scaling

If you need to scale your environment to accommodate more vectors, you can modify your existing index to scale it vertically or create a new index and scale horizontally. This article will describe both methods and how to scale your index effectively.

Vertical scaling

Scaling vertically is fast and involves no downtime. This is a good choice when you can't pause upserts and must continue serving traffic. It also allows you to double your capacity instantly. However, there are some factors to consider.

By changing the pod size, you can scale to x2, x4, and x8 pod sizes, which means you are doubling your capacity at each step. Moving up to a new capacity will effectively double the number of pods used at each step. If you need to scale by smaller increments, then consider horizontal scaling.

The number of base pods you specify when you initially create the index is static and cannot be changed. For example, if you start with 10 pods of p1.x1and vertically scale to p1.x2, this equates to 20 pods worth of usage. Neither can you change pod types with vertical scaling. If you want to change your pod type while scaling, then horizontal scaling is the better option.

You can only scale index sizes up and cannot scale them back down.

See our learning center for more information on vertical scaling.

Horizontal scaling

There are two approaches to horizontal scaling in Pinecone: adding pods and adding replicas. Adding pods increases all resources but requires a pause in upserts; adding replicas only increases throughput and requires no pause in upserts.

Adding pods

Adding pods to an index increases all resources, including available capacity. Adding pods to an existing index is possible using our collectionsfeature. A collection is an immutable snapshot of your index in time: a collection stores the data but not the original index definition.

When you create an index from a collection, you define the new index configuration. This allows you to scale the base pod count horizontally without scaling vertically. The main advantage of this approach is that you can scale incrementally instead of doubling capacity as with vertical scaling. Also, you can redefine pod types if you are experimenting or if you need to use a different pod type, such asperformance-optimized pods or storage-optimized pods. Another advantage of this method is that you can change your metadata configurationto redefine metadata fields as indexed or stored-only. This is important when tuning your indexfor the best throughput.

Here are the general steps to make a copy of your index and create a new index while changing the pod type, pod count, metadata configuration, replicas, and all typical parameters when creating a new collection:

1. Pause upserts.
2. Create a collection from the current index.
3. Create an index from the collection with new parameters.
4. Continue upserts to the newly created index. Note: the URL has likely changed.
5. Delete the old index if desired.

Adding replicas

Each replica duplicates the resources and data in an index. This means that adding additional replicas increases the throughput of the index but not its capacity. However, adding replicas does not require downtime.

Throughput in terms of queries per second (QPS) scales linearly with the number of replicas per index.

To add replicas, use the configure_indexoperation to increase the number of replicas for your index.

---

sparse-dense vectors allow keyword-aware semantic search combining with keyword search results can improve relevance

https://arxiv.org/pdf/2210.11934.pdf?

An Analysis of Fusion Functions for Hybrid Retrieval SEBASTIAN BRUCH, Pinecone, USA SIYU GAI∗ , University of California, Berkeley, USA AMIR INGBER, Pinecone, Israel We study hybrid search in text retrieval where lexical and semantic search are fused together with the intuition that the two are complementary in how they model relevance. In particular, we examine fusion by a convex combination (CC) of lexical and semantic scores, as well as the Reciprocal Rank Fusion (RRF) method, and identify their advantages and potential pitfalls. Contrary to existing studies, we find RRF to be sensitive to its parameters; that the learning of a CC fusion is generally agnostic to the choice of score normalization; that CC outperforms RRF in in-domain and out-of-domain settings; and finally, that CC is sample efficient, requiring only a small set of training examples to tune its only parameter to a target domain. CCS Concepts: • Information systems → Retrieval models and ranking; Combination, fusion and federated search. Additional Key Words and Phrases: Hybrid Retrieval, Lexical and Semantic Search, Fusion Functions 1 INTRODUCTION Retrieval is the first stage in a multi-stage ranking system [1, 2, 45], where the objective is to find the top-k set of documents, that are the most relevant to a given query q, from a large collection of documents D. Implicit in this task are two major research questions: a) how do we measure the relevance between a query q and a document d ∈ D?; and, b) how do we find the top-k documents according to a given similarity metric efficiently? In this work, we are primarily concerned with the former question in the context of text retrieval. As a fundamental problem in Information Retrieval (IR), the question of the similarity between queries and documents has been explored extensively. Early methods model text as a Bag of Words (BoW) and compute the similarity of two pieces of text using a statistical measure such as the term frequency-inverse document frequency (TF-IDF) family, with BM25 [33, 34] being its most prominent member. We refer to retrieval with a BoW model as lexical search and the similarity scores computed by such a system as lexical scores. Lexical search is simple, efficient, (naturally) “zero-shot,” and generally effective, but has important limitations: it is susceptible to the vocabulary mismatch problem and, moreover, does not take into account the semantic similarity of queries and documents [5]. That, it turns out, is what deep learning models are excellent at. With the rise of pre-trained language models such as BERT [8], it is now standard practice to learn a vector representation of queries and documents that does capture their semantics, and thereby, reduce top-k retrieval to the problem of finding k nearest neighbors in the resulting vector space [9, 13, 16, 32, 40, 44]—where closeness is measured using vector similarity or distance. We refer to this method as semantic search and the similarity scores computed by such a system as semantic scores. Hypothesizing that lexical and semantic search are complementary in how they model relevance, recent works [5, 13, 14, 19, 20, 42] began exploring methods to fuse together lexical and semantic retrieval: for a query q and ranked lists of documents RLex and RSem retrieved separately by lexical and semantic search systems respectively, the task is to construct a final ranked list RFusion so as to improve retrieval quality. This is often referred to as hybrid search. ∗Contributed to this work during a research internship at Pinecone. Authors’ addresses: Sebastian Bruch, sbruch@acm.org, Pinecone, New York, NY, USA; Siyu Gai, catherine_gai@berkeley.edu, University of California, Berkeley, Berkeley, CA, USA; Amir Ingber, Pinecone, Tel Aviv, Israel, ingber@pinecone.io. arXiv:2210.11934v2 [cs.IR] 4 May 2023

becoming increasingly clear that hybrid search does indeed lead to meaningful gains in retrieval quality, especially when applied to out-of-domain datasets [5, 40]—settings in which the semantic retrieval component uses a model that was not trained or fine-tuned on the target dataset. What is less clear and is worthy of further investigation, however, is how this fusion is done. One intuitive and common approach is to linearly combine lexical and semantic scores [13, 20, 40]. If fLex (q, d) and fSem(q, d) represent the lexical and semantic scores of document d with respect to query q, then a linear (or more accurately, convex) combination is expressed as fConvex = α fSem + (1 − α)fLex where 0 ≤ α ≤ 1. Because lexical scores (such as BM25) and semantic scores (such as dot product) may be unbounded, often they are normalized with min-max scaling [16, 40] prior to fusion. A recent study [5] argues that convex combination is sensitive to its parameter α and the choice of score normalization.1 They claim and show empirically, instead, that Reciprocal Rank Fusion (RRF) [6] may be a more suitable fusion as it is non-parametric and may be utilized in a zero-shot manner. They demonstrate its impressive performance even in zero-shot settings on a number of benchmark datasets. This work was inspired by the claims made in [5]; whereas [5] addresses how various hybrid methods perform relative to one another in an empirical study, we re-examine their findings and analyze why these methods work and what contributes to their relative performance. Our contributions thus can best be summarized as an in-depth examination of fusion functions and their behavior. As our first research question (RQ1), we investigate whether the convex combination fusion is a reasonable choice and study its sensitivity to the normalization protocol. We show that, while normalization is essential to create a bounded function and thereby bestow consistency to the fusion across domains, the specific choice of normalization is a rather small detail: there always exist convex combinations of scores normalized by min-max, standard score, or any other linear transformation that are rank-equivalent. In fact, when formulated as a per-query learning problem, the solution found for a dataset that is normalized with one scheme can be transformed to a solution for a different choice. We next investigate the properties of RRF. We first unpack RRF and examine its sensitivity to its parameters as our second research question (RQ2)—contrary to [5], we adopt a parametric view of RRF where we have as many parameters as there are retrieval functions to fuse, a quantity that is always one more than that in a convex combination. We find that, in contrast to a convex combination, a tuned RRF generalizes poorly to out-of-domain datasets. We then intuit that, because RRF is a function of ranks, it disregards the distribution of scores and, as such, discards useful information. Observe that the distance between raw scores plays no role in determining their hybrid score—a behavior we find counter-intuitive in a metric space where distance does matter. Examining this property constitutes our third and final research question (RQ3). Finally, we empirically demonstrate an unsurprising yet important fact: tuning α in a convex combination fusion function is extremely sample-efficient, requiring just a handful of labeled queries to arrive at a value suitable for a target domain, regardless of the magnitude of shift in the data distribution. RRF, on the other hand, is relatively less sample-efficient and converges to a relatively less effective retrieval system. We believe our findings, both theoretical and empirical, are important and pertinent to the research in this field. Our analysis leads us to believe that the convex combination formulation is theoretically sound, empirically effective, sample-efficient, and robust to domain shift. Moreover, 1 c.f. Section 3.1 in [5]: “This fusion method is sensitive to the score scales . . .which needs careful score normalization” (emphasis ours). An Analysis of Fusion Functions for

Hybrid Retrieval 3 unlike the parameters in RRF, the parameter(s) of a convex function are highly interpretable and, if no training samples are available, can be adjusted to incorporate domain knowledge. We organized the remainder of this article as follows. In Section 2, we review the relevant literature on hybrid search. Section 3 then introduces our adopted notation and provides details of our empirical setup, thereby providing context for the theoretical and empirical analysis of fusion functions. In Section 4, we begin our analysis by a detailed look at the convex combination of retrieval scores. We then examine RRF in Section 5. In Section 6, we summarize our observations and identify the properties a fusion function should have to behave well in hybrid retrieval. We then conclude this work and state future research directions in Section 7. 2 RELATED WORK A multi-stage ranking system is typically comprised of a retrieval stage and several subsequent reranking stages, where the retrieved candidates are ordered using more complex ranking functions [2, 39]. Conventional wisdom has that retrieval must be recall-oriented while improving ranking quality may be left to the re-ranking stages, which are typically Learning to Rank (LtR) models [17, 24, 29, 39, 41]. There is indeed much research on the trade-offs between recall and precision in such multi-stage cascades [7, 21], but a recent study [46] challenges that established convention and presents theoretical analysis that suggests retrieval must instead optimize precision. We therefore report both recall and NDCG [11], but focus on NDCG where space constraints prevent us from presenting both or when similar conclusions can be reached regardless of the metric used. One choice for retrieval that remains popular to date is BM25 [33, 34]. This additive statistic computes a weighted lexical match between query and document terms: it computes, for each query term, the product of its "importance" (i.e., frequency of a term in a document, normalized by document and global statistics such as average length) and its propensity—a quantity that is inversely proportionate to the fraction of documents that contain the term—and adds the scores of query terms to arrive at the final similarity or relevance score. Because BM25, like other lexical scoring functions, insists on an exact match of terms, even a slight typo can throw the function off. This vocabulary mismatch problem has been subject to much research in the past, with remedies ranging from pseudo-relevance feedback to document and query expansion techniques [15, 30, 36]. Trying to address the limitations of lexical search can only go so far, however. After all, they additionally do not capture the semantic similarity between queries and documents, which may be an important signal indicative of relevance. It has been shown that both of these issues can be remedied by Transformer-based [38] pre-trained language models such as BERT [8]. Applied to the ranking task, such models [25, 27–29] have advanced the state-of-the-art dramatically on benchmark datasets [26]. The computationally intensive inference of these deep models often renders them too inefficient for first-stage retrieval, however, making them more suitable for re-ranking stages. But by cleverly disentangling the query and document transformations into the so-called dual-encoder architecture, where, in the resulting design, the "embedding" of a document can be computed independently of queries, we can pre-compute document vectors and store them offline. In this way, we substantially reduce the computational cost during inference as it is only necessary to obtain the vector representation of the query during inference. At a high level, these models project queries and documents onto a low-dimensional vector space where semantically-similar points stay closer to each other. By doing so we transform the retrieval problem to one of similarity search or Approximate Nearest Neighbor (ANN) search—the k nearest neighbors to a query vector are the desired top-k documents. This

ANN problem can be solved efficiently using a number of algorithms such as FAISS [12] or Hierarchical Navigable Small World Graphs [22] available as open source 4 Sebastian Bruch, Siyu Gai, and Amir Ingber packages or through a managed service such as Pinecone2 , creating an opportunity to use deep models and vector representations for first-stage retrieval [13, 44]—a setup that we refer to as semantic search. Semantic search, however, has its own limitations. Previous studies [5, 37] have shown, for example, that when applied to out-of-domain datasets, their performance is often worse than BM25. Observing that lexical and semantic retrievers can be complementary in the way they model relevance [5], it is only natural to consider a hybrid approach where lexical and semantic similarities both contribute to the makeup of final retrieved list. To date there have been many studies [13, 14, 18–20, 40, 42, 47] that do just that, where most focus on in-domain tasks with one exception [5] that considers a zero-shot application too. Most of these works only use one of the many existing fusion functions in experiments, but none compares the main ideas comprehensively. We review the popular fusion functions from these works in the subsequent sections and, through a comparative study, elaborate what about their behavior may or may not be problematic. 3 SETUP In the sections that follow, we study fusion functions with a mix of theoretical and empirical analysis. For that reason, we present our notation as well as empirical setup and evaluation measures here to provide sufficient context for our arguments. 3.1 Notation We adopt the following notation in this work. We use fo(q, d) : Q × D → R to denote the score of document d ∈ D to query q ∈ Q according to the retrieval system o ∈ O. If o is a semantic retriever, Sem, then Q and D are the space of (dense) vectors in R d and fSem is typically cosine similarity or inner product. Similarly, when o is a lexical retriever, Lex, Q and D are high-dimensional sparse vectors in R $|V|$, with $|V|$ denoting the size of the vocabulary, and fLex is typically BM25. A retrieval system o is the space Q × D equipped with a metric fo(·, ·)—which need not be a proper metric. We denote the set of top-k documents retrieved for query q by retrieval system o by $R\ k$ o (q). We write πo(q, d) to denote the rank of document d with respect to query q according to retrieval system o. Note that, πo(q, di) can be expressed as the sum of indicator functions: πo(q, di) = 1 + $\sum dj \in R\ k$ o (q) 1fo (q,dj)>fo (q,di) , (1) where 1c is 1 when the predicate c holds and 0 otherwise. In words, and ignoring the subtleties introduced by the presence of score ties, the rank of document d is the count of documents whose score is larger than the score of d. Hybrid retrieval operates on the product space of Î oi with metric fFusion : Î foi → R. Without loss of generality, in this work, we restrict Î oi to be Lex × Sem. That is, we only consider the problem of fusing two retrieval scores, but note that much of the anlysis can be trivially extended to the fusion of multiple retrieval systems. We refer to this hybrid metric as a fusion function. A fusion function fFusion is typically applied to documents in the union of retrieved sets Uk (q) = Ð o $R\ k$ o (q), which we simply call the union set. When a document d in the union set is not present in one of the top-k sets (i.e., d ∈ Uk (q) but $d \notin R\ k$ oi (q) for some oi), we compute its missing score (i.e., foi (q, d)) prior to fusion. An Analysis of Fusion Functions for Hybrid Retrieval 5 Table 1. Datasets used in this work for evaluation purposes along with select statistics. Dataset Document Count Query Count MS MARCO Passage v1 8.8M 6,980 Natural Questions (NQ) 2.68M 3,452 Quora 523K 10,000 NFCorpus 3.6K 323 HotpotQA 5.23M 7,405 Fever 5.42M 6,666 SciFact 5K 300 DBPedia 4.63M 400 FiQA 57K 648 3.2 Empirical Setup Datasets: We evaluate our methods on a variety of publicly available benchmark datasets, summarized in Table 1 both in in-domain and out-of-domain, zero-shot settings. One

of the datasets is the MS MARCO3 Passage Retrieval v1 dataset [26], a publicly available retrieval and ranking collection from Microsoft. It consists of roughly 8.8 million short passages which, along with queries in natural language, originate from Bing. The queries are split into train, dev, and eval non-overlapping subsets. We use the train queries for any learning or tuning and evaluate exclusively on the small dev query set (consisting of 6,980 queries) in our analysis. Included in the dataset also are relevance labels. We additionally experiment with 8 datasets from the BeIR collection [37] 4 : Natural Questions (NQ, question answering), Quora (duplicate detection), NFCorpus (medical), HotpotQA (question answering), Fever (fact extraction), SciFact (scientific claim verification), DBPedia (entity search), and FiQA (financial). For a more detailed description of each dataset, we refer the reader to [37]. Lexical search: We use PISA [23] for keyword-based lexical retrieval. We tokenize queries and documents by space and apply stemming available in PISA—we do not employ any other preprocessing steps such as stopword removal, lemmatization, or expansion. We use BM25 with the same hyperparameters as [5] (k1=0.9 and b=0.4) to retrieve the top 1,000 candidates. Semantic search: We use the all-MiniLM-L6-v2 model checkpoint available on HuggingFace5 to project queries and documents into 384-dimensional vectors, which can subsequently be used for indexing and top-k retrieval using cosine similarity. This model has been shown to achieve competitive quality on an array of benchmark datasets while remaining compact in size and efficient to infer6 , thereby allowing us to conduct extensive experiments with results that are competitive with existing state-of-the-art models. This model has been fine-tuned on a large number of datasets, exceeding a total of 1 billion pairs of text, including NQ, MS MARCO Passage, and Quora. As such, we consider all experiments on these three datasets as in-domain, and the rest as out-of-domain. We use the exact search for inner product algorithm (IndexFlatIP) from FAISS [12] to retrieve top 1,000 nearest neighbors. 2http://pinecone.io 3Available at https://microsoft.github.io/msmarco/ 4Available at https://github.com/beir-cellar/beir 5Available at https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2 6 c.f. https://sbert.net for details. 6 Sebastian Bruch, Siyu Gai, and Amir Ingber Supplementary models and fusions: Our primary set of experiments presented in the body of this manuscript focus exclusively on the fusion of BM25 with the all-MiniLM-L6-v2 model as representatives of lexical and semantic retrieval functions. We do, however, note that our analysis of fusion functions is not limited to lexical-semantic search per se: all normalization and fusion functions studied in this work can be applied to arbitrary scoring functions! As such, we conduct additional experiments using a variety of pairs of retrieval models to confirm the generality of the main theoretical findings of this work. We report these results in Appendix A through Appendix D. In Appendix A, we examine the fusion of the Splade model with BM25. Splade7 [9] is a deep learning model that produces sparse representations for a given piece of text, where each non-zero entry in the resulting embedding is the importance weight of a term in the BERT [8] WordPiece [43] vocabulary comprising of roughly 30, 000 terms. Appendix B studies the fusion of BM25 with the Tas-B [10] model.8 Tas-B is a bi-encoder model that was trained using supervision from a cross-encoder and a ColBERT model. In Appendix C we fuse Splade and Tas-B, and in Appendix D Tas-B and all-MiniLM-L6-v2. We note that, both Splade and Tas-B were fine-tuned on the MS MARCO dataset. As such, in all the supplementary experiments, results reported on the MS MARCO dataset should be considered “in-domain” while the remaining datasets represent out-of-domain distributions. Evaluation: Unless noted otherwise,

we form the union set for every query from the candidates retrieved by the lexical and semantic search systems. We then compute missing scores where required, compute fFusion on the union set, and re-order according to the hybrid scores. We then measure Recall@1000 and NDCG@1000 to quantify ranking quality, as recommended by Zamani et al. [46]. Due to the much smaller size of SciFact and NFCorpus, we evaluate Recall and NDCG at rank cutoff 100 instead, retrieving roughly 2% and 2.7% of the size of the dataset, respectively. We note that, this choice of cutoff does not affect the outcome of our experiments or change our conclusions, but it more clearly highlights the differences between the various methods; recall approaches 1 regardless of the retrieval method if rank cutoff was 1,000 (or 20% and 27% of the size of the datasets). Further note that, we choose to evaluate deep (i.e., with a larger rank cut-off) rather than shallow metrics per the discussion in [40] to understand the performance of each system more completely. 4 ANALYSIS OF CONVEX COMBINATION OF RETRIEVAL SCORES We are interested in understanding the behavior and properties of fusion functions. In the remainder of this work, we study through that lens two popular methods that are representative of existing ideas in the literature, beginning with a convex combination of scores. As noted earlier, most existing works use a convex combination of lexical and semantic scores as follows: fConvex (q, d) = α fSem(q, d) + (1 − α)fLex (q, d) for some 0 ≤ α ≤ 1. When α = 1 the above collapses to semantic scores and when it is 0, to lexical scores. An interesting property of this fusion is that it takes into account the distribution of scores. In other words, the distance between lexical (or semantic) scores of two documents plays a significant role in determining their final hybrid score. One disadvantage, however, is that the range of fSem can be very different from fLex. Moreover, as with TF-IDF in lexical search or with inner product in semantic search, the range of individual functions fo may depend on the norm of the query and document vectors (e.g., BM25 is a function of the number of query terms). As such any constant α is likely to yield inconsistently-scaled hybrid scores. 7Pre-trained checkpoint from HuggingFace available at https://huggingface.co/naver/splade-cocondenser-ensembledistil 8Available at https://huggingface.co/sentence-transformers/msmarco-distilbert-base-tas-b An Analysis of Fusion Functions for Hybrid Retrieval 7 The problem above is trivially addressed by applying score normalization prior to fusion [16, 40]. Suppose we have collected a union set Uk (q) for q, and that for every candidate we have computed both lexical and semantic scores. Now, consider the min-max scaling of scores ϕmm : R → [0, 1] below: ϕmm(fo(q, d)) = fo(q, d) − mq Mq − mq , (2) where mq = min$d$ ∈Uk (q) fo(q, d) and Mq = max$d$ ∈Uk (q) fo(q, d). We note that, min-max scaling is the de facto method in the literature, but other choices of ϕo(·) in the more general expression below: fConvex (q, d) = $\alpha\phi$Sem(fSem(q, d)) + (1 − α)ϕLex (fLex (q, d)), (3) are valid as well so long as ϕSem, ϕLex : R → R are monotone in their argument. For example, for reasons that will become clearer later, we can redefine the normalization by replacing the minimum of the set with the theoretical minimum of the function (i.e., the maximum value that is always less than or equal to all values attainable by the scoring function, or its infimum) to arrive at: ϕtmm(fo(q, d)) = fo(q, d) − inf fo(q, ·) Mq − inf fo(q, ·) . (4) As an example, when fLex is BM25, then its infimum is 0. When fSem is cosine similarity, then that quantity is −1. Another popular choice is the standard score (z-score) normalization which is defined as follows: ϕz (fo(q, d)) = fo(q, d) − μ σ , (5) where μ and σ denote the mean and standard deviation of the set of scores fo(q, ·) for query q. We will return to normalization shortly, but we make note of one small but important fact: in cases where the variance of lexical (semantic) scores in the union set is 0, we

may skip the fusion step altogether because retrieval quality will be unaffected by lexical (semantic) scores. The case where the variance is arbitrarily close to 0, however, creates challenges for certain normalization functions. While this would make for an interesting theoretical analysis, we do not study this particular setting in this work as, empirically, we do observe a reasonably large variance among scores in the union set on all datasets using state-of-the-art lexical and semantic retrieval functions. 4.1 Suitability of Convex Combination A convex combination of scores is a natural choice for creating a mixture of two retrieval systems, but is it a reasonable choice? It has been established in many past empirical studies that fConvex with min-max normalization often serves as a strong baseline. So the answer to our question appears to be positive. Nonetheless, we believe it is important to understand precisely why this fusion works. We investigate this question empirically, by visualizing lexical and semantic scores of querydocument pairs from an array of datasets. Because we operate in a two-dimensional space, observing the pattern of positive (where document is relevant to query) and negative samples in a plot can reveal a lot about whether and how they are separable and how the fusion function behaves. To that end, we sample up to 20,000 positive and up to the same number of negative query-document pairs from the validation split of each dataset, and illustrate the collected points in a scatter plot in Figure 1. From these figures, it is clear that positive and negative samples form clusters that are, with some error, separable by a linear function. What is different between datasets is the slope of this separating line. For example, in MS MARCO, Quora, and NQ, which are in-domain datasets, the separating line is almost vertical, suggesting that the semantic scores serve as a sufficiently strong 8 Sebastian Bruch, Siyu Gai, and Amir Ingber (a) MS MARCO (b) Quora (c) NQ (d) FiQA (e) HotpotQA (f) Fever Fig. 1. Visualization of the normalized lexical (ϕtmm(fLex)) and semantic (ϕtmm(fSem)) scores of querydocument pairs sampled from the validation split of each dataset. Shown in red are up to 20,000 positive samples where document is relevant to query, and in black up to the same number of negative samples. Adding a lexical (semantic) dimension to query-document pairs helps tease out the relevant documents that would be statistically indistinguishable in a one-dimensional semantic (lexical) view of the data—when samples are projected onto the x (y) axis. signal for relevance. This is somewhat true of FiQA. In other out-of-domain datasets, however, the line is rotated counter-clockwise, indicating a more balanced weighting of lexical and semantic scores. Said differently, adding a lexical (semantic) dimension to query-document pairs helps tease An Analysis of Fusion Functions for Hybrid Retrieval 9 (a) α = 0.6 (b) α = 0.8 Fig. 2. Effect of fConvex on pairs of lexical and semantic scores. out the relevant documents that would be statistically indistinguishable in a one-dimensional semantic (lexical) view of the data. Interestingly, across all datasets, there is a higher concentration of negative samples where lexical scores vanish. This empirical evidence suggests that lexical and semantic scores may indeed be complementary— an observation that is in agreement with prior work [5]—and a line may be a reasonable choice for distinguishing between positive and negative samples. But while these figures shed light on the shape of positive and negative clusters and their separability, our problem is not classification but ranking. We seek to order query-document pairs and, as such, separability is less critical and, in fact, not required. It is therefore instructive to understand the effect of a particular convex combination on pairs of lexical and semantic scores. This is visualized in Figure 2 for two values of α in fConvex. The plots in Figure 2 illustrate how the parameter α determines how different regions of the plane are ranked relative

to each other. This is a trivial fact, but it is nonetheless interesting to map these patterns to the distributions in Figure 1. In-domain datasets, for example, form a pattern of positives and negatives that is unsurprisingly more in tune with the α = 0.8 setting of fConvex than α = 0.6. 4.2 Role of Normalization We have thus far used min-max normalization to be consistent with the literature. In this section, we ask the question first raised by Chen et al. [5] on whether and to what extent the choice of normalization matters and how carefully one must choose the normalization protocol. In other words, we wish to examine the effect of ϕSem(·) and ϕLex (·) on the convex combination in Equation (3). Before we begin, let us consider the following suite of functions: • ϕmm: Min-max scaling of Equation (2); • ϕtmm: Theoretical min-max scaling of Equation (4); • ϕz: z-score normalization of Equation (5); • ϕmm−Lex: Min-max scaling of lexical scores, unnormalized semantic scores; • ϕtmm−Lex: Theoretical min-max normalized lexical scores, unnormalized semantic scores; • ϕz−Lex: z-score normalized lexical scores, unnormalized semantic scores; and, • I: The identity transformation, leaving both semantic and lexical scores unnormalized. 10 Sebastian Bruch, Siyu Gai, and Amir Ingber We believe these transformations together test the various conditions in our upcoming arguments. Let us first state the notion of rank-equivalence more formally: Definition 4.1. We say two functions f and g are rank-equivalent on the set U and write $f \pi = g$, if the order among documents in a set U induced by f is the same as that induced by g. For example, when ϕSem(x) = ax + b and ϕLex (x) = cx + d are linear transformations of scores for some positive coefficients a, b and real intercepts b, c, then they can be reduced to the following rank-equivalent form: fConvex (q, d) π = ($a\alpha$)fSem(q, d) + c(1 − α)fLex (q, d). In fact, letting α ′ = $c\alpha$/[$c\alpha$ + c(1 − α)] transforms the problem to one of learning a convex combination of the original scores with a modified weight. This family of functions includes ϕmm, ϕz, and ϕtmm, and as such solutions for one family can be transformed to solutions for another normalization protocol. More formally: Lemma 4.2. For every query, given an arbitrary α, there exists a α ′ such that the convex combination of min-max normalized scores with parameter α is rank-equivalent to a convex combination of z-score normalized scores with α ′ , and vice versa. Proof. Write mo and Mo for the minimum and maximum scores retrieved by system o, and μo and σo for their mean and standard deviation. We also write Ro = Mo − mo for brevity. For every document d, we have the following: α fSem(q, d) − mSem RSem + (1 − α) fLex (q, d) − mLex RLex π = α Rsem fSem(q, d) + 1 − α RLex fLex (q, d) π = 1 σSemσLex α RSem fSem(q, d) + 1 − α RLex fLex (q, d) − α RSem μSem − 1 − α RLex μLex π = α RSemσLex fSem(q, d) − μSem σSem + 1 − α RLexσSem fLex (q, d) − μLex σLex , where in every step we either added a constant or multiplied the expression by a positive constant, both rank-preserving operations. Finally, setting α ′ = α RSemσLex /(α RSemσLex + 1 − α RLexσSem) completes the proof. The other direction is similar. □ The fact above implies that the problem of tuning α for a query in a min-max normalization regime is equivalent to learning α ′ in a z-score normalized setting. In other words, there is a one-to-one relationship between these parameters, and as a result solutions can be mapped from one problem space to the other. However, this statement is only true for individual queries and does not have any implications for the learning of the weight in the convex combination over an entire collection of queries. Let us now consider this more complex setup. The question we wish to answer is as follows: under what conditions is fConvex with parameter α and a pair of normalization functions (ϕSem, ϕLex) rank-equivalent to an f ′ Convex of a new pair of normalization functions (ϕ ′ Sem, ϕ′ Lex) with weight α ′ ? That is, for a constant α with one normalization protocol, when is there a

constant α' that produces the same ranked lists for every query but with a different normalization protocol? The answer to this question helps us understand whether and when changing normalization schemes from min-max to z-score, for example, matters. We state the following definitions followed by a theorem that answers this questio An Analysis of Fusion Functions for Hybrid Retrieval 11 Definition 4.3. We say f : R → R is a δ-expansion with respect to g : R → R if for any x and y in the domains of f and g we have that $|f(y) - f(x)| \geq \delta |g(y) - g(x)|$ for some $\delta \geq 1$. For example, ϕmm(·) is an expansion with respect to ϕtmm(·) with a factor δ that depends on the range of the scores. As another example, ϕz (·) is an expansion with respect to ϕmm(·). Definition 4.4. For two pairs of functions f , g : R → R and f' , g' : R → R, and two points x and y in their domains, we say that f' expands with respect to f more rapidly than g' expands with respect to g, with a relative expansion rate of $\lambda \geq 1$, if the following condition holds: $|f'(y) - f'(x)| \; |f(y) - f(x)| = \lambda \; |g'(y) - g'(x)| \; |g(y) - g(x)|$. When λ is independent of the points x and y, we call this relative expansion uniform: $|\Delta f'|/|\Delta f| \; |\Delta g'|/|\Delta g| = \lambda, \; \forall x, y$. As an example, if f and g are min-max scaling and f' and g' are z-score normalization, then their respective rate of expansion is roughly similar. We will later show that this property often holds empirically across different transformations. Theorem 4.5. For every choice of α, there exists a constant α' such that the following functions are rank-equivalent on a collection of queries Q: fConvex = $\alpha\phi$ (fSem(q, d)) + (1 − α)ω(fLex (q, d)), and f' Convex = $\alpha' \phi'$ (fSem(q, d)) + (1 − α')ω' (fLex(q, d)), if for the monotone functions ϕ, ω, ϕ' , ω' : R → R, ϕ' expands with respect to ϕ more rapidly than ω' expands with respect to ω with a uniform rate λ. Proof. Consider a pair of documents di and dj in the ranked list of a query q such that di is ranked above dj according to fConvex. Shortening fo(q, dk) to f (k) o for brevity, we have that: f (i) Convex > f (j) Convex ⟹ α (ϕ (f (i) Sem) − ϕ (f (j) Sem)) | {z } $\Delta\phi ij$ + (ω(f (j) Lex) − ω(f (i) Lex)) | {z } $\Delta\omega ji$ > ω(f (j) Lex) − ω(f (i) Lex) This holds if and only if we have the following: (α > 1/(1 + $\Delta\phi ij$ $\Delta\omega ji$), if $\Delta\phi ij$ + $\Delta\omega ji$ > 0, α < 1/(1 + $\Delta\phi ij$ $\Delta\omega ji$), otherwise. (6) Observe that, because of the monotonicity of a convex combination and the monotonicity of the normalization functions, the case $\Delta\phi ij$ < 0 and $\Delta\omega ji$ > 0 (which implies that the semantic and lexical scores of dj are both larger than di) is not valid as it leads to a reversal of ranks. Similarly, the opposite case $\Delta\phi ij$ > 0 and $\Delta\omega ji$ < 0 always leads to the correct order regardless of the weight in the convex combination. We consider the other two cases separately below. Case 1: $\Delta\phi ij$ > 0 and $\Delta\omega ji$ > 0. Because of the monotonicity property, we can deduce that $\Delta\phi' ij$ > 0 and $\Delta\omega' ji$ > 0. From Equation (6), for the order between di and dj to be preserved under the image of f' Convex, we must therefore have the following: α' > 1/(1 + $\Delta\phi' ij$ $\Delta\omega' ji$). 12 Sebastian Bruch, Siyu Gai, and Amir Ingber By assumption, using Definition 4.4, we observe that: $\Delta\phi' ij$ $\Delta\phi ij$ ≥ $\Delta\omega' ji$ $\Delta\omega ji$ ⟹ $\Delta\phi' ij$ $\Delta\omega' ji$ ≥ $\Delta\phi ij$ $\Delta\omega ji$. As such, the lower-bound on α' imposed by documents di and dj of query q, $L' ij$ (q), is smaller than the lower-bound on α, Lij (q). Like α, this case does not additionally constrain α' from above (i.e., the upper-bound does not change: $U' ij$ (q) = Uij (q) = 1). Case 2: $\Delta\phi ij$ < 0, $\Delta\omega ji$ < 0. Once again, due to monotonicity, it is easy to see that $\Delta\phi' ij$ < 0 and $\Delta\omega' ji$ < 0. Equation (6) tells us that, for the order to be preserved under f' Convex, we must similarly have that: α' < 1/(1 + $\Delta\phi' ij$ $\Delta\omega' ji$). Once again, by assumption we have that the upper-bound on α' is a translation of the upper-bound on α to the left. The lower-bound is unaffected and remains 0. For f' Convex to induce the same order as fConvex among all pairs of documents for all queries in Q, the intersection of the intervals produced by the constraints on α' has to be non-empty: $I' \triangleq \bigcup_q$ [max$_{ij}$ $L' ij$ (q), min$_{ij}$ $U' ij$ (q)] = [max$_{q,ij}$ $L' ij$ (q), min$_{q,ij}$ $U' ij$ (q)] ≠ ∅. We next prove that I' is

always non-empty to conclude the proof of the theorem. By Equation (6) and the existence of α, we know that max$_{q,ij}$ L_{ij} (q) ≤ min$_{q,ij}$ U_{ij} (q). Suppose that documents d_i and d_j of query $q1$ maximize the lower-bound, and that documents d_m and d_n of query $q2$ minimize the upper-bound. We therefore have that: $1/(1 + \Delta\phi_{ij}\ \Delta\omega_{ji}) \leq 1/(1 + \Delta\phi_{mn}\ \Delta\omega_{nm}) \implies \Delta\phi_{ij}\ \Delta\omega_{ji} \geq \Delta\phi_{mn}\ \Delta\omega_{nm}$ Because of the uniformity of the relative expansion rate, we can deduce that: $\Delta\phi'_{ij}\ \Delta\omega'_{ji} \geq \Delta\phi'_{mn}\ \Delta\omega'_{nm} \implies \max_{q,ij} L'_{ij}(q) \leq \min_{q,ij} U'_{ij}(q)$. □ It is easy to show that the theorem above also holds when the condition is updated to reflect a shift of lower- and upper-bounds to the right, which happens when ϕ' contracts with respect to ϕ more rapidly than ω' does with respect to ω. The picture painted by Theorem 4.5 is that switching from min-max scaling to z-score normalization or any other linear transformation that is bounded and does not severely distort the distribution of scores, especially among the top-ranking documents, results in a rank-equivalent function. At most, for any given value of the ranking metric of interest such as NDCG, we should observe a shift of the weight in the convex combination to the right or left. Figure 3 illustrates this effect empirically on select datasets. As anticipated, the peak performance in terms of NDCG shifts to the left or right depending on the type of normalization. The uniformity requirement on the relative expansion rate, λ, in Theorem 4.5 is not as strict and restrictive as it may appear. First, it is only necessary for λ to be stable on the set of ordered pairs of documents as ranked by fConvex: $|\Delta\phi'_{ij}|/|\Delta\phi_{ij}\ |\ |\Delta\omega'_{ji}|/|\Delta\omega_{ji}| = \lambda,\ \forall (d_i, d_j)$ st $f\text{Convex}(d_i) > f\text{Convex}(d_j)$. An Analysis of Fusion Functions for Hybrid Retrieval 13 (a) MS MARCO (b) Quora (c) HotpotQA (d) FiQA Fig. 3. Effect of normalization on the performance of fConvex as a function of α on the validation set. Second, as we will observe experimentally, λ being concentrated around one value, rather than being the same constant everywhere as uniformity requires, is often sufficient for the effect to materialize in practice. We observe this phenomenon empirically by fixing the parameter α in fConvex with one transformation and forming ranked lists, then choosing another transformation and computing its relative expansion rate λ on all ordered pairs of documents. We show the measured relative expansion rate in Figure 4 for various transformations. Figure 4 shows that most pairs of transformations yield a stable relative expansion rate. For example, if fConvex uses ϕtmm and f' Convex uses ϕmm—denoted by $\phi\text{tmm} \rightarrow \phi\text{mm}$—for every choice of α, the relative expansion rate λ is concentrated around a constant value. This implies that any ranked list obtained from fConvex can be reconstructed by f' Convex. Interestingly, ϕz−Lex $\rightarrow$ ϕmm−Lex has a comparatively less stable λ, but removing normalization altogether (i.e., ϕmm−Lex $\rightarrow I$) dramatically distorts the expansion rates. This goes some way to explain why normalization and boundedness are important properties. This connection between boundedness and the effect that changing the normalization function has on ranking quality, is clearer in the experiments presented in Appendix A through Appendix D. 14 Sebastian Bruch, Siyu Gai, and Amir Ingber (a) MS MARCO (b) Quora (c) HotpotQA (d) FiQA Fig. 4. Relative expansion rate of semantic scores with respect to lexical scores, λ, when changing from one transformation to another, with 95% confidence intervals. Prior to visualization, we normalize values of λ to bring them into a similar scale—this only affects aesthetics and readability, but is the reason why the vertical axis is not scaled. For most transformations and every value of α, we observe a stable relative rate of expansion where λ concentrates around one value for the vast majority of queries. In general, when a function is unbounded, fusing with normalization versus without results in a relative expansion rate λ with a high variance, which leads to relatively different classes of rankings:

there exists ranked lists which are produced by a fusion with normalization that cannot be reconstructed by a fusion without normalization. For pairs of normalization functions whose relative expansion rate is stable and highly concentrated, on the other hand, the curves showing the effect of α on fConvex are translations of each other, as Theorem 4.5 predicts: every ranked list produced by a fusion using one normalization function can be reproduced by a fusion using another. In the last two sections, we have answered RQ1: convex combination is an appropriate fusion function and its performance is not sensitive to the choice of normalization so long as the transformation has reasonable properties. Interestingly, the behavior of ϕtmm appears to be more robust to An Analysis of Fusion Functions for Hybrid Retrieval 15 the data distribution—its peak remains within a small neighborhood as we move from one dataset to another. We believe the reason ϕtmm-normalized scores are more stable is because it has one fewer data-dependent statistic in the transformation (i.e., minimum score in the retrieved set is replaced with minimum feasible value regardless of the candidate set). In the remainder of this work, we use ϕtmm and denote a convex combination of scores normalized by it by TM2C2 for brevity. Where the theoretical minimum does not exist (e.g., with the Tas-B model), we use ϕmm instead and denote it by M2C2. 5 ANALYSIS OF RECIPROCAL RANK FUSION Chen et al. [5] show that RRF performs better and more reliably than a convex combination of normalized scores. RRF is computed as follows: fRRF $(q, d) = 1 \eta + \pi$Lex $(q, d) + 1 \eta + \pi$Sem(q, d) , (7) where η is a free parameter. The authors of [5] take a non-parametric view of RRF, where the parameter η is set to its default value 60, in order to apply the fusion to out-of-domain datasets in a zero-shot manner. In this work, we additionally take a parametric view of RRF, where as we elaborate later, the number of free parameters is the same as the number of functions being fused together, a quantity that is always larger than the number of parameters in a convex combination. Let us begin by comparing the performance of RRF and TM2C2 empirically to get a sense of their relative efficacy. We first verify whether hybrid retrieval leads to significant gains in in-domain and out-of-domain experiments. In a way, we seek to confirm the findings reported in [5] and compare the two fusion functions in the process. Table 2 summarizes our results for our primary models, with results for the remaining fusions reported in the appendices. We note that, we set RRF's η to 60 per [5] but tuned TM2C2's α on the validation set of the in-domain datasets and found that α = 0.8 works well for the three datasets. In the experiments leading to Table 2, we fix α = 0.8 and evaluate methods on the test split of the datasets. Per [5, 40], we have also included the performance of an oracle system that uses a per-query α, to establish an upper-bound—the oracle knows which value of α works best for any given query. Our results show that hybrid retrieval using RRF outperforms pure-lexical and pure-semantic retrieval on most datasets. This fusion method is particularly effective on out-of-domain datasets, rendering the observation of [5] a robust finding and asserting once more the remarkable performance of RRF in zeros-shot settings. Contrary to [5], however, we find that TM2C2 significantly outperforms RRF on all datasets in terms of NDCG, and does generally better in terms of Recall. Our observation is consistent with [40] that TM2C2 substantially boosts NDCG even on in-domain datasets. To contextualize the effect of α on ranking quality, we visualize a parameter sweep on the validation split of in-domain datasets in Figure 5(a), and for completeness, on the test split of out-of-domain datasets in Figure 5(b). These figures also compare the performance of TM2C2 with RRF by reporting the difference between NDCG of the two methods. These plots show that there always exists an interval of α for which fTM2C2 $>$ fRRF with $>$ indicating better

rank quality. 5.1 Effect of Parameters Chen et al. [5] rightly argue that because RRF is merely a function of ranks, rather than scores, it naturally addresses the scale and range problem without requiring normalization—which, as we showed, is not a consequential choice anyway. While that statement is accurate, we believe it introduces new problems that must be recognized too. 16 Sebastian Bruch, Siyu Gai, and Amir Ingber Table 2. Recall@1000 and NDCG@1000 (except SciFact and NFCorpus where cutoff is 100) on the test split of various datasets for lexical and semantic search as well as hybrid retrieval using RRF [5] (η = 60) and TM2C2 (α = 0.8). The symbols ‡ and ∗ indicate statistical significance (p-value < 0.01) with respect to TM2C2 and RRF respectively, according to a paired two-tailed t-test. Recall NDCG Dataset Lex. Sem. TM2C2 RRF Lex. Sem. TM2C2 RRF Oracle in-domain MS MARCO 0.836‡∗ 0.964‡∗ 0.974 0.969‡ 0.309‡∗ 0.441‡∗ 0.454 0.425‡ 0.547 NQ 0.886‡∗ 0.978‡∗ 0.985 0.984 0.382‡∗ 0.505‡ 0.542 0.514‡ 0.637 Quora 0.992‡∗ 0.999 0.999 0.999 0.800‡∗ 0.889‡∗ 0.901 0.877‡ 0.936 zero-shot NFCorpus 0.255‡∗ 0.320‡∗ 0.338 0.327 0.268‡∗ 0.296‡∗ 0.327 0.312‡ 0.371 HotpotQA 0.878‡∗ 0.756‡∗ 0.884 0.888 0.682‡∗ 0.520‡∗ 0.699 0.675‡ 0.767 FEVER 0.969‡∗ 0.931‡∗ 0.972 0.972 0.689‡∗ 0.558‡∗ 0.744 0.721‡ 0.814 SciFact 0.900‡∗ 0.932‡∗ 0.958 0.955 0.698‡∗ 0.681‡∗ 0.753 0.730‡ 0.796 DBPedia 0.540‡∗ 0.408‡∗ 0.564 0.567 0.415‡∗ 0.425‡∗ 0.512 0.489‡ 0.553 FiQA 0.720‡∗ 0.908 0.907 0.904 0.315‡∗ 0.467‡ 0.496 0.464‡ 0.561 (a) in-domain (b) out-of-domain Fig. 5. Difference in NDCG@1000 of TM2C2 and RRF (positive indicates better ranking quality by TM2C2) as a function of α. When α = 0 the model is rank-equivalent to lexical search while α = 1 is rank-equivalent to semantic search. The first, more minor issue is that ranks cannot be computed exactly unless the entire collection D is ranked by retrieval system o for every query. That is because, there may be documents that appear in the union set, but not in one of the individual top-k sets. Their true rank is therefore unknown, though is often approximated by ranking documents within the union set. We take this approach when computing ranks. The second issue is that, unlike TM2C2, RRF ignores the raw scores and discards information about their distribution. In this regime, whether or not a document has a low or high semantic score does not matter so long as its rank in $R\ k$ Sem stays the same. It is arguable in this case whether rank is An Analysis of Fusion Functions for Hybrid Retrieval 17 a stronger signal of relevance than score, a measurement in a metric space where distance matters greatly. We intuit that, such distortion of distances may result in a loss of valuable information that would lead to better final ranked lists. To understand these issues better, let us first repeat the exercise in Section 4.1 for RRF. In Figure 6, we have plotted the reciprocal rank (i.e., $rr(\pi$o) = 1/(η + πo) with η = 60) for sampled querydocument pairs as before. We choose η = 60 per the setup in [5], but note that changing this value leads to different distributions: as η approaches ∞, for example, all scores will collapse to a single point regardless of the original ranks (πo). From the figure, we can see that samples are pulled towards one of the poles at (0, 0) and (1/61, 1/61). The former attracts a higher concentration of negative samples while the latter positive samples. While this separation is somewhat consistent across datasets, the concentration around poles and axes changes. Indeed, on HotpotQA and Fever there is a higher concentration of positive documents near the top, whereas on FiQA and the in-domain datasets more positive samples end up along the vertical line at $rr(\pi$Sem) = 1/61, indicating that lexical ranks matter less. This suggests that a simple addition of reciprocal ranks does not behave consistently across domains. We argued earlier that RRF is parametric and that it, in fact, has as many parameters as there are retrieval

functions to fuse. To see this more clearly, let us rewrite Equation (7) as follows: fRRF (q, d) = 1 ηLex + πLex (q, d) + 1 ηSem + πSem(q, d) . (8) We study the effect of parameters on fRRF by comparing the NDCG obtained from RRF with a particular choice of ηLex and ηSem against a realization of RRF with ηLex = ηSem = 60. In this way, we are able to visualize the impact on performance relative to the baseline configuration that is typically used in the literature. This difference in NDCG is rendered as a heatmap in Figure 7 for select datasets—figures for all other datasets show a similar pattern. We note that, we select ηLex and ηSem from the set {1, 2, . . . , 100}, but selectively illustrated a subset of values in Figure 7 to make the figures more readable; the omitted combinations of ηLex and ηSem do not bring more insight than what can already be deduced from these figures. As a general observation, we note that NDCG swings wildly as a function of RRF parameters. Crucially, performance improves off-diagonal, where the parameter takes on different values for the semantic and lexical components. On MS MARCO, shown in Figure 7(a), NDCG improves when ηLex > ηSem, while the opposite effect can be seen for HotpotQA, an out-of-domain dataset. This can be easily explained by the fact that increasing ηo for retrieval system o effectively discounts the contribution of ranks from o to the final hybrid score. On in-domain datasets where the semantic model already performs strongly, for example, discounting the lexical system by increasing ηLex leads to better performance. Having observed that tuning RRF potentially leads to gains in NDCG, we ask if tuned parameters generalize on out-of-domain datasets. To investigate that question, we tune RRF on in-domain datasets and pick the value of parameters that maximize NDCG on the validation split of in-domain datasets, and measure the performance of the resulting function on the test split of all (in-domain and out-of-domain) datasets. We present the results in Table 3. While tuning a parametric RRF does indeed lead to gains in NDCG on in-domain datasets, the tuned function does not generalize well to out-of-domain datasets. The poor generalization can be explained by the reversal of patterns observed in Figure 7 where ηLex > ηSem suits in-domain datasets better but the opposite is true for out-of-domain datasets. By modifying ηLex and ηSem we modify the fusion of ranks and boost certain regions and discount others in an imbalanced manner. Figure 8 visualizes this effect on fRRF for particular values of its parameters. This addresses RQ2. 18 Sebastian Bruch, Siyu Gai, and Amir Ingber (a) MS MARCO (b) Quora (c) NQ (d) FiQA (e) HotpotQA (f) Fever Fig. 6. Visualization of the reciprocal rank determined by lexical (rr(πLex) = 1/(60 + πLex)) and semantic (rr(πSem) = 1/(60 + πSem)) retrieval for query-document pairs sampled from the validation split of each dataset. Shown in red are up to 20,000 positive samples where document is relevant to query, and in black up to the same number of negative samples. 5.2 Effect of Lipschitz Continuity In the previous section, we stated an intuition that because RRF does not preserve the distribution of raw scores, it loses valuable information in the process of fusing retrieval systems. In our final research question, RQ3, we investigate if this indeed matters in practice. An Analysis of Fusion Functions for Hybrid Retrieval 19 (a) MS MARCO (b) HotpotQA Fig. 7. Difference in NDCG@1000 of fRRF with distinct values ηLex and ηSem, and fRRF with ηLex = ηSem = 60 (positive indicates better ranking quality by the former). On MS MARCO, an in-domain dataset, NDCG improves when ηLex > ηSem, while the opposite effect can be seen for HotpotQA, an out-of-domain dataset. (a) ηLex = 60, ηSem = 60 (b) ηLex = 10, ηSem = 4 (c) ηLex = 3, ηSem = 5 Fig. 8. Effect of fRRF with select configurations of ηLex and ηSem on pairs of ranks from lexical and semantic systems. When ηLex > ηSem, the fusion function discounts the lexical system's contribution.

The notion of “preserving” information is well captured by the concept of Lipschitz continuity.9 When a function is Lipschitz continuous with a small Lipschitz constant, it does not oscillate wildly with a small change to its input. Intuitively, and in the context of this work, this means that a small change to the individual scores does not lead to a sudden and exaggerated effect on the fused score; the fusion function does not dramatically distort the distribution of scores or ranks, an effect we characterize informally as “preserving information.” RRF does not have this property because the moment one lexical (or semantic) score becomes larger than another the function makes a hard transition to a new value. 9A function f is Lipschitz continous with constant L if $|f(y) - f(x)|_o \leq L|y - x|_i$ for some norm $|\cdot|_o$ and $|\cdot|_i$ on the output and input space of f. 20 Sebastian Bruch, Siyu Gai, and Amir Ingber Table 3. Mean NDCG@1000 (NDCG@100 for SciFact and NFCorpus) on the test split of various datasets for hybrid retrieval using TM2C2 (α = 0.8) and RRF (ηLex, ηSem). The symbols ‡ and ∗ indicate statistical significance (p-value < 0.01) with respect to TM2C2 and baseline RRF (60, 60) respectively, according to a paired two-tailed t-test. NDCG Dataset TM2C2 RRF (60, 60) RRF (5, 5) RRF (10, 4) in-domain MS MARCO 0.454 0.425‡ 0.435‡∗ 0.451∗ NQ 0.542 0.514‡ 0.521‡∗ 0.528‡∗ Quora 0.901 0.877‡ 0.885‡∗ 0.896∗ zero-shot NFCorpus 0.327 0.312‡ 0.318‡∗ 0.310‡ HotpotQA 0.699 0.675‡ 0.693∗ 0.621‡∗ FEVER 0.744 0.721‡ 0.727‡∗ 0.649‡∗ SciFact 0.753 0.730‡ 0.738‡ 0.715‡∗ DBPedia 0.512 0.489‡ 0.489‡ 0.480‡∗ FiQA 0.496 0.464‡ 0.470‡∗ 0.482‡∗ We can therefore cast RQ3 as a question of whether Lipschitz continuity is an important property in practice. To put that hypothesis to the test, we design a smooth approximation of RRF using known techniques [4, 31]. As expressed in Equation (1), the rank of a document is simply the sum of indicators. It is thus trivial to approximate this quantity using a generalized sigmoid with parameter β: $\sigma_\beta(x) = 1/(1 + \exp(-\beta x))$. As β approaches 1, the sigmoid takes its usual S shape, while $\beta \to \infty$ produces a very close approximation of the indicator. Interestingly, the Lipschitz constant of $\sigma_\beta(\cdot)$ is, in fact, β. As β increases, the approximation of ranks becomes more accurate, but the Lipschitz constant becomes larger. When β is too small, however, the approximation breaks down but the function transitions more slowly, thereby preserving much of the characteristics of the underlying data distribution. RRF being a function of ranks can now be approximated by plugging in approximate ranks in Equation (7), resulting in SRRF: fSRRF $(q, d) = \frac{1}{\eta + \tilde{\pi}\text{Lex}(q, d)} + \frac{1}{\eta + \tilde{\pi}\text{Sem}(q, d)}$, (9) where $\tilde{\pi}\text{o}(q, di) = 0.5 + \sum_{dj \in R_k \text{ o } (q)} \sigma_\beta(f\text{o}(q, dj) - f\text{o}(q, di))$. By increasing β we increase the Lipschitz constant of fSRRF. This is the lever we need to test the idea that Lipschitz continuity matters and that functions that do not distort the distributional properties of raw scores lead to better ranking quality. Figures 9 and 10 visualize the difference between SRRF and RRF for two settings of η selected based on the results in Table 3. As anticipated, when β is too small, the approximation error is large and ranking quality degrades. As β becomes larger, ranking quality trends in the direction of RRF. Interestingly, as β becomes gradually smaller, the performance of SRRF improves over the RRF baseline. This effect is more pronounced for the η = 60 setting of RRF, as well as on the out-of-domain datasets. Empirical evidence reported for other fusions in Appendices A through D shows this finding to be robust. An Analysis of Fusion Functions for Hybrid Retrieval 21 (a) in-domain (b) out-of-domain Fig. 9. The difference in NDCG@1000 of fSRRF and fRRF with η = 60 (positive indicates better ranking quality by SRRF) as a function of β. (a) in-domain (b) out-of-domain Fig. 10. The difference in NDCG@1000 of fSRRF and fRRF with η = 5 (positive indicates better ranking quality by SRRF) as a function of β. While we acknowledge the

possibility that the approximation in Equation (9) may cause a change in ranking quality, we expected that change to be a degradation, not an improvement. However, given we do observe gains by smoothing the function, and that the only other difference between SRRF and RRF is their Lipschitz constant, we believe these results highlight the role of Lipschitz continuity in ranking quality. For completeness, we have also included a comparison of SRRF, RRF, and TM2C2 in Table 4. 22 Sebastian Bruch, Siyu Gai, and Amir Ingber Table 4. Mean NDCG@1000 (NDCG@100 for SciFact and NFCorpus) on the test split of various datasets for hybrid retrieval using TM2C2 (α = 0.8), RRF (η), and SRRF(η, β). The parameters β are fixed to values that maximize NDCG on the validation split of in-domain datasets. The symbols ‡ and $*$ indicate statistical significance (p-value < 0.01) with respect to TM2C2 and RRF respectively, according to a paired two-tailed t-test. NDCG Dataset TM2C2 RRF(60) SRRF (60, 40) RRF(5) SRRF (5, 100) in-domain MS MARCO 0.454 0.425‡ 0.431‡$*$ 0.435‡ 0.431‡$*$ NQ 0.542 0.514‡ 0.516‡ 0.521‡ 0.517‡ Quora 0.901 0.877‡ 0.889‡$*$ 0.885‡ 0.889‡$*$ zero-shot NFCorpus 0.327 0.312‡ 0.323‡$*$ 0.318‡ 0.322‡ HotpotQA 0.699 0.675‡ 0.695$*$ 0.693‡ 0.705‡$*$ FEVER 0.744 0.721‡ 0.725‡ 0.727‡ 0.735‡$*$ SciFact 0.753 0.730‡ 0.740‡ 0.738‡ 0.740‡ DBPedia 0.512 0.489‡ 0.501‡$*$ 0.489‡ 0.492‡ FiQA 0.496 0.464‡ 0.468‡ 0.470‡ 0.469‡ 6 DISCUSSION The analysis in this work motivates us to identify and document the properties of a well-behaved fusion function, and present the principles that, we hope, will guide future research in this space. These desiderata are stated below. Monotonicity: When fo is positively correlated with a target ranking metric (i.e., ordering documents in decreasing order of fo must lead to higher quality), then it is natural to require that fHybrid be monotone increasing in its arguments. We have already seen and indeed used this property in our analysis of the convex combination fusion function. It is trivial to show why this property is crucial. Homogeneity: The order induced by a fusion function must be unaffected by a positive rescaling of query and document vectors. That is: fHybrid(q, d) π = fHybrid(q,γd) π = fHybrid(γq, d) where π = denotes rank-equivalence and γ > 0. This property prevents any retrieval system from inflating its contribution to the final hybrid score by simply boosting its document or query vectors. Boundedness: Recall that, a convex combination without score normalization is often ineffective and inconsistent because BM25 is unbounded and that lexical and semantic scores are on different scales. To see this effect we turn to Figure 11. We observe in Figure 11(a) that, for in-domain datasets, adding the unnormalized lexical scores using a convex combination leads to a severe degradation of ranking quality. We believe this is because of the fact that the semantic retrieval model, which is fine-tuned on these datasets, already produces ranked lists of high quality, and that adding the lexical scores which are on a very different scale distorts the rankings and leads to poor performance. In out-of-domain experiments as shown in Figure 11(b), however, the addition of lexical scores leads to often significant gains in quality. We believe this can be explained exactly as the in-domain observations: the semantic model generally does poorly on out-of-domain datasets while the lexical retriever does well. But because the semantic scores are bounded and relatively small, they do not significantly distort the rankings produced by the lexical retriever. An Analysis of Fusion Functions for Hybrid Retrieval 23 (a) in-domain (b) out-of-domain Fig. 11. The difference in NDCG of convex combination of unnormalized scores and a pure semantic search (positive indicates better ranking quality by a convex combination) as a function of α. To avoid that pitfall, we require that fHybrid be bounded: |fHybrid| ≤ M for some M > 0. As we have seen before, normalizing the raw scores addresses this issue.

Lipschitz Continuity: We argued that because RRF does not take into consideration the raw scores, it distorts their distribution and thereby loses valuable information. On the other hand, TM2C2 (or any convex combination of scores) is a smooth function of scores and preserves much of the characteristics of its underlying distribution. We formalized this idea using the notion of Lipschitz continuity: A larger Lipschitz constant leads to a larger distortion of retrieval score distribution. Interpretability and Sample Efficiency: The question of hybrid retrieval is an important topic in IR. What makes it particularly pertinent is its zero-shot applicability, a property that makes deep models reusable, reducing computational costs and emissions as a result [3, 35], and enabling resource-constrained research labs to innovate. Given the strong evidence supporting the idea that hybrid retrieval is most valuable when applied to out-of-domain datasets [5], we believe that fHybrid should be robust to distributional shifts and should not need training or fine-tuning on target datasets. This implies that either the function must be non-parametric, that its parameters can be tuned efficiently with respect to the training samples required, or that they are highly interpretable such that their value can be guided by expert knowledge. In the absence of a truly non-parametric approach, however, we believe a fusion that is more sample-efficient to tune is preferred. Because convex combination has fewer parameters than the fully parameterized RRF, we believe it should have this property. To confirm, we ask how many training queries it takes to converge to the correct α on a target dataset. Figure 12 visualizes our experiments, where we plot NDCG of RRF (η = 60) and TM2C2 with α = 0.8 from Table 2. Additionally, we take the train split of each dataset and sample from it progressively larger subsets (with a step size of 5%), and use it to tune the parameters of each function. We then measure NDCG of the tuned functions on the test split. For the depicted datasets as well as all other datasets in this work, we observe a similar trend: with less than 5% of the training data, which is often a small set of queries, TM2C2's α converges, regardless of the magnitude of domain shift. This sample efficiency is remarkable because it enables significant gains with little 24 Sebastian Bruch, Siyu Gai, and Amir Ingber (a) MS MARCO (b) Quora (c) HotpotQA (d) Fever (e) NFCorpus (f) FiQA Fig. 12. Sample efficiency of TM2C2 and the parameterized variants of RRF (single parameter where ηSem = ηLex, and two parameters where we allow different values of ηSem and ηLex, and a third variation that is a convex combination of RRF terms defined in Equation 10). We sample progressively larger subsets of the training set (with a step size of 5%), tune the parameters of each function on the resulting set, and evaluate the resulting function on the test split. These figures depict NDCG@1000 as a function of the size of the tuning set, averaged over 5 trials with the shaded regions illustrating the 95% confidence intervals. For reference, we have also plotted NDCG on the test split for RRF (η = 60) and TM2C2 with α = 0.8 from Table 2. An Analysis of Fusion Functions for Hybrid Retrieval 25 labeling effort. Finally, while RRF does not settle on a value and its parameters are sensitive to the training sample, its performance does more or less converge. However, the performance of the fully parameterized RRF is still sub-optimal compared with TM2C2. In Figure 12, we also include a convex combination of fully parameterized RRF terms, denoted by RRF-CC and defined as: fRRF (q, d) = (1 − α) 1 ηLex + πLex (q, d) + α 1 ηSem + πSem(q, d) , (10) where α, ηLex, and ηSem are tunable parameters. The question this particular formulation tries to answer is whether adding an additional weight to the combination of the RRF terms affects retrieval quality. From the figure, it is clear that the addition of this parameter does not have a significant impact on the overall performance. This also serves as additional evidence supporting the claim

that Lipschitz continuity is an important property. 7 CONCLUSION We studied the behavior of two popular functions that fuse together lexical and semantic retrieval to produce hybrid retrieval, and identified their advantages and pitfalls. Importantly, we investigated several questions and claims in prior work. We established theoretically that the choice of normalization is not as consequential as once thought for a convex combination-based fusion function. We found that RRF is sensitive to its parameters. We also observed empirically that convex combination of normalized scores outperforms RRF on in-domain and out-of-domain datasets—a finding that is in disagreement with [5]. We believe that a convex combination with theoretical minimum-maximum normalization (TM2C2) indeed enjoys properties that are important in a fusion function. Its parameter, too, can be tuned sample-efficiently or set to a reasonable value based on domain knowledge. In our experiments, for example, we found the range $\alpha \in [0.6, 0.8]$ to consistently lead to improvements. While we observed that a line appears to be appropriate for a collection of query-document pairs, we acknowledge that, that may change if our analysis was conducted on a per-query basis—itself a rather non-trivial effort. For example, it is unclear if bringing non-linearity to the design of the fusion function or the normalization itself leads to a more accurate prediction of α on a per-query basis. We leave an exploration of this question to future work. We also note that, while our analysis does not exclude the use of multiple retrieval engines as input, and indeed can be extended, both theoretically and empirically, to a setting where we have more than just lexical and semantic scores, it is nonetheless important to conduct experiments and validate that our findings generalize. In particular, we ask if the role of normalization changes when fusing three or more models, and if so, what is the behavior of convex combination given a particular normalization function, and how does that compare with RRF. We believe, however, that our current assumptions are practical and are reflective of the current state of hybrid search where we typically fuse only lexical and semantic retrieval systems. As such, we leave an extended analysis of fusion on multiple retrieval systems to future work. ACKNOWLEDGMENTS We benefited greatly from conversations with Brian Hentschel, Edo Liberty, and Michael Bendersky. We are grateful to them for their insight and time. We further thank the anonymous reviewers for their meticulous examination of our claims and for their insightful feedback. REFERENCES [1] Nima Asadi. 2013. Multi-Stage Search Architectures for Streaming Documents. University of Maryland. [2] Nima Asadi and Jimmy Lin. 2013. Effectiveness/Efficiency Tradeoffs for Candidate Generation in Multi-Stage Retrieval Architectures. In Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval (Dublin, Ireland). 997–1000. [3] Sebastian Bruch, Claudio Lucchese, and Franco Maria Nardini. 2022. ReNeuIR: Reaching Efficiency in Neural Information Retrieval. In Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval (Madrid, Spain). 3462–3465. [4] Sebastian Bruch, Masrour Zoghi, Michael Bendersky, and Marc Najork. 2019. Revisiting Approximate Metric Optimization in the Age of Deep Neural Networks. In Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval (Paris, France). 1241–1244. [5] Tao Chen, Mingyang Zhang, Jing Lu, Michael Bendersky, and Marc Najork. 2022. Out-of-Domain Semantics to the Rescue! Zero-Shot Hybrid Retrieval Models. In Advances in Information Retrieval: 44th European Conference on IR Research, ECIR 2022, Stavanger, Norway, April 10–14, 2022, Proceedings, Part I (Stavanger, Norway).

95–110. [6] Gordon V. Cormack, Charles L A Clarke, and Stefan Buettcher. 2009. Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods. In Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval (Boston, MA, USA). 758–759. [7] Van Dang, Michael Bendersky, and W Bruce Croft. 2013. Two-Stage learning to rank for information retrieval. In Advances in Information Retrieval. Springer, 423–434. [8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). 4171–4186. [9] Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. 2022. From Distillation to Hard Negative Sampling: Making Sparse Neural IR Models More Effective. In Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval (Madrid, Spain). 2353–2359. [10] Sebastian Hofstätter, Sheng-Chieh Lin, Jheng-Hong Yang, Jimmy Lin, and Allan Hanbury. 2021. Efficiently Teaching an Effective Dense Retriever with Balanced Topic Aware Sampling. In Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval (Virtual Event, Canada). 113–122. [11] Kalervo Järvelin and Jaana Kekäläinen. 2000. IR evaluation methods for retrieving highly relevant documents. In Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval. ACM, 41–48. [12] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. IEEE Transactions on Big Data 7 (2021), 535–547. [13] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP). [14] Saar Kuzi, Mingyang Zhang, Cheng Li, Michael Bendersky, and Marc Najork. 2020. Leveraging Semantic and Lexical Matching to Improve the Recall of Document Retrieval Systems: A Hybrid Approach. (2020). arXiv:2010.01195 [cs.IR] [15] Hang Li, Shuai Wang, Shengyao Zhuang, Ahmed Mourad, Xueguang Ma, Jimmy Lin, and Guido Zuccon. 2022. To Interpolate or Not to Interpolate: PRF, Dense and Sparse Retrievers. In Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval (Madrid, Spain). 2495–2500. [16] Jimmy Lin, Rodrigo Nogueira, and Andrew Yates. 2021. Pretrained Transformers for Text Ranking: BERT and Beyond. arXiv:2010.06467 [cs.IR] [17] Tie-Yan Liu. 2009. Learning to Rank for Information Retrieval. Foundations and Trends in Information Retrieval 3, 3 (2009), 225–331. [18] Yi Luan, Jacob Eisenstein, Kristina Toutanova, and Michael Collins. 2021. Sparse, Dense, and Attentional Representations for Text Retrieval. Transactions of the Association for Computational Linguistics 9 (2021), 329–345. [19] Ji Ma, Ivan Korotkov, Keith Hall, and Ryan T. McDonald. 2020. Hybrid First-stage Retrieval Models for Biomedical Literature. In CLEF. [20] Xueguang Ma, Kai Sun, Ronak Pradeep, and Jimmy J. Lin. 2021. A Replication Study of Dense Passage Retriever. (2021). arXiv:2004.04906 [cs.CL] [21] Craig Macdonald, Rodrygo LT Santos, and Iadh Ounis. 2013. The whens and hows of learning to rank for web search. Information Retrieval 16, 5 (2013), 584–628. [22] Yu. A. Malkov and D. A. Yashunin. 2016. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. arXiv:1603.09320 [cs.DS] [23] Antonio Mallia, Michal Siedlaczek, Joel Mackenzie, and Torsten Suel. 2019. PISA: Performant Indexes and Search for

Academia. In Proceedings of the Open-Source IR Replicability Challenge co-located with 42nd International ACM SIGIR An Analysis of Fusion Functions for Hybrid Retrieval 27 Conference on Research and Development in Information Retrieval Paris, France, July 25, 2019. 50–56. [24] Yoshitomo Matsubara, Thuy Vu, and Alessandro Moschitti. 2020. Reranking for Efficient Transformer-Based Answer Selection. In Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval. 1577–1580. [25] Bhaskar Mitra, Eric Nalisnick, Nick Craswell, and Rich Caruana. 2016. A dual embedding space model for document ranking. (2016). arXiv:1602.01137 [cs.IR] [26] Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. 2016. MS MARCO: A Human Generated MAchine Reading COmprehension Dataset. (November 2016). [27] Rodrigo Nogueira and Kyunghyun Cho. 2020. Passage Re-ranking with BERT. arXiv:1901.04085 [cs.IR] [28] Rodrigo Nogueira, Zhiying Jiang, Ronak Pradeep, and Jimmy Lin. 2020. Document Ranking with a Pretrained Sequence-to-Sequence Model. In Findings of the Association for Computational Linguistics: EMNLP 2020. 708–718. [29] Rodrigo Nogueira, Wei Yang, Kyunghyun Cho, and Jimmy Lin. 2019. Multi-stage document ranking with BERT. (2019). arXiv:1910.14424 [cs.IR] [30] Rodrigo Nogueira, Wei Yang, Jimmy Lin, and Kyunghyun Cho. 2019. Document Expansion by Query Prediction. (2019). arXiv:1904.08375 [cs.IR] [31] Tao Qin, Tie-Yan Liu, and Hang Li. 2010. A general approximation framework for direct optimization of information retrieval measures. Information retrieval 13, 4 (2010), 375–397. [32] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics. [33] Stephen Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. Foundations and Trends in Information Retrieval 3, 4 (April 2009), 333–389. [34] Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. 1994. Okapi at TREC-3. In TREC (NIST Special Publication, Vol. 500-225). National Institute of Standards and Technology (NIST), 109–126. [35] Harrisen Scells, Shengyao Zhuang, and Guido Zuccon. 2022. Reduce, Reuse, Recycle: Green Information Retrieval Research. In Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval (Madrid, Spain). 2825–2837. [36] Tao Tao, Xuanhui Wang, Qiaozhu Mei, and ChengXiang Zhai. 2006. Language Model Information Retrieval with Document Expansion. In Proceedings of the Main Conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics (New York, New York). 407–414. [37] Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. 2021. BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models. In Proceedings of the 35th Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2). [38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In Proceedings of the 31st International Conference on Neural Information Processing Systems (Long Beach, California, USA). 6000–6010. [39] Lidan Wang, Jimmy Lin, and Donald Metzler. 2011. A cascade ranking model for efficient ranked retrieval. In Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval. ACM, 105–114. [40] Shuai Wang, Shengyao Zhuang, and Guido Zuccon. 2021. BERT-Based Dense Retrievers Require Interpolation with BM25 for Effective Passage

Retrieval. In Proceedings of the 2021 ACM SIGIR International Conference on Theory of Information Retrieval (Virtual Event, Canada). 317–324. [41] Qiang Wu, Christopher J.C. Burges, Krysta M. Svore, and Jianfeng Gao. 2010. Adapting boosting for information retrieval measures. Information Retrieval (2010). [42] Xiang Wu, Ruiqi Guo, David Simcha, Dave Dopson, and Sanjiv Kumar. 2019. Efficient Inner Product Approximation in Hybrid Spaces. (2019). arXiv:1903.08690 [cs.LG] [43] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. arXiv:1609.08144 [cs.CL] [44] Lee Xiong, Chenyan Xiong, Ye Li, Kwok-Fung Tang, Jialin Liu, Paul Bennett, Junaid Ahmed, and Arnold Overwijk. 2021. Approximate Nearest Neighbor Negative Contrastive Learning for Dense Text Retrieval. In International Conference on Learning Representations. [45] Dawei Yin, Yuening Hu, Jiliang Tang, Tim Daly, Mianwei Zhou, Hua Ouyang, Jianhui Chen, Changsung Kang, Hongbo Deng, Chikashi Nobata, et al. 2016. Ranking relevance in yahoo search. In Proceedings of the 22nd ACM SIGKDD 28 Sebastian Bruch, Siyu Gai, and Amir Ingber International Conference on Knowledge Discovery and Data Mining. ACM, 323–332. [46] Hamed Zamani, Mike Bendersky, Donald Metzler, Honglei Zhuang, and Marc Najork. 2022. Stochastic RetrievalConditioned Reranking. In Proceedings of the 2022 ACM SIGIR International Conference on the Theory of Information Retrieval (Madrid, Spain). [47] Jingtao Zhan, Jiaxin Mao, Yiqun Liu, Min Zhang, and Shaoping Ma. 2020. RepBERT: Contextualized Text Embeddings for First-Stage Retrieval. arXiv:2006.15498 [cs.IR] An Analysis of Fusion Functions for Hybrid Retrieval 29 A FUSION OF SPLADE AND BM25 (a) MS MARCO (b) Quora (c) HotpotQA (d) FiQA Fig. 13. Effect of normalization on fConvex = $\alpha\, fA$ + (1 − α)fB, where fA is Splade and fB is BM25, as a function of α (left); and, the relative expansion rate of Splade scores with respect to BM25 scores (i.e., λ in Definition 4.4), with 95% confidence intervals (right). ϕ-fo indicates that the normalization function ϕ was only applied to fo , with the other function entering fusion without normalization. 30 Sebastian Bruch, Siyu Gai, and Amir Ingber Table 5. NDCG@1000 (except SciFact and NFCorpus where cutoff is 100) on the test split of various datasets for individual systems and their fusion using RRF [5] (η = 60) and TM2C2 (α = 0.8 in fConvex = αSplade + (1 − α)BM25). The symbols ‡ and ∗ indicate statistical significance (p-value < 0.01) with respect to TM2C2 and RRF respectively, according to a paired two-tailed t-test. NDCG Dataset BM25 Splade TM2C2 RRF MS MARCO 0.309‡∗ 0.508∗ 0.507 0.444‡ NQ 0.382‡∗ 0.591‡∗ 0.587 0.520‡ Quora 0.798‡∗ 0.853‡ 0.876 0.859‡ NFCorpus 0.269‡∗ 0.314∗ 0.317 0.304‡ HotpotQA 0.682‡∗ 0.727‡∗ 0.751 0.737‡ FEVER 0.689‡∗ 0.806‡∗ 0.825 0.786‡ SciFact 0.698‡∗ 0.723‡∗ 0.740 0.732‡ DBPedia 0.415‡∗ 0.546‡∗ 0.556 0.526‡ FiQA 0.315‡∗ 0.442∗ 0.446 0.406‡ (a) in-domain (b) out-of-domain Fig. 14. The difference in NDCG@1000 of fSRRF and fRRF with η = 60 (positive indicates better ranking quality by SRRF) as a function of β. B FUSION OF TAS-B AND BM25 An Analysis of Fusion Functions for Hybrid Retrieval 31 (a) MS MARCO (b) Quora (c) HotpotQA (d) FiQA Fig. 15. Effect of normalization on fConvex = $\alpha\, fA$ + (1 − α)fB, where fA is the Tas-B function and fB is BM25, as a function of α (left); and, the relative expansion rate of Tas-B scores with respect to BM25 scores (i.e., λ in Definition 4.4),

with 95% confidence intervals (right). ϕ-fo indicates that the normalization function ϕ was only applied to fo , with the other function entering fusion without normalization. 32 Sebastian Bruch, Siyu Gai, and Amir Ingber Table 6. NDCG@1000 (except SciFact and NFCorpus where cutoff is 100) on the test split of various datasets for individual systems and their fusion using RRF [5] (η = 60) and M2C2 (α = 0.8 in αTas-B+ (1−α)BM25). The symbols ‡ and ∗ indicate statistical significance (p-value < 0.01) with respect to M2C2 and RRF respectively, according to a paired two-tailed t-test. NDCG Dataset BM25 Tas-B M2C2 RRF MS MARCO 0.309‡∗ 0.477‡∗ 0.486 0.434‡ NQ 0.382‡∗ 0.522‡∗ 0.552 0.513‡ Quora 0.798‡∗ 0.856‡ 0.881 0.860‡ NFCorpus 0.269‡∗ 0.292‡ 0.307 0.299 HotpotQA 0.682‡∗ 0.631‡∗ 0.702 0.711‡ FEVER 0.689‡∗ 0.725‡∗ 0.785 0.776‡ SciFact 0.698‡∗ 0.670‡∗ 0.715 0.719 DBPedia 0.415‡∗ 0.495‡∗ 0.533 0.512‡ FiQA 0.315‡∗ 0.396‡∗ 0.422 0.401‡ (a) in-domain (b) out-of-domain Fig. 16. The difference in NDCG@1000 of fSRRF and fRRF with η = 60 (positive indicates better ranking quality by SRRF) as a function of β. An Analysis of Fusion Functions for Hybrid Retrieval 33 C FUSION OF TAS-B AND SPLADE (a) MS MARCO (b) Quora (c) HotpotQA (d) FiQA Fig. 17. Effect of normalization on fConvex = α fA + (1 −α)fB, where fA is the Tas-B function and fB is Splade, as a function of α (left); and, the relative expansion rate of Tas-B scores with respect to Splade scores (i.e., λ in Definition 4.4), with 95% confidence intervals (right). ϕ-fo indicates that the normalization function ϕ was only applied to fo , with the other function entering fusion without normalization. 34 Sebastian Bruch, Siyu Gai, and Amir Ingber Table 7. NDCG@1000 (except SciFact and NFCorpus where cutoff is 100) on the test split of various datasets for individual systems and their fusion using RRF [5] (η = 60) and M2C2 (α = 0.2 in αTas-B+(1−α)Splade). The symbols ‡ and ∗ indicate statistical significance (p-value < 0.01) with respect to M2C2 and RRF respectively, according to a paired two-tailed t-test. NDCG Dataset Splade Tas-B M2C2 RRF MS MARCO 0.508 0.477‡∗ 0.512 0.504‡ NQ 0.589 0.522‡∗ 0.593 0.581‡ Quora 0.853 0.856 0.859 0.857 NFCorpus 0.313 0.292‡∗ 0.318 0.314 HotpotQA 0.727∗ 0.631‡∗ 0.728 0.686‡ FEVER 0.806∗ 0.725‡∗ 0.811 0.795‡ SciFact 0.723∗ 0.670‡∗ 0.730 0.709‡ DBPedia 0.546‡ 0.495‡∗ 0.555 0.545‡ FiQA 0.442‡ 0.396‡∗ 0.453 0.440‡ (a) in-domain (b) out-of-domain Fig. 18. The difference in NDCG@1000 of fSRRF and fRRF with η = 60 (positive indicates better ranking quality by SRRF) as a function of β. D FUSION OF TAS-B AND ALL-MINILM-L6-V2 An Analysis of Fusion Functions for Hybrid Retrieval 35 (a) MS MARCO (b) Quora (c) HotpotQA (d) FiQA Fig. 19. Effect of normalization on fConvex = α fA + (1 − α)fB, where fA is Tas-B and fB is All-MiniLM-l6-v2, as a function of α (left); and, the relative expansion rate of Tas-B scores with respect to All-MiniLM-l6-v2 scores (i.e., λ in Definition 4.4), with 95% confidence intervals (right). ϕ-fo indicates that the normalization function ϕ was only applied to fo , with the other function entering fusion without normalization. 36 Sebastian Bruch, Siyu Gai, and Amir Ingber Table 8. NDCG@1000 (except SciFact and NFCorpus where cutoff is 100) on the test split of various datasets for individual systems and their fusion using RRF [5] (η = 60) and M2C2 (α = 0.8 in αTas-B + (1 − α)All-MiniLM-l6-v2). The symbols ‡ and ∗ indicate statistical significance (p-value < 0.01) with respect to M2C2 and RRF respectively, according to a paired two-tailed t-test. NDCG Dataset All-MiniLM-l6-v2 Tas-B M2C2 RRF MS MARCO 0.441‡∗ 0.477‡ 0.483 0.474‡ NQ 0.505‡∗ 0.522‡∗ 0.547 0.550 Quora 0.889‡∗ 0.856‡ 0.872 0.877 NFCorpus 0.296‡∗ 0.292‡∗ 0.307 0.316‡ HotpotQA 0.520‡∗ 0.631‡∗ 0.646 0.610‡ FEVER 0.558‡∗ 0.725∗ 0.730 0.669‡ SciFact 0.681‡∗ 0.670‡∗ 0.704 0.712 DBPedia 0.425‡∗ 0.495‡ 0.510 0.496‡ FiQA 0.467‡ 0.396‡∗ 0.438 0.467‡ (a) in-domain (b) out-of-domain Fig. 20. The

difference in NDCG@1000 of fSRRF and fRRF with η = 60 (positive indicates better ranking quality by SRRF) as a function of β.

Concepts - Projects

This document explains the concepts related to Pinecone projects.

Projects contain indexes and users

Each Pinecone project contains a number of indexesand users. Only a user who belongs to the project can access the indexes in that project. Each project also has at least one project owner. All of the pods in a single project are located in a single environment.

Project settings

When you create a new project, you can choose the name, deployment environment, and pod limit.

Project environment

When creating a project, you must choose a cloud environment for the indexes in that project. The following table lists the available cloud regions, the corresponding values of the environmentparameter for the init() operation, and which billing tier has access to each environment:

Cloud region	environmentvalue	Tier availability
GCP US-West-1 Free (N. California)	us-west1-gcp-free	Starter
GCP Asia-Southeast-1 (Singapore)	asia-southeast1-gcp-free	Starter
GCP US-West-4 (Las Vegas)	us-west4-gcp	Starter
GCP US-West-1 (N. California)	us-west1-gcp	Standard / Enterprise
GCP US-Central-1 (Iowa)	us-central1-gcp	Standard / Enterprise
GCP US-West-4 (Las Vegas)	us-west4-gcp	Standard /

		Enterprise
GCP US-East-4 (Virginia)	us-east4-gcp	Standard / Enterprise
GCP northamerica-northeast-1	northamerica-northeast1-gcp	Standard / Enterprise
GCP Asia-Northeast-1 (Japan)	asia-northeast1-gcp	Standard / Enterprise
GCP Asia-Southeast-1 (Singapore)	asia-southeast1-gcp	Standard / Enterprise
GCP US-East-1 (South Carolina)	us-east1-gcp	Standard / Enterprise
GCP EU-West-1 (Ireland)	eu-west1-gcp	Standard / Enterprise
GCP EU-West-4 (Netherlands)	eu-west4-gcp	Standard / Enterprise
AWS US-East-1 (Virginia)	us-east1-aws	Standard / Enterprise

Contact usif you need a dedicated deployment in other regions.

The environment cannot be changed after the project is created.

Project pod limit

You can set the maximum number of pods that can be used in total across all indexes in a project. Use this to control costs.

The pod limit can be changed only by the project owner.

Project roles

There are two project roles: Project ownerand project member.Table 1 below summarizes the permissions for each role.

Table 1: Project roles and permissions

Project role	Permissions in organization

Project owner	Manage project members
	Manage project API keys
	Manage pod limits
Project member	Access API keys
	Create indexes in project
	Use indexes in project

Concepts - Organizations

A Pinecone organization is a set of projectsthat use the same billing. Organizations allow one or more users to control billing and project permissions for all of the projects belonging to the organization. Each project belongs to an organization.

For a guide to adding users to an organization, see Add users to a project or organization.

Projects in an organization

Each organization contains one or more projects that share the same organization owners and billing settings. Each project belongs to exactly one organization. If you need to move a project from one organization to another, contact Pinecone support.

Billing settings

All of the projects in an organization share the same billing method and settings. The billing settings for the organization are controlled by the organization owners.

Organization roles

There are two organization roles: organization owner and organization user.

Organization owners

Organization owners manage organization billing, users, and projects. Organization owners are also project ownersfor every project belonging to the organization. This means that organization owners have all permissions to manage project members, API keys, and quotas for these projects.

Organization users

Unlike organization owners, organization users cannot edit billing settings or invite new users to the organization. Organization users can create new projects, and project owners can add organization members to a project. New users have whatever role the organization owners and project owners grant them. Project owners can add users to a project if those users belong to the same organization as the project.

Table 1: Organization roles and permissions

Organization role	Permissions in organization
Organization owner	Project owner for all projects
	Create projects
	Manage billing
	Manags organization members
Organization member	Create projects
	Join projects when invited
	Read access to billing

Organization single sign-on (SSO)

SSO allows organizations to manage their teams' access to Pinecone through their identity management solution. Once your integration is configured, you can require that users from your domain sign in through SSO, and you can specify a default role for teammates when they sign up. Only organizations in the enterprise tier can set up SSO. To set up your SSO integration, contact Pinecone support at support@pinecone.io.

Sparse-dense embeddings - Keyword search algorithms like the BM25 algorithm compute

the relevance of text documents based on the number of keyword matches, their frequency, and other factors.

In information retrieval, **Okapi BM25** (*BM* is an abbreviation of *best matching*) is a ranking function used by search engines to estimate the relevance of documents to a given search query. It is based on the probabilistic retrieval framework developed in the 1970s and 1980s by Stephen E. Robertson, Karen Spärck Jones, and others.

The name of the actual ranking function is *BM25*. The fuller name, *Okapi BM25*, includes the name of the first system to use it, which was the Okapi information retrieval system, implemented at London's City University in the 1980s and 1990s. BM25 and its newer variants, e.g. BM25F (a version of BM25 that can take document structure and anchor text into account), represent TF-IDF-like retrieval functions used in document retrieval.[*citation needed*]

The ranking function[edit]

BM25 is a bag-of-words retrieval function that ranks a set of documents based on the query terms appearing in each document, regardless of their proximity within the document. It is a family of scoring functions with slightly different components and parameters. One of the most prominent instantiations of the function is as follows.

Given a query *Q*, containing keywords , the BM25 score of a document *D* is:
where is the number of times that occurs in the document *D*, is the length of the document *D* in words, and avgdl is the average document length in the text collection from which documents are drawn. and *b* are free parameters, usually chosen, in absence of an advanced optimization, as and .[1] is the IDF (inverse document frequency) weight of the query term . It is usually computed as:
where *N* is the total number of documents in the collection, and is the number of documents containing .

There are several interpretations for IDF and slight variations on its formula. In the original BM25 derivation, the IDF component is derived from the Binary Independence Model.

IDF information theoretic interpretation[edit]

Here is an interpretation from information theory. Suppose a query term appears in documents. Then a randomly picked document will contain the term with probability (where is again the

cardinality of the set of documents in the collection). Therefore, the information content of the message " contains " is:
Now suppose we have two query terms and . If the two terms occur in documents entirely independently of each other, then the probability of seeing both and in a randomly picked document is:

and the information content of such an event is:

With a small variation, this is exactly what is expressed by the IDF component of BM25.

Modifications[edit]

- At the extreme values of the coefficient *b* BM25 turns into ranking functions known as **BM11** (for) and **BM15** (for).[2]
- **BM25F**[3][4] (or the **BM25 model with Extension to Multiple Weighted Fields**[5]) is a modification of BM25 in which the document is considered to be composed from several fields (such as headlines, main text, anchor text) with possibly different degrees of importance, term relevance saturation and length normalization. BM25F defines each type of field as a *stream*, applying a per-stream weighting to scale each stream against the calculated score.
- **BM25+**[6] is an extension of BM25. BM25+ was developed to address one deficiency of the standard BM25 in which the component of term frequency normalization by document length is not properly lower-bounded; as a result of this deficiency, long documents which do match the query term can often be scored unfairly by BM25 as having a similar relevancy to shorter documents that do not contain the query term at all. The scoring formula of BM25+ only has one additional free parameter (a default value is 1.0 in absence of a training data) as compared with BM25:

Concepts - Indexes

Overview

This document describes concepts related to Pinecone indexes. To learn how to create or modify an index, see Manage indexes.

An index is the highest-level organizational unit of vector data in Pinecone. It accepts and stores vectors, serves queries over the vectors it contains, and does other vector operations over its contents. Each index runs on at least one pod.

Pods, pod types, and pod sizes

Pods are pre-configured units of hardware for running a Pinecone service. Each index runs on one or more pods. Generally, more pods mean more storage capacity, lower latency, and higher throughput. You can also create pods of different sizes.

Once an index is created using a particular pod type, you cannot change the pod type for that index. However, you can create a new index from that collectionwith a different pod type.

Different pod types are priced differently. See pricingfor more details.

Starter plan

When using the starter plan, you can create one pod with enough resources to support approximately 100,000 vectors with 1536-dimensional embeddings and metadata; the capacity is proportional for other dimensions.

When using a starter plan, all create_indexcalls ignore the pod_typeparameter.

s1 pods

These storage-optimized pods provide large storage capacity and lower overall costs with slightly higher query latencies than p1 pods. They are ideal for very large indexes with moderate or relaxed latency requirements.

Each s1 pod has enough capacity for around 5M vectors of 768 dimensions.

p1 pods

These performance-optimized pods provide very low query latencies, but hold fewer vectors per pod than s1 pods. They are ideal for applications with low latency requirements (<100ms).

Each p1 pod has enough capacity for around 1M vectors of 768 dimensions.

p2 pods

The p2 pod type provides greater query throughput with lower latency. For vectors with fewer than 128 dimension and queries where topKis less than 50, p2 pods support up to 200 QPS per replica and return queries in less than 10ms. This means that query throughput and latency are better than s1 and p1.

Each p2 pod has enough capacity for around 1M vectors of 768 dimensions. However, capacity may vary with dimensionality.

The data ingestion rate for p2 pods is significantly slower than for p1 pods; this rate decreases as the number of dimensions increases. For example, a p2 pod containing vectors with 128 dimensions can upsert up to 300 updates per second; a p2 pod containing vectors with 768 dimensions or more supports upsert of 50 updates per second. Because query latency and throughput for p2 pods vary from p1 pods, test p2 pod performance with your dataset.

The p2 pod type does not support sparse vector values.

Pod size and performance

Pod performance varies depending on a variety of factors. To observe how your workloads perform on a given pod type, experiment with your own data set.

Each pod type supports four pod sizes: x1, x2, x4, and x8. Your index storage and compute capacity doubles for each size step. The default pod size is x1. You can increase the size of a pod after index creation.

To learn about changing the pod size of an index, see Manage indexes.

Distance metrics

You can choose from different metrics when creating a vector index:

- euclidean
    - This is used to calculate the distance between two data points in a plane. It is one of the most commonly used distance metric. For an example, see our image similarity search example.
    - When you use metric='euclidean', the most similar results are those with the lowest score.
- cosine
    - This is often used to find similarities between different documents. The advantage is that the scores are normalized to [-1,1] range.
- dotproduct
    - This is used to multiply two vectors. You can use it to tell us how similar the two vectors are. The more positive the answer is, the closer the two vectors are in terms of their directions.

For the full list of parameters available to customize an index, see the create_index API reference.

Depending on your application, some metrics have better recall and precision performance than others. For more information, see: What is Vector Similarity Search?

Concepts - Sparse-dense embeddings

Overview

Pinecone supports vectors with sparse and dense values, which allows you to perform semantic and keyword search over your data in one query and combine the results for more relevant results. This topic describes how sparse-dense vectors work in Pinecone.

To see sparse-dense embeddings in action, see the Ecommerce hybrid search example.

Pinecone sparse-dense vectors allow keyword-aware semantic search

Pinecone sparse-dense vectors allows you to perform keyword-aware semantic search. Semantic
search results for out-of-domain queries can be less relevant; combining these
with keyword search results can improve relevance.
Because Pinecone allows you to create your own sparse vectors, you can use sparse-dense queries to solve the Maximum Inner Product Search (MIPS) problem for sparse-dense vectors of any real values. This includes emerging use-cases such as retrieval over learnt sparse representations for text data using SPLADE.

Sparse-dense workflow

Using sparse-dense vectors involves the following general steps:

1. Create dense vectors using an external embedding model.
2. Create sparse vectors using an external model.
3. Create an index that supports sparse-dense vectors (s1 or p1 with the dotproductmetric).
4. Upsert dense and sparse vectors to your index.
5. Search the index using sparse-dense vectors.
6. Pinecone returns sparse-dense vectors.

Sparse versus dense vectors in Pinecone

Pinecone supports dense and sparse embeddingsas a single vector. These types of embeddings represent different types of information and enable distinct kinds of search. Dense vectorsenable semantic search. Semantic search returns the most similar results according to a specific distance metric even if no exact matches are present. This is possible because dense vectors generated by embedding models such as SBERTare numerical representations of semantic meaning.

Sparse vectors have very large number of dimensions, where only a small proportion of values are non-zero. When used for keywords search, each sparse vector represents a document; the dimensions represent words from a dictionary, and the values represent the importance of these words in the document. Keyword search algorithms like the BM25algorithm compute the relevance of text documents based on the number of keyword matches, their frequency, and other factors.

Creating sparse vector embeddings

Keyword-aware semantic search requires vector representations of documents. Because Pinecone indexes accept sparse indexes rather than documents, you can control the generation of sparse vectors to represent documents.

For examples of sparse vector generation, see SPLADE for Sparse Vector Search Explained, our SPLADE generation notebook, and our BM25 generation notebook.

ℹ️Note

Pinecone supports sparse vector values of sizes up to 1000 non-zero values.

Pinecone creates sparse-dense vectors from your sparse and dense embeddings

In Pinecone, each vector consists of dense vector values and, optionally, sparse vector values as well. Pinecone does not support vectors with only sparse values.

p1 and s1 indexes using the dotproductmetric support sparse-dense vectors

Pinecone stores sparse-dense vectors in p1 and s1 indexes. In order to query an index using sparse values, the index must use the dotproductmetric. Attempting to query any other index with sparse values returns an error.

Indexes created before February 22, 2023 do not support sparse values.

Sparse-dense queries include sparse and dense vector values

To query your sparse-dense vectors, you provide a query vector containing both sparse and dense values. Pinecone ranks vectors in your index by considering the full dot product over the entire vector; the score of a vector is the sum of the dot product of its dense values with the dense part of the query, together with the dot product of its sparse values with the sparse part of the query.

Sparse-dense vector format

Pinecone represents sparse values as a dictionary of two arrays: indicesand values. You can upsert these values inside a vector parameter to upsert a sparse-dense vector.

Example

The following example upserts two vectors with sparse and dense values.

Python

```
index = pinecone.Index('example-index')

upsert_response = index.upsert(
vectors=[
{'id': 'vec1',
'values': [0.1, 0.2, 0.3],
'metadata': {'genre': 'drama'},
'sparse_values': {
'indices': [10, 45, 16],
'values': [0.5, 0.5, 0.2]
}},
{'id': 'vec2',
'values': [0.2, 0.3, 0.4],
'metadata': {'genre': 'action'},
'sparse_values': {
'indices': [15, 40, 11],
'values': [0.4, 0.5, 0.2]
}}
],
namespace='example-namespace'
```

)

The following example queries an index using a sparse-dense vector.

Python
query_response = index.query(
namespace="example-namespace",
top_k=10,
vector=[0.1, 0.2, 0.3],
sparse_vector={
'indices': [10, 45, 16],
'values': [0.5, 0.5, 0.2]
}
)

Sparse-dense queries do not support explicit weighting

Because Pinecone's index views your sparse-dense vector as a single vector, it does not offer a built-in parameter to adjust the weight of a query's dense part against its sparse part; the index is agnostic to density or sparsity of coordinates in your vectors. You may, however, incorporate a linear weighting scheme by customizing your query vector, as we demonstrate in the function below.

Examples

The following example transforms vector values using an alpha parameter.

Python
def hybrid_score_norm(dense, sparse, alpha: float):
"""Hybrid score using a convex combination

alpha * dense + (1 - alpha) * sparse

Args:
dense: Array of floats representing
sparse: a dict of `indices` and `values`
alpha: scale between 0 and 1
"""
if alpha < 0 or alpha > 1:
raise ValueError("Alpha must be between 0 and 1")
hs = {
'indices': sparse['indices'],
'values': [v * (1 - alpha) for v in sparse['values']]
}

```
return [v * alpha for v in dense], hs
```

The following example transforms a vector using the above function, then queries a Pinecone index.

```
Python
sparse_vector = {
'indices': [10, 45, 16],
'values': [0.5, 0.5, 0.2]
}
dense_vector = [0.1, 0.2, 0.3]

hdense, hsparse = hybrid_score_norm(dense_vector, sparse_vector, alpha=0.75)

query_response = index.query(
namespace="example-namespace",
top_k=10,
vector=hdense,
sparse_vector=hsparse
)
```

Vector Search - Use Examples

These examples demonstrate how you might build vector search into your applications with Pinecone. You can view their source code to jumpstart your own application.

Our Learn section explains the basics of vector search and vector databases.

Example	Data	Model(s)	Stack
Semantic SearchHow to perform a simple semantic search.	text	all-MiniLM-L6-v2	PythonSentence Transformers
Generative QA with OpenAIBuild retrieval enhanced generative QA systems with OpenAI.	text	text-davinci-003text-embedding-ada-002	PythonOpenAI
Chatbot Agents with LangChainHow to create conversational agents with LangChain and Pinecone.	text	gpt-3.5-turbotext-embedding-ada-002	PythonOpenAILangChain

Extractive Question AnsweringBuild an extractive QA application with similarity search.	text	SQuAD	PythonSentence Transformers
Abstractive Question AnsweringBuild an abstractive (generative) QA application with similarity search.	text	MPNetBART	Python
Tabular Question AnsweringExtract answers from tables through similarity search.	text	MPNettapas-base-finetuned-wtq	PythonSentence Transformers
Basic Keyword Filter Hybrid SearchHow to pair semantic search with a basic keyword filter.	text	MPNet	PythonSentence Transformers
Advanced Hybrid E-Commerce SearchUse Pinecone's advanced sparse-dense index for hybrid e-commerce search.	textimages	CLIPBM25	PythonSentence Transformers
NER-Powered Semantic SearchAutomatically extract entities from text and use them to improve serach results.	text	MPNetbert-base-NER	Python
Video Transcription SearchCreate an app that searches video transcription data.	text	MPNet	PythonSentence Transformers
GIF Description SearchCreate an app that helps users to search through GIFs.	text	BERTall-MiniLM-L6-v2	PythonSentence Transformers
Image Similarity SearchHow to build advanced image search applications.	images	SqueezeNet	Python
Facial Similarity SearchFind your celebrity doppelganger with facial similarity search.	images	MTCNN	PythonPyTorch
Audio Similarity SearchBuild advanced audio search applications.	audio	PANNs	Python
Personalized Content RecommendationsUse Pinecone to create a simple personalized article or content recommender.	text	AWE-komninos	Python

Movie RecommenderCreate a movie recommendation system with the MovieLens dataset.	text	Custom	PythonTensorFlow
Document DeduplicationCreate a simple application for identifying duplicate documents.	text	AWE	Python
IT Threat DetectionBuild an application for detecting rare events in IT threat detection.	text	Custom	Python
Extreme ClassificationLabel new texts automatically given an enormous number of potential labels.	text	AWE-komninos	Python
Time Series SearchLabel new texts automatically given an enormous number of potential labels.	series	Kats	Python
Satellite Image SearchHow to perform a similarity search across satellite imagery.	image	clip-rsicd-v2	PythonTransformers

---

Collections - Back up indexes

Overview

This document describes how to make backup copies of your indexes using collections.

To learn how to create an index from a collection, see Manage indexes.

⚠️Warning

This document uses collections. This is a public preview feature. Test thoroughly before using this feature with production workloads.

Create a backup using a collection

To create a backup of your index, use the create_collectionoperation. A collection is a static copy of your index that only consumes storage.

Example

The following example creates a collection named example-collectionfrom an index named example-index.

PythonJavaScriptcurl
pinecone.create_collection("example-collection", "example-index")

Check the status of a collection

To retrieve the status of the process creating a collection and the size of the collection, use the describe_collectionoperation. Specify the name of the collection to check. You can only call describe_collectionon a collection in the current project.

The describe_collectionoperation returns an object containing key-value pairs representing the name of the collection, the size in bytes, and the creation status of the collection.

Example

The following example gets the creation status and size of a collection named example-collection.

PythonJavaScriptcurl
pinecone.describe_collection("example-collection")
Results:

Shell
CollectionDescription(name='test-collection', size=3818809, status='Ready')

List your collections

To get a list of the collections in the current project, use the list_collectionsoperation.

Example

The following example gets a list of all collections in the current project.

PythonJavaScriptcurl
pinecone.list_collections()
Results

Shell
example-collection

Delete a collection

To delete a collection, use the delete_collectionoperation. Specify the name of the collection to delete.

Deleting the collection takes several minutes. During this time, the describe_collectionoperation returns the status "deleting".

Example

The following example deletes the collection example-collection.

PythonJavaScriptcurl
pinecone.delete_collection("example-collection")

Collections

This document explains the concepts related to collections in Pinecone.

⚠️Warning

This is a public preview("Beta") feature. Test thoroughly before using this feature for production workloads. No SLAs or technical support commitments are provided for this feature.
A collection is a static copy of an index.It is a non-queryable representation of a set of vectors and metadata. You can create a collection from an index, and you can create a new index from a collection. This new index can differ from the original source index: the new index can have a different number of pods, a different pod type, or a different similarity metric.

Use cases for collections

Creating a collection from your index is useful when performing tasks like the following:

- Temporarily shutting down an index
- Copying the data from one index into a different index;
- Making a backup of your index
- Experimenting with different index configurations

To learn about creating backups with collections, see Back up indexes.

To learn about creating indexes from collections, see Manage indexes.

Performance

Collections operations perform differently with different pod types.

- Creating a collection from an index takes approximately 10 minutes.
- Creating a p1 or s1 index from a collection takes approximately 10 minutes.
- Creating a p2 index from a collection can take several hours.

Limitations

You cannot query or write to a collection after its creation. For this reason, a collection only incurs storage costs.

You can only perform operations on collections in the current Pinecone project.

Node.JS Client - Github README.md

README.md from https://github.com/pinecone-io/pinecone-ts-client

README.md

Pinecone Node.js Client

This is the Node.js client for Pinecone, written in Typescript. It is a wrapper around the Pinecone OpenAPI spec.

https://github.com/pinecone-io/pinecone-ts-client/actions/workflows/PR.yml
https://github.com/pinecone-io/pinecone-ts-client/graphs/commit-activity
https://npmjs.com/package/@pinecone-database/pinecone
⚠️ *Warning*

This is a public preview("Beta") client. Test thoroughly before using this client for production workloads. No SLAs or technical support commitments are provided for this client. Expect potential breaking changes in future releases.

Installation

npm i @pinecone-database/pinecone

Usage

Set the following environment variables:

PINECONE_API_KEY=your_api_key
PINECONE_ENVIRONMENT=your_environment

Initializing the client

```
import { PineconeClient } from "@pinecone-database/pinecone";

// Create a client
const client = new PineconeClient();

// Initialize the client
await client.init({
apiKey: process.env.PINECONE_API_KEY,
environment: process.env.PINECONE_ENVIRONMENT,
});
```

Control plane operations

The Pinecone control plane allows you to perform the following operations:

1. Create, configure and delete indexes
2. Get information about an existing indexes
3. Create and delete collections
4. Select an index to operate on

Indexes

Create Index

```
const createRequest: CreateRequest = {
```

```
name: indexName,
dimension: dimensions,
metric,
};

await client.createIndex({ createRequest });
```

Delete Index

```
await client.deleteIndex({ indexName });
```

Describe Index

```
const indexDescription = await client.describeIndex({ indexName });
```

Example result:


```
{
"database": {
"name": "my-index",
"metric": "cosine",
"dimension": 10,
"replicas": 1,
"shards": 1,
"pods": 1,
"pod_type": "p1.x1"
},
"status": {
"waiting": [],
"crashed": [],
"host": "my-index-[project-id].svc.[environment].pinecone.io",
"port": 433,
"state": "Ready",
"ready": true
}
}
```


List Indexes

```
const list = await client.listIndexes();
```

Example result:


```
["index1", "index2"]
```


Select an index

To operate on an index, you must select it. This is done by calling the Indexmethod on the client.

const index = client.Index(indexName);

Collections

Create Collection

const createCollectionRequest: CreateCollectionRequest = {
name: collection,
source: indexName,
};
await client.createCollection({ createCollectionRequest });

Delete Collection

await client.deleteCollection(collection);

Describe Collection

const describeCollection = await client.describeCollection({ collectionName });
Example result:

{
"name": "my-collection",
"status": "Ready",
"size": 3059815,
"dimension": 10
}

List Collections

const list = await client.listCollections();
Example result:

["collection1", "collection2"]

Index operations

The Pinecone index operations allow you to perform the following operations instances of Vector.

A Vectoris defined as follows:

type Vector = {
id: string;
values: number[];
metadata?: object;

```
sparseValues: {
indices: [15, 30, 11];
values: [0.1, 0.2, 0.3];
}; // optional sparse values
};
```

After selecting an index to operate on, you can:

Upsert vectors

```
const upsertRequest: UpsertRequest = {
vectors,
namespace,
};
await index.upsert({ upsertRequest });
```

Query vectors

```
const vector = [...] // a vector

const queryRequest: QueryRequest = {
topK: 1,
vector,
namespace,
includeMetadata: true,
includeValues: true,
}
```

To query with a sparse vector:

```
const queryRequest: QueryRequest = {
topK: 1,
vector,
namespace,
includeMetadata: true,
includeValues: true,
sparseVector: {
indices: [15, 30, 11],
values: [0.1, 0.2, 0.3],
},
};
```

To execute the query:

```
const queryResponse = await index.query({ queryRequest });
```

Update a vector

```
const updateRequest: UpdateRequest = {
id: vectorId, // the ID of the vector to update
values: [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0], // the new vector values
sparseValues: {
indices: [15, 30, 11],
values: [0.1, 0.2, 0.3],
}, // optional sparse values
setMetadata: metadata, // the new metadata
namespace,
};
await index.update({ updateRequest });
```

Fetch vectors by their IDs

```
const fetchResult = await index.fetch({
ids: [vectorIDs],
namespace,
});
```

Delete vectors

```
await index.delete1({
ids: [vectorIDs],
namespace,
});
```

Delete all vectors in a namespace

```
await index.delete1({
deleteAll: true,
namespace,
});
```

Python Client - Github README.md

README.md from https://github.com/pinecone-io/pinecone-python-client

pinecone-client

The Pinecone python client

For more information, see the docs at https://www.pinecone.io/docs/

Installation

Install a released version from pip:

pip3 install pinecone-client
Or the gRPC version of the client for tuning performance

pip3 install "pinecone-client[grpc]"
Or the latest development version:

pip3 install git+https://git@github.com/pinecone-io/pinecone-python-client.git
Or a specific development version:

pip3 install git+https://git@github.com/pinecone-io/pinecone-python-client.git
pip3 install
git+https://git@github.com/pinecone-io/pinecone-python-client.git@example-branch-name
pip3 install git+https://git@github.com/pinecone-io/pinecone-python-client.git@259deff

Creating an index

The following example creates an index without a metadata configuration. By default, Pinecone indexes all metadata.

import pinecone

pinecone.init(api_key="YOUR_API_KEY",
environment="us-west1-gcp")

pinecone.create_index("example-index", dimension=1024)
The following example creates an index that only indexes the "color" metadata field. Queries against this index cannot filter based on any other metadata field.

metadata_config = {
"indexed": ["color"]
}

pinecone.create_index("example-index-2", dimension=1024,
metadata_config=metadata_config)

List indexes

The following example returns all indexes in your project.

```
import pinecone

pinecone.init(api_key="YOUR_API_KEY", environment="us-west1-gcp")

active_indexes = pinecone.list_indexes()
```

Describe index

The following example returns information about the index example-index.

```
import pinecone

pinecone.init(api_key="YOUR_API_KEY", environment="us-west1-gcp")

index_description = pinecone.describe_index("example-index")
```

Delete an index

The following example deletes example-index.

```
import pinecone

pinecone.init(api_key="YOUR_API_KEY", environment="us-west1-gcp")

pinecone.delete_index("example-index")
```

Scale replicas

The following example changes the number of replicas for example-index.

```
import pinecone

pinecone.init(api_key="YOUR_API_KEY", environment="us-west1-gcp")

new_number_of_replicas = 4
pinecone.configure_index("example-index", replicas=new_number_of_replicas)
```

Describe index statistics

The following example returns statistics about the index example-index.

```
import pinecone

pinecone.init(api_key="YOUR_API_KEY", environment="us-west1-gcp")
index = pinecone.Index("example-index")

index_stats_response = index.describe_index_stats()
```

Upsert vectors

The following example upserts vectors to example-index.

```
import pinecone

pinecone.init(api_key="YOUR_API_KEY", environment="us-west1-gcp")
index = pinecone.Index("example-index")

upsert_response = index.upsert(
vectors=[
("vec1", [0.1, 0.2, 0.3, 0.4], {"genre": "drama"}),
("vec2", [0.2, 0.3, 0.4, 0.5], {"genre": "action"}),
],
namespace="example-namespace"
)
```

Query an index

The following example queries the index example-indexwith metadata filtering.

```
import pinecone

pinecone.init(api_key="YOUR_API_KEY", environment="us-west1-gcp")
index = pinecone.Index("example-index")

query_response = index.query(
namespace="example-namespace",
top_k=10,
include_values=True,
include_metadata=True,
vector=[0.1, 0.2, 0.3, 0.4],
filter={
"genre": {"$in": ["comedy", "documentary", "drama"]}
}
)
```

Delete vectors

The following example deletes vectors by ID.

```
import pinecone

pinecone.init(api_key="YOUR_API_KEY", environment="us-west1-gcp")
index = pinecone.Index("example-index")

delete_response = index.delete(ids=["vec1", "vec2"], namespace="example-namespace")
```

Fetch vectors

The following example fetches vectors by ID.

```
import pinecone

pinecone.init(api_key="YOUR_API_KEY", environment="us-west1-gcp")
index = pinecone.Index("example-index")

fetch_response = index.fetch(ids=["vec1", "vec2"], namespace="example-namespace")
```

Update vectors

The following example updates vectors by ID.

```
import pinecone

pinecone.init(api_key="YOUR_API_KEY", environment="us-west1-gcp")
index = pinecone.Index("example-index")

update_response = index.update(
id="vec1",
values=[0.1, 0.2, 0.3, 0.4],
set_metadata={"genre": "drama"},
namespace="example-namespace"
)
```

Create collection

The following example creates the collection example-collectionfrom example-index.

```
import pinecone
```

```
pinecone.init(api_key="YOUR_API_KEY",
environment="us-west1-gcp")

pinecone.create_collection("example-collection", "example-index")
```

List collections

The following example returns a list of the collections in the current project.

```
import pinecone

pinecone.init(api_key="YOUR_API_KEY", environment="us-west1-gcp")

active_collections = pinecone.list_collections()
```

Describe a collection

The following example returns a description of the collection example-collection.

```
import pinecone

pinecone.init(api_key="YOUR_API_KEY", environment="us-west1-gcp")

collection_description = pinecone.describe_collection("example-collection")
```

Delete a collection

The following example deletes the collection example-collection.

```
import pinecone

pinecone.init(api_key="YOUR_API_KEY", environment="us-west1-gcp")

pinecone.delete_collection("example-collection")
```

Workflow - Insert data

After creating a Pinecone index, you can start inserting vector embeddings and metadata into the index.

Inserting the vectors

1. Connect to the index:

Pythoncurl
index = pinecone.Index("pinecone-index")

2. Insert the data as a list of (id, vector)tuples. Use the Upsertoperation to write vectors into a namespace:

PythonJavaScriptcurl
Insert sample data (5 8-dimensional vectors)
index.upsert([
("A", [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]),
("B", [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]),
("C", [0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3]),
("D", [0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4]),
("E", [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5])
])
Immediately after the upsert response is received, vectors may not be visible to queries yet. In most situations, you can check if the vectors have been received by checking for the vector counts returned by describe_index_stats()to be updated. This technique may not work if the index has multiple replicas. The database is eventually consistent.

Batching upserts

For clients upserting larger amounts of data, you should insert data into an index in batches of 100 vectors or fewer over multiple upsert requests.

Example

Python
import random
import itertools

def chunks(iterable, batch_size=100):
"""A helper function to break an iterable into chunks of size batch_size."""
it = iter(iterable)
chunk = tuple(itertools.islice(it, batch_size))
while chunk:
yield chunk
chunk = tuple(itertools.islice(it, batch_size))

```
vector_dim = 128
vector_count = 10000

# Example generator that generates many (id, vector) pairs
example_data_generator = map(lambda i: (f'id-{i}', [random.random() for _ in
range(vector_dim)]), range(vector_count))

# Upsert data with 100 vectors per upsert request
for ids_vectors_chunk in chunks(example_data_generator, batch_size=100):
index.upsert(vectors=ids_vectors_chunk) # Assuming `index` defined elsewhere
```

Sending upserts in parallel

By default, all vector operations block until the response has been received. But using our client they can be made asynchronous. For the Batching Upsertsexample this can be done as follows:

```
PythonShell
# Upsert data with 100 vectors per upsert request asynchronously
# - Create pinecone.Index with pool_threads=30 (limits to 30 simultaneous requests)
# - Pass async_req=True to index.upsert()
with pinecone.Index('example-index', pool_threads=30) as index:
# Send requests in parallel
async_results = [
index.upsert(vectors=ids_vectors_chunk, async_req=True)
for ids_vectors_chunk in chunks(example_data_generator, batch_size=100)
]
# Wait for and retrieve responses (this raises in case of error)
[async_result.get() for async_result in async_results]
```

Pinecone is thread-safe, so you can launch multiple read requests and multiple write requests in parallel. Launching multiple requests can help with improving your throughput. However, reads and writes can't be performed in parallel, therefore writing in large batches might affect query latency and vice versa.

If you experience slow uploads, see Performance tuningfor advice.

Partitioning an index into namespaces

You can organize the vectors added to an index into partitions, or "namespaces," to limit queries and other vector operations to only one such namespace at a time. For more information, see: Namespaces.

Inserting vectors with metadata

You can insert vectors that contain metadata as key-value pairs.

You can then use the metadata to filter for those criteria when sending the query. Pinecone will search for similar vector embeddings only among those items that match the filter. For more information, see: Metadata Filtering.

PythonJavaScriptcurl
```
index.upsert([
("A", [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1], {"genre": "comedy", "year": 2020}),
("B", [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2], {"genre": "documentary", "year": 2019}),
("C", [0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3], {"genre": "comedy", "year": 2019}),
("D", [0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4], {"genre": "drama"}),
("E", [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5], {"genre": "drama"})
])
```

Upserting vectors with sparse values

Sparse vector valuescan be upserted alongside dense vector values.

You cannot upsert sparse vector values without a dense vector values.

⚠️Warning

Only s1and p1pod types using the dotproductmetric support querying sparse vectors. There is no error at upsert time: if you attempt to query any other pod type using sparse vectors, Pinecone returns an error.
Indexes created before February 22, 2023 do not support sparse values.

Pythoncurl
```
index = pinecone.Index('example-index')

upsert_response = index.upsert(
vectors=[
{'id': 'vec1',
'values': [0.1, 0.2, 0.3, 0.4],
'metadata': {'genre': 'drama'},
'sparse_values': {
'indices': [10, 45, 16],
'values': [0.5, 0.5, 0.2]
}},
{'id': 'vec2',
'values': [0.2, 0.3, 0.4, 0.5],
```

'metadata': {'genre': 'action'},
'sparse_values': {
'indices': [15, 40, 11],
'values': [0.4, 0.5, 0.2]
}}
],
namespace='example-namespace'
)

Troubleshooting index fullness errors

When upserting data, you may receive the following error:
console
Index is full, cannot accept data.
New upserts may fail as the capacity becomes exhausted. While your index can still serve queries, you need to scale your environment to accommodate more vectors.

To resolve this issue, you can scale your index.

Workflow - Metadata filtering

You can limit your vector search based on metadata. Pinecone lets you attach metadata key-value pairs to vectors in an index, and specify filter expressions when you query the index.

Searches with metadata filters retrieve exactly the number of nearest-neighbor results that match the filters. For most cases, the search latency will be even lower than unfiltered searches.

For more background information on metadata filtering, see: The Missing WHERE Clause in Vector Search.

Supported metadata types

You can associate a metadata payload with each vector in an index, as key-value pairs in a JSON object where keys are strings and values are one of:

- String
- Number (integer or floating point, gets converted to a 64 bit floating point)
- Booleans (true, false)
- List of String

ℹ️Note

High cardinality consumes more memory:Pinecone indexes metadata to allow for filtering. If the metadata contains many unique values — such as a unique identifier for each vector — the index will consume significantly more memory. Consider using selective metadata indexingto avoid indexing high-cardinality metadata that is not needed for filtering.

⚠️Warning

Null metadata values are not supported. Instead of setting a key to hold a null value, we recommend you remove that key from the metadata payload. For example, the following would be valid metadata payloads:

JSON
{
"genre": "action",
"year": 2020,
"length_hrs": 1.5
}

{
"color": "blue",
"fit": "straight",
"price": 29.99,
"is_jeans": true
}

Supported metadata size

Pinecone supports 40kb of metadata per vector.

Metadata query language

ℹ️Note

Pinecone's filtering query language is based on MongoDB's query and projection operators. We currently support a subset of those selectors.
The metadata filters can be combined with AND and OR:

- $eq- Equal to *(number, string, boolean)*
- $ne- Not equal to *(number, string, boolean)*
- $gt- Greater than *(number)*

- $gte- Greater than or equal to *(number)*
- $lt- Less than *(number)*
- $lte- Less than or equal to *(number)*
- $in- In array *(string or number)*
- $nin- Not in array *(string or number)*

Using arrays of strings as metadata values or as metadata filters

A vector with metadata payload...

JSON
{ "genre": ["comedy", "documentary"] }
...means the "genre"takes on both values.

For example, queries with the following filters will match the vector:

JSON
{"genre":"comedy"}

{"genre": {"$in":["documentary","action"]}}

{"$and": [{"genre": "comedy"}, {"genre":"documentary"}]}
Queries with the following filter will notmatch the vector:

JSON
{ "$and": [{ "genre": "comedy" }, { "genre": "drama" }] }
And queries with the following filters will notmatch the vector because they are invalid. They will result in a query compilation error:

INVALID QUERY:
{"genre": ["comedy", "documentary"]}

INVALID QUERY:
{"genre": {"$eq": ["comedy", "documentary"]}}

Inserting metadata into an index

Metadata can be included in upsert requests as you insert your vectors.

For example, here's how to insert vectors with metadata representing movies into an index:

PythonJavaScriptcurl
import pinecone

```
pinecone.init(api_key="your-api-key", environment="us-west1-gcp")
index = pinecone.Index("example-index")

index.upsert([
("A", [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1], {"genre": "comedy", "year": 2020}),
("B", [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2], {"genre": "documentary", "year": 2019}),
("C", [0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3], {"genre": "comedy", "year": 2019}),
("D", [0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4], {"genre": "drama"}),
("E", [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5], {"genre": "drama"})
])
```

Querying an index with metadata filters

Metadata filter expressions can be included with queries to limit the search to only vectors matching the filter expression.

For example, we can search the previous movies index for documentaries from the year 2019. This also uses the include_metadataflag so that vector metadata is included in the response.

⚠️Warning

For performance reasons, do not return vector data and metadata when top_k>1000. Queries with top_kover 1000 should not contain include_metadata=Trueor include_data=True.

PythonJavaScript

```
index.query(
vector=[0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1],
filter={
"genre": {"$eq": "documentary"},
"year": 2019
},
top_k=1,
include_metadata=True
)

# Returns:
# {'matches': [{'id': 'B',
# 'metadata': {'genre': 'documentary', 'year': 2019.0},
# 'score': 0.0800000429,
# 'values': []}],
# 'namespace': ''}
```

curl

```
curl -i -X POST
https://YOUR_INDEX-YOUR_PROJECT.svc.YOUR_ENVIRONMENT.pinecone.io/query \
-H 'Api-Key: YOUR_API_KEY' \
-H 'Content-Type: application/json' \
-d '{
"vector": [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1],
"filter": {"genre": {"$in": ["comedy", "documentary", "drama"]}},
"topK": 1,
"includeMetadata": true
}'

# Output:
# {
# "matches": [
# {
# "id": "B",
# "score": 0.0800000429,
# "values": [],
# "metadata": {
# "genre": "documentary",
# "year": 2019
# }
# }
# ],
# "namespace": ""
# }
```

More example filter expressions

A comedy, documentary, or drama:

JSON
```
{
"genre": { "$in": ["comedy", "documentary", "drama"] }
}
```
A drama from 2020:

JSON
```
{
"genre": { "$eq": "drama" },
"year": { "$gte": 2020 }
}
```
A drama from 2020 (equivalent to the previous example):

JSON

```
{
"$and": [{ "genre": { "$eq": "drama" } }, { "year": { "$gte": 2020 } }]
}
```

A drama or a movie from 2020:

JSON

```
{
"$or": [{ "genre": { "$eq": "drama" } }, { "year": { "$gte": 2020 } }]
}
```

Deleting vectors by metadata filter

To specify vectors to be deleted by metadata values, pass a metadata filter expression to the delete operation. This deletes all vectors matching the metadata filter expression.

Example

This example deletes all vectors with genre "documentary" and year 2019 from an index.

PythonJavaScriptcurl

```
index.delete(
filter={
"genre": {"$eq": "documentary"},
"year": 2019
}
)
```

Limits

This is a summary of current Pinecone limitations. For many of these, there is a workaround or we're working on increasing the limits.

Upserts

Max vector dimensionality is 20,000.

Max size for an upsert request is 2MB. Recommended upsert limit is 100 vectors per request.

Vectors may not be visible to queries immediately after upserting. You can check if the vectors were indexed by looking at the total with describe_index_stats(), although this method may not work if the index has multiple replicas. The database is eventually consistent.

Pinecone supports sparse vector values of sizes up to 1000 non-zero values.

Queries

Max value for top_k, the number of results to return, is 10,000. Max value for top_kfor queries with include_metadata=Trueor include_data=Trueis 1,000.

Fetch and Delete

Max vectors per fetch or delete request is 1,000.

Namespaces

There is no limit to the number of namespacesper index.

Pod storage capacity

Each p1pod has enough capacity for 1M vectors with 768 dimensions.

Each s1pod has enough capacity for 5M vectors with 768 dimensions.

Metadata

Max metadata size per vector is 40 KB.

Null metadata values are not supported. Instead of setting a key to hold a null value, we recommend you remove that key from the metadata payload.

Metadata with high cardinality, such as a unique value for every vector in a large index, uses more memory than expected and can cause the pods to become full.

Retention

In general, indexes on the Starter (free) plan are archived as collections and deleted after 7 days of inactivity; for indexes created by certain open source projects such as AutoGPT, indexes are archived and deleted after 1 day of inactivity. To prevent this, you can send any API request to Pinecone and the counter will reset.

Workflow - Query data

After your data is indexed, you can start sending queries to Pinecone.

The Queryoperation searches the index using a query vector. It retrieves the IDs of the most similar vectors in the index, along with their similarity scores. It can optionally include the result vectors' values and metadata too. You specify the number of vectors to retrieve each time you send a query. They are always ordered by similarity from most similar to least similar.

Sending a query

When you send a query, you provide a vectorand retrieve the top-kmost similar vectors for each query. For example, this example sends a query vector and retrieves three matching vectors:

PythonJavaScriptcurl

```
index.query(
vector=[0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3],
top_k=3,
include_values=True
)

# Returns:
# {'matches': [{'id': 'C',
# 'score': -1.76717265e-07,
# 'values': [0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3]},
# {'id': 'B',
# 'score': 0.080000028,
# 'values': [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]},
# {'id': 'D',
# 'score': 0.0800001323,
# 'values': [0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4]}],
# 'namespace': ''}
```

Depending on your data and your query, you may not get top_k results. This happens when top_k is larger than the number of possible matching vectors for your query.

Querying by namespace

You can organize the vectors added to an index into partitions, or "namespaces," to limit queries and other vector operations to only one such namespace at a time. For more information, see: Namespaces.

Using metadata filters in queries

You can add metadata to document embeddings within Pinecone, and then filter for those criteria when sending the query. Pinecone will search for similar vector embeddings only among those items that match the filter. For more information, see: Metadata Filtering.

PythonJavaScriptcurl
index.query(
vector=[0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1],
filter={
"genre": {"$eq": "documentary"},
"year": 2019
},
top_k=1,
include_metadata=True
)

Querying vectors with sparse and dense values

When querying an index containing sparse and dense vectors, use the query()operation with the sparse_vectorparameter present.

⚠️Warning

The Updateoperation does not validate the existence of ids within an index. If a non-existent id is given then no changes are made and a 200 OK will be returned.
Examples

The following example queries the index example-index with a sparse-dense vector.

Pythoncurl
query_response = index.query(
namespace="example-namespace",
top_k=10,
vector=[0.1, 0.2, 0.3, 0.4],
sparse_vector={

'indices': [10, 45, 16],
'values': [0.5, 0.5, 0.2]
}
)

Limitations

Avoid returning vector data and metadata when top_kis greater than 1000. This means queries with top_kover 1000 should not contain include_metadata=Trueor include_data=True. For more limitations, see: Limits.

Workflow - Manage data

In addition to insertingand queryingdata, there are other ways you can interact with vector data in a Pinecone index. This section walks through the various vector operations available.

Connect to an index

If you're using a Pinecone client library to access an index, you'll need to open a session with the index:

PythonJavaScriptcurl
# Connect to the index
index = pinecone.Index("pinecone-index")

Specify an index endpoint

Pinecone indexes each have their own DNS endpoint. For cURL and other direct API calls to a Pinecone index, you'll need to know the dedicated endpoint for your index.
Index endpoints take the following form:

https://{index-name}-{project-name}.svc.YOUR_ENVIRONMENT.pinecone.io

- {index-name}is the name you gave your index when you created it.
- {project-name}is the Pinecone project name that your API key is associated with. This can be retrieved using the whoamioperation below.
- YOUR_ENVIRONMENTis the cloud region for your Pinecone project..

Call whoamito retrieve your project name.

The following command retrieves your Pinecone project name.

Pythoncurl
pinecone.whoami()

Describe index statistics

Get statistics about an index, such as vector count per namespace:

PythonJavaScriptcurl
index.describe_index_stats()

Fetching vectors

The Fetchoperation looks up and returns vectors, by id, from an index. The returned vectors include the vector data and/or metadata. Typical fetch latency is under 5ms.

Fetch items by their ids:

PythonJavaScriptcurl
index.fetch(["id-1", "id-2"])

Returns:
{'namespace': '',
'vectors': {'id-1': {'id': 'id-1',
'values': [0.568879, 0.632687092, 0.856837332, ...]},
'id-2': {'id': 'id-2',
'values': [0.00891787093, 0.581895, 0.315718859, ...]}}}

Updating vectors

There are two methods for updating vectors and metadata, using *full*or *partial*updates.

Full update

Full updates modify the entire item, that is vectors and metadata. Updating an item by id is done the same way as inserting items. (Write operations in Pinecone are idempotent.)

The Upsertoperation writes vectors into an index.

ℹ️Note

If a new value is upserted for an existing vector id, it will overwrite the

previous value.

1. Update the value of the item ("id-3", [3.3, 3.3]):

PythonJavaScriptcurl
index.upsert([("id-3", [3.3, 3.3])])

2. Fetch the item again. We should get ("id-3", [3.3, 3.3]):

PythonJavaScriptcurl
index.fetch(["id-3"])

Partial update

The Updateoperation performs partial updates that allow changes to *part*of an item. Given an id, we can update the vector value with the valuesargument or update metadata with the set_metadataargument.

⚠️Warning

The Updateoperation does not validate the existence of ids within an index. If a non-existent id is given then no changes are made and a 200 OK will be returned.
To update the value of item ("id-3", [3., 3.], {"type": "doc", "genre": "drama"}):

PythonJavaScriptcurl
index.update(id="id-3", values=[4., 2.])
The updated item would now be ("id-3", [4., 2.], {"type": "doc", "genre": "drama"}).

When updating metadata only specified fields will be modified. If a specified field does not exist, it is added.

ℹ️Note

Metadata updates apply *only*to fields passed to the set_metadata argument. Any other fields will remain unchanged.
To update the metadata of item ("id-3", [4., 2.], {"type": "doc", "genre": "drama"}):

PythonJavaScriptcurl
index.update(id="id-3", set_metadata={"type": "web", "new": "true"})
The updated item would now be ("id-3", [4., 2.], {"type": "web", "genre": "drama", "new": "true"}).

Both vector and metadata can be updated at once by including both valuesand set_metadataarguments. To update the "id-3"item we write:

PythonJavaScriptcurl
index.update(id="id-3", values=[1., 2.], set_metadata={"type": "webdoc"})
The updated item would now be ("id-3", [1., 2.], {"type": "webdoc", "genre": "drama", "new": "true"}).

Deleting vectors

The Deleteoperation deletes vectors, by ID, from an index.

Alternatively, it can also delete all vectors from an index or namespace.

When deleting large numbers of vectors, limit the scope of delete operations to hundreds of vectors per operation.

Instead of deleting all vectors in an index, delete the indexand recreate it.

Delete vectors by ID

To delete vectors by their IDs, specify an idsparameter to delete. The idsparameter is an array of strings containing vector IDs.

Example

PythonJavaScriptcurl
index.delete(ids=["id-1", "id-2"], namespace='example-namespace')

Delete vectors by namespace

To delete all vectors from a namespace, specify deleteAll='true'and provide a namespaceparameter.

ℹ️Note

If you delete all vectors from a single namespace, it will also delete the namespace.
Example:

PythonJavaScriptcurl
index.delete(deleteAll='true', namespace='example-namespace')

Delete vectors by metadata

To delete vectors by metadata, pass a metadata filter expression to the delete operation.

Workflow - Manage indexes

In this section, we explain how you can get a list of your indexes, create an index, delete an index, and describe an index.

To learn about the concepts related to indexes, see Indexes.

⚠️Warning

Indexes on the Starter (free) plan are deleted after 7 days of inactivity. To prevent this, send any API request or log into the console. This will count as activity.

Getting information on your indexes

List all your Pinecone indexes:

PythonJavaScriptcurl
pinecone.list_indexes()
Get the configuration and current status of an index named "pinecone-index":

PythonJavaScriptcurl
pinecone.describe_index("pinecone-index")

Creating an index

The simplest way to create an index is as follows. This gives you an index with a single pod that will perform approximate nearest neighbor (ANN) search using cosine similarity:

PythonJavaScriptcurl
pinecone.create_index("example-index", dimension=128)
A more complex index can be created as follows. This creates an index that measures similarity by Euclidean distance and runs on 4 s1 (storage-optimized) pods of size x1:

PythonJavaScriptcurl
pinecone.create_index("example-index", dimension=128, metric="euclidean", pods=4, pod_type="s1.x1")

Create an index from a collection

To create an index from a collection, use the create_indexoperation and provide a source_collectionparameter containing the name of the collection from which you wish to create an index. The new index is queryable and writable.

Creating an index from a collection generally takes about 10 minutes. Creating a p2 index from a collection can take several hours when the number of vectors is on the order of 1M.

Example

The following example creates an index named example-indexwith 128 dimensions from a collection named example-collection.

PythonJavaScriptcurl
pinecone.create_index("example-index", dimension=128,
source_collection="example-collection")
For more information about each pod type and size, see Indexes.

For the full list of parameters available to customize an index, see the create_index API reference.

Create an index from a public collection

To create an index from a public collection, follow these steps:

1. Open the Pinecone console.
2. Click the name of the project in which you want to create the index.
3. In the left menu, click Public Collections.
4. Find the public collection from which you want to create an index. Next to that public collection, click CREATE INDEX.
5. When index creation is complete, the console redirects you to view the new index.

To learn more about using specific public collections, see the example documentation for the OpenAI Trec, Cohere Trec, and SQuADcollections.

Changing pod sizes

The default pod size is x1. After index creation, you can increase the pod size for an index.

Increasing the pod size of your index does not result in downtime. Reads and writes continue uninterrupted during the scaling process. Currently, you cannot reduce the pod size of your indexes. Your number of replicas and your total number of pods remain the same, but each pod changes size. Resizing completes in about 10 minutes.

To learn more about pod sizes, see Indexes.

Increasing the pod size for an index

To change the pod size of an existing index, use the configure_indexoperation and append the new size to the pod_typeparameter, separated by a period (.).

Example

The following example assumes that my_indexhas size x1and changes the size to x2.

PythonJavaScriptcurl
pinecone.configure_index("my_index", pod_type="s1.x2")

Checking the status of a pod size change

To check the status of a pod size change, use the describe_indexoperation. The statusfield in the results contains the key-value pair "state":"ScalingUp"or "state":"ScalingDown"during the resizing process and the key-value pair "state":"Ready"after the process is complete.

The index fullness metric provided by describe_index_statsmay be inaccurate until the resizing process is complete.

Example

The following example uses describe_indexto get the index status of the index example-index. The statusfield contains the key-value pair "state":"ScalingUp", indicating that the resizing process is still ongoing.

PythonJavaScriptcurl
pinecone.describe_index("example-index")
Results:

JSON
{
"database": {
"name": "example-index",
"dimensions": "768",
"metric": "cosine",
"pods": 6,
"replicas": 2,
"shards": 3,
"pod_type": "p1.x2",
"index_config": {},
"status": {
"ready": true,
"state": "ScalingUp"

```
}
}
}
```

Replicas

You can increase the number of replicas for your index to increase throughput (QPS). All indexes start with replicas=1.

Example

The following example uses the configure_indexoperation to set the number of replicas for the index example-indexto 4.

PythonJavaScriptcurl

```
pinecone.configure_index("example-index", replicas=4)
```

See the configure_index API referencefor more details.

Selective metadata indexing

By default, Pinecone indexes all metadata. When you index metadata fields, you can filter vector search queries using those fields. When you store metadata fields without indexing them, you keep memory utilization low, especially when you have many unique metadata values, and therefore can fit more vectors per pod.

When you create a new index, you can specify which metadata fields to index using the metadata_configparameter.

PythonJavaScriptcurl

```
metadata_config = {
"indexed": ["metadata-field-name"]
}

pinecone.create_index("example-index", dimension=128,
metadata_config=metadata_config)
```

The value for the metadata_configparameter is a JSON object containing the names of the metadata fields to index.

JSON

```
{
"indexed": [
"metadata-field-1",
"metadata-field-2",
```

"metadata-field-n"
]
}
When you provide a metadata_configobject, Pinecone only indexes the metadata fields present in that object: any metadata fields absent from the metadata_configobject are not indexed.

When a metadata field is indexed, you can filter your queriesusing that metadata field; if a metadata field is not indexed, metadata filtering ignores that field.

Examples

The following example creates an index that only indexes the genremetadata field. Queries against this index that filter for the genremetadata field may return results; queries that filter for other metadata fields behave as though those fields do not exist.

PythonJavaScriptcurl
metadata_config = {
"indexed": ["genre"]
}

pinecone.create_index("example-index", dimension=128,
metadata_config=metadata_config)

Deleting an index

This operation will delete all of the data and the computing resources associated with the index.

ℹ️Note

When you create an index, it runs as a service until you delete it. Users are billed for running indexes, so we recommend you delete any indexes you're not using. This will minimize your costs.
Delete a Pinecone index named "pinecone-index":

PythonJavaScriptcurl
pinecone.delete_index("example-index")

Pricing

For each existing index, hourly billing is determined by the per-hour-priceof a pod multiplied by the number of podsthe index uses. You will be sent an invoice at the end of the month for the total minutes your indexes have been running, regardless of activity. Reach outto an expert if you have any questions.

Pods

An index is made up of pods, which are units of cloud resources (vCPU, RAM, disk) that provide storage and compute for each index. Choose the pod typethat works best for your use case.

Indexes

Indexes store your vector embeddings and metadata. Each index uses at least one pod, but you can add more to increase storage capacity.

Scaling

As you grow, you can scale storage capacity by increasing your pod sizes. You can also increase or decrease throughput when needed, by adding replicasto an index.

Pricing Plans

Starter

Free
Limited to one index and one project.
No credit card required.

Get Started

For trying out and for small applications.
Hardware:

- Single starter pod

Features:

- Single project
- Shared environment
- Single Availability Zone

Community Support:

- Community Support

Standard

starting at $70/month

Estimated for one index on one s1 pod running for 30 days at $0.096/hour

Get Started
View pricing details

For production applications at any scale.
Hardware:

- Any number of pods and replicas
- Zero-downtime scaling

Features:

- Save vector data in Collections ($0.025/GB/month)
- Multiple projects and users
- Choose your cloud region
- Single availability zone
- Access to a free Starter pod

Standard Support:

- Email support during business hours
- Up to 2 technical contacts
- Response time SLA

Enterprise

starting at $104/month

Estimated for one index on one s1 pod running for 30 days at $0.144/hour

Get Started
View pricing details

For mission-critical production applications.
Hardware:

- Any number of pods and replicas
- Zero-downtime scaling

Features:

- Everything in Standard
- Prometheus metrics
- Multiple availability zones
- Single sign-on
- Multiple payment options

Premium Support:

- 24/7/365 dedicated support
- Up to 4 technical contacts
- Response time SLA
- Uptime SLA (99.9%)

Enterprise Dedicated: Everything in Enterprise, deployed in a single-tenant VPC environment in the region of your choice. Contact usfor details and pricing.

Use Case - Product Recommender

Learn how to build a product recommendation engine using collaborative filtering and Pinecone.

In this example, we will generate product recommendations for ecommerce customers based on previous orders and trending items. This example covers preparing the vector embeddings, creating and deploying the Pinecone service, writing data to Pinecone, and finally querying Pinecone to receive a ranked list of recommended products.

Data Preparation

Import Python Libraries

Python
import osimport timeimport randomimport numpy as npimport pandas as pdimport scipy.sparse as sparseimport itertools
Load the (Example) Instacart Data

We are going to use the Instacart Market Basket Analysisdataset for this task.

The data used throughout this example is a set of files describing customers' orders over time. The main focus is on the *orders.csv*file, where each line represents a relation between a user and the order. In other words, each line has information on *userid*(user who made the order)

and *orderid*. Note there is no information about products in this table. Product information related to specific orders is stored in the *order_product__*.csv* dataset.

Python
order_products_train = pd.read_csv('data/order_products__train.csv')order_products_prior = pd.read_csv('data/order_products__prior.csv')products = pd.read_csv('data/products.csv')orders = pd.read_csv('data/orders.csv')order_products = order_products_train.append(order_products_prior)
Preparing data for the model

The Collaborative Filtering model used in this example requires only users' historical preferences on a set of items. As there is no explicit rating in the data we are using, the purchase quantity can represent a "confidence" in terms of how strong the interaction was between the user and the products.

The dataframe data will store this data and will be the base for the model.

Python
customer_order_products = pd.merge(orders, order_products, how='inner',on='order_id')# creating a table with "confidences"data = customer_order_products.groupby(['user_id', 'product_id'])[['order_id']].count().reset_index()data.columns=["user_id", "product_id", "total_orders"]data.product_id = data.product_id.astype('int64')# Create a lookup frame so we can get the product names back in readable form later.products_lookup = products[['product_id', 'product_name']].drop_duplicates()products_lookup['product_id'] = products_lookup.product_id.astype('int64')
We will create three prototype users here and add them to our data dataframe. Each user will be buying only a specific product:

- The first user will be buying only Mineral Water
- The second user will be buying baby products: No More Tears Baby Shampooand Baby Wash & Shampoo

These users will be later used for querying and examination of the model results.

Python
data_new = pd.DataFrame([[data.user_id.max() + 1, 22802, 97], [data.user_id.max() + 2, 26834, 89], [data.user_id.max() + 2, 12590, 77]], columns=['user_id', 'product_id', 'total_orders'])data_new

	user_id	product_id	total_orders
0	206210	22802	97

1	206211	26834	89
2	206211	12590	77

Python
data = data.append(data_new).reset_index(drop = True)data.tail()

	user_id	product_id	total_orders
13863744	206209	48697	1
13863745	206209	48742	2
13863746	206210	22802	97
13863747	206211	26834	89
13863748	206211	12590	77

In the next step, we will first extract user and item unique ids, in order to create a CSR (Compressed Sparse Row) matrix.

Python
users = list(np.sort(data.user_id.unique()))items = list(np.sort(products.product_id.unique()))purchases = list(data.total_orders)# create zero-based index position <-> user/item ID mappingsindex_to_user = pd.Series(users)# create reverse mappings from user/item ID to index positionsuser_to_index = pd.Series(data=index_to_user.index + 1, index=index_to_user.values)# create zero-based index position <-> item/user ID mappingsindex_to_item = pd.Series(items)# create reverse mapping from item/user ID to index positionsitem_to_index = pd.Series(data=index_to_item.index, index=index_to_item.values)# Get the rows and columns for our new matrixproducts_rows = data.product_id.astype(int)users_cols = data.user_id.astype(int)# Create a sparse matrix for our users and products containing number of purchasessparse_product_user = sparse.csr_matrix((purchases, (products_rows, users_cols)), shape=(len(items) + 1, len(users) + 1))sparse_product_user.data = np.nan_to_num(sparse_product_user.data, copy=False)sparse_user_product = sparse.csr_matrix((purchases, (users_cols, products_rows)), shape=(len(users) + 1, len(items) + 1))sparse_user_product.data = np.nan_to_num(sparse_user_product.data, copy=False)

Implicit Model

In this section we will demonstrate creation and training of a recommender model using the implicitlibrary. The recommendation model is based off the algorithms described in the paper Collaborative Filtering for Implicit Feedback Datasetswith performance optimizations described in Applications of the Conjugate Gradient Method for Implicit Feedback Collaborative Filtering.

Python
```
!pip install --quiet -U implicit
```
Python
```
import implicitfrom implicit import evaluation#split data into train and test setstrain_set, test_set = evaluation.train_test_split(sparse_product_user, train_percentage=0.9)# initialize a modelmodel = implicit.als.AlternatingLeastSquares(factors=100, regularization=0.05, iterations=50, num_threads=1)alpha_val = 15train_set = (train_set * alpha_val).astype('double')# train the model on a sparse matrix of item/user/confidence weightsmodel.fit(train_set, show_progress = True)
```

```
WARNING:root:OpenBLAS detected. Its highly recommend to set the environment variable 'export OPENBLAS_NUM_THREADS=1' to disable its internal multithreading
```

```
0%| | 0/50 [00:00<?, ?it/s]
```

We will evaluate the model using the inbuilt library function

Python
```
test_set = (test_set * alpha_val).astype('double')evaluation.ranking_metrics_at_k(model, train_set.T, test_set.T, K=100, show_progress=True, num_threads=1)
```

```
0%| | 0/206212 [00:00<?, ?it/s]
```

```
{'precision': 0.27489359984895717,
'map': 0.04460861877969595,
'ndcg': 0.14436536385146576,
'auc': 0.6551648380086259}
```

This is what item and user factors look like. These vectors will be stored in our vector index later and used for recommendation.

Python
```
model.item_factors[1:3]
```

```
array([[ 0.01009897, 0.00260342, 0.00165942, 0.01748168, 0.00649343,
-0.01647822, 0.01860397, -0.01009837, 0.01125452, 0.01987451,
-0.00579512, 0.00421128, 0.01707346, -0.00212536, 0.01915585,
0.03640049, -0.01142028, 0.01023709, 0.00446458, -0.00143529,
```

-0.00024208, 0.00909473, -0.01408565, 0.02619351, 0.00210135,
-0.00378899, 0.01231347, 0.00278133, 0.00071992, 0.00915809,
0.01640408, 0.00880539, -0.00648519, -0.01160682, 0.00664212,
-0.00406996, 0.01543106, 0.00690582, 0.00898032, 0.00277333,
0.00626428, -0.01610408, 0.01018737, 0.0008459 , 0.02026955,
-0.01055363, -0.00107795, 0.01484767, 0.01800155, -0.00275021,
-0.0018283 , -0.00346971, 0.00077051, -0.01080908, 0.00037001,
-0.00290308, 0.00491365, -0.01362148, -0.00129594, 0.00192484,
0.00101756, -0.00051836, 0.00603317, 0.01611738, 0.00511096,
-0.0053055 , 0.01907502, 0.01232757, 0.01042075, 0.01301588,
0.00567376, 0.0152219 , 0.02414433, 0.01395251, 0.00916175,
0.01294622, 0.00187435, 0.01768819, 0.01806206, 0.01500281,
0.01065951, 0.02733074, 0.00765102, 0.00435439, -0.01976543,
0.01680202, 0.00840835, 0.00042277, -0.00216795, 0.00113048,
-0.00012699, 0.01142939, 0.01374972, -0.00985129, 0.00935802,
0.00541372, 0.01037668, 0.02024015, -0.00793628, -0.00261189],
[0.00088747, 0.00581244, 0.00074211, 0.00428396, 0.00124957,
0.00699728, 0.00304013, 0.00676518, 0.00414387, 0.00205417,
0.0029335 , 0.00505301, 0.00522107, 0.00404108, 0.00236721,
0.00406507, 0.00101947, 0.00298186, 0.00049156, 0.00279067,
0.00343525, 0.00175488, 0.00907208, 0.00276436, 0.00414505,
0.00458229, 0.00363405, 0.00375954, 0.00198171, 0.00270804,
0.00479605, 0.00120687, 0.00249341, 0.00051512, -0.00110135,
0.00844493, 0.00641403, 0.00101385, 0.00484058, 0.00632413,
0.00334539, 0.00232208, 0.00288551, 0.00755766, 0.00279979,
0.00587453, 0.00742234, 0.00580525, 0.00412665, 0.00347631,
0.00433106, 0.00427196, 0.00670939, 0.00304596, 0.00385384,
0.00222394, 0.00511582, 0.00354225, 0.00200116, 0.00717725,
0.00186237, 0.00434178, 0.00102088, 0.00222063, 0.00230367,
0.00420666, 0.00698098, 0.00549557, 0.00345657, 0.00642341,
0.00036 , 0.00464778, 0.00284442, 0.00530352, 0.00218676,
0.00493103, 0.00179086, 0.0041003 , 0.00497837, 0.0068793 ,
0.00429972, 0.00396508, 0.00451153, 0.00486684, 0.00272128,
0.00467645, 0.00423267, 0.00388015, 0.00339444, 0.00115735,
0.00807636, 0.00298532, 0.00143811, 0.00293057, 0.00590145,
0.00418158, 0.00488713, 0.00097365, -0.00083799, 0.00363581]],
dtype=float32)

Python
model.user_factors[1:3]

array([[7.24285245e-01, 5.59004486e-01, 4.96992081e-01,
-4.15437818e-01, -1.94785964e+00, -2.23764396e+00,
-1.76767483e-02, -2.21530461e+00, -6.52559578e-01,

2.78620571e-01, 6.03808701e-01, 1.27670407e-01,
3.06052566e-01, -9.93388355e-01, -5.34315288e-01,
1.20948291e+00, -2.11217976e+00, 1.67127061e+00,
1.03314137e+00, 8.54326487e-01, 1.85733151e+00,
5.69297194e-01, -8.93577933e-01, 1.76394248e+00,
1.28939009e+00, 3.32375497e-01, -2.60327369e-01,
4.21450347e-01, -1.72091925e+00, 1.10491872e+00,
-1.86411276e-01, -3.51959467e-02, -1.41517222e+00,
-9.19971287e-01, 4.63204056e-01, -4.07809407e-01,
1.23038590e+00, -8.25872004e-01, -1.50579488e+00,
8.65903348e-02, -7.29649186e-01, -5.21384776e-01,
1.59157085e+00, -8.51297379e-01, 2.81686401e+00,
-8.55669677e-01, -3.48052949e-01, -5.16085029e-01,
8.01080287e-01, 1.04207866e-01, -2.72860657e-02,
-5.18645883e-01, -1.77561533e+00, -1.22266948e+00,
-1.74415603e-01, 3.58568132e-01, -8.37117255e-01,
-1.45265543e+00, 2.43810445e-01, 5.80842435e-01,
-5.91480255e-01, 1.29645097e+00, 1.47483099e+00,
-6.84086800e-01, -7.20921755e-01, -1.11399984e+00,
2.38089368e-01, 2.19725475e-01, 3.29073220e-01,
-6.45937538e-03, 2.44079873e-01, 1.26761782e+00,
7.07967520e-01, 1.21964478e+00, 1.10735869e+00,
1.02583379e-01, -2.92189389e-01, 5.52688181e-01,
1.61700773e+00, 5.11932790e-01, -2.67194122e-01,
1.47362947e+00, -1.13380539e+00, 1.40330446e+00,
4.91484731e-01, 1.36100423e+00, 1.80482656e-01,
9.14917171e-01, 6.22740746e-01, -1.88607132e+00,
-1.34071469e+00, -2.27820247e-01, 1.15018475e+00,
-1.23491549e+00, -4.78476077e-01, -4.65549737e-01,
9.11170244e-01, 2.07606936e+00, 1.04314007e-01,
1.81862903e+00],
[8.30793440e-01, 3.86868089e-01, -1.63957000e-01,
6.93703368e-02, 1.53786719e+00, -5.87535620e-01,
3.72619987e+00, 1.22163899e-01, -8.54973614e-01,
1.11186251e-01, -1.42095876e+00, -8.75619590e-01,
-1.81247914e+00, -9.44502056e-01, 8.14570427e-01,
-5.43736219e-01, -6.02845371e-01, 2.01962996e+00,
1.60777140e+00, 2.20254612e+00, 2.08239055e+00,
8.16642225e-01, -4.42571700e-01, 6.22263908e-01,
6.29432023e-01, -1.16571808e+00, 2.32731175e+00,
-1.12640738e+00, 1.60938001e+00, 4.67458010e+00,
-1.46235943e+00, 1.46000063e+00, 1.11922979e-01,
-2.55218220e+00, 7.85077095e-01, 8.50843608e-01,
-1.10671151e+00, -6.06540870e-03, 2.76003122e-01,

-9.57318366e-01, -1.30121040e+00, -3.81188631e-01,
2.17489243e+00, 8.48001361e-01, 2.24089599e+00,
-1.32857335e+00, 9.44799244e-01, 2.29169533e-01,
1.10746622e+00, -3.48530680e-01, -2.12854624e+00,
4.96270150e-01, -1.30754066e+00, 1.41697776e+00,
2.73206377e+00, 1.48888981e+00, -1.58728147e+00,
1.58903934e-03, 1.66406441e+00, -1.75263867e-01,
2.02891684e+00, -1.95949566e+00, 1.52711666e+00,
8.71322572e-01, 1.82597125e+00, 1.37408182e-01,
-1.81464672e+00, -1.04905093e+00, -2.37590694e+00,
8.15740228e-01, 1.64217085e-01, 1.99734032e+00,
-1.54955173e+00, -5.57012379e-01, 1.32525837e+00,
-1.30014801e+00, 1.32985008e+00, -3.50400567e+00,
2.45490909e-01, -2.43037295e+00, -2.74685884e+00,
-2.12384558e+00, -1.42703640e+00, -6.69254959e-01,
1.30702591e+00, -2.15909433e+00, 1.44703603e+00,
-2.29611732e-02, 1.82583869e+00, 1.57409739e+00,
-3.97216320e-01, -6.94107652e-01, 2.89623165e+00,
2.33722359e-01, -5.27708590e-01, 1.04344904e+00,
8.51706207e-01, -4.50546294e-01, 1.38413882e+00,
2.07552814e+00]], dtype=float32)

Configure Pinecone

Install and setup Pinecone

Python
!pip install --quiet -U pinecone-client
Python
import pinecone
Python
Load Pinecone API keyapi_key = os.getenv('PINECONE_API_KEY') or 'YOUR_API_KEY'# Set Pinecone environment.env = os.getenv('PINECONE_ENVIRONMENT') or 'YOUR_ENVIRONMENT'pinecone.init(api_key=api_key, environment=env)
Get a Pinecone API keyif you don't have one.

Python
#List all present indexes associated with your key, should be empty on the first runpinecone.list_indexes()

[]

Create an Index

Python
Set a name for your indexindex_name = 'shopping-cart-demo'
Python
# Make sure service with the same name does not existif index_name in pinecone.list_indexes():
pinecone.delete_index(index_name)pinecone.create_index(name=index_name, dimension=100)
Connect to the new index

Python
index = pinecone.Index(index_name=index_name)

Load Data

Uploading all items (products that one can buy) and displaying some examples of products and their vector representations.

Python
Get all of the itemsall_items = [title for title in products_lookup['product_name']]# Transform items into factorsitems_factors = model.item_factors# Prepare item factors for uploaditems_to_insert = list(zip(all_items, items_factors[1:].tolist()))display(items_to_insert[:2])

[('Chocolate Sandwich Cookies',
[0.010098974220454693,
0.0026034200564026833,
0.0016594183398410678,
0.017481675371527672,
0.006493427790701389,
-0.016478220000863075,
0.018603969365358353,
-0.010098369792103767,
0.01125451922416687,
0.019874505698680878,
-0.005795117933303118,
0.00421128049492836,
0.017073458060622215,
-0.0021253626327961683,
0.019155845046043396,
0.036400485783815384,
-0.01142028160393238,
0.010237086564302444,
0.004464581608772278,
-0.0014352924190461636,

-0.00024208369723055512,
0.009094727225601673,
-0.014085653237998486,
0.02619350701570511,
0.002101349411532283,
-0.0037889881059527397,
0.012313470244407654,
0.002781332703307271,
0.0007199185783974826,
0.009158086962997913,
0.016404075548052788,
0.008805392310023308,
-0.006485185585916042,
-0.01160681527107954,
0.006642122287303209,
-0.004069960676133633,
0.015431062318384647,
0.006905817426741123,
0.008980315178632736,
0.002773326588794589,
0.0062642814591526985,
-0.0161040760576725,
0.010187366977334023,
0.0008458984084427357,
0.02026955410838127,
-0.010553630068898201,
-0.0010779497679322958,
0.014847667887806892,
0.018001552671194077,
-0.0027502067387104034,
-0.0018282983219251037,
-0.0034697114024311304,
0.000770510989241302,
-0.010809078812599182,
0.0003700107627082616,
-0.002903081476688385,
0.004913648124784231,
-0.01362148392945528,
-0.001295942347496748,
0.0019248360767960548,
0.0010175565257668495,
-0.0005183601751923561,
0.006033174227923155,
0.016117379069328308,

0.005110959522426128,
-0.00530549930408597,
0.019075021147727966,
0.012327569536864758,
0.01042074803262949,
0.01301588024944067,
0.005673760548233986,
0.015221904963254929,
0.024144325405359268,
0.01395251415669918,
0.009161749854683876,
0.012946223840117455,
0.0018743481487035751,
0.017688188701868057,
0.018062060698866844,
0.015002812258899212,
0.010659514926373959,
0.02733074128627777,
0.0076510170474648476,
0.0043543861247599125,
-0.019765431061387062,
0.016802024096250534,
0.008408350870013237,
0.0004227694298606366,
-0.002167945960536599,
0.0011304811341688037,
-0.0001269889180548489,
0.01142938993871212,
0.013749724254012108,
-0.00985129363834858,
0.009358019568026066,
0.0054137222468853,
0.010376684367656708,
0.020240148529410362,
-0.007936276495456696,
-0.0026118927635252476]),
('All-Seasons Salt',
[0.0008874664781615138,
0.005812443327158689 5,
0.0007421106565743685,
0.00428396463394165,
0.001249574706889689,
0.006997276097536087,
0.0030401344411075115,

0.006765175145119429,
0.004143866710364819,
0.0020541702397167683,
0.002933498937636614,
0.005053007043898106,
0.00522107258439064,
0.004041083622723818,
0.002367211040109396,
0.004065068904310465,
0.0010194696951657534,
0.0029818632174283266,
0.0004915563040412962,
0.0027906731702387333,
0.0034352506045252085,
0.0017548849573358893,
0.009072077460587025,
0.002764355158433318,
0.004145053215324879,
0.004582288675010204,
0.003634049789980054,
0.0037595359608531,
0.00198170798830688,
0.002708042971789837,
0.004796050023287535,
0.0012068713549524546,
0.0024934052489697933,
0.0005151224322617054,
-0.001101348432712257,
0.00844493042677641,
0.006414031144231558,
0.001013854518532753,
0.0048405807465314865,
0.006324129644781351,
0.0033453928772360086,
0.0023220758885145187,
0.002885512774810195,
0.007557660341262817,
0.002799794776365161,
0.005874533671885729,
0.007422335911542177,
0.0058052497915923595,
0.004126648418605328,
0.0034763067960739136,
0.004331058822572231,

0.004271955695003271,
0.00670938566327095,
0.0030459642875939608,
0.0038538381922990084,
0.0022239401005208492,
0.005115816835314035,
0.003542253514751792,
0.002001164946705103,
0.007177253719419241,
0.0018623704090714455,
0.004341782070696354,
0.0010208759922534227,
0.0022206329740583897,
0.002303670858964324,
0.004206661134958267,
0.006980976089835167,
0.005495565943419933,
0.003456572536379099,
0.006423408165574074,
0.0003599990450311452,
0.004647782538086176,
0.0028444179333746433,
0.005303522571921349,
0.0021867596078664064,
0.004931030794978142,
0.0017908598529174924,
0.0041002980433404446,
0.004978368990123272,
0.006879299879074097,
0.004299724940210581,
0.0039650811813771725,
0.004511528182774782,
0.00486684450879693,
0.0027212793938815594,
0.004676445387303829,
0.0042326669208705425,
0.003880152478814125,
0.003394442144781351,
0.0011573455994948745,
0.008076360449194908,
0.0029853193555027246,
0.0014381115324795246,
0.0029305710922926664,
0.005901449825614691,

0.004181584343314171,
0.004887125454843044,
0.0009736462379805744,
-0.0008379911305382848,
0.00363580626435577787])]

Insert items into the index

Python
def chunks(iterable, batch_size=100): it = iter(iterable) chunk = tuple(itertools.islice(it, batch_size)) while chunk: yield chunk chunk = tuple(itertools.islice(it, batch_size))
Python
print('Index statistics before upsert:', index.describe_index_stats())for e, batch in enumerate(chunks([(ii[:64],x) for ii,x in items_to_insert])): index.upsert(vectors=batch)print('Index statistics after upsert:', index.describe_index_stats())

Index statistics before upsert: {'dimension': 0, 'namespaces': {}}
Index statistics after upsert: {'dimension': 100, 'namespaces': {'': {'vector_count': 49677}}}

This is a helper method for analysing recommendations later.This method returns top N products that someone bought in the past (based on product quantity).

Python
def products_bought_by_user_in_the_past(user_id: int, top: int = 10): selected = data[data.user_id == user_id].sort_values(by=['total_orders'], ascending=False) selected['product_name'] = selected['product_id'].map(products_lookup.set_index('product_id')['product_name']) selected = selected[['product_id', 'product_name', 'total_orders']].reset_index(drop=True) if selected.shape[0] < top: return selected return selected[:top]
Python
data.tail()

	user_id	product_id	total_orders
13863744	206209	48697	1
13863745	206209	48742	2
13863746	206210	22802	97
13863747	206211	26834	89
13863748	206211	12590	77

Query for Recommendations

We are now retrieving user factors for users that we have manually created before for testing purposes. Besides these users, we are adding a random existing user. We are also displaying these users so you can see what these factors look like.

Python
user_ids = [206210, 206211, 103593]user_factors =
model.user_factors[user_to_index[user_ids]]display(user_factors[1:])

array([[-2.446773 , -0.62870413, -0.9166386 , -1.0933994 , 0.9897131 ,
-2.166681 , 0.09873585, 1.1049409 , 1.6753025 , 1.5794269 ,
1.8142459 , 1.5048354 , 0.7157051 , -0.7888281 , 0.06156079,
-1.6539581 , -0.15790005, 0.5999737 , -1.4803663 , -0.03179923,
0.91451246, 0.14260213, -1.1541293 , -0.01566206, -1.3449577 ,
-2.232925 , -0.88052607, 0.19183849, 0.3109626 , 1.32479 ,
0.16483077, -0.8045166 , 1.36922 , 0.81774026, 1.3368418 ,
2.8871357 , 2.4540865 , -1.908394 , 2.8208447 , -1.3499558 ,
-0.90089166, 1.0632626 , 1.8107275 , -0.83986664, 1.1764408 ,
-1.6621239 , -1.4636188 , -2.3367987 , -1.2510904 , 0.4349534 ,
0.08233324, 1.0688674 , -0.41190436, 1.6045849 , -2.3667567 ,
-1.8557758 , -0.1931467 , 0.10383442, 1.3932719 , 1.3465406 ,
-0.17274773, 0.41542327, -1.0992794 , 1.7954347 , -0.9157203 ,
-0.3183454 , 0.7724282 , -0.5658835 , 1.0758705 , -1.7377888 ,
2.0294137 , -2.1382923 , 1.0606468 , 1.800927 , -1.3713943 ,
1.0659586 , 0.31013912, -0.5963934 , 0.69738954, 1.383554 ,
1.0078012 , -2.7117298 , -1.7087 , 0.4050448 , 3.548174 ,
0.27247337, -0.16570352, -0.92676795, -1.2243328 , 0.63455725,
-1.5337977 , -2.8735108 , 1.2812912 , -0.11600056, 1.2358317 ,
0.5591759 , -0.63913107, 1.2325013 , 1.3712876 , -1.3370212],
[1.70396 , -1.5320156 , 2.8847353 , 0.32170388, 1.3340172 ,
-1.1947397 , 1.9013127 , -0.4816413 , -2.0899863 , -1.2761233 ,
-1.8430734 , -0.6221577 , 0.8063771 , 1.2961249 , 0.18268324,
-3.2958453 , -0.31202024, 3.8049164 , 0.73393685, 1.7682556 ,
0.372242 , 1.002703 , 0.32070097, 0.2046866 , 0.9008953 ,
1.3807229 , 1.1176021 , 0.1957425 , -1.3196671 , 2.1180258 ,
0.48846507, 0.76666814, -0.30274457, -2.5167181 , 0.3489467 ,
2.0131872 , -1.5119745 , -0.91736513, 1.3228838 , -1.5192536 ,
-1.1463904 , -1.0334512 , 1.2355485 , -0.21977787, 2.3017268 ,
-1.4751832 , -0.6216355 , 0.3089897 , -0.85497165, -0.31444585,
-3.100829 , 2.390458 , 0.07399248, -0.09938905, -1.0162137 ,
1.9475894 , -0.9248195 , -1.084834 , 0.39212215, 0.6491842 ,
1.2028612 , -1.0323097 , 2.6522071 , -0.8172474 , 1.0873827 ,
-2.9416876 , -0.06957518, -0.7316911 , -0.7430743 , 0.319504 ,

-0.9984044 , 0.06710945, -3.003772 , 0.6744962 , 2.1210036 ,
-0.4559903 , 0.6154137 , -1.7743443 , 0.5672013 , 1.004357 ,
-1.8588076 , 0.05864619, 0.01209994, 2.0575655 , -1.1680491 ,
0.3783967 , 1.6527759 , 1.5397102 , -0.2965242 , 2.5335467 ,
-0.40009058, -0.66989446, -1.6143844 , 0.7761751 , -1.0538983 ,
0.48226374, 1.2432365 , 2.1671696 , 1.7070205 , 0.2968687]],
dtype=float32)

Model recommendations

We will now retrieve recommendations from our model directly, just to have these results as a baseline.

Python
print("Model recommendations\n")start_time = time.process_time()recommendations0 = model.recommend(userid=user_ids[0], user_items=sparse_user_product)recommendations1 = model.recommend(userid=user_ids[1], user_items=sparse_user_product)recommendations2 = model.recommend(userid=user_ids[2], user_items=sparse_user_product)print("Time needed for retrieving recommended products: " + str(time.process_time() - start_time) + ' seconds.\n')print('\nRecommendations for person 0:')for recommendation in recommendations0: product_id = recommendation[0] print(products_lookup[products_lookup.product_id == product_id]['product_name'].values)print('\nRecommendations for person 1:')for recommendation in recommendations1: product_id = recommendation[0] print(products_lookup[products_lookup.product_id == product_id]['product_name'].values)print('\nRecommendations for person 2:')for recommendation in recommendations2: product_id = recommendation[0] print(products_lookup[products_lookup.product_id == product_id]['product_name'].values)

Model recommendations

Time needed for retrieving recommended products: 0.0625 seconds.

Recommendations for person 0:
['Sparkling Water']
['Soda']
['Smartwater']
['Zero Calorie Cola']
['Natural Artesian Water']
['Natural Spring Water']
['Distilled Water']
['Sparkling Natural Mineral Water']
['Spring Water']
['Drinking Water']

Recommendations for person 1:
['Baby Wipes Sensitive']
['YoKids Squeezers Organic Low-Fat Yogurt, Strawberry']
['Organic Blackberries']
['Organic Whole Milk']
['Eggo Pancakes Minis']
['Natural California Raisins Mini Snack Boxes']
['100% Raw Coconut Water']
['White Buttermints']
['Danimals Strawberry Explosion Flavored Smoothie']
['Strawberry Explosion/Banana Split Smoothie']

Recommendations for person 2:
['Organic Golden Delicious Apple']
['Organic Red Delicious Apple']
['Bartlett Pears']
['Organic Blackberries']
['Bag of Organic Bananas']
['Black Seedless Grapes']
['Organic Braeburn Apple']
['Organic Blueberries']
["Organic D'Anjou Pears"]
['White Peach']

Query the index

Let's now query the index to check how quickly we retrieve results. Please note that query speed depends in part on your internet connection.

Python
Query by user factorsstart_time = time.process_time()query_results = index.query(queries=user_factors[:-1].tolist(), top_k=10)print("Time needed for retrieving recommended products using Pinecone: " + str(time.process_time() - start_time) + ' seconds.\n')for _id, res in zip(user_ids, query_results.results): print(f'user_id={_id}') df = pd.DataFrame({ 'products': [match.id for match in res.matches], 'scores': [match.score for match in res.matches] }) print("Recommendation: ") display(df) print("Top buys from the past: ") display(products_bought_by_user_in_the_past(_id, top=15))

Time needed for retrieving recommended products using Pinecone: 0.03125 seconds.

user_id=206210
Recommendation:

	products	scores
0	Mineral Water	0.919242
1	Zero Calorie Cola	0.716640
2	Orange & Lemon Flavor Variety Pack Sparkling F...	0.631119
3	Sparkling Water	0.603575
4	Milk Chocolate Almonds	0.577868
5	Extra Fancy Unsalted Mixed Nuts	0.577714
6	Popcorn	0.565397
7	Organic Coconut Water	0.547605
8	Drinking Water	0.542832
9	Tall Kitchen Bag With Febreze Odor Shield	0.538533

Top buys from the past:

	product_id	product_name	total_orders
0	22802	Mineral Water	97

user_id=206211
Recommendation:

	products	scores
0	Baby Wash & Shampoo	0.731054
1	No More Tears Baby Shampoo	0.695655
2	Size 6 Baby Dry Diapers	0.526953
3	Natural Applesauce Snack & Go Pouches	0.478145
4	White Buttermints	0.475006

5	Size 5 Cruisers Diapers Super Pack	0.474203
6	Go-Gurt SpongeBob SquarePants Strawberry Ripti...	0.461982
7	Baby Wipes Sensitive	0.461840
8	Original Detergent	0.456813
9	Stage 1 Newborn Hypoallergenic Liquid Detergent	0.456143

Top buys from the past:

	product_id	product_name	total_orders
0	26834	No More Tears Baby Shampoo	89
1	12590	Baby Wash & Shampoo	77

*Note*The inference using Pinecone is much faster compared to retrieving recommendations from a model directly. Please note that this result depends on your internet connection as well.

All that's left to do is surface these recommendations on the shopping site, or feed them into other applications.

Clean up

Delete the index once you are sure that you do not want to use it anymore. Once it is deleted, you cannot reuse it.

Python
pinecone.delete_index(index_name)

Summary

In this example we used Pineconeto build and deploy a product recommendation engine that uses collaborative filtering, relatively quickly.

Once deployed, the product recommendation engine can index new data, retrieve recommendations in milliseconds, and send results to production applications.

Use Case - Image Search

Background

What is Image Search and how will we use it?

One may find themselves with an image, looking for similar images among a large image corpus. The difficult part of this requirement is instantly retrieving, at scale, similar images, especially when there are tens of millions or billions of images from which to choose.

In this example, we will walk you through the mechanics of how to solve this problem using an off-the-shelf, pretrained, neural network to generate data structures known as vector embeddings. We will use Pinecone's vector database offering to find images with similar vector embeddings to an *query image*.

Learning Goals and Estimated Reading Time

By the end of this 15 minute demo (on a recent MacBook Pro, or up to an hour on Google Colab), you will have:

1. Learned about Pinecone's value for solving realtime image search requirements!
2. Stored and retrieved vectors from Pinecone your very-own Pinecone Vector Database.
3. Encoded images as vectors using a pretrained neural network (i.e. no model training necessary).
4. Queried Pinecone's Vector Database to find similar images to the query in question.

Once all data is encoded as vectors, and is in your Pinecone Index, results of Pinecone queries are returned, on average, in tens of milliseconds.

Setup: Prerequisites and Image Data

Python 3.7+

This code has been tested with Python 3.7. It is recommended to run this code in a virtual environment or Google Colab.

Acquiring your Pinecone API Key

A Pinecone API key is required. You can obtain a complimentary key on our our website. Either add PINECONE_EXAMPLE_API_KEYto your list of environmental variables, or manually enter

it after running the below cell (a prompt will pop up requesting the API key, storing the result within this kernel (session)).

Installing and Importing Prerequisite Libraries:

All prerequisites are installed and listed in the next cell.

Installing via pip

Python

```
!pip install -qU pinecone-client \
torchvision \
seaborn \
tqdm \
httpimport \
requests
```

Importing and Defining Constants

Python

```
import os
import requests

import tqdm
import httpimport
import pinecone
import numpy as np
from PIL import Image

import torch
import torchvision

DATA_DIRECTORY = 'tmp'
INDEX_NAME = 'image-search'
INDEX_DIMENSION = 1000
BATCH_SIZE=200
```

Helper Module

This helper module will be imported and will enable this notebook to be self-contained.

Python

```
# There is a helper module required for this notebook to run.
# When not present with this notebook, it will be streamed in from Pinecone's Example Repository.
```

```
# You can find the module at
https://github.com/pinecone-io/examples/tree/master/image_search

if os.path.isfile('helper.py'):
import helper as h
else:
print('importing `helper.py` from https://github.com/pinecone-io')
with httpimport.github_repo(
username='startakovsky',
repo='pinecone-examples-fork',
module=['image_search'],
branch='may-2022-image-search-refresh'):
from image_search import helper as h
Extracting API Key from environmental variable PINECONE_EXAMPLE_API_KEY...

Pinecone API Key available at h.pinecone_api_key
```

Downloading Data

To demonstrate image search using Pinecone, we will download 100,000 small images using [built-in datasets](#)available with the torchvisionlibrary.

```
Python
datasets = {
'CIFAR10': torchvision.datasets.CIFAR10(DATA_DIRECTORY, transform=h.preprocess,
download=True),
'CIFAR100': torchvision.datasets.CIFAR100(DATA_DIRECTORY, transform=h.preprocess,
download=True)
}

Files already downloaded and verified
Files already downloaded and verified
```

Inspecting Images

These are some of the images from what was just downloaded. If interested, read about the CIFAR image dataset [here](#).

```
Python
h.show_random_images_from_full_dataset(datasets['CIFAR100'])
```

Generating Embeddings and Sending them to Pinecone

Loading a Pretrained Computer Vision Model

We will use a pretrained model that, like the dataset above, is shipped with PyTorch. This model will create a 1000-dimensional sequence of floats for each input image. We will use this output as an embedding associated with an image.

Python
model = torchvision.models.squeezenet1_1(pretrained=True).eval()

Why SqueezeNet?

We chose the SqueezeNetmodel because it is a very small model and basic model that has been trained on millions of imagesacross 1000 classes. It is easy to instantiate with one line of code and generates embeddings quite a bit faster than deeper models.

On Comparing Embeddings

Two embeddings might look like something like this:

- [-0.02, 0.06, 0.0, 0.01, 0.08, -0.03, 0.01, 0.02, 0.01, 0.02, -0.07, -0.11, -0.01, 0.08, -0.04]
- [-0.04, -0.09, 0.04, -0.1, -0.05, -0.01, -0.06, -0.04, -0.02, -0.04, -0.04, 0.07, 0.03, 0.02, 0.03]

In order to determine how similar they are, we use a simpleformula that takes a very short time to compute. Similarity scores are, in general, an excellent proxy for image similarity.

Creating Our Pinecone Index

The process for creating a Pinecone Index requires your Pinecone API key, the name of your index, and the number of dimensions of each vector (1000).

In this example, to compare embeddings, we will use the cosine similarity scorebecause this model generates un-normalized probability vectors. While this calculation is trivial when comparing two vectors, it will take quite a long time when needing to compare a query vector against millions or billions of vectors and determine those most similar with the query vector.

What is Pinecone for?

There is often a technical requirement to compare one vector to tens or hundreds of millions or more vectors, to do so with low latency (less than 50ms) and a high throughput. Pinecone solves this problem with its managed vector database service, and we will demonstrate this below.

Python
```
# authenticate with Pinecone API, keys and environment available at your project at https://app.pinecone.io
pinecone.init(h.pinecone_api_key, environment='YOUR_ENVIRONMENT')
# if the index does not already exist, we create it
if INDEX_NAME not in pinecone.list_indexes():
pinecone.create_index(name=INDEX_NAME, dimension=INDEX_DIMENSION)
# instantiate connection to your Pinecone index
index = pinecone.Index(INDEX_NAME)
```

Preparing Vector Embeddings

We will encode the downloaded images for upload to Pinecone, and store the associated class of each image as metadata.

Creating Vector IDs

Each vector ID will have a prefix corresponding to *CIFAR10*or *CIFAR100*.

Python
```
def get_vector_ids(batch_number, batch_size, prefix):
"""Return vector ids."""
start_index = batch_number * batch_size
end_index = start_index + batch_size
ids = np.arange(start_index, end_index)
# create id based on prefix
# eg. if id == 5, prefix == 'CIFAR10', then create 'CIFAR10.5' as vector id.
ids_with_prefix = map(lambda x: f'{prefix}.{str(x)}', ids)
return ids_with_prefix
```

Creating metadata for each vector containing class label

Python
```
def get_vector_metadata(label_indices, class_list):
"""Return list of {'label': <class name>}."""
get_class_name = lambda index: {'label': class_list[index]}
return map(get_class_name, label_indices)
```

Constructing Vector Embeddings

In a Pinecone Vector Database, there are three components to every Pinecone vector embedding:

- a vector ID
- a sequence of floats of a user-defined, fixed dimension
- vector metadata (a key-value mapping, used for filtering at runtime)

Python
```
def get_vectors_from_batch(preprocessed_data, label_indices, batch_number, dataset):
"""Return list of tuples like (vector_id, vector_values, vector_metadata)."""
num_records = len(preprocessed_data)
prefix = dataset.__class__.__name__
with torch.no_grad():
# generate image embeddings with PyTorch model
vector_values = model(preprocessed_data).tolist()
# return respective IDs/metadata for each image embedding
vector_metadata = get_vector_metadata(label_indices, dataset.classes)
vector_ids = get_vector_ids(batch_number, num_records, prefix)
return list(zip(vector_ids, vector_values, vector_metadata))
```

Example Vector Embedding

The below code is an example of a vector embedding, showing just the first 3 components of the associated vector.

Python
```
dataset = datasets['CIFAR100']
list_of_preprocessed_tensors, label_indices = list(zip(*[dataset[i] for i in range(BATCH_SIZE)]))
preprocessed_data = torch.stack(list_of_preprocessed_tensors)
vectors = get_vectors_from_batch(preprocessed_data, label_indices, 0, dataset)
id_, embedding, metadata = vectors[123]
print(id_, embedding[:3], metadata, sep=', ')
```

```
CIFAR100.123, [4.237038612365723, 11.179943084716797, 1.3662679195404053], {'label':
'orange'}
```

Upsert Vectors to Pinecone

This function iterates through a dataset in batches, generates a list of vector embeddings (as in the the above example) and upserts in batches to Pinecone.

Python
```
def upsert_image_embeddings(dataset, pinecone_index, batch_size=BATCH_SIZE,
num_rows=None):
"""Iterate through dataset, generate embeddings and upsert in batches to Pinecone index.
Args:
- dataset: a PyTorch Dataset
- pinecone_index: your Pinecone index
- batch_size: batch size
- num_rows: Number of initial rows to use of dataset, use all rows if None.
"""
if num_rows > len(dataset):
```

```
raise ValueError(f`num_rows` should not exceed length of dataset: {len(dataset)}')
if num_rows:
sampler = range(num_rows)
else:
sampler = None
dataloader = torch.utils.data.DataLoader(dataset, batch_size=BATCH_SIZE, sampler=sampler)
tqdm_kwargs = h.get_tqdm_kwargs(dataloader)
for batch_number, (data, label_indices) in tqdm.notebook.tqdm(enumerate(dataloader),
**tqdm_kwargs):
vectors = get_vectors_from_batch(
data,
label_indices,
batch_number,
dataloader.dataset)
pinecone_index.upsert(vectors)
```

Begin Upsert for all 100,000 Images

One progress bar is generated per dataset. Truncate number of rows in each dataset by modifying num_rowsparameter value in the cell below. Each of the CIFAR datasets have 50,000 rows.

Python

```
for dataset in datasets.values():
upsert_image_embeddings(dataset, index, num_rows=50_000)
```

0%| | 0/250 [00:00<?, ?chunk of 200 CIFAR10 vectors/s]

0%| | 0/250 [00:00<?, ?chunk of 200 CIFAR100 vectors/s]

View Progress On The Pinecone Console(sample screenshot below)

Querying Pinecone

Now that all the embeddings of the images are on Pinecone's database, it's time to demonstrate Pinecone's lightning fast query capabilities.

Pinecone Example Usage

In the below example we query Pinecone's API with an embedding of a query image to return the vector embeddings that have the highest similarity score. Pinecone effeciently estimates which of the uploaded vector embeddings have the highest similarity when paired with the query term's embedding, and the database will scale to billions of embeddings maintaining low-latency and high throughput. In this example we have upserted 100,000 embeddings. Our starter plan supports up to one million.

Example: Pinecone API Request and Response

Let's find images similar to the query_imagevariable, shown below.

Example Query Image

Python
```
url = 'https://www.cs.toronto.edu/~kriz/cifar-10-sample/dog4.png'
r = requests.get(url, stream=True)
query_image = Image.open(r.raw)
h.printmd("#### A sample image")
query_image.resize((125,125))
```

A sample image

Python
```
query_embedding = model(h.preprocess(query_image).unsqueeze(0)).tolist()
response = index.query(query_embedding, top_k=4, include_metadata=True)
h.printmd(f"#### A sample response from Pinecone \n ==============\n \n")
h.printmd(f"```python\n{response}\n```")
```

A sample response from Pinecone

==========

Python
```
{'matches': [{'id': 'CIFAR10.11519',
'metadata': {'label': 'dog'},
'score': 1.00000012,
'values': []},
{'id': 'CIFAR10.21059',
'metadata': {'label': 'dog'},
'score': 0.982942224,
'values': []},
{'id': 'CIFAR10.48510',
'metadata': {'label': 'dog'},
'score': 0.982879698,
'values': []},
```

{'id': 'CIFAR100.32560',
'metadata': {'label': 'seal'},
'score': 0.982618093,
'values': []}],
'namespace': ''}

Enriched Response

In the next few lines, we look up the actual images associated to the vector embeddings.

Python
h.show_response_as_grid(response, datasets, 1, 4, figsize=(10, 10))

Results

We invite the reader to explore various queries to see how they come up. In the one above, we chose one of the CIFAR-10 images as the query image. Note that the query image embedding need not exist in your Pinecone index in order to find similar images. Additionally, the search results are only as good as the embeddings, which are based on the quality and quantity of the images as well as how expressive the model used is. There are plenty of other out of the box, pretrained models in PyTorch and elsewhere!

Pinecone Example Usage with Metadata

Extensive predicate logic can be applied to metadata filtering, just like the WHERE clausein SQL! Pinecone's metadata featureprovides easy-to-implement filtering.

Example using Metadata

For demonstration, let's use metadata to find all images classified as a *seal*that look like the query_imagevariable shown above.

Python
response = index.query(
query_embedding,
top_k=25,
filter={"label": {"$eq": "seal"}},
include_metadata=True
)
h.show_response_as_grid(response, datasets, 5, 5, figsize=(10, 10))

Results

All of the results returned are indeed seals, and many of them do look like the query image! Note how the cosine similarity scores are returned in descending order.

Additional Note On Querying Pinecone

In this example, you queried your Pinecone index with an embedding that was already in the index, however that is not necessary at all. For this index, *any 1000-dimensional embedding*can be used to query Pinecone.

Conclusion

In this example, we demonstrated how Pinecone makes it possible to do realtime image similarity search using a pre-trained computer vision model! We also demonstrated the use of metadata filtering with querying Pinecone's vector database.

Like what you see? Explore our community

Learn more about semantic search and the rich, performant, and production-level feature set of Pinecone's Vector Database by visiting https://pinecone.io, connecting with us hereand following us on LinkedIn. If interested in some of the algorithms that allow for effecient estimation of similar vectors, visit our Algorithms and Libraries section of our Learning Center.

Use Case - Ecommerce Hybrid Search

Hybrid Search for E-Commerce with Pinecone

Hybrid vector search is combination of traditional keyword search and modern dense vector search. It has emerged as a powerful tool for e-commerce companies looking to improve the search experience for their customers.

By combining the strengths of traditional text-based search algorithms with the visual recognition capabilities of deep learning models, hybrid vector search allows users to search for products using a combination of text and images. This can be especially useful for product searches, where customers may not know the exact name or details of the item they are looking for.

Pinecone's new sparse-dense indexallows you to seamlessly perform hybrid search for e-commerce or in any other context. This notebook demonstrates how to use the new hybrid search feature to improve e-commerce search.

Install Dependencies

First, let's import the necessary libraries

Python
!pip install -qU datasets transformers sentence-transformers pinecone-client[grpc]

Connect to Pinecone

Let's initiate a connection and create an index. For this, we need a free API key, and then we initialize the connection like so:

Python
import pinecone# init connection to pineconepinecone.init(api_key='YOUR_API_KEY', # app.pinecone.io environment='YOUR_ENV' # find next to api key)
To use the sparse-denseindex in Pinecone we must set metric="dotproduct"and use either s1or p1pods. We also align the dimensionvalue to that of our retrieval model, which outputs 512-dimensional vectors.

Python
choose a name for your indexindex_name = "hybrid-image-search"if index_name not in pinecone.list_indexes(): # create the index pinecone.create_index(index_name, dimension=512, metric="dotproduct", pod_type="s1")
Now we have created the sparse-dense enabled index, we connect to it:

Python
index = pinecone.GRPCIndex(index_name)
Note: we are using GRPCIndexrather than Indexfor the improved upsert speeds, either can be used with the sparse-dense index.

Load Dataset

We will work with a subset of the Open Fashion Product Imagesdataset, consisting of ~44K fashion products with images and category labels describing the products. The dataset can be loaded from the Huggigface Datasets hub as follows:

Python
from datasets import load_dataset# load the dataset from huggingface datasets hubfashion = load_dataset("ashraq/fashion-product-images-small", split="train")fashion

Dataset({
features: ['id', 'gender', 'masterCategory', 'subCategory', 'articleType', 'baseColour', 'season', 'year', 'usage', 'productDisplayName', 'image'],
num_rows: 44072
})

We will first assign the images and metadata into separate variables and then convert the metadata into a pandas dataframe.

Python
assign the images and metadata to separate variablesimages = fashion["image"]metadata = fashion.remove_columns("image")
Python
display a product imageimages[900]

Python
convert metadata into a pandas dataframemetadata = metadata.to_pandas()metadata.head()

	id	gender	masterCategory	subCategory	articleType	baseColour	season	year	usage	productDisplayName
0	15970	Men	Apparel	Topwear	Shirts	Navy Blue	Fall	2011.0	Casual	Turtle Check Men Navy Blue Shirt
1	39386	Men	Apparel	Bottomwear	Jeans	Blue	Summer	2012.0	Casual	Peter England Men Party Blue Jeans
2	59263	Women	Accessories	Watches	Watches	Silver	Winter	2016.0	Casual	Titan Women Silver Watch
3	21123	Men	Apparel	Bottomwear	Track Pants	Black	Fall	2011.0	Casual	Manchester United Men Solid Black Track Pants

	79									
4	53759	Men	Apparel	Topwear	Tshirts	Grey	Summer	2012.0	Casual	Puma Men Grey T-shirt

We need both sparse and dense vectors to perform hybrid search. We will use all the metadata fields except for the idand yearto create sparse vectors and the product images to create dense vectors.

Sparse Vectors

To create the sparse vectors we'll use BM25.

⚠️Warning

For now we'll implement BM25 using a temporary helper function. This will be updated with a more permanent solution in the near future.
Python
import requestswith open('pinecone_text.py' ,'w') as fb:
fb.write(requests.get('https://storage.googleapis.com/gareth-pinecone-datasets/pinecone_text.py').text)
The above helper function requires that we pass a tokenizer that will handle the splitting of text into tokens before building the BM25 vectors.

We will use a bert-base-uncasedtokenizer from Hugging Face tokenizers:

Python
from transformers import BertTokenizerFastimport pinecone_text# load bert tokenizer from huggingfacetokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')def tokenize_func(text): token_ids = tokenizer(text, add_special_tokens=False)['input_ids'] return tokenizer.convert_ids_to_tokens(token_ids)bm25 = pinecone_text.BM25(tokenize_func)
Python
tokenize_func('Turtle Check Men Navy Blue Shirt')

['turtle', 'check', 'men', 'navy', 'blue', 'shirt']

BM25 requires training on a representative portion of the dataset. We do this like so:

Python

bm25.fit(metadata['productDisplayName'])

BM25(avgdl=7.983640406607369,
doc_freq=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 3.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
30.0, 9.0, 2.0, 0.0, ...],
ndocs=44072)

Let's create a test sparse vector using a productDisplayName.

Python
metadata['productDisplayName'][0]

'Turtle Check Men Navy Blue Shirt'

Python
bm25.transform_query(metadata['productDisplayName'][0])

{'indices': [3837, 7163, 19944, 29471, 32256, 55104],
'values': [0.11231091182274018,
0.33966052569334826,
0.10380364782761624,
0.21489054354050624,
0.04188327394634689,
0.18745109716944222]}

And for the stored docs, we only need the "IDF" part:

Python
bm25.transform_doc(metadata['productDisplayName'][0])

{'indices': [3837, 7163, 19944, 29471, 32256, 55104],
'values': [0.4344342631341496,
0.4344342631341496,
0.4344342631341496,
0.4344342631341496,
0.4344342631341496,
0.4344342631341496]}

Dense Vectors

We will use the CLIP to generate dense vectors for product images. We can directly pass PIL images to CLIP as it can encode both images and texts. We can load CLIP like so:

Python
```
from sentence_transformers import SentenceTransformerimport torchdevice = 'cuda' if torch.cuda.is_available() else 'cpu'# load a CLIP model from huggingfacemodel = SentenceTransformer( 'sentence-transformers/clip-ViT-B-32', device=device)model
```

```
SentenceTransformer(
(0): CLIPModel()
)
```

Python
```
dense_vec = model.encode([metadata['productDisplayName'][0]])dense_vec.shape
```

```
(1, 512)
```

The model gives us a 512dimensional dense vector.

Upsert Documents

Now we can go ahead and generate sparse and dense vectors for the full dataset and upsert them along with the metadata to the new hybrid index. We can do that easily as follows:

Python
```
from tqdm.auto import tqdmbatch_size = 200for i in tqdm(range(0, len(fashion), batch_size)): # find end of batch i_end = min(i+batch_size, len(fashion)) # extract metadata batch meta_batch = metadata.iloc[i:i_end] meta_dict = meta_batch.to_dict(orient="records") # concatinate all metadata field except for id and year to form a single string meta_batch = [" ".join(x) for x in meta_batch.loc[:, ~meta_batch.columns.isin(['id', 'year'])].values.tolist()] # extract image batch img_batch = images[i:i_end] # create sparse BM25 vectors sparse_embeds = [bm25.transform_doc(text) for text in meta_batch] # create dense vectors dense_embeds = model.encode(img_batch).tolist() # create unique IDs ids = [str(x) for x in range(i, i_end)] upserts = [] # loop through the data and create dictionaries for uploading documents to pinecone index for _id, sparse, dense, meta in zip(ids, sparse_embeds, dense_embeds, meta_dict): upserts.append({ 'id': _id, 'sparse_values': sparse, 'values': dense, 'metadata': meta }) # upload the documents to the new hybrid index index.upsert(upserts)# show index description after uploading the documentsindex.describe_index_stats()
```

Querying

Now we can query the index, providing the sparse and dense vectors. We do this directly with an equal weighting between sparse and dense like so:

Python

query = "dark blue french connection jeans for men"# create sparse and dense vectorssparse = bm25.transform_query(query)dense = model.encode(query).tolist()# searchresult = index.query(top_k=14, vector=dense, sparse_vector=sparse, include_metadata=True)# used returned product ids to get imagesimgs = [images[int(r["id"])] for r in result["matches"]]imgs

[<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=60x80 at 0x7FBFC46B4610>,
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=60x80 at 0x7FBFAB8FECA0>,
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=60x80 at 0x7FBFA6D774F0>,
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=60x80 at 0x7FBFA4E46970>,
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=60x80 at 0x7FBFA428DF70>,
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=60x80 at 0x7FBFABA41880>,
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=60x80 at 0x7FBFA4C79310>,
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=60x80 at 0x7FBFA6E02730>,
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=60x80 at 0x7FC0522358B0>,
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=60x80 at 0x7FBFC4A00880>,
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=60x80 at 0x7FBFA6FDA190>,
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=60x80 at 0x7FBFC41FBBE0>,
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=60x80 at 0x7FBFC48A9B20>,
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=60x80 at 0x7FBFAABB1AF0>]

We return a list of PIL image objects, to view them we will define a function called display_result.

Python
from IPython.core.display import HTMLfrom io import BytesIOfrom base64 import b64encode# function to display product imagesdef display_result(image_batch): figures = [] for img in image_batch: b = BytesIO() img.save(b, format='png') figures.append(f''' <figure style="margin: 5px !important;"> <img src="{b64encode(b.getvalue()).decode('utf-8')}" style="width: 90px; height: 120px" > </figure> ''') return HTML(data=f''' {''.join(figures)} ''')
And now we can view them:

Python
display_result(imgs)

It's possible to prioritize our search based on sparse vs. dense vector results. To do so, we scale the vectors, for this we'll use a function named hybrid_scale.

Python
def hybrid_scale(dense, sparse, alpha: float): """Hybrid vector scaling using a convex combination alpha * dense + (1 - alpha) * sparse Args: dense: Array of floats representing sparse: a dict of `indices` and `values` alpha: float between 0 and 1 where 0 == sparse only and 1 == dense only """ if alpha < 0 or alpha > 1: raise ValueError("Alpha must be between 0 and 1") # scale sparse and dense vectors to create hybrid search vecs hsparse = { 'indices':

sparse['indices'], 'values': [v * (1 - alpha) for v in sparse['values']] } hdense = [v * alpha for v in dense] return hdense, hsparse
First, we will do a pure sparse vector search by setting the alpha value as 0.

Python
question = "dark blue french connection jeans for men"# scale sparse and dense vectorshdense, hsparse = hybrid_scale(dense, sparse, alpha=0)# searchresult = index.query(top_k=14, vector=hdense, sparse_vector=hsparse, include_metadata=True)# used returned product ids to get imagesimgs = [images[int(r["id"])] for r in result["matches"]]# display the imagesdisplay_result(imgs)

Let's take a look at the description of the result.

Python
for x in result["matches"]: print(x["metadata"]['productDisplayName'])

French Connection Men Blue Jeans
French Connection Men Blue Jeans
French Connection Men Blue Jeans
French Connection Men Blue Jeans
French Connection Men Blue Jeans
French Connection Men Blue Jeans
French Connection Women Blue Jeans
French Connection Women Blue Jeans
French Connection Men Navy Blue Jeans
French Connection Men Black Jeans
French Connection Men Grey Jeans
French Connection Men Black Jeans
French Connection Men Black Jeans
French Connection Men Black Jeans

We can observe that the keyword search returned French Connection jeans but failed to rank the men's French Connection jeans higher than a few of the women's. Now let's do a pure semantic image search by setting the alpha value to 1.

Python
scale sparse and dense vectorshdense, hsparse = hybrid_scale(dense, sparse, alpha=1)# searchresult = index.query(top_k=14, vector=hdense, sparse_vector=hsparse, include_metadata=True)# used returned product ids to get imagesimgs = [images[int(r["id"])] for r in result["matches"]]# display the imagesdisplay_result(imgs)

Python
for x in result["matches"]: print(x["metadata"]['productDisplayName'])

Locomotive Men Radley Blue Jeans
Locomotive Men Race Blue Jeans
Locomotive Men Eero Blue Jeans
Locomotive Men Cam Blue Jeans
Locomotive Men Ian Blue Jeans
French Connection Men Blue Jeans
Locomotive Men Cael Blue Jeans
Locomotive Men Lio Blue Jeans
French Connection Men Blue Jeans
Locomotive Men Rafe Blue Jeans
Locomotive Men Barney Grey Jeans
Spykar Men Actif Fit Low Waist Blue Jeans
Spykar Men Style Essentials Kns 0542 Blue Jeans
Wrangler Men Blue Skanders Jeans

The semantic image search correctly returned blue jeans for men, but mostly failed to match the exact brand we are looking for — French Connection. Now let's set the alpha value to 0.05to try a hybrid search that is slightly more dense than sparse search.

Python
```
# scale sparse and dense vectorshdense, hsparse = hybrid_scale(dense, sparse, alpha=0.05)# searchresult = index.query( top_k=14, vector=hdense, sparse_vector=hsparse, include_metadata=True)# used returned product ids to get imagesimgs = [images[int(r["id"])] for r in result["matches"]]# display the imagesdisplay_result(imgs)
```

Python
```
for x in result["matches"]: print(x["metadata"]['productDisplayName'])
```

French Connection Men Blue Jeans
French Connection Men Blue Jeans
French Connection Men Blue Jeans
Locomotive Men Radley Blue Jeans
French Connection Men Navy Blue Jeans
Locomotive Men Race Blue Jeans
Locomotive Men Cam Blue Jeans
Locomotive Men Eero Blue Jeans
Locomotive Men Ian Blue Jeans
French Connection Men Blue Jeans
French Connection Men Blue paint Stained Regular Fit Jeans
Locomotive Men Cael Blue Jeans
Locomotive Men Lio Blue Jeans
French Connection Men Blue Jeans

By performing a mostly sparse search with some help from our image-based dense vectors, we get a strong number of French Connection jeans, that are for men, and visually are almost all aligned to blue jeans.

Let's try more queries.

Python
query = "small beige handbag for women"# create sparse and dense vectorssparse = bm25.transform_query(query)dense = model.encode(query).tolist()# scale sparse and dense vectors - keyword search firsthdense, hsparse = hybrid_scale(dense, sparse, alpha=0)# searchresult = index.query(top_k=14, vector=hdense, sparse_vector=hsparse, include_metadata=True)# used returned product ids to get imagesimgs = [images[int(r["id"])] for r in result["matches"]]# display the imagesdisplay_result(imgs)

We get a lot of small handbags for women, but they're not beige. Let's use the image dense vectors to weigh the colors higher.

Python
scale sparse and dense vectorshdense, hsparse = hybrid_scale(dense, sparse, alpha=0.05)# searchresult = index.query(top_k=14, vector=hdense, sparse_vector=hsparse, include_metadata=True)# used returned product ids to get imagesimgs = [images[int(r["id"])] for r in result["matches"]]# display the imagesdisplay_result(imgs)

Python
for x in result["matches"]: print(x["metadata"]['productDisplayName'])

Rocky S Women Beige Handbag
Kiara Women Beige Handbag
Baggit Women Beige Handbag
Lino Perros Women Beige Handbag
Kiara Women Beige Handbag
Kiara Women Beige Handbag
French Connection Women Beige Handbag
Rocia Women Beige Handbag
Murcia Women Beige Handbag
Baggit Women Beige Handbag
French Connection Women Beige Handbag
Murcia Women Mauve Handbag
Baggit Women Beige Handbag
Baggit Women Beige Handbag

Here we see better aligned handbags.

Python

scale sparse and dense vectorshdense, hsparse = hybrid_scale(dense, sparse, alpha=1)# searchresult = index.query(top_k=14, vector=hdense, sparse_vector=hsparse, include_metadata=True)# used returned product ids to get imagesimgs = [images[int(r["id"])] for r in result["matches"]]# display the imagesdisplay_result(imgs)

If we go too far with dense vectors, we start to see a few purses, rather than handbags.

Let's run some more interesting queries. This time we will use a product imageto create our dense vector, we'll provide a text query as before that will be used to create the sparse vector, and then we'll select a specific color as per the metadata attached to each image, with metadata filtering.

Python
images[36254]

Python
query = "soft purple topwear"# create the sparse vectorsparse = bm25.transform_query(query)# now create the dense vector using the imagedense = model.encode(images[36254]).tolist()# scale sparse and dense vectorshdense, hsparse = hybrid_scale(dense, sparse, alpha=0.3)# searchresult = index.query(top_k=14, vector=hdense, sparse_vector=hsparse, include_metadata=True)# use returned product ids to get imagesimgs = [images[int(r["id"])] for r in result["matches"]]# display the imagesdisplay_result(imgs)

Our "purple" component isn't being considered strongly enough, let's add this to the metadata filtering:

Python
query = "soft purple topwear"# create the sparse vectorsparse = bm25.transform_query(query)# now create the dense vector using the imagedense = model.encode(images[36254]).tolist()# scale sparse and dense vectorshdense, hsparse = hybrid_scale(dense, sparse, alpha=0.3)# searchresult = index.query(top_k=14, vector=hdense, sparse_vector=hsparse, include_metadata=True, filter={"baseColour": "Purple"} # add to metadata filter)# used returned product ids to get imagesimgs = [images[int(r["id"])] for r in result["matches"]]# display the imagesdisplay_result(imgs)

Let's try with another image:

Python
images[36256]

Python
query = "soft green color topwear"# create the sparse vectorsparse = bm25.transform_query(query)# now create the dense vector using the imagedense = model.encode(images[36256]).tolist()# scale sparse and dense vectorshdense, hsparse =

hybrid_scale(dense, sparse, alpha=0.6)# searchresult = index.query(top_k=14, vector=hdense, sparse_vector=hsparse, include_metadata=True, filter={"baseColour": "Green"} # add to metadata filter)# use returned product ids to get imagesimgs = [images[int(r["id"])] for r in result["matches"]]# display the imagesdisplay_result(imgs)

Here we did not specify the gender but the search results are accurate and we got products matching our query image and description.

Delete the Index

If you're done with the index, we delete it to save resources.

Python
pinecone.delete_index(index_name)

Use Case - Generative QA with OpenAI

In this notebook we will learn how to build a retrieval enhanced generative question-answering systemwith Pinecone and OpenAI. This will allow us to retrieve relevant contexts to our queries from Pinecone, and pass these to a generative OpenAI model to generate an answer backed by real data sources. Required installs for this notebook are:

Python
!pip install -qU openai pinecone-client datasets
Initialize OpenAI connection with:

Python
import openai# get API key from top-right dropdown on OpenAI websiteopenai.api_key = "OPENAI_API_KEY"
For many questions *state-of-the-art (SOTA)*LLMs are more than capable of answering correctly.

Python
query = "who was the 12th person on the moon and when did they land?"# now query text-davinci-003 WITHOUT contextres = openai.Completion.create(engine='text-davinci-003', prompt=query, temperature=0, max_tokens=400, top_p=1, frequency_penalty=0, presence_penalty=0, stop=None)res['choices'][0]['text'].strip()

'The 12th person on the moon was Harrison Schmitt, and he landed on December 11, 1972.'

However, that isn't always the case. Let's first rewrite the above into a simple function so we're not rewriting this every time.

Python
def complete(prompt): # query text-davinci-003 res = openai.Completion.create(engine='text-davinci-003', prompt=prompt, temperature=0, max_tokens=400, top_p=1, frequency_penalty=0, presence_penalty=0, stop=None) return res['choices'][0]['text'].strip()
Now let's ask a more specific question about training a specific type of transformer model called a *sentence-transformer*. The ideal answer we'd be looking for is *"Multiple Negatives Ranking (MNR) loss"*.

Don't worry if this is a new term to you, it isn't required to understand what we're doing or demoing here.

Python
query = ("Which training method should I use for sentence transformers when " + "I only have pairs of related sentences?")complete(query)

'If you only have pairs of related sentences, then the best training method to use for sentence transformers is the supervised learning approach. This approach involves providing the model with labeled data, such as pairs of related sentences, and then training the model to learn the relationships between the sentences. This approach is often used for tasks such as natural language inference, semantic similarity, and paraphrase identification.'

Another common answer produced by the LLM is:

The best training method to use for fine-tuning a pre-trained model with sentence transformers is the Masked Language Model (MLM) training. MLM training involves randomly masking some of the words in a sentence and then training the model to predict the masked words. This helps the model to learn the context of the sentence and better understand the relationships between words.

Both answers seem convincing. Yet, they're both wrong. For the former about supervised learning approachbeing the most suitable. This is completely true, but it's not specific and doesn't answer the question.

For the latter, MLM is typically used in the pretraining step of a transformer model but *cannot*be used to fine-tune a sentence-transformer, and has nothing to do with having *"pairs of related sentences"*.

We have two options for enabling our LLM in understanding and correctly answering this question:

1. We fine-tune the LLM on text data covering the topic mentioned, likely on articles and papers talking about sentence transformers, semantic search training methods, etc.
2. We use Retrieval Augmented Generation (RAG), a technique that implements an information retrieval component to the generation process. Allowing us to retrieve relevant information and feed this information into the generation model as a *secondary*source of information.

We will demonstrate option 2.

Building a Knowledge Base

With option 2, the retrieval of relevant information requires an external *"Knowledge Base"*, a place where we can store and use to efficiently retrieve information. We can think of this as the external *long-term memory*of our LLM.

We will need to retrieve information that is semantically related to our queries, to do this we need to use *"dense vector embeddings"*. These can be thought of as numerical representations of the *meaning*behind our sentences.

There are many options for creating these dense vectors, like open source sentence transformersor OpenAI's ada-002 model. We will use OpenAI's offering in this example.

We have already authenticated our OpenAI connection, to create an embedding we just do:

Python
embed_model = "text-embedding-ada-002"res = openai.Embedding.create(input=["Sample document text goes here", "there will be several phrases in each batch"], engine=embed_model)
In the response reswe will find a JSON-like object containing our new embeddings within the 'data'field.

Python
res.keys()

dict_keys(['object', 'data', 'model', 'usage'])

Inside 'data'we will find two records, one for each of the two sentences we just embedded. Each vector embedding contains 1536dimensions (the output dimensionality of the text-embedding-ada-002model.

Python
len(res['data'])

2

Python
len(res['data'][0]['embedding']), len(res['data'][1]['embedding'])

(1536, 1536)

We will apply this same embedding logic to a dataset containing information relevant to our query (and many other queries on the topics of ML and AI).

Data Preparation

The dataset we will be using is the jamescalam/youtube-transcriptionsfrom Hugging Face *Datasets*. It contains transcribed audio from several ML and tech YouTube channels. We download it with:

Python
from datasets import load_datasetdata = load_dataset('jamescalam/youtube-transcriptions', split='train')data

Using custom data configuration jamescalam--youtube-transcriptions-6a482f3df0aedcdb
Reusing dataset json (/Users/jamesbriggs/.cache/huggingface/datasets/jamescalam___json/jamescalam--youtube-transcriptions-6a482f3df0aedcdb/0.0.0/ac0ca5f5289a6cf108e706efcf040422dbbfa8e658dee6a819f20d76bb84d26b)

Dataset({
features: ['title', 'published', 'url', 'video_id', 'channel_id', 'id', 'text', 'start', 'end'],
num_rows: 208619
})

Python
data[0]

{'title': 'Training and Testing an Italian BERT - Transformers From Scratch #4',
'published': '2021-07-06 13:00:03 UTC',
'url': 'https://youtu.be/35Pdoyi6ZoQ',
'video_id': '35Pdoyi6ZoQ',
'channel_id': 'UCv83tO5cePwHMt1952IVVHw',

'id': '35Pdoyi6ZoQ-t0.0',
'text': 'Hi, welcome to the video.',
'start': 0.0,
'end': 9.36}

The dataset contains many small snippets of text data. We will need to merge many snippets from each video to create more substantial chunks of text that contain more information.

Python
from tqdm.auto import tqdmnew_data = []window = 20 # number of sentences to combinestride = 4 # number of sentences to 'stride' over, used to create overlapfor i in tqdm(range(0, len(data), stride)): i_end = min(len(data)-1, i+window) if data[i]['title'] != data[i_end]['title']: # in this case we skip this entry as we have start/end of two videos continue text = ' '.join(data[i:i_end]['text']) # create the new merged dataset new_data.append({ 'start': data[i]['start'], 'end': data[i_end]['end'], 'title': data[i]['title'], 'text': text, 'id': data[i]['id'], 'url': data[i]['url'], 'published': data[i]['published'], 'channel_id': data[i]['channel_id'] })

0%| | 0/52155 [00:00<?, ?it/s]

Python
new_data[0]

{'start': 0.0,
'end': 74.12,
'title': 'Training and Testing an Italian BERT - Transformers From Scratch #4',
'text': "Hi, welcome to the video. So this is the fourth video in a Transformers from Scratch mini series. So if you haven't been following along, we've essentially covered what you can see on the screen. So we got some data. We built a tokenizer with it. And then we've set up our input pipeline ready to begin actually training our model, which is what we're going to cover in this video. So let's move over to the code. And we see here that we have essentially everything we've done so far. So we've built our input data, our input pipeline. And we're now at a point where we have a data loader, PyTorch data loader, ready. And we can begin training a model with it. So there are a few things to be aware of. So I mean, first, let's just have a quick look at the structure of our data.",
'id': '35Pdoyi6ZoQ-t0.0',
'url': 'https://youtu.be/35Pdoyi6ZoQ',
'published': '2021-07-06 13:00:03 UTC',
'channel_id': 'UCv83tO5cePwHMt1952IVVHw'}

Indexing Data in Vector DB

Now we need a place to store these embeddings and enable a efficient *vector search*through them all. To do that we use Pinecone, we can get a free API keyand enter it below where we will

initialize our connection to Pinecone and create a new index. You can find your environment in the Pinecone consoleunder API Keys.

Python
import pineconeindex_name = 'openai-youtube-transcriptions'# initialize connection to pinecone (get API key at app.pinecone.io)pinecone.init(api_key="PINECONE_API_KEY", environment="YOUR_ENVIRONMENT")# check if index already exists (it shouldn't if this is first time)if index_name not in pinecone.list_indexes(): # if does not exist, create index pinecone.create_index(index_name, dimension=len(res['data'][0]['embedding']), metric='cosine', metadata_config={'indexed': ['channel_id', 'published']})# connect to indexindex = pinecone.Index(index_name)# view index statsindex.describe_index_stats()

{'dimension': 1536,
'index_fullness': 0.0,
'namespaces': {},
'total_vector_count': 0}

We can see the index is currently empty with a total_vector_countof 0. We can begin populating it with OpenAI text-embedding-ada-002built embeddings like so:

Python
from tqdm.auto import tqdmimport datetimefrom time import sleepbatch_size = 100 # how many embeddings we create and insert at oncefor i in tqdm(range(0, len(new_data), batch_size)): # find end of batch i_end = min(len(new_data), i+batch_size) meta_batch = new_data[i:i_end] # get ids ids_batch = [x['id'] for x in meta_batch] # get texts to encode texts = [x['text'] for x in meta_batch] # create embeddings (try-except added to avoid RateLimitError) try: res = openai.Embedding.create(input=texts, engine=embed_model) except: done = False while not done: sleep(5) try: res = openai.Embedding.create(input=texts, engine=embed_model) done = True except: pass embeds = [record['embedding'] for record in res['data']] # cleanup metadata meta_batch = [{ 'start': x['start'], 'end': x['end'], 'title': x['title'], 'text': x['text'], 'url': x['url'], 'published': x['published'], 'channel_id': x['channel_id'] } for x in meta_batch] to_upsert = list(zip(ids_batch, embeds, meta_batch)) # upsert to Pinecone index.upsert(vectors=to_upsert)

0%| | 0/487 [00:00<?, ?it/s]

Making Queries

Now we search, for this we need to create a *query vector*xq:

Python
res = openai.Embedding.create(input=[query], engine=embed_model)# retrieve from Pineconexq = res['data'][0]['embedding']# get relevant contexts (including the questions)res = index.query(xq, top_k=2, include_metadata=True)
Python

res

{'matches': [{'id': 'pNvujJ1XyeQ-t418.88',
'metadata': {'channel_id': 'UCv83tO5cePwHMt1952IVVHw',
'end': 568.4,
'published': datetime.date(2021, 11, 24),
'start': 418.88,
'text': 'pairs of related sentences you can go '
'ahead and actually try training or '
'fine-tuning using NLI with multiple '
"negative ranking loss. If you don't have "
'that fine. Another option is that you have '
'a semantic textual similarity data set or '
'STS and what this is is you have so you '
'have sentence A here, sentence B here and '
'then you have a score from from 0 to 1 '
'that tells you the similarity between '
'those two scores and you would train this '
'using something like cosine similarity '
"loss. Now if that's not an option and your "
'focus or use case is on building a '
'sentence transformer for another language '
'where there is no current sentence '
'transformer you can use multilingual '
'parallel data. So what I mean by that is '
'so parallel data just means translation '
'pairs so if you have for example a English '
'sentence and then you have another '
'language here so it can it can be anything '
"I'm just going to put XX and that XX is "
'your target language you can fine-tune a '
'model using something called multilingual '
'knowledge distillation and what that does '
'is takes a monolingual model for example '
'in English and using those translation '
'pairs it distills the knowledge the '
'semantic similarity knowledge from that '
'monolingual English model into a '
'multilingual model which can handle both '
'English and your target language. So '
"they're three options quite popular very "
'common that you can go for and as a '
'supervised methods the chances are that '
'probably going to outperform anything you '

'do with unsupervised training at least for '
'now. So if none of those sound like '
'something',
'title': 'Today Unsupervised Sentence Transformers, '
'Tomorrow Skynet (how TSDAE works)',
'url': 'https://youtu.be/pNvujJ1XyeQ'},
'score': 0.865277052,
'sparseValues': {},
'values': []},
{'id': 'WS1uVMGhlWQ-t737.28',
'metadata': {'channel_id': 'UCv83tO5cePwHMt1952IVVHw',
'end': 900.72,
'published': datetime.date(2021, 10, 20),
'start': 737.28,
'text': "were actually more accurate. So we can't "
"really do that. We can't use this what is "
'called a mean pooling approach. Or we '
"can't use it in its current form. Now the "
'solution to this problem was introduced by '
'two people in 2019 Nils Reimers and Irenia '
'Gurevich. They introduced what is the '
'first sentence transformer or sentence '
'BERT. And it was found that sentence BERT '
'or S BERT outformed all of the previous '
'Save the Art models on pretty much all '
'benchmarks. Not all of them but most of '
'them. And it did it in a very quick time. '
'So if we compare it to BERT, if we wanted '
'to find the most similar sentence pair '
'from 10,000 sentences in that 2019 paper '
'they found that with BERT that took 65 '
'hours. With S BERT embeddings they could '
'create all the embeddings in just around '
'five seconds. And then they could compare '
'all those with cosine similarity in 0.01 '
"seconds. So it's a lot faster. We go from "
'65 hours to just over five seconds which '
'is I think pretty incredible. Now I think '
"that's pretty much all the context we need "
'behind sentence transformers. And what we '
'do now is dive into a little bit of how '
'they actually work. Now we said before we '
'have the core transform models and what S '
'BERT does is fine tunes on sentence pairs '

'using what is called a Siamese '
'architecture or Siamese network. What we '
'mean by a Siamese network is that we have '
'what we can see, what can view as two BERT '
'models that are identical and the weights '
'between those two models are tied. Now in '
'reality when implementing this we just use '
'a single BERT model. And what we do is we '
'process one sentence, a sentence A through '
'the model and then we process another '
'sentence, sentence B through the model. '
"And that's the sentence pair. So with our "
'cross-linked we were processing the '
'sentence pair together. We were putting '
'them both together, processing them all at '
'once. This time we process them '
'separately. And during training what '
'happens is the weights',
'title': 'Intro to Sentence Embeddings with '
'Transformers',
'url': 'https://youtu.be/WS1uVMGhlWQ'},
'score': 0.85855335,
'sparseValues': {},
'values': []}],
'namespace': ''}

To make this and the next step of building an expanded query simpler, we pack everything into a function named retrieve.

Python
limit = 3750def retrieve(query): res = openai.Embedding.create(input=[query], engine=embed_model) # retrieve from Pinecone xq = res['data'][0]['embedding'] # get relevant contexts res = index.query(xq, top_k=3, include_metadata=True) contexts = [x['metadata']['text'] for x in res['matches']] # build our prompt with the retrieved contexts included prompt_start = ("Answer the question based on the context below.\n\n"+ "Context:\n") prompt_end = (f"\n\nQuestion: {query}\nAnswer:") # append contexts until hitting limit for i in range(1, len(contexts)): if len("\n\n---\n\n".join(contexts[:i])) >= limit: prompt = (prompt_start + "\n\n---\n\n".join(contexts[:i-1]) + prompt_end) break elif i == len(contexts)-1: prompt = (prompt_start + "\n\n---\n\n".join(contexts) + prompt_end) return prompt
Using retrievewe return the expanded query:

Python
first we retrieve relevant items from Pineconequery_with_contexts = retrieve(query)query_with_contexts

"Answer the question based on the context below.\n\nContext:\npairs of related sentences you can go ahead and actually try training or fine-tuning using NLI with multiple negative ranking loss. If you don't have that fine. Another option is that you have a semantic textual similarity data set or STS and what this is is you have so you have sentence A here, sentence B here and then you have a score from from 0 to 1 that tells you the similarity between those two scores and you would train this using something like cosine similarity loss. Now if that's not an option and your focus or use case is on building a sentence transformer for another language where there is no current sentence transformer you can use multilingual parallel data. So what I mean by that is so parallel data just means translation pairs so if you have for example a English sentence and then you have another language here so it can it can be anything I'm just going to put XX and that XX is your target language you can fine-tune a model using something called multilingual knowledge distillation and what that does is takes a monolingual model for example in English and using those translation pairs it distills the knowledge the semantic similarity knowledge from that monolingual English model into a multilingual model which can handle both English and your target language. So they're three options quite popular very common that you can go for and as a supervised methods the chances are that probably going to outperform anything you do with unsupervised training at least for now. So if none of those sound like something\n\n---\n\nwere actually more accurate. So we can't really do that. We can't use this what is called a mean pooling approach. Or we can't use it in its current form. Now the solution to this problem was introduced by two people in 2019 Nils Reimers and Irenia Gurevich. They introduced what is the first sentence transformer or sentence BERT. And it was found that sentence BERT or S BERT outformed all of the previous Save the Art models on pretty much all benchmarks. Not all of them but most of them. And it did it in a very quick time. So if we compare it to BERT, if we wanted to find the most similar sentence pair from 10,000 sentences in that 2019 paper they found that with BERT that took 65 hours. With S BERT embeddings they could create all the embeddings in just around five seconds. And then they could compare all those with cosine similarity in 0.01 seconds. So it's a lot faster. We go from 65 hours to just over five seconds which is I think pretty incredible. Now I think that's pretty much all the context we need behind sentence transformers. And what we do now is dive into a little bit of how they actually work. Now we said before we have the core transform models and what S BERT does is fine tunes on sentence pairs using what is called a Siamese architecture or Siamese network. What we mean by a Siamese network is that we have what we can see, what can view as two BERT models that are identical and the weights between those two models are tied. Now in reality when implementing this we just use a single BERT model. And what we do is we process one sentence, a sentence A through the model and then we process another sentence, sentence B through the model. And that's the sentence pair. So with our cross-linked we were processing the sentence pair together. We were putting them both together, processing them all at once. This time we process them separately. And during training what happens is the weights\n\n---\n\nTransformer-based Sequential Denoising Autoencoder. So what we'll do is jump straight into it and take a look at where we might want to use this training approach and and how we can actually implement it. So the first question we need to ask is do we really need to resort to unsupervised training? Now what we're going to do here is just have a look at a few of the most popular training approaches and what sort of data we need for that. So the first one

we're looking at here is Natural Language Inference or NLI and NLI requires that we have pairs of sentences that are labeled as either contradictory, neutral which means they're not necessarily related or as entailing or as inferring each other. So you have pairs that entail each other so they are both very similar pairs that are neutral and also pairs that are contradictory. And this is the traditional NLI data. Now using another version of fine-tuning with NLI called a multiple negatives ranking loss you can get by with only entailment pairs so pairs that are related to each other or positive pairs and it can also use contradictory pairs to improve the performance of training as well but you don't need it. So if you have positive pairs of related sentences you can go ahead and actually try training or fine-tuning using NLI with multiple negative ranking loss. If you don't have that fine. Another option is that you have a semantic textual similarity data set or STS and what this is is you have so you have sentence A here, sentence B\n\nQuestion: Which training method should I use for sentence transformers when I only have pairs of related sentences?\nAnswer:"

Now we pass our expanded query to the LLM:

Python
then we complete the context-infused querycomplete(query_with_contexts)

'You should use Natural Language Inference (NLI) with multiple negative ranking loss.'

And we get a pretty great answer straight away, specifying to use *multiple-rankings loss*(also called *multiple negatives ranking loss*).

Use Case - Semantic Search

Semantic Search

In this walkthrough we will see how to use Pinecone for semantic search. To begin we must install the required prerequisite libraries:

Python
!pip install -qU pinecone-client[grpc] datasets sentence-transformers

Data Preprocessing

The dataset preparation process requires a few steps:

1. We download the Quora dataset from Hugging Face Datasets.

2. The text content of the dataset is embedded into vectors.
3. We reformat into a (id, vector, metadata)structure to be added to Pinecone.

We will see how steps 1, 2, and 3are done in this section, but we won't implement 2and 3across the whole dataset until we reach the *upsert loop*as we will iteratively perform these two steps.

In either case, this can take some time. If you'd rather skip the data preparation step and get straight to upserts and testing the semantic search functionality, you should refer to the fast notebook.

Python
```
from datasets import load_dataset

dataset = load_dataset('quora', split='train')
dataset
```

```
WARNING:datasets.builder:Found cached dataset quora
(/root/.cache/huggingface/datasets/quora/default/0.0.0/36ba4cd42107f051a158016f1bea6ae3f4
685c5df843529108a54e42d86c1e04)
```

```
Dataset({
features: ['questions', 'is_duplicate'],
num_rows: 404290
})
```

The dataset contains ~400K pairs of natural language questions from Quora.

Python
```
dataset[:5]
```

```
{'questions': [{'id': [1, 2],
'text': ['What is the step by step guide to invest in share market in india?',
'What is the step by step guide to invest in share market?']},
{'id': [3, 4],
'text': ['What is the story of Kohinoor (Koh-i-Noor) Diamond?',
'What would happen if the Indian government stole the Kohinoor (Koh-i-Noor) diamond back?']},
```

{'id': [5, 6],
'text': ['How can I increase the speed of my internet connection while using a VPN?',
'How can Internet speed be increased by hacking through DNS?']},
{'id': [7, 8],
'text': ['Why am I mentally very lonely? How can I solve it?',
'Find the remainder when [math]23^{24}[/math] is divided by 24,23?']},
{'id': [9, 10],
'text': ['Which one dissolve in water quikly sugar, salt, methane and carbon di oxide?',
'Which fish would survive in salt water?']}],
'is_duplicate': [False, False, False, False, False]}

Whether or not the questions are duplicates is not so important, all we need for this example is the text itself. We can extract them all into a single questionslist.

Python
questions = []

for record in dataset['questions']:
questions.extend(record['text'])
# remove duplicates
questions = list(set(questions))
print('\n'.join(questions[:5]))
print(len(questions))

How do I write an essay in 200 words for placement test?
How do actors learn accents?
What are crime control policies? What are examples?
How can a small company grow big?
537362

With our questions ready to go we can move on to demoing steps 2and 3above.

Building Embeddings and Upsert Format

To create our embeddings we will us the MiniLM-L6sentence transformer model. This is a very efficient semantic similarity embedding model from the sentence-transformerslibrary. We initialize it like so:

Python
from sentence_transformers import SentenceTransformer
import torch

device = 'cuda' if torch.cuda.is_available() else 'cpu'
if device != 'cuda':

```
print(f"You are using {device}. This is much slower than using "
"a CUDA-enabled GPU. If on Colab you can change this by "
"clicking Runtime > Change runtime type > GPU.")

model = SentenceTransformer('all-MiniLM-L6-v2', device=device)
model
```

```
SentenceTransformer(
(0): Transformer({'max_seq_length': 256, 'do_lower_case': False}) with Transformer model:
BertModel
(1): Pooling({'word_embedding_dimension': 384, 'pooling_mode_cls_token': False,
'pooling_mode_mean_tokens': True, 'pooling_mode_max_tokens': False,
'pooling_mode_mean_sqrt_len_tokens': False})
(2): Normalize()
)
```

There are *three*interesting bits of information in the above model printout. Those are:

- max_seq_lengthis 256. That means that the maximum number of tokens (like words) that can be encoded into a single vector embedding is 256. Anything beyond this *must*be truncated.
- word_embedding_dimensionis 384. This number is the dimensionality of vectors output by this model. It is important that we know this number later when initializing our Pinecone vector index.
- Normalize(). This final normalization step indicates that all vectors produced by the model are normalized. That means that models that we would typical measure similarity for using *cosine similarity*can also make use of the *dotproduct*similarity metric. In fact, with normalized vectors *cosine*and *dotproduct*are equivalent.

Moving on, we can create a sentence embedding using this model like so:

Python
```
query = 'which city is the most populated in the world?'

xq = model.encode(query)
xq.shape
```

```
(384,)
```

Encoding this single sentence leaves us with a 384dimensional sentence embedding (aligned to the word_embedding_dimensionabove).

To prepare this for upsertto Pinecone, all we do is this:

Python
_id = '0'
metadata = {'text': query}

vectors = [(_id, xq, metadata)]
Later when we do upsert our data to Pinecone, we will be doing so in batches. Meaning vectorswill be a list of (id, embedding, metadata)tuples.

Creating an Index

Now the data is ready, we can set up our index to store it.

We begin by initializing our connection to Pinecone. To do this we need a free API key.

Python
import os
import pinecone

# get api key from app.pinecone.io
PINECONE_API_KEY = os.environ.get('PINECONE_API_KEY') or 'PINECONE_API_KEY'
# find your environment next to the api key in pinecone console
PINECONE_ENV = os.environ.get('PINECONE_ENVIRONMENT') or
'PINECONE_ENVIRONMENT'

pinecone.init(
api_key=PINECONE_API_KEY,
environment=PINECONE_ENV
)
Now we create a new index called semantic-search. It's important that we align the index dimensionand metricparameters with those required by the MiniLM-L6model.

Python
index_name = 'semantic-search-fast'

# only create index if it doesn't exist
if index_name not in pinecone.list_indexes():
pinecone.create_index(
name=index_name,
dimension=model.get_sentence_embedding_dimension(),
metric='cosine'
)

now connect to the index

index = pinecone.GRPCIndex(index_name)
Now we upsert the data, we will do this in batches of 128.

Note:On Google Colab with GPU expected runtime is ~7 minutes. If using CPU this will be significantly longer. If you'd like to get this running faster refer to the fast notebook.

Python
```
from tqdm.auto import tqdm

batch_size = 128

for i in tqdm(range(0, len(questions), batch_size)):
# find end of batch
i_end = min(i+batch_size, len(questions))
# create IDs batch
ids = [str(x) for x in range(i, i_end)]
# create metadata batch
metadatas = [{'text': text} for text in questions[i:i_end]]
# create embeddings
xc = model.encode(questions[i:i_end])
# create records list for upsert
records = zip(ids, xc, metadatas)
# upsert to Pinecone
index.upsert(vectors=records)

# check number of records in the index
index.describe_index_stats()
```

0%| | 0/4199 [00:00<?, ?it/s]

```
{'dimension': 384,
'index_fullness': 0.4,
'namespaces': {'': {'vector_count': 537919}},
'total_vector_count': 537919}
```

Making Queries

Now that our index is populated we can begin making queries. We are performing a semantic search for *similar questions*, so we should embed and search with another question. Let's begin.

Python
```
query = "which city has the highest population in the world?"

# create the query vector
xq = model.encode(query).tolist()

# now query
xc = index.query(xq, top_k=3, include_metadata=True)
xc
```

```
{'matches': [{'id': '229220',
'metadata': {'text': 'Which is the most populated city in the '
'world.?'},
'score': 0.8828482,
'sparse_values': {'indices': [], 'values': []},
'values': []},
{'id': '371413',
'metadata': {'text': 'What is the most populated city in the '
'world?'},
'score': 0.8788713,
'sparse_values': {'indices': [], 'values': []},
'values': []},
{'id': '177352',
'metadata': {'text': 'What are the most populated cities in the '
'world?'},
'score': 0.84194744,
'sparse_values': {'indices': [], 'values': []},
'values': []}],
'namespace': ''}
```

In the returned response xcwe can see the most relevant questions to our particular query. We can reformat this response to be a little easier to read:

Python
```
for result in xc['matches']:
print(f"{round(result['score'], 2)}: {result['metadata']['text']}")
```

```
0.88: Which is the most populated city in the world.?
0.88: What is the most populated city in the world?
```

0.84: What are the most populated cities in the world?

These are clearly very relevant results. All of these questions either share the exact same meaning as our question, or are related. We can make this harder by using more complicated language, but as long as the "meaning" behind our query remains the same, we should see similar results.

Python
```
query = "which urban locations have the highest concentration of homo sapiens?"

# create the query vector
xq = model.encode(query).tolist()

# now query
xc = index.query(xq, top_k=3, include_metadata=True)
for result in xc['matches']:
print(f"{round(result['score'], 2)}: {result['metadata']['text']}")
```

0.64: What are the most populated cities in the world?
0.62: Which is the most populated city in the world.?
0.62: What is the most populated city in the world?

Here we used *very different*language with completely different terms in our query than that of the returned documents. We substituted "city"for "urban location"and "populated"for "concentration of homo sapiens".

Despite these very different terms and *lack*of term overlap between query and returned documents — we get highly relevant results — this is the power of *semantic search*.

You can go ahead and ask more questions above. When you're done, delete the index to save resources:

Python
```
pinecone.delete_index(index_name)
```

Basic Info - What is a Vector Database?

We’re in the midst of the AI revolution. It’s upending any industry it touches, promising great innovations - but it also introduces new challenges. Efficient data processing has become more crucial than ever for applications that involve large language models, generative AI, and semantic search.

All of these new applications rely on **vector embeddings**, a type of data representation that carries within it semantic information that's critical for the AI to gain understanding and maintain a long-term memory they can draw upon when executing complex tasks.

Embeddingsare generated by AI models (such as Large Language Models) and have a large number of attributes or features, making their representation challenging to manage. In the context of AI and machine learning, these features represent different dimensions of the data that are essential for understanding patterns, relationships, and underlying structures.

That is why we need a specialized database designed specifically for handling this type of data. **Vector databases**like Pineconefulfill this requirement by offering optimized storage and querying capabilities for embeddings. Vector databases have the capabilities of a traditional database that are absent in standalone vector indexes and the specialization of dealing with vector embeddings, which traditional scalar-based databases lack.

The challenge of working with vector embeddings is that traditional scalar-based databases can't keep up with the complexity and scale of such data, making it difficult to extract insights and perform real-time analysis. That's where vector databases come into play – they are intentionally designed to handle this type of data and offer the performance, scalability, and flexibility you need to make the most out of your data.

With a vector database, we can add advanced features to our AIs, like semantic information retrieval, long-term memory, and more. The diagram below gives us a better understanding of the role of vector databases in this type of application:

Let's break this down:

1. First, we use the **embedding model**to create **vector embeddings**for the **content**we want to index.
2. The **vector embedding**is inserted into the **vector database**, with some reference to the original **content**the embedding was created from.
3. When the **application**issues a query, we use the same **embedding model**to create embeddings for the query, and use those embeddings to query the **database**for *similar*vector embeddings. And as mentioned before, those similar embeddings are associated with the original **content**that was used to create them.

What's the difference between a vector index and a vector database?

Standalone vector indices like FAISS(Facebook AI Similarity Search) can significantly improve search and retrieval of vector embeddings, but they lack capabilities that exist in any database.

Vector databases, on the other hand, are purpose-built to *manage*vector embeddings, providing several advantages over using standalone vector indices:

1. **Data management:**Vector databases offer well-known and easy-to-use features for data storage, like inserting, deleting, and updating data. This makes managing and maintaining vector data easier than using a standalone vector *index*like FAISS, which requires additional work to integrate with a storage solution.
2. **Metadata storage and filtering:**Vector databases can store metadata associated with each vector entry. Users can then query the database using additional metadata filters for finer-grained queries.
3. **Scalability:**Vector databases are designed to scale with growing data volumes and user demands, providing better support for distributed and parallel processing. Standalone vector indices may require custom solutions to achieve similar levels of scalability (such as deploying and managing them on Kubernetes clusters or other similar systems).
4. **Real-time updates:**Vector databases often support real-time data updates, allowing for dynamic changes to the data, whereas standalone vector indexes may require a full re-indexing process to incorporate new data, which can be time-consuming and computationally expensive.
5. **Backups and collections:**Vector databases handle the routine operation of backing up all the data stored in the database. Pinecone also allows users to selectively choose specific indexes that can be backed up in the form of “collections,” which store the data in that index for later use.
6. **Ecosystem integration:**Vector databases can more easily integrate with other components of a data processing ecosystem, such as ETL pipelines (like Spark), analytics tools (like Tableauand Segment), and visualization platforms (like Grafana) – streamlining the data management workflow. It also enables easy integration with other AI related tools like LangChain, LlamaIndexand ChatGPT’s Plugins.
7. **Data security and access control:**Vector databases typically offer built-in data security features and access control mechanisms to protect sensitive information, which may not be available in standalone vector index solutions.

In short, a vector database provides a superior solution for handling vector embeddings by addressing the limitations of standalone vector indices, such as scalability challenges, cumbersome integration processes, and the absence of real-time updates and built-in security measures, ensuring a more effective and streamlined data management experience.

How does a vector database work?

We all know how traditional databases work (more or less)—they store strings, numbers, and other types of scalar data in rows and columns. On the other hand, a vector database operates on vectors, so the way it’s optimized and queried is quite different.

In traditional databases, we are usually querying for rows in the database where the value usually exactly matches our query. In vector databases, we apply a similarity metric to find a vector that is the **most similar**to our query.

A vector database uses a combination of different algorithms that all participate in Approximate Nearest Neighbor (ANN) search. These algorithms optimize the search through hashing, quantization, or graph-based search.

These algorithms are assembled into a pipeline that provides fast and accurate retrieval of the neighbors of a queried vector. Since the vector database provides **approximate**results, the main trade-offs we consider are between accuracy and speed. The more accurate the result, the slower the query will be. However, a good system can provide ultra-fast search with near-perfect accuracy.

Here's a common pipeline for a vector database:

1. **Indexing**: The vector database indexes vectors using an algorithm such as PQ, LSH, or HNSW (more on these below). This step maps the vectors to a data structure that will enable faster searching.
2. **Querying**: The vector database compares the indexed query vector to the indexed vectors in the dataset to find the nearest neighbors (applying a similarity metric used by that index)
3. **Post Processing**: In some cases, the vector database retrieves the final nearest neighbors from the dataset and post-processes them to return the final results. This step can include re-ranking the nearest neighbors using a different similarity measure.

In the following sections, we will discuss each of these algorithms in more detail and explain how they contribute to the overall performance of a vector database.

Algorithms

Several algorithms can facilitate the creation of a vector index. Their common goal is to enable fast querying by creating a data structure that can be traversed quickly. They will commonly transform the representation of the original vector into a compressed form to optimize the query process.

However, as a user of Pinecone, you don't need to worry about the intricacies and selection of these various algorithms. Pinecone is designed to handle all the complexities and algorithmic decisions behind the scenes, ensuring you get the best performance and results without any hassle. By leveraging Pinecone's expertise, you can focus on what truly matters – extracting valuable insights and delivering powerful AI solutions.

The following sections will explore several algorithms and their unique approaches to handling vector embeddings. This knowledge will empower you to make informed decisions and appreciate the seamless performance Pinecone delivers as you unlock the full potential of your application.

Random Projection

The basic idea behind random projection is to project the high-dimensional vectors to a lower-dimensional space using a **random projection matrix**. We create a matrix of random numbers. The size of the matrix is going to be the target low-dimension value we want. We then calculate the dot product of the input vectors and the matrix, which results in a **projected matrix**that has fewer dimensions than our original vectors but still preserves their similarity.

When we query, we use the same projection matrix to project the query vector onto the lower-dimensional space. Then, we compare the projected query vector to the projected vectors in the database to find the nearest neighbors. Since the dimensionality of the data is reduced, the search process is significantly faster than searching the entire high-dimensional space.

Just keep in mind that random projection is an approximate method, and the projection quality depends on the properties of the projection matrix. In general, the more random the projection matrix is, the better the quality of the projection will be. But generating a truly random projection matrix can be computationally expensive, especially for large datasets. Learn more about random projection.

Product Quantization

Another way to build an index is product quantization (PQ), which is a *lossy*compression technique for high-dimensional vectors (like vector embeddings). It takes the original vector, breaks it up into smaller chunks, simplifies the representation of each chunk by creating a representative “code” for each chunk, and then puts all the chunks back together - without losing information that is vital for similarity operations. The process of PQ can be broken down into four steps: splitting, training, encoding, and querying.

1. **Splitting**-The vectors are broken into segments.
2. **Training**- we build a “codebook” for each segment. Simply put - the algorithm generates a pool of potential “codes” that could be assigned to a vector. In practice - this “codebook” is made up of the center points of clusters created by performing k-means clustering on each of the vector’s segments. We would have the same number of values in the segment codebook as the value we use for the k-means clustering.
3. **Encoding**- The algorithm assigns a specific code to each segment. In practice, we find the nearest value in the codebook to each vector segment after the training is complete. Our PQ code for the segment will be the identifier for the corresponding value in the codebook. We could use as many PQ codes as we’d like, meaning we can pick multiple values from the codebook to represent each segment.

4. **Querying**- When we query, the algorithm breaks down the vectors into sub-vectors and quantizes them using the same codebook. Then, it uses the indexed codes to find the nearest vectors to the query vector.

The number of representative vectors in the codebook is a trade-off between the accuracy of the representation and the computational cost of searching the codebook. The more representative vectors in the codebook, the more accurate the representation of the vectors in the subspace, but the higher the computational cost to search the codebook. By contrast, the fewer representative vectors in the codebook, the less accurate the representation, but the lower the computational cost. Learn more about PQ.

Locality-sensitive hashing

Locality-Sensitive Hashing (LSH) is a technique for indexing in the context of an approximate nearest-neighbor search. It is optimized for speed while still delivering an approximate, non-exhaustive result. LSH maps similar vectors into “buckets” using a set of hashing functions, as seen below:
To find the nearest neighbors for a given query vector, we use the same hashing functions used to “bucket” similar vectors into hash tables. The query vector is hashed to a particular table and then compared with the other vectors in that same table to find the closest matches. This method is much faster than searching through the entire dataset because there are far fewer vectors in each hash table than in the whole space.

It’s important to remember that LSH is an approximate method, and the quality of the approximation depends on the properties of the hash functions. In general, the more hash functions used, the better the approximation quality will be. However, using a large number of hash functions can be computationally expensive and may not be feasible for large datasets. Learn more about LSH.

Hierarchical Navigable Small World (HSNW)

HSNW creates a hierarchical, tree-like structure where each node of the tree represents a set of vectors. The edges between the nodes represent the **similarity**between the vectors. The algorithm starts by creating a set of nodes, each with a small number of vectors. This could be done randomly or by clustering the vectors with algorithms like k-means, where each cluster becomes a node.
The algorithm then examines the vectors of each node and draws an edge between that node and the nodes that have the most similar vectors to the one it has.
When we query an HSNW index, it uses this graph to navigate through the tree, visiting the nodes that are most likely to contain the closest vectors to the query vector. Learn more about HSNW.

Similarity Measures

Building on the previously discussed algorithms, we need to understand the role of similarity measures in vector databases. These measures are the foundation of how a vector database compares and identifies the most relevant results for a given query.

Similarity measures are mathematical methods for determining how similar two vectors are in a vector space. Similarity measures are used in vector databases to compare the vectors stored in the database and find the ones that are most similar to a given query vector.

Several similarity measures can be used, including:

- **Cosine similarity:**measures the cosine of the angle between two vectors in a vector space. It ranges from -1 to 1, where 1 represents identical vectors, 0 represents orthogonal vectors, and -1 represents vectors that are diametrically opposed.
- **Euclidean distance:**measures the straight-line distance between two vectors in a vector space. It ranges from 0 to infinity, where 0 represents identical vectors, and larger values represent increasingly dissimilar vectors.
- **Dot product:**measures the product of the magnitudes of two vectors and the cosine of the angle between them. It ranges from -∞ to ∞, where a positive value represents vectors that point in the same direction, 0 represents orthogonal vectors, and a negative value represents vectors that point in opposite directions.

The choice of similarity measure will have an effect on the results obtained from a vector database. It is also important to note that each similarity measure has its own advantages and disadvantages, and it is important to choose the right one depending on the use case and requirements. Learn more about similarity measures.

Filtering

Every vector stored in the database also includes metadata. In addition to the ability to query for similar vectors, vector databases can also filter the results based on a metadata query. To do this, the vector database usually maintains two indexes: a vector index and a metadata index. It then performs the metadata filtering either before or after the vector search itself, but in either case, there are difficulties that cause the query process to slow down.

The filtering process can be performed either before or after the vector search itself, but each approach has its own challenges that may impact the query performance:

- **Pre-filtering:**In this approach, metadata filtering is done before the vector search. While this can help reduce the search space, it may also cause the system to overlook relevant results that don't match the metadata filter criteria. Additionally, extensive metadata filtering may slow down the query process due to the added computational overhead.

- **Post-filtering:**In this approach, the metadata filtering is done after the vector search. This can help ensure that all relevant results are considered, but it may also introduce additional overhead and slow down the query process as irrelevant results need to be filtered out after the search is complete.

To optimize the filtering process, vector databases use various techniques, such as leveraging advanced indexing methods for metadata or using parallel processing to speed up the filtering tasks. Balancing the trade-offs between search performance and filtering accuracy is essential for providing efficient and relevant query results in vector databases. Learn more about vector search filtering.

Database Operations

Unlike vector indexes, vector databases are equipped with a set of capabilities that makes them better qualified to be used in high scale production settings. Let's take a look at an overall overview of the components that are involved in operating the database.

Performance and Fault tolerance

Performance and fault tolerance are tightly related. The more data we have, the more nodes that are required - and the bigger chance for errors and failures. As is the case with other types of databases, we want to ensure that queries are executed as quickly as possible even if some of the underlying nodes fail. This could be due to hardware failures, network failures, or other types of technical bugs. This kind of failure could result in downtime or even incorrect query results.

To ensure both high performance and fault tolerance, vector databases use sharding and replication apply the following:

1. **Sharding**- partitioning the data across multiple nodes. There are different methods for partitioning the data - for example, it can be partitioned by the similarity of different clusters of data so that similar vectors are stored in the same partition. When a query is made, it is sent to all the shards and the results are retrieved and combined. This is called the "scatter-gather" pattern.
2. **Replication**- creating multiple copies of the data across different nodes. This ensures that even if a particular node fails, other nodes will be able to replace it. There are two main consistency models: *eventual*consistency and *strong*consistency. Eventual consistency allows for temporary inconsistencies between different copies of the data which will improve availability and reduce latency but may result in conflicts and even data loss. On the other hand, strong consistency requires that all copies of the data are updated before a write operation is considered complete. This approach provides stronger consistency but may result in higher latency.

Monitoring

To effectively manage and maintain a vector database, we need a robust monitoring system that tracks the important aspects of the database's performance, health, and overall status. Monitoring is critical for detecting potential problems, optimizing performance, and ensuring smooth production operations. Some aspects of monitoring a vector database include the following:

1. **Resource usage**- monitoring resource usage, such as CPU, memory, disk space, and network activity, enables the identification of potential issues or resource constraints that could affect the performance of the database.
2. **Query performance**- query latency, throughput, and error rates may indicate potential systemic issues that need to be addressed.
3. **System health**- overall system health monitoring includes the status of individual nodes, the replication process, and other critical components.

Access-control

Access control is the process of managing and regulating user access to data and resources. It is a vital component of data security, ensuring that only authorized users have the ability to view, modify, or interact with sensitive data stored within the vector database.

Access control is important for several reasons:

1. **Data protection:**As AI applications often deal with sensitive and confidential information, implementing strict access control mechanisms helps safeguard data from unauthorized access and potential breaches.
2. **Compliance:**Many industries, such as healthcare and finance, are subject to strict data privacy regulations. Implementing proper access control helps organizations comply with these regulations, protecting them from legal and financial repercussions.
3. **Accountability and auditing:**Access control mechanisms enable organizations to maintain a record of user activities within the vector database. This information is crucial for auditing purposes, and when security breaches happen, it helps trace back any unauthorized access or modifications.
4. **Scalability and flexibility:**As organizations grow and evolve, their access control needs may change. A robust access control system allows for seamless modification and expansion of user permissions, ensuring that data security remains intact throughout the organization's growth.

Backups and collections

When all else fails, vector databases offer the ability to rely on regularly created backups. These backups can be stored on external storage systems or cloud-based storage services, ensuring the safety and recoverability of the data. In case of data loss or corruption, these backups can

be used to restore the database to a previous state, minimizing downtime and impact on the overall system. With Pinecone, users can choose to back up specific indexes as well and save them as “collections,” which can later be used to populate new indexes.

API and SDKs

This is where the rubber meets the road: Developers who interact with the database want to do so with an easy-to-use API, using a toolset that is familiar and comfortable. By providing a user-friendly interface, the vector database API layer simplifies the development of high-performance vector search applications.

In addition to the API, vector databases would often provide programming language specific SDKs that wrap the API. The SDKs make it even easier for developers to interact with the database in their applications. This allows developers to concentrate on their specific use cases, such as semantic text search, generative question-answering, hybrid search, image similarity search, or product recommendations, without having to worry about the underlying infrastructure complexities.

Summary

The exponential growth of vector embeddings in fields such as NLP, computer vision, and other AI applications has resulted in the emergence of vector databases as the computation engine that allows us to interact effectively with vector embeddings in our applications.

Vector databases are purpose-built databases that are specialized to tackle the problems that arise when managing vector embeddings in production scenarios. For that reason, they offer significant advantages over traditional scalar-based databases and standalone vector indexes.

In this post, we reviewed the key aspects of a vector database, including how it works, what algorithms it uses, and the additional features that make it operationally ready for production scenarios. We hope this helps you understand the inner workings of vector databases. Luckily, this isn’t something you must know to use Pinecone. Pinecone takes care of all of these considerations (and then some) and frees you to focus on the rest of your application.

Basic Info - What are Vector Embeddings?

Introduction

Vector embeddings are one of the most fascinating and useful concepts in machine learning. They are central to many NLP, recommendation, and search algorithms. If you've ever used things like recommendation engines, voice assistants, language translators, you've come across systems that rely on embeddings.

ML algorithms, like most software algorithms, need numbers to work with. Sometimes we have a dataset with columns of numeric values or values that can be translated into them (ordinal, categorical, etc). Other times we come across something more abstract like an entire document of text. We create vector embeddings, which are just lists of numbers, for data like this to perform various operations with them. A whole paragraph of text or any other object can be reduced to a vector. Even numerical data can be turned into vectors for easier operations. But there is something special about vectors that makes them so useful. This representation makes it possible to translate semantic similarity as perceived by humans to proximity in a vector space.

In other words, when we represent real-world objects and concepts such as images, audio recordings, news articles, user profiles, weather patterns, and political views as vector embeddings, the semantic similarity of these objects and concepts can be quantified by how close they are to each other as points in vector spaces. Vector embedding representations are thus suitable for common machine learning tasks such as clustering, recommendation, and classification.

Source: DeepAI

For example, in a clustering task, clustering algorithms assign similar points to the same cluster while keeping points from different clusters as dissimilar as possible. In a recommendation task, when making recommendations for an unseen object, the recommender system would look for objects that are most similar to the object in question, as measured by their similarity as vector embeddings. In a classification task, we classify the label of an unseen object by the major vote over labels of the most similar objects.

Creating Vector Embeddings

One way of creating vector embeddings is to engineer the vector values using domain knowledge. This is known as feature engineering. For example, in medical imaging, we use medical expertise to quantify a set of features such as shape, color, and regions in an image that capture the semantics. However, engineering vector embeddings requires domain knowledge, and it is too expensive to scale.

Instead of engineering vector embeddings, we often train models to translate objects to vectors. A deep neural network is a common tool for training such models. The resulting embeddings are usually high dimensional (up to two thousand dimensions) and dense (all values are non-zero).

For text data, models such as Word2Vec, GLoVE, and BERT transform words, sentences, or paragraphs into vector embeddings.

Images can be embedded using models such as convolutional neural networks (CNNs), Examples of CNNs include VGG, and Inception. Audio recordings can be transformed into vectors using image embedding transformations over the audio frequencies visual representation (e.g., using its Spectrogram).

Example: Image Embedding with a Convolutional Neural Network

Consider the following example, in which raw images are represented as greyscale pixels. This is equivalent to a matrix (or table) of integer values in the range `0` to `255`. Wherein the value `0` corresponds to a black color and `255` to white color. The image below depicts a greyscale image and its corresponding matrix.

Source: Serena Young

The left sub-image depicts the grayscale pixels, the middle sub-image contains the pixel grayscale values, and the rightmost sub-image defines the matrix. Notice the matrix values define a vector embedding in which its first coordinate is the matrix upper-left cell, then going left-to-right until the last coordinate which corresponds to the lower-right matrix cell.

Such embeddings are great at maintaining the semantic information of a pixel's neighborhood in an image. However, they are very sensitive to transformations like shifts, scaling, cropping and other image manipulation operations. Therefore they are often used as raw inputs to learn more robust embeddings.

Convolutional Neural Network (CNN or ConvNet) is a class of deep learning architectures that are usually applied to visual data transforming images into embeddings.

CNNs are processing the input via hierarchical small local sub-inputs which are termed receptive fields. Each neuron in each network layer processes a specific receptive field from the former layer. Each layer either applies a convolution on the receptive field or reduces the input size, which is called subsampling.

The image below depicts a typical CNN structure. Notice the receptive fields, depicted as sub-squares in each layer, service as an input to a single neuron within the preceding layer. Notice also the subsampling operations reduce the layer size, while the convolution operations extend the layer size. The resulting vector embedding is received via a fully connected layer.

Source: Aphex34, CC BY-SA 4.0

Learning the network weights (i.e., the embedding model) requires a large set of labeled images. The weights are being optimized in a way that images with the same labels are embedded closer compared to images with different labels. Once we learn the CNN embedding model we can transform the images into vectors and store them with a K-Nearest-Neighbor index. Now, given a new unseen image, we can transform it with the CNN model, retrieve its k-most similar vectors, and thus the corresponding similar images.

Although we used images and CNNs as examples, vector embeddings can be created for any kind of data and there are multiple models/methods that we can use to create them.

Using Vector Embeddings

The fact that embeddings can represent an object as a dense vector that contains its semantic information makes them very useful for a wide range of ML applications.

Similarity search is one of the most popular uses of vector embeddings. Search algorithms like KNN and ANN require us to calculate distance between vectors to determine similarity. Vector embeddings can be used to calculate these distances. Nearest neighbor search in turn can be used for tasks like de-duplication, recommendations, anomaly detection, reverse image search, etc.

Even if we don't use embeddings directly for an application, many popular ML models and methods internally rely on them. For example in encoder-decoder architectures, embeddings produced by encoder contain the necessary information for the decoder to produce a result. This architecture is widely used in applications, such as machine translation and caption generation.

Background Info - What is the Tanimoto Similarity?

Tanimoto similarity is a measure of similarity between two sets of data. It is a metric used to compare the similarity of two sets of data, and is often used in machine learning and data science.

The Tanimoto similarity is calculated by taking the intersection of two sets and dividing it by the sum of the sizes of the two sets. This gives a value between 0 and 1, where 0 indicates no similarity and 1 indicates perfect similarity.

Tanimoto similarity is often used in machine learning and data science to compare the similarity of two sets of data. For example, it can be used to compare the similarity of two sets of images, or two sets of text documents. It can also be used to compare the similarity of two sets of data points, such as two sets of customer data. In this case, the Tanimoto similarity can be used to identify customers who are similar in terms of their purchase history or other characteristics.

Tanimoto Similarity vs. Jaccard Coefficient

It is similar to the Jaccard coefficient, which is the ratio of the intersection of two sets to the union of two sets.

The Tanimoto similarity and Jaccard coefficient are both measures of similarity between two sets of data. The main difference between the two is that the Tanimoto similarity takes into account the size of the sets, while the Jaccard Index does not. The Tanimoto similarity is calculated by dividing the number of elements that are common to both sets by the total number of elements in both sets, while the Jaccard Index is calculated by dividing the number of elements that are common to both sets by the number of elements that are unique to either set.

Background Info - What is the Jaccard Similarity?

The Jaccard similarity coefficient, or Jaccard Index, is a measure of similarity between two sets of data. It is calculated by taking the size of the intersection of the two sets and dividing it by the size of the union of the two sets. This gives us a value between 0 and 1, where 0 indicates no similarity and 1 indicates perfect similarity.

The Jaccard similarity coefficient can be used in machine learning and data science in a variety of ways. For example, it can be used to measure the similarity between two documents, or to measure the similarity between two sets of data points. It can also be used to measure the similarity between two clusters of data points, or to measure the similarity between two sets of features. In addition, it can be used to measure the similarity between two sets of words or phrases.

Background Info - What is the Euclidean Distance?

The Euclidean Distance is a measure of the distance between two points in a Euclidean space. It is calculated by taking the square root of the sum of the squared differences between the coordinates of the two points. For example, if two points have coordinates (x1, y1) and (x2, y2), then the Euclidean Distance between them is given by:

d
=
s
q
r
t
(
(
x
2
−
x
1
)
2
+
(
y
2
−
y
1
)
2
)

The Euclidean Distance is a useful measure for many applications, such as clustering, classification, and regression. It is also used in machine learning algorithms such as k-means clustering and k-nearest neighbors.

Background Info - What is the Distance Between Two Vectors?

The distance between two vectors is a measure of how different they are from each other. It is a numerical value that can be calculated using a variety of methods, such as Euclidean distance, dot product, or cosine similarity.

This measure of distance is used in many machine learning and data science applications, such as clustering, classification, and recommendation systems.

For example, in a clustering algorithm, the distance between two vectors can be used to determine which cluster they should be assigned to. In a classification algorithm, the distance between two vectors can be used to determine which class they should be assigned to. In a recommendation system, the distance between two vectors can be used to determine which items should be recommended to a user.

Background Info - What is Maximum a Posteriori Estimation?

Maximum a posteriori (MAP) estimation is a method of estimating the parameters of a statistical model. It is a type of Bayesian estimation, which uses Bayes' theorem to update the probability for a hypothesis as more evidence or information becomes available.

MAP estimation can be used in machine learning and data science to estimate the parameters of a model, such as the weights of a neural network or the coefficients of a linear regression model. MAP estimation can also be used to estimate the probability of a given hypothesis, such as the probability of a given data point belonging to a certain class.

Background Info - What is Inductive Learning?

Inductive learning is a type of machine learning that uses data to make predictions or generalizations about a given problem. It is based on the idea that if a set of data points have certain characteristics, then future data points will also have those characteristics.

This type of learning is used in many areas of machine learning and data science, such as supervised learning, unsupervised learning, and reinforcement learning. For example, in supervised learning, inductive learning can be used to build a model that can predict the outcome of a given problem based on the data it has been trained on.

In unsupervised learning, inductive learning can be used to identify patterns in data and make predictions about future data points. In reinforcement learning, inductive learning can be used to identify the best action to take in a given situation.

Background Info - What is Cosine Similarity?

Cosine similarity is a measure of similarity between two non-zero vectors of an inner product space that measures the cosine of the angle between them. Two vectors with the same orientation have a cosine similarity of
1
, two vectors at
90
°
have a similarity of
0
, and two vectors diametrically opposed have a similarity of
−
1
, independent of their magnitude.

The formula for cosine similarity is:

Cosine similarity
=
(
Dot product of two vectors
)
/
(
Product of Euclidean lengths of two vectors
)
Cosine similarity is a commonly used measure in information retrieval and text mining, where it is often used to measure the similarity of documents. It is also used in collaborative filtering, where it is used to measure the similarity of users or items.

Background Info - What is an Inner Product?

An inner product is a mathematical operation that takes two vectors of the same size and produces a scalar value. It is calculated by multiplying each element of one vector with the corresponding element of the other vector and then summing up the results.

It is used in many areas of mathematics, including linear algebra, calculus, and geometry. In machine learning and data science, inner products are used to measure the similarity between two vectors.

For example, in a neural network, the inner product of two vectors can be used to measure the similarity between two input vectors, or between an input vector and a weight vector.

Inner products can also be used to measure the similarity between two images, or between two text documents.

Background Info - What is a Sparse Vector?

A sparse vector is a vector that contains mostly zeros, with only a few non-zero elements. It is a useful data structure for representing data that is mostly empty or has a lot of zeros. For example, if you have a vector of length 10,000 and only 10 elements are non-zero, then it is a sparse vector.

It is used in machine learning and data science when dealing with large datasets, as it can reduce the amount of memory needed to store the data. For example, if you have a dataset with millions of features where only a few features are important for each data point, you can represent it as a sparse vector, which will take up much less memory than a dense vector.

Sparse vectors can also be used in algorithms such as linear regression, where they can help reduce the computational complexity of the algorithm.

Choosing index type and size

Introduction

When planning your Pinecone deployment, it is important to understand the approximate storage requirements of your vectors to choose the appropriate pod type and number. This page will give guidance on sizing to help you plan accordingly.

As with all guidelines, these considerations are general and may not apply to your specific use case. We caution you to always test your deployment and ensure that the index configuration you are using is appropriate to your requirements.

Collectionsmake it easy to create new versions of your index with different pod types and sizes, and we encourage you to take advantage of that feature to test different configurations. This guide is merely an overview of sizing considerations and should not be taken as a definitive guide.

Users on the Standard, Enterprise, and Enterprise Dedicated plans can contact supportfor further help with sizing and testing.

Overview

There are five main considerations when deciding how to configure your Pinecone index:

- Number of vectors
- Dimensionality of your vectors
- Size of metadata on each vector
- QPS throughput
- Cardinality of indexed metadata

Each of these considerations comes with requirements for index size, pod type, and replication strategy.

Number of vectors

The most important consideration in sizing is the number of vectorsyou plan on working with. As a rule of thumb, a single p1 pod can store approximately 1M vectors, while a s1 pod can store 5M vectors. However, this can be affected by other factors, such as dimensionality and metadata, which are explained below.

Dimensionality of vectors

The rules of thumb above for how many vectors can be stored in a given pod assumes a typical configuration of 768 dimensions per vector. As your individual use case will dictate the dimensionality of your vectors, the amount of space required to store them may necessarily be larger or smaller.

Each dimension on a single vector consumes 4 bytes of memory and storage per dimension, so if you expect to have 1M vectors with 768 dimensions each, that's about 3GB of storage without factoring in metadata or other overhead. Using that reference, we can estimate the typical pod size and number needed for a given index. Table 1 below gives some examples of this.

Table 1: Estimated number of pods per 1M vectors by dimensionality

Pod type	Dimensions	Estimated max vectors per pod

p1	512	1,250,000
	768	1,000,000
	1024	675,000
p2	512	1,250,000
	768	1,100,000
	1024	1,000,000
s1	512	8,000,000
	768	5,000,000
	1024	4,000,000

Pinecone does not support fractional pod deployments, so always round up to the next nearest whole number when choosing your pods.

Queries per second (QPS)

QPS speeds are governed by a combination of the pod typeof the index, the number of replicas, and the top_kvalue of queries. The pod type is the primary factor driving QPS, as the different pod types are optimized for different approaches.

The p1 podsare performance-optimized pods which provide very low query latencies, but hold fewer vectors per pod than s1 pods. They are ideal for applications with low latency requirements (<100ms). The s1 pods are optimized for storage and provide large storage capacity and lower overall costs with slightly higher query latencies than p1 pods. They are ideal for very large indexes with moderate or relaxed latency requirements.

The p2 pod typeprovides greater query throughput with lower latency. They support 200 QPS per replica and return queries in less than 10ms. This means that query throughput and latency are better than s1 and p1, especially for low dimension vectors (<512D).

As a rule, a single p1 pod with 1M vectors of 768 dimensions each and no replicas can handle about 20 QPS. It's possible to get greater or lesser speeds, depending on the size of your metadata, number of vectors, the dimensionality of your vectors, and the top_Kvalue for your search. See Table 2 below for more examples.

Table 2: QPS by pod type and top_kvalue*

Pod type	top_k10	top_k250	top_k1000
p1	30	25	20
p2	150	50	20
s1	10	10	10

*The QPS values in Table 2 represent baseline QPS with 1M vectors and 768 dimensions.

Adding replicas is the simplest way to increase your QPS. Each replica increases the throughput potential by roughly the same QPS, so aiming for 150 QPS using p1 pods means using the primary pod and 5 replicas. Using threading or multiprocessing in your application is also important, as issuing single queries sequentially still subjects you to delays from any underlying latency. The Pinecone gRPC clientcan also be used to increase throughput of upserts.

Metadata cardinality and size

The last consideration when planning your indexes is the cardinality and size of your metadata. While the increases are small when talking about a few million vectors, they can have a real impact as you grow to hundreds of millions or billions of vectors.

Indexes with very high cardinality, like those storing a unique user ID on each vector, can have significant memory requirements, resulting in fewer vectors fitting per pod. Also, if the size of the metadata per vector is larger, the index requires more storage. Limiting which metadata fields are indexed using selective metadata indexingcan help lower memory usage.

Pod sizes

You can also start with one of the larger pod sizes, like p1.x2. Each step up in pod size doubles the space available for your vectors. We recommend starting with x1 pods and scaling as you grow. This way, you don't start with too large a pod size and have nowhere else to go up, meaning you have to migrate to a new index before you're ready.

Example applications

The following examples will showcase how to use the sizing guidelines above to choose the appropriate type, size, and number of pods for your index.

Example 1: Semantic search of news articles

In our first example, we'll use the demo app for semantic searchfrom our documentation. In this case, we're only working with 204,135 vectors. The vectors use 300 dimensions each, well under the general measure of 768 dimensions. Using the rule of thumb above of up to 1M vectors per p1 pod, we can run this app comfortably with a single p1.x1 pod.

Example 2: Facial recognition

For this example, suppose you're building an application to identify customers using facial recognition for a secure banking app. Facial recognition can work with as few as 128 dimensions, but in this case, because the app will be used for access to finances, we want to make sure we're certain that the person using it is the right one. We plan for 100M customers and use 2048 dimensions per vector.

We know from our rules of thumb above that 1M vectors with 768 dimensions fit nicely in a p1.x1 pod. We can just divide those numbers into the new targets to get the ratios we'll need for our pod estimate:

100M / 1M = 100 base p1 pods
2048 / 768 = 2.667 vector ratio
2.667 * 100 = 267 rounding up

So we need 267 p1.x1 pods. We can reduce that by switching to s1 pods instead, sacrificing latency by increasing storage availability. They hold five times the storage of p1.x1, so the math is simple:

267 / 5 = 54 rounding up

So we estimate that we need 54 s1.x1 pods to store very high dimensional data for the face of each of the bank's customers.

Index Operations - configure_index

PATCHhttps://controller.{environment}.pinecone.io/databases/{indexName}

This operation specifies the pod type and number of replicas for an index.
PATH PARAMS
indexName
string
required

The name of the index
BODY PARAMS

The desired pod type and replica configuration for the index.
replicas
integer

The desired number of replicas for the index.
pod_type
string

The new pod type for the index. One of s1, p1, or p2appended with .and one of x1, x2, x4, or x8.
RESPONSES
202
The index has been successfully updated

400
Bad request,not enough quota.
404
Index not found.
500
Internal error. Can be caused by invalid parameters.

Index Operations - delete_index

delete_index

DELETEhttps://controller.{environment}.pinecone.io/databases/{indexName}

This operation deletes an existing index.
PATH PARAMS
indexName
string
required

The name of the index
RESPONSES
202
The index has been successfully deleted.

404
Index not found.

500
Internal error. Can be caused by invalid parameters.

Index Operations - describe_index

describe_index

GEThttps://controller.{environment}.pinecone.io/databases/{indexName}

Get a description of an index.
PATH PARAMS
indexName
string
required

The name of the index
RESPONSES
200
Configuration information and deployment status of the index

404
Index not found
500
Internal error. Can be caused by invalid parameters.

Index Operations - create_index

create_index

POSThttps://controller.{environment}.pinecone.io/databases

This operation creates a Pinecone index. You can use it to specify the measure of similarity, the dimension of vectors to be stored in the index, the numbers of replicas to use, and more.
BODY PARAMS

name
string
required

The name of the index to be created. The maximum length is 45 characters.
dimension
integer
required

The dimensions of the vectors to be inserted in the index
metric
string

The distance metric to be used for similarity search. You can use 'euclidean', 'cosine', or 'dotproduct'.
pods
integer

The number of pods for the index to use,including replicas.
replicas
integer

The number of replicas. Replicas duplicate your index. They provide higher availability and throughput.
pod_type
string

The type of pod to use. One of s1, p1, or p2appended with .and one of x1, x2, x4, or x8.
metadata_config
object | null
Configuration for the behavior of Pinecone's internal metadata index. By default, all metadata is indexed; when metadata_configis present, only specified metadata fields are indexed. To specify metadata fields to index, provide a JSON object of the following form:
{"indexed": ["example_metadata_field"]}

METADATA_CONFIG OBJECT | NULL

source_collection
string

The name of the collection to create an index from
SHOW DEPRECATED
RESPONSES
201

The index has been successfully created

400
Bad request. Encountered when request exceeds quota or an invalid index name.
409
Index of given name already exists.
500
Internal error. Can be caused by invalid parameters.

Index Operations - list_indexes

GEThttps://controller.{environment}.pinecone.io/databases

This operation returns a list of your Pinecone indexes.
RESPONSE
200
This operation returns a list of all the indexes that you have previously created, and which are associated with the given API key

---

Index Operations - delete_collection

delete_collection

DELETE https://controller.{environment}.pinecone.io/collections/{collectionName}

This operation deletes an existing collection.
PATH PARAMS
collectionName
string
required

The name of the collection
RESPONSES
202
The index has been successfully deleted.

404

Collection not found.
500
Internal error. Can be caused by invalid parameters.

Index Operations - describe_collection

describe_collection

GEThttps://controller.{environment}.pinecone.io/collections/{collectionName}

Get a description of a collection.
PATH PARAMS
collectionName
string
required

The name of the collection
RESPONSES
200
Configuration information and deployment status of the index

404
Index not found.
500
Internal error. Can be caused by invalid parameters.

Index Operations - create_collection

POSThttps://controller.{environment}.pinecone.io/collections

This operation creates a Pinecone collection.
BODY PARAMS
name
string
required

The name of the collection to be created.

source
string
required

The name of the source index to be used as the source for the collection.
RESPONSES
201
The collection has been successfully created.

400
Bad request. Request exceeds quota or collection name is invalid.
409
A collection with the name provided already exists.
500
Internal error. Can be caused by invalid parameters.

Index Operations - list_collections

GET https://controller.{environment}.pinecone.io/collections

This operation returns a list of your Pinecone collections.

Vector Operations - Upsert

Upsert

POSThttps://index_name-project_id.svc.environment.pinecone.io/vectors/upsert
The Upsertoperation writes vectors into a namespace.
If a new value is upserted for an existing vector id, it will overwrite the previous value.
BODY PARAMS
vectors
array of objects
required

An array containing the vectors to upsert. Recommended batch limit is 100 vectors.
ADD OBJECT

namespace
string

This is the namespace name where you upsert vectors.
RESPONSES
200
A successful response.

Default
An unexpected error response.

Vector Operations - Fetch

Fetch

GEThttps://index_name-project_id.svc.environment.pinecone.io/vectors/fetch
The Fetch operation looks up and returns vectors, by ID, from a single namespace.
The returned vectors include the vector data and/or metadata.
QUERY PARAMS
ids
array of strings
required

The vector IDs to fetch. Does not accept values containing spaces.
ADD STRING

namespace
string
RESPONSES
200
A successful response.

Default
An unexpected error response.

Vector Operations - Update

Update

POSThttps://index_name-project_id.svc.environment.pinecone.io/vectors/update
The Updateoperation updates vector in a namespace.
If a value is included, it will overwrite the previous value.
If a set_metadata is included, the values of the fields specified in it will be added or overwrite the previous value.
BODY PARAMS
id
string
required

Vector's unique id.
values
array of floats

Vector data.
ADD FLOAT

sparseValues
object

Vector sparse data. Represented as a list of indices and a list of corresponded values, which must be the same length.
SPARSEVALUES OBJECT

setMetadata
object

Metadata to *set*for the vector.
SETMETADATA OBJECT

namespace
string

The namespace containing the vector to update.
RESPONSES
200
A successful response.

Default
An unexpected error response.

Vector Operations - Delete

Delete

POSThttps://index_name-project_id.svc.environment.pinecone.io/vectors/delete
The Delete operation deletes vectors, by id, from a single namespace.
You can delete items by their id, from a single namespace.
BODY PARAMS
ids
array of strings

Vectors to delete.
ADD STRING

deleteAll
boolean

This indicates that all vectors in the index namespace should be deleted.
truefalse
namespace
string

The namespace to delete vectors from, if applicable.
filter
object
If specified, the metadata filter here will be used to select the vectors to delete. This is mutually exclusive
with specifying ids to delete in the ids param or using delete_all=True.
See https://www.pinecone.io/docs/metadata-filtering/.
FILTER OBJECT

RESPONSES
200
A successful response.

Default
An unexpected error response.

Vector Operations - Query

POSThttps://index_name-project_id.svc.environment.pinecone.io/query
The Queryoperation searches a namespace, using a query vector.
It retrieves the ids of the most similar items in a namespace, along with their similarity scores.
BODY PARAMS
namespace
string

The namespace to query.
topK
int64
required

The number of results to return for each query.
filter
object

The filter to apply. You can use vector metadata to limit your search. See https://www.pinecone.io/docs/metadata-filtering/.
FILTER OBJECT

includeValues
boolean

Indicates whether vector values are included in the response.
truefalse
includeMetadata
boolean

Indicates whether metadata is included in the response as well as the ids.
truefalse
vector
array of floats

The query vector. This should be the same length as the dimension of the index being queried.
Each query()request can contain only one of the parameters idor vector.

ADD FLOAT

sparseVector
object

Vector sparse data. Represented as a list of indices and a list of corresponded values, which must be the same length.
SPARSEVECTOR OBJECT

id
string

The unique ID of the vector to be used as a query vector. Each query()request can contain only one of the parameters queries, vector, or id.
DEPRECATED PARAMETERSHIDE
queries
array of objects
DEPRECATED

DEPRECATED. The query vectors. Each query()request can contain only one of the parameters queries, vector, or id.
ADD OBJECT

RESPONSES
200
A successful response.

Default
An unexpected error response.

Vector Operations - DescribeIndexStats

POSThttps://index_name-project_id.svc.environment.pinecone.io/describe_index_stats

The DescribeIndexStatsoperation returns statistics about the index's contents, including the vector count per namespace and the number of dimensions.
BODY PARAMS
filter
object
If this parameter is present, the operation only returns statistics
for vectors that satisfy the filter.

See https://www.pinecone.io/docs/metadata-filtering/.
FILTER OBJECT

RESPONSES
200
A successful response.

Default
An unexpected error response.

Node.JS Client

This page provides installation instructions, usage examples, and a reference for the Pinecone Node.JS client.

⚠️Warning

This is a public preview("Beta") client. Test thoroughly beforeusing this client for production workloads. No SLAs or technical supportcommitments are provided for this client. Expect potential breakingchanges in future releases.

Getting Started

Installation

Use the following shell command to install the Node.JS client for use with Node.JS versions 17 and above:

Shell
npm install @pinecone-database/pinecone
Alternatively, you can install Pinecone with Yarn:

Shell
yarn add @pinecone-database/pinecone

Usage

Initialize the client

To initialize the client, instantiate the PineconeClientclass and call the initmethod. The initmethod takes an object with the apiKeyand environmentproperties:

JavaScript
```
import { PineconeClient } from "@pinecone-database/pinecone";const pinecone = new PineconeClient();await pinecone.init({ environment: "YOUR_ENVIRONMENT", apiKey: "YOUR_API_KEY",});
```

Create index

The following example creates an index without a metadata configuration. By default, Pinecone indexes all metadata.

JavaScript
```
await pinecone.createIndex({ createRequest: { name: "example-index", dimension: 1024, },});
```

The following example creates an index that only indexes the "color" metadata field. Queries against this index cannot filter based on any other metadata field.

JavaScript
```
await pinecone.createIndex({ createRequest: { name: "example-index-2", dimension: 1024, metadataConfig: { indexed: ["color"], }, },});
```

List indexes

The following example logs all indexes in your project.

JavaScript
```
const indexesList = await pinecone.listIndexes();
```

Describe index

The following example logs information about the index example-index.

JavaScript
```
const indexDescription = await pinecone.describeIndex({ indexName: "example-index",});
```

Delete index

The following example deletes example-index.

JavaScript
```
await pinecone.deleteIndex({ indexName: "example-index",});
```

Scale replicas

The following example sets the number of replicas and pod type for example-index.

JavaScript
```
await pinecone.configureIndex({ indexName: "example-index", patchRequest: { replicas: 2, podType: "p2", },});
```

Describe index statistics

The following example returns statistics about the index example-index.

JavaScript
```
const index = pinecone.Index("example-index");const indexStats = index.describeIndexStats({ describeIndexStatsRequest: { filter: {}, },});
```

Upsert vectors
The following example upserts vectors to example-index.

JavaScript
```
const index = pinecone.Index("example-index");const upsertRequest = { vectors: [ { id: "vec1", values: [0.1, 0.2, 0.3, 0.4], metadata: { genre: "drama", }, }, { id: "vec2", values: [0.2, 0.3, 0.4, 0.5], metadata: { genre: "action", }, }, ], namespace: "example-namespace",};const upsertResponse = await index.upsert({ upsertRequest });
```

Query an index
The following example queries the index example-indexwith metadata filtering.

JavaScript
```
const index = pinecone.Index("example-index");const queryRequest = { vector: [0.1, 0.2, 0.3, 0.4], topK: 10, includeValues: true, includeMetadata: true, filter: { genre: { $in: ["comedy", "documentary", "drama"] }, }, namespace: "example-namespace",};const queryResponse = await index.query({ queryRequest });
```

Delete vectors
The following example deletes vectors by ID.

JavaScript
```
const index = pinecone.Index("example-index");await index.delete1({ ids: ["vec1", "vec2"], namespace: "example-namespace",});
```

Fetch vectors
The following example fetches vectors by ID.

JavaScript
```
const index = pinecone.Index("example-index");const fetchResponse = await index.fetch({ ids: ["vec1", "vec2"], namespace: "example-namespace",});
```

Update vectors
The following example updates vectors by ID.

JavaScript

```
const index = pinecone.Index("example-index");const updateRequest = { id: "vec1", values: [0.1, 0.2, 0.3, 0.4], setMetadata: { genre: "drama" }, namespace: "example-namespace",};const updateResponse = await index.update({ updateRequest });
```

Create collection

The following example creates the collection example-collectionfrom example-index.

JavaScript

```
const createCollectionRequest = { name: "example-collection", source: "example-index",};await pinecone.createCollection({ createCollectionRequest,});
```

List collections

The following example returns a list of the collections in the current project.

JavaScript

```
const collectionsList = await pinecone.listCollections();
```

Describe a collection

The following example returns a description of the collection example-collection.

JavaScript

```
const collectionDescription = await pinecone.describeCollection({ collectionName: "example-collection",});
```

Delete a collection

The following example deletes the collection example-collection.

JavaScript

```
await pinecone.deleteCollection({ collectionName: "example-collection",});
```

Reference

For the REST API or other clients, see the API reference.

init()

pinecone.init(configuration: PineconeClientConfiguration)

Initialize the Pinecone client.

Parameters	Type	Description
configuration	PineconeClientConfiguration	The configuration for the Pinecone client.

Types

PineconeClientConfiguration

Parameters	Type	Description
apiKey	string	The API key for the Pinecone service.
environment	string	The cloud environment of your Pinecone project.

Example:

JavaScript
import { PineconeClient } from "@pinecone-database/pinecone";const pinecone = new PineconeClient();await pinecone.init({ apiKey: "YOUR_API_KEY", environment: "YOUR_ENVIRONMENT",});

configureIndex()

pinecone.configure_index(indexName: string, patchRequest?: PatchRequest)

Configure an index to change pod type and number of replicas.

Parameters	Type	Description
requestParameters	ConfigureIndexRequest	Index configuration parameters.

Types

ConfigureIndexRequest

Parameters	Type	Description
indexName	string	The name of the index.
patchRequest	PatchRequest	(Optional) Patch request parameters.

PatchRequest

Parameters	Type	Description
replicas	number	(Optional) The number of replicas to configure for this index.
podType	string	(Optional) The new pod type for the index. One of s1, p1, or p2appended with .and one of x1, x2, x4, or x8.

Example:

JavaScript

const newNumberOfReplicas = 4;const newPodType = "s1.x4";await pinecone.configureIndex({ indexName: "example-index", patchRequest: { replicas: newNumberOfReplicas, podType: newPodType, },});

createCollection()

pinecone.createCollection(requestParameters: CreateCollectionOperationRequest)

Create a collection from an index.

Parameters	Type	Description
requestParameters	CreateCollectionOperationRequest	Create collection operation wrapper

Types

CreateCollectionOperationRequest

Parameters	Type	Description
createCollectionRequest	CreateCollectionRequest	Collection request parameters.

CreateCollectionRequest

Parameters	Type	Description
name	string	The name of the collection to be created.
source	string	The name of the source index to be used as the source for the collection.

Example:

JavaScript
await pinecone.createCollection({ createCollectionRequest: { name: "example-collection", source: "example-index", },});

createIndex()

pinecone.createIndex(requestParameters?: CreateIndexRequest)

Create an index.

Parameters	Type	Description
requestParameters	CreateIndexRequest	Create index operation wrapper

Types

CreateIndexRequest

Parameters	Type	Description
createRequest	CreateRequest	Create index request parameters

CreateRequest

Parameters	Type	Description
name	str	The name of the index to be created. The maximum length is 45 characters.
dimension	integer	The dimensions of the vectors to be inserted in the index.
metric	str	(Optional) The distance metric to be used for similarity search: 'euclidean', 'cosine', or 'dotproduct'.
pods	int	(Optional) The number of pods for the index to use, including replicas.
replicas	int	(Optional) The number of replicas.
pod_type	str	(Optional) The new pod type for the index. One of s1, p1, or p2appended with .and one of x1, x2, x4, or x8.
metadata_config	object	(Optional) Configuration for the behavior of Pinecone's internal metadata index. By default, all metadata is indexed; when metadata_config is present, only specified metadata fields are indexed. To specify metadata fields to index, provide a JSON object of the following form: {"indexed": ["example_metadata_field"]}
source_collection	str	(Optional) The name of the collection to create an index from.

Example:

JavaScript
// The following example creates an index without a metadata// configuration. By default, Pinecone indexes all metadata.await pinecone.createIndex({ createRequest: { name: "pinecone-index", dimension: 1024, },});// The following example creates an index that only indexes// the 'color' metadata field. Queries against this index// cannot filter based on any other metadata field.await pinecone.createIndex({ createRequest: { name: "example-index-2", dimension: 1024, metadata_config: { indexed: ["color"], }, },});

deleteCollection()

pinecone.deleteCollection(requestParameters: DeleteCollectionRequest)

Delete an existing collection.

Types

Parameters	Type	Description
requestParameters	DeleteCollectionRequest	Delete collection request parameters

DeleteCollectionRequest

Parameters	Type	Description
collectionName	string	The name of the collection to delete.

Example:

JavaScript
```
await pinecone.deleteCollection({ collectionName: "example-collection",});
```

deleteIndex()

pinecone.deleteIndex(requestParameters: DeleteIndexRequest)

Delete an index.

Types

Parameters	Type	Description
requestParameters	DeleteIndexRequest	Delete index request parameters

DeleteIndexRequest

Parameters	Type	Description
indexName	string	The name of the index to delete.

Example:

JavaScript
```
await pinecone.deleteIndex({ indexName: "example-index",});
```

describeCollection()

pinecone.describeCollection(requestParameters: DescribeCollectionRequest)

Get a description of a collection.

Types

Parameters	Type	Description
requestParameters	DescribeCollectionRequest	Describe collection request parameters

DescribeCollectionRequest

Parameters	Type	Description
collectionName	string	The name of the collection.

Example:

JavaScript
const collectionDescription = await pinecone.describeCollection({ collectionName: "example-collection",});
Return:

- collectionMeta: objectConfiguration information and deployment status of the collection.
    - name: stringThe name of the collection.
    - size: integerThe size of the collection in bytes.
    - status: stringThe status of the collection.

describeIndex()

pinecone.describeIndex(requestParameters: DescribeIndexRequest)

Get a description of an index.

Types

Parameters	Type	Description
requestParameters	DescribeIndexRequest	Describe index request parameters

DescribeIndexRequest

Parameters	Type	Description
indexName	string	The name of the index.

Types
Returns:

- database: object
- name: stringThe name of the index.
- dimension: integerThe dimensions of the vectors to be inserted in the index.
- metric: stringThe distance metric used for similarity search: 'euclidean', 'cosine', or 'dotproduct'.
- pods: integerThe number of pods the index uses, including replicas.
- replicas: integerThe number of replicas.
- pod_type: stringThe pod type for the index. One of s1, p1, or p2appended with .and one of x1, x2, x4, or x8.
- metadata_config: objectConfiguration for the behavior of Pinecone's internal metadata index. By default, all metadata is indexed; when metadata_config is present, only specified metadata fields are indexed. To specify metadata fields to index, provide a JSON object of the following form: {"indexed": ["example_metadata_field"]}
- status: object
- ready: booleanWhether the index is ready to serve queries.
- state: stringOne of Initializing, ScalingUp, ScalingDown, Terminating, or Ready.

Example:

JavaScript
```
const indexDescription = await pinecone.describeIndex({ indexName: "example-index",});
```

listCollections

pinecone.listCollections()

Return a list of the collections in your project.

Example:

JavaScript
```
const collections = await pinecone.listCollections();
```
Returns:

- arrayof stringsThe names of the collections in your project.

listIndexes

pinecone.listIndexes()

Return a list of your Pinecone indexes.

Returns:

- arrayof stringsThe names of the indexes in your project.

Example:

JavaScript
```
const indexesList = await pinecone.listIndexes();
```

Index()

pinecone.Index(indexName: string)

Construct an Index object.

Parameters	Type	Description
indexName	string	The name of the index.

Example:

JavaScript
```
const index = pinecone.Index("example-index");
```

Index.delete1()

index.delete(requestParameters: Delete1Request)

Delete items by their ID from a single namespace.

Parameters	Type	Description
requestParameters	Delete1Request	Delete request parameters

Types

Delete1Request

Parameters	Type	Description
ids	Array	(Optional) The IDs of the items to delete.
deleteAll	boolean	(Optional) Indicates that all vectors in the index namespace should be deleted.

namespace	str	(Optional) The namespace to delete vectors from, if applicable.

Types

Example:

JavaScript
```
await index.delete1({ ids: ["example-id-1", "example-id-2"], namespace:
"example-namespace",});
```

Index.describeIndexStats()

index.describeIndexStats(requestParameters: DescribeIndexStatsOperationRequest)

Returns statistics about the index's contents, including the vector count per namespace and the number of dimensions.

Parameters	Type	Description
requestParameters	DescribeIndexStatsOperationRequest	Describe index stats request wrapper

Types

DescribeIndexStatsOperationRequest

Parameters	Type	Description
describeIndexStatsRequest	DescribeIndexStatsRequest	Describe index stats request parameters

DescribeIndexStatsRequest

parameter	Type	Description
filter	object	(Optional) A metadata filter expression.

Returns:

- namespaces: objectA mapping for each namespace in the index from the namespace name to a

  summary of its contents. If a metadata filter expression is present, the summary will reflect only vectors matching that expression.
- dimension: int64The dimension of the indexed vectors.

- indexFullness: floatThe fullness of the index, regardless of whether a metadata filter expression was passed. The granularity of this metric is 10%.
- totalVectorCount: int64The total number of vectors in the index.

Example:

JavaScript
const indexStats = await index.describeIndexStats({ describeIndexStatsRequest: {},});
Read more about filteringfor more detail.

Index.fetch()

index.fetch(requestParameters: FetchRequest)

The Fetch operation looks up and returns vectors, by ID, from a single namespace. The returned vectors include the vector data and metadata.

Parameters	Type	Description
requestParameters	FetchRequest	Fetch request parameters

Types

FetchRequest

Parameters	Type	Description
ids	Array	The vector IDs to fetch. Does not accept values containing spaces.
namespace	string	(Optional) The namespace containing the vectors.

Returns:

- vectors: objectContains the vectors.
- namespace: stringThe namespace of the vectors.

Example:

JavaScript
const fetchResponse = await index.fetch({ ids: ["example-id-1", "example-id-2"], namespace: "example-namespace",});

Index.query()

index.query(requestParameters: QueryOperationRequest)

Search a namespace using a query vector. Retrieves the ids of the most similar items in a namespace, along with their similarity scores.

Parameters	Type	Description
requestParameters	QueryOperationRequest	The query operation request wrapper.

Types

Parameters	Type	Description
queryRequest	QueryRequest	The query operation request.

QueryRequest

Parameter	Type	Description
namespace	string	(Optional) The namespace to query.
topK	number	The number of results to return for each query.
filter	object	(Optional) The filter to apply. You can use vector metadata to limit your search. See https://www.pinecone.io/docs/metadata-filtering/.
includeValues	boolean	(Optional) Indicates whether vector values are included in the response. Defaults to false.
includeMetadata	boolean	(Optional) Indicates whether metadata is included in the response as well as the ids. Defaults to false.
vector	Array	(Optional) The query vector. This should be the same length as the dimension of the index being queried. Each query()request can contain only one of the parameters idor vector.
id	string	(Optional) The unique ID of the vector to be used as a query vector. Each query()request can contain only one of the parameters vectoror id.

Example:

JavaScript
```
const queryResponse = await index.query({ queryRequest: { namespace: "example-namespace", topK: 10, filter: { genre: { $in: ["comedy", "documentary", "drama"] } }, includeValues: true, includeMetadata: true, vector: [0.1, 0.2, 0.3, 0.4], },});
```

Index.update()

index.update(requestParameters: UpdateOperationRequest)

Updates vectors in a namespace. If a value is included, it will overwrite the previous value.If setMetadatais included in the updateRequest, the values of the fields specified in it will be added or overwrite the previous value.

Parameters	Type	Description
requestParameters	UpdateOperationRequest	The update operation wrapper

Types

UpdateOperationRequest

Parameters	Type	Description
updateRequest	UpdateRequest	The update request.

UpdateRequest

Parameter	Type	Description
id	string	The vector's unique ID.
values	Array	(Optional) Vector data.
setMetadata	object	(Optional) Metadata to set for the vector.
namespace	string	(Optional) The namespace containing the vector.

Example:

JavaScript
const updateResponse = await index.update({ updatedRequest: { id: "vec1", values: [0.1, 0.2, 0.3, 0.4], setMetadata: { genre: "drama", }, namespace: "example-namespace", },});

Index.upsert()

index.upsert(requestParameters: UpsertOperationRequest)

Writes vectors into a namespace. If a new value is upserted for an existing vector ID, it will overwrite the previous value.

Parameters	Type	Description

requestParameters	UpsertOperationRequest	Upsert operation wrapper

Types

UpsertOperationRequest

Parameters	Type	Description
upsertRequest	UpsertRequest	The upsert request.

UpsertRequest

| Parameter | Type | Description || vectors| Array | An array containing the vectors to upsert. Recommended batch limit is 100 vectors.id(str) - The vector's unique id.values([float]) - The vector data.metadata(object) - (Optional) Metadata for the vector. || namespace| string | (Optional) The namespace name to upsert vectors. |

Vector

Parameter	Type	Description
id	string	The vector's unique ID.
values	Array	Vector data.
metadata	object	(Optional) Metadata for the vector.

Returns:

- upsertedCount: int64The number of vectors upserted.

Example:

JavaScript
```
const upsertResponse = await index.upsert({ upsertRequest: { vectors: [ { id: "vec1", values: [0.1, 0.2, 0.3, 0.4], metadata: { genre: "drama", }, }, { id: "vec2", values: [0.1, 0.2, 0.3, 0.4], metadata: { genre: "comedy", }, }, ], namespace: "example-namespace", },});
```

Python Client

This page provides installation instructions, usage examples, and a reference for the Pinecone Python client.

Getting Started

Installation

Use the following shell command to install the Python client for use with Python versions 3.6+:

Python
pip3 install pinecone-client
Alternatively, you can install Pinecone in a Jupyter notebook:

Python
!pip3 install pinecone-client
We strongly recommend installing Pinecone in a virtual environment. For more information on using Python virtual environments, see:

- PyPA Python Packaging User Guide
- Python Virtual Environments: A Primer

There is a gRPC flavor of the client available, which comes with more dependencies in return for faster upload speeds. To install it, use the following command:

Python
pip3 install "pinecone-client[grpc]"
For the latest development version:

Python
pip3 install git+https://git@github.com/pinecone-io/pinecone-python-client.git
For a specific development version:

Python
pip3 install git+https://git@github.com/pinecone-io/pinecone-python-client.gitpip3 install git+https://git@github.com/pinecone-io/pinecone-python-client.git@example-branch-namepip3 install git+https://git@github.com/pinecone-io/pinecone-python-client.git@259deff

Usage

Create index

The following example creates an index without a metadata configuration. By default, Pinecone indexes all metadata.

Python

import pineconepinecone.init(api_key="YOUR_API_KEY", environment="YOUR_ENVIRONMENT")pinecone.create_index("example-index", dimension=1024)
The following example creates an index that only indexes the "color" metadata field. Queries against this index cannot filter based on any other metadata field.

Python
metadata_config = { "indexed": ["color"]}pinecone.create_index("example-index-2", dimension=1024, metadata_config=metadata_config)

List indexes
The following example returns all indexes in your project.

Python
import pineconepinecone.init(api_key="YOUR_API_KEY", environment="YOUR_ENVIRONMENT")active_indexes = pinecone.list_indexes()

Describe index
The following example returns information about the index example-index.

Python
import pineconepinecone.init(api_key="YOUR_API_KEY", environment="YOUR_ENVIRONMENT")index_description = pinecone.describe_index("example-index")

Delete index
The following example deletes example-index.

Python
import pineconepinecone.init(api_key="YOUR_API_KEY", environment="YOUR_ENVIRONMENT")pinecone.delete_index("example-index")

Scale replicas
The following example changes the number of replicas for example-index.

Python
import pineconepinecone.init(api_key="YOUR_API_KEY", environment="YOUR_ENVIRONMENT")new_number_of_replicas = 4pinecone.configure_index("example-index", replicas=new_number_of_replicas)

Describe index statistics
The following example returns statistics about the index example-index.

Python

```
import pineconepinecone.init(api_key="YOUR_API_KEY",
environment="YOUR_ENVIRONMENT")index =
pinecone.Index("example-index")index_stats_response = index.describe_index_stats()
```

Upsert vectors

The following example upserts dense vectors to example-index.

Python

```
import pineconepinecone.init(api_key="YOUR_API_KEY",
environment="YOUR_ENVIRONMENT")index =
pinecone.Index("example-index")upsert_response = index.upsert( vectors=[ ( "vec1", # Vector
ID [0.1, 0.2, 0.3, 0.4], # Dense vector values {"genre": "drama"} # Vector metadata ), ( "vec2",
[0.2, 0.3, 0.4, 0.5], {"genre": "action"} ) ], namespace="example-namespace")
```

Query an index

The following example queries the index example-indexwith metadata filtering.

Python

```
import pineconepinecone.init(api_key="YOUR_API_KEY",
environment="YOUR_ENVIRONMENT")index =
pinecone.Index("example-index")query_response = index.query(
namespace="example-namespace", top_k=10, include_values=True, include_metadata=True,
vector=[0.1, 0.2, 0.3, 0.4], filter={ "genre": {"$in": ["comedy", "documentary", "drama"]} })
```

Delete vectors

The following example deletes vectors by ID.

Python

```
import pineconepinecone.init(api_key="YOUR_API_KEY",
environment="YOUR_ENVIRONMENT")index =
pinecone.Index("example-index")delete_response = index.delete(ids=["vec1", "vec2"],
namespace="example-namespace")
```

Fetch vectors

The following example fetches vectors by ID.

Python

```
import pineconepinecone.init(api_key="YOUR_API_KEY",
environment="YOUR_ENVIRONMENT")index =
pinecone.Index("example-index")fetch_response = index.fetch(ids=["vec1", "vec2"],
namespace="example-namespace")
```

Update vectors

The following example updates vectors by ID.

Python
```
import pineconepinecone.init(api_key="YOUR_API_KEY", environment="YOUR_ENVIRONMENT")index = pinecone.Index("example-index")update_response = index.update( id="vec1", values=[0.1, 0.2, 0.3, 0.4], set_metadata={"genre": "drama"}, namespace="example-namespace")
```

Create collection

The following example creates the collection example-collectionfrom example-index.

Python
```
import pineconepinecone.init(api_key="YOUR_API_KEY", environment="YOUR_ENVIRONMENT")pinecone.create_collection("example-collection", "example-index")
```

List collections

The following example returns a list of the collections in the current project.

Python
```
import pineconepinecone.init(api_key="YOUR_API_KEY", environment="YOUR_ENVIRONMENT")active_collections = pinecone.list_collections()
```

Describe a collection

The following example returns a description of the collection example-collection.

Python
```
import pineconepinecone.init(api_key="YOUR_API_KEY", environment="YOUR_ENVIRONMENT")collection_description = pinecone.describe_collection("example-collection")
```

Delete a collection

The following example deletes the collection example-collection.

Python
```
import pineconepinecone.init(api_key="YOUR_API_KEY", environment="YOUR_ENVIRONMENT")pinecone.delete_collection("example-collection")
```

Reference

For the REST API or other clients, see the API reference.

init()

pinecone.init(**kwargs)

Initialize Pinecone.

Parameters	Type	Description
api_key	str	Your Pinecone API key.
environment	str	The cloud environment of your Pinecone project.

Example:

Python
import pineconepinecone.init(api_key="YOUR_API_KEY", environment="YOUR_ENVIRONMENT")

configure_index()

pinecone.configure_index(index_name, **kwargs)

Configure an index to change pod type and number of replicas.

Parameters	Type	Description
index_name	str	The name of the index
replicas	int	(Optional) The number of replicas to configure for this index.
pod_type	str	(Optional) The new pod type for the index. One of s1, p1, or p2appended with .and one of x1, x2, x4, or x8.

Example:

Python
import pineconepinecone.init(api_key='YOUR_API_KEY', environment='YOUR_ENVIRONMENT')new_number_of_replicas = 4pinecone.configure_index('example-index', replicas=new_number_of_replicas)

create_collection()

pinecone.create_collection(**kwargs)

Create a collection from an index.

Parameters	Type	Description
name	str	The name of the collection to be created.

source	str	The name of the source index to be used as the source for the collection.

Example:

Python
import pineconepinecone.init(api_key='YOUR_API_KEY', environment='YOUR_ENVIRONMENT')pinecone.create_collection('example-collection', 'example-index')

create_index()

pinecone.create_index(**kwargs)

Create an index.

Parameters	Type	Description
name	str	The name of the index to be created. The maximum length is 45 characters.
dimension	integer	The dimensions of the vectors to be inserted in the index.
metric	str	(Optional) The distance metric to be used for similarity search: 'euclidean', 'cosine', or 'dotproduct'.
pods	int	(Optional) The number of pods for the index to use, including replicas.
replicas	int	(Optional) The number of replicas.
pod_type	str	(Optional) The new pod type for the index. One of s1, p1, or p2appended with .and one of x1, x2, x4, or x8.
metadata_config	object	(Optional) Configuration for the behavior of Pinecone's internal metadata index. By default, all metadata is indexed; when metadata_config is present, only specified metadata fields are indexed. To specify metadata fields to index, provide a JSON object of the following form: {"indexed": ["example_metadata_field"]}
source_collection	str	(Optional) The name of the collection to create an index from.

Example:

Python

import pineconepinecone.init(api_key='YOUR_API_KEY', environment='YOUR_ENVIRONMENT')## The following example creates an index without a metadata## configuration. By default, Pinecone indexes all metadata.pinecone.create_index('example-index', dimension=1024)## The following example creates an index that only indexes## the 'color' metadata field. Queries against this index## cannot filter based on any other metadata field.metadata_config = { 'indexed': ['color']}pinecone.create_index('example-index-2', dimension=1024, metadata_config=metadata_config)

delete_collection()

pinecone.delete_collection('example-collection')

Delete an existing collection.

Parameters	Type	Description
collectionName	str	The name of the collection to delete.

Example:

Python
import pineconepinecone.init(api_key='YOUR_API_KEY', environment='YOUR_ENVIRONMENT')pinecone.delete_collection('example-collection')

delete_index()

pinecone.delete_index(indexName)

Delete an existing index.

Parameters	Type	Description
index_name	str	The name of the index.

Example:

Python
import pineconepinecone.init(api_key='YOUR_API_KEY', environment='YOUR_ENVIRONMENT')pinecone.delete_index('example-index')

describe_collection()

pinecone.describe_collection(collectionName)

Get a description of a collection.

Parameters	Type	Description
collection_name	str	The name of the collection.

Example:

Python
import pineconepinecone.init(api_key='YOUR_API_KEY',
environment='YOUR_ENVIRONMENT')collection_description =
pinecone.describe_collection('example-collection')
Returns:

- collectionMeta: objectConfiguration information and deployment status of the collection.
    - name: stringThe name of the collection.
    - size: integerThe size of the collection in bytes.
    - status: stringThe status of the collection.

describe_index()

pinecone.describe_index(indexName)

Get a description of an index.

Parameters	Type	Description
index_name	str	The name of the index.

Returns:

- database: object
- name: stringThe name of the index.
- dimension: integerThe dimensions of the vectors to be inserted in the index.
- metric: stringThe distance metric used for similarity search: 'euclidean', 'cosine', or 'dotproduct'.
- pods: integerThe number of pods the index uses, including replicas.
- replicas: integerThe number of replicas.
- pod_type: stringThe pod type for the index. One of s1, p1, or p2appended with .and one of x1, x2, x4, or x8.
- metadata_config: objectConfiguration for the behavior of Pinecone's internal metadata index. By default, all metadata is indexed; when metadata_config is present, only specified metadata fields are indexed. To specify metadata fields to index, provide a JSON object of the following form: {"indexed": ["example_metadata_field"]}
- status: object
- ready: booleanWhether the index is ready to serve queries.

- state: stringOne of Initializing, ScalingUp, ScalingDown, Terminating, or Ready.

Example:

Python
import pineconepinecone.init(api_key='YOUR_API_KEY',
environment='YOUR_ENVIRONMENT')index_description =
pinecone.describe_index('example-index')

list_collections()

pinecone.list_collections()

Return a list of the collections in your project.

Returns:

- arrayof stringsThe names of the collections in your project.

Example:

Python
import pineconepinecone.init(api_key='YOUR_API_KEY',
environment='us-east1-gcp')active_collections = pinecone.list_collections()

list_indexes()

pinecone.list_indexes()

Return a list of your Pinecone indexes.

Returns:

- arrayof stringsThe names of the indexes in your project.

Example:

Python
import pineconepinecone.init(api_key='YOUR_API_KEY',
environment='YOUR_ENVIRONMENT')active_indexes = pinecone.list_indexes()

Index()

pinecone.Index(indexName)

Construct an Index object.

Parameters	Type	Description
indexName	str	The name of the index.

Example:

Python
index = pinecone.Index("example-index")

Index.delete()

Index.delete(**kwargs)

Delete items by their ID from a single namespace.

Parameters	Type	Description
ids	array	(Optional) arrayof stringsvectors to delete.
delete_all	boolean	(Optional) Indicates that all vectors in the index namespace should be deleted.
namespace	str	(Optional) The namespace to delete vectors from, if applicable.
filter	object	(Optional) If specified, the metadata filter here will be used to select the vectors to delete. This is mutually exclusive with specifying ids to delete in the ids param or using delete_all=True. See https://www.pinecone.io/docs/metadata-filtering/.

Example:

Python
import pineconepinecone.init(api_key='YOUR_API_KEY', environment='YOUR_ENVIRONMENT')index = pinecone.Index('example-index')delete_response = index.delete(ids=['vec1', 'vec2'], namespace='example-namespace')

Index.describe_index_stats()

Index.describe_index_stats()

Returns statistics about the index's contents, including the vector count per namespace and the number of dimensions.

Returns:

- namespaces: objectA mapping for each namespace in the index from the namespace name to a

  summary of its contents. If a metadata filter expression is present, the summary will reflect only vectors matching that expression.
- dimension: int64The dimension of the indexed vectors.
- indexFullness: floatThe fullness of the index, regardless of whether a metadata filter expression was passed. The granularity of this metric is 10%.
- totalVectorCount: int64The total number of vectors in the index.

Example:

Python
import pineconepinecone.init(api_key='YOUR_API_KEY', environment='YOUR_ENVIRONMENT')index = pinecone.Index('example-index')index_stats_response = index.describe_index_stats()

Index.fetch()

Index.fetch(ids, **kwargs)

The Fetch operation looks up and returns vectors, by ID, from a single namespace. The returned vectors include the vector data and metadata.

Parameters	Type	Description
ids	[str]	The vector IDs to fetch. Does not accept values containing spaces.
namespace	str	(Optional) The namespace containing the vectors.

Returns:

- vectors: objectContains the vectors.
- namespace: stringThe namespace of the vectors.

Example:

Python
import pineconepinecone.init(api_key='YOUR_API_KEY', environment='YOUR_ENVIRONMENT')index = pinecone.Index('example-index')fetch_response = index.fetch(ids=['vec1', 'vec2'], namespace='example-namespace')

Index.query()

Index.query(**kwargs)

Search a namespace using a query vector. Retrieves the ids of the most similar items in a namespace, along with their similarity scores.

Parameters	Type	Description
namespace	str	(Optional) The namespace to query.
top_k	int64	The number of results to return for each query.
filter	object	(Optional) The filter to apply. You can use vector metadata to limit your search. See https://www.pinecone.io/docs/metadata-filtering/.
include_values	boolean	(Optional) Indicates whether vector values are included in the response. Defaults to false.
include_metadata	boolean	(Optional) Indicates whether metadata is included in the response as well as the ids. Defaults to false.
vector	[floats]	(Optional) The query vector. This should be the same length as the dimension of the index being queried. Each query()request can contain only one of the parameters idor vector.
sparse_vector	dictionary	(Optional) The sparse query vector. This must contain an array of integers named indicesand an array of floats named values. These two arrays must be the same length.
id	string	(Optional) The unique ID of the vector to be used as a query vector. Each query()request can contain only one of the parameters vectoror id.

Example:

Python
import pinecone pinecone.init(api_key='YOUR_API_KEY', environment='YOUR_ENVIRONMENT') index = pinecone.Index('example-index') query_response = index.query(namespace='example-namespace', top_k=10, include_values=True, include_metadata=True, vector=[0.1, 0.2, 0.3, 0.4], filter={ 'genre': {'$in': ['comedy', 'documentary', 'drama']} })
The following example queries the index example-indexwith a sparse-dense vector.

Python

```
import pineconepinecone.init(api_key="YOUR_API_KEY",
environment="YOUR_ENVIRONMENT")index =
pinecone.Index("example-index")query_response = index.query(
namespace="example-namespace", top_k=10, include_values=True, include_metadata=True,
vector=[0.1, 0.2, 0.3, 0.4], sparse_vector={ 'indices': [10, 45, 16], 'values': [0.5, 0.5, 0.2] }, filter={
"genre": {"$in": ["comedy", "documentary", "drama"]} })
```

Index.update()

Index.update(**kwargs)

Updates vectors in a namespace. If a value is included, it will overwrite the previous value.If set_metadatais included, the values of the fields specified in it will be added or overwrite the previous value.

Parameters	Type	Description
id	str	The vector's unique ID.
values	[float]	(Optional) Vector data.
set_metadata	object	(Optional) Metadata to set for the vector.
namespace	str	(Optional) The namespace containing the vector.

Example:

Python

```
import pineconepinecone.init(api_key='YOUR_API_KEY',
environment='YOUR_ENVIRONMENT')index =
pinecone.Index('example-index')update_response = index.update( id='vec1', values=[0.1, 0.2,
0.3, 0.4], set_metadata={'genre': 'drama'}, namespace='example-namespace')
```

Index.upsert()

Index.upsert(**kwargs)

Writes vectors into a namespace. If a new value is upserted for an existing vector ID, it will overwrite the previous value.

Parameters	Type	Description

vectors	[object]	An array containing the vectors to upsert. Recommended batch limit is 100 vectors. id(str) - The vector's unique id. values([float]) - The vector data. metadata(object) - (Optional) Metadata for the vector. sparse_vector(object) - (Optional) A dictionary containing the index and values arrays containing the sparse vector values.
namespace	str	(Optional) The namespace name to upsert vectors.

Returns:

- upsertedCount: int64The number of vectors upserted.

Example:

Python
import pineconepinecone.init(api_key='YOUR_API_KEY', environment='YOUR_ENVIRONMENT')index = pinecone.Index('example-index')upsert_response = index.upsert(vectors=[{'id': "vec1", "values":[0.1, 0.2, 0.3, 0.4], "metadata": {'genre': 'drama'}}, {'id': "vec2", "values":[0.2, 0.3, 0.4, 0.5], "metadata": {'genre': 'action'}},], namespace='example-namespace')
The following example upserts vectors with sparse and dense values to example-index.

Python
import pineconepinecone.init(api_key="YOUR_API_KEY", environment="YOUR_ENVIRONMENT")index = pinecone.Index("example-index")upsert_response = index.upsert(vectors=[{'id': 'vec1', 'values': [0.1, 0.2, 0.3, 0.4], 'metadata': {'genre': 'drama'}, 'sparse_values': { 'indices': [10, 45, 16], 'values': [0.5, 0.5, 0.2] }}, {'id': 'vec2', 'values': [0.2, 0.3, 0.4, 0.5], 'metadata': {'genre': 'action'}, 'sparse_values': { 'indices': [15, 40, 11], 'values': [0.4, 0.5, 0.2] }}], namespace='example-namespace')

Quickstart

How to get started with the Pinecone vector database.
Suggest Edits

This guide explains how to set up a Pinecone vector database in minutes.

1. Install Pinecone client (optional)

This step is optional. Do this step only if you want to use the Python client.

Use the following shell command to install Pinecone:

Python
pip install pinecone-client

2. Get and verify your Pinecone API key

To use Pinecone, you must have an API key. To find your API key, open the Pinecone consoleand click API Keys. This view also displays the environment for your project. Note both your API key and your environment.

To verify that your Pinecone API key works, use the following commands:

Pythoncurl
import pinecone

pinecone.init(api_key="YOUR_API_KEY", environment="YOUR_ENVIRONMENT")
If you don't receive an error message, then your API key is valid.

3. Hello, Pinecone!

You can complete the remaining steps in three ways:

- Use the "Hello, Pinecone!" colab notebookto write and execute Python in your browser.
- Copy the commands below into your local installation of Python.
- Use the cURL API commands below.

1. Initialize Pinecone

Pythoncurl
import pinecone
pinecone.init(api_key="YOUR_API_KEY", environment="YOUR_ENVIRONMENT")
2. Create an index.

The commands below create an index named "quickstart" that performs approximate nearest-neighbor search using the Euclidean distance metricfor 8-dimensional vectors.

Index creation takes roughly a minute.

Pythoncurl
pinecone.create_index("quickstart", dimension=8, metric="euclidean")

⚠️Warning

In general, indexes on the Starter (free) plan are archived as collections and deleted after 7 days of inactivity; for indexes created by certain open source projects such as AutoGPT, indexes are archived and deleted after 1 day of inactivity. To prevent this, you can send any API request to Pinecone and the counter will reset.
3. Retrieve a list of your indexes.

Once your index is created, its name appears in the index list.

Use the following commands to return a list of your indexes.

Pythoncurl
pinecone.list_indexes()
# Returns:
# ['quickstart']
4. Connect to the index (Client only).

Before you can query your index using a client, you must connect to the index.

Use the following commands to connect to your index.

Pythoncurl
index = pinecone.Index("quickstart")
5. Insert the data.

To ingest vectors into your index, use the upsertoperation.

The upsert operation inserts a new vector in the index or updates the vector if a vector with the same ID is already present.

The following commands upsert 5 8-dimensional vectors into your index.

Pythoncurl
Upsert sample data (5 8-dimensional vectors)
index.upsert([
("A", [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]),

("B", [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]),
("C", [0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3]),
("D", [0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4]),
("E", [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5])
])
The cURL command above uses the endpointfor your Pinecone index.

ℹ️Note

When upserting larger amounts of data, upsert data in batchesof 100 vectors or fewer over multiple upsert requests.
6. Get statistics about your index.

The following commands return statistics about the contents of your index.

Pythoncurl
index.describe_index_stats()
# Returns:
# {'dimension': 8, 'index_fullness': 0.0, 'namespaces': {'': {'vector_count': 5}}}
7. Query the index and get similar vectors.

The following example queries the index for the three (3) vectors that are most similar to an example 8-dimensional vector using the Euclidean distance metric specified in step 2 ("Create an index.") above.

Pythoncurl
index.query(
vector=[0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3],
top_k=3,
include_values=True
)
Returns:
{'matches': [{'id': 'C',
'score': 0.0,
'values': [0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3]},
{'id': 'D',
'score': 0.0799999237,
'values': [0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4]},
{'id': 'B',
'score': 0.0800000429,
'values': [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]}],
'namespace': ''}
8. Delete the index.

Once you no longer need the index, use the delete_indexoperation to delete it.

The following commands delete the index.

Pythoncurl
pinecone.delete_index("quickstart")

⚠️Warning

After you delete an index, you cannot use it again.

Next steps

Now that you're successfully making indexes with your API key, you can start inserting dataor view more examples.

Overview

Overview

An introduction to the Pinecone vector database.
Suggest Edits
Pinecone makes it easy to build high-performance vector searchapplications. It's a managed, cloud-native vector database with a simple APIand no infrastructure hassles.

Pinecone has the following attributes:

- Fast: Get ultra-low query latency at any scale, even with billions of items.
- Fresh: Get live index updates when you add, edit, or delete data.
- Filtered: Combine vector search with metadata filters for more relevant, faster results.
- Fully managed: Get started, use, and scale with ease, while we keep things running smoothly and securely.

Get started using Pinecone.

Use cases

Pinecone is useful for a broad variety of applications. The following are some of the most common:

- Semantic text search: Convert text data into vector embeddings using an NLPtransformer such as a sentence embedding model, then index and search through those vectors using Pinecone.
- Generative question-answering: Retrieve relevant contexts to queries from Pinecone and pass these to a generative model like OpenAI to generate an answer backed by real data sources.
- Hybrid search: Perform semantic and keyword search over your data in one query and combine the results for more relevant results.
- Image similarity search: Transform image data into vector embeddings and build an index with Pinecone. Then convert query images into vectors and retrieve similar images.
- Product recommendations: Generate product recommendations for ecommerce based on vectors representing users.

Want to see more and start with working example notebooks? See our example applications.

Key concepts

Vector search

Unlike traditional search methods that revolve around keywords, vector databasesindex and search through ML-generated representations of data, called vector embeddings, to find items most similar to the query.

Vector embeddings

Vector embeddingsare sets of numbers that represent objects. They are generated by embedding modelstrained to capture the semantic similarity of objects in a given set. Pinecone supports two kinds of vector embeddings: dense embeddingsand sparse embeddings.

You need to have vector embeddings to use Pinecone.

Vector database

A vector databaseindexes and stores vector embeddings for efficient management and fast retrieval. Unlike a standalone vector index, a vector database like Pinecone provides additional capabilities such as index management, data management, metadata storage and filtering, and horizontal scaling.

Learn more about vector databases.

Workflow

Follow these guides to set up your index:

1. Create an index
2. Connect to an index
3. Insert the dataand vectors into the index

Once you have an index with data, follow these guides to start using your index:

- Query the data
    - Filter the data
- Fetch data
- Insert more dataor update existing vectors
- Manage the index
- Manage data

Pricing and deployment options

Visit the pricing pagefor pricing and deployment options.

Pinecone client libraries, connectors, and SDKs

This page contains information about Pinecone client libraries, connectors, and SDKs.

Client libraries

Pinecone supports official client libraries for the following languages:

Language	Repository	Package manager	Documentation
Python	GitHub	PyPi	Reference
Node.js (Public Preview)	GitHub	NPM	Reference

Connectors

Pinecone supports official connectors for the following databases:

Database	Language	Repository	Documentation
Apache Spark	Scala / Python	GitHub	Readme

Community libraries

The following libraries are community-maintained.

Language	GitHub	Package manager	Documentation
Ruby	GitHub	RubyGems	Readme