Quick tip: Review the prerequisites before you run the lab

Start Lab 01:30:00

Streaming Updates into Vertex AI Vector Search

Lab 1 hour 30 minutes No cost Intermediate

★★★★☆

This lab may incorporate AI tools to support your learning.

Previous Next

Overview

This lab demonstrates how to use the Vertex AI Vector Search Stream Update Service. The dataset used for this tutorial is the GloVe dataset.

Here, you will learn how to create Approximate Nearest Neighbor (ANN) Index, update index using stream update, and query against indexes.

The steps performed in this lab include:

- Create ANN Index
- Create an IndexEndpoint with VPC Network
- Deploy ANN Index
- Perform online query
- Update index by using stream udpate
- Compute recall

Setup and requirements

Read these instructions. Labs are timed and you cannot pause them. The timer, which starts when you click **Start Lab**, shows how long Google Cloud resources will be made available to you.

This Qwiklabs hands-on lab lets you do the lab activities yourself in a real cloud environment, not in a simulation or demo environment. It does so by giving you new, temporary credentials that you use to sign in and access Google Cloud for the duration of the lab.

What you need

To complete this lab, you need:

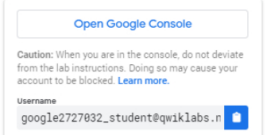
- Access to a standard internet browser (Chrome browser recommended).
- Time to complete the lab.

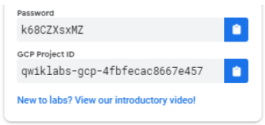
Note: If you already have your own personal Google Cloud account or project, do not use it for this lab.

Note: If you are using a Pixelbook, open an Incognito window to run this lab.

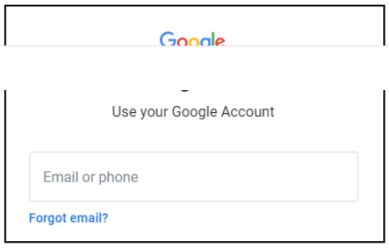
How to start your lab and sign in to the Google Cloud Console

1. Click the **Start Lab** button. If you need to pay for the lab, a pop-up opens for you to select your payment method. On the left is a panel populated with the temporary credentials that you must use for this lab.



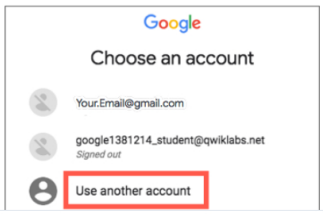


2. Copy the username, and then click **Open Google Console**. The lab spins up resources, and then opens another tab that shows the **Sign in** page.



Tip: Open the tabs in separate windows, side-by-side.

If you see the **Choose an account** page, click **Use Another Account**.



3. In the **Sign in** page, paste the username that you copied from the Connection Details panel. Then copy and paste the password.

Important: You must use the credentials from the Connection Details panel. Do not use your Qwiklabs credentials. If you have your own Google Cloud account, do not use it for this lab (avoids incurring charges).

4. Click through the subsequent pages:

- Accept the terms and conditions.
- Do not add recovery options or two-factor authentication (because this is a temporary account).
- Do not sign up for free trials.

After a few moments, the Cloud Console opens in this tab.

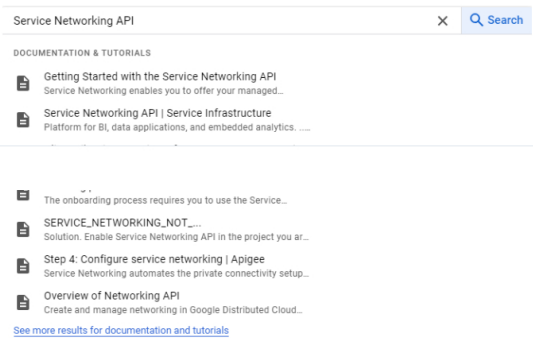
Note: You can view the menu with a list of Google Cloud Products and Services by clicking the **Navigation menu** at the top-left.

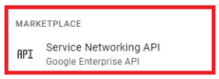
Task 1. Enable APIs

In this section, let's enable the service networking API.

To enable the Service Networking API, follow these steps:

1. Type **Service Networking API** into the top search bar of the Google Cloud Console, choose the result as shown in the following image.



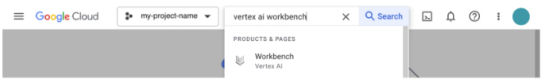


2. Click **Enable**.

Task 2. Prepare VPC Network and Install Packages

To reduce any network overhead that might lead to unnecessary increase in overhead latency, it is best to call the ANN endpoints from your VPC via a direct VPC Peering connection. The following section describes how to setup a VPC Peering connection.

1. In the Google Cloud console, navigate to **Vertex AI Workbench**. In the top search bar of the Google Cloud console, enter **Vertex AI Workbench**, and click on the first result.



2. Click on **User managed notebooks** and then click on **Open JupyterLab** for **generative-ai-jupyterlab** notebook.

The JupyterLab will run in a new tab.

3. On the **Launcher**, under **Notebook**, click on **Python 3** to open a new python notebook.

4. Run the following code snippet in the first cell by clicking the play ▶ button at the top or enter **SHIFT+ENTER** on your keyboard to execute the cell.

```
PROJECT_ID = "GCP Project ID"
NETWORK_NAME = "Network Name"
PEERING_RANGE_NAME = "VPC Peering Range Name"

# Reserve IP range
! gcloud compute addresses create {PEERING_RANGE_NAME} --global -
-prefix-length=16 --network={NETWORK_NAME} --purpose=VPC_PEERING
--project={PROJECT_ID} --description="peering range for cymbal
demo"

# Set up peering with service networking
--ranges={PEERING_RANGE_NAME} --project={PROJECT_ID}
```

5. Download and install the latest (preview) version of the Vertex AI SDK for Python.

```
! pip install -U git+https://github.com/googleapis/python-
aiplatform.git@main --user
```

6. Install the `h5py` to prepare sample dataset, and the `grpcio-tools` for querying against the index.

```
! pip install -U grpcio-tools --user
! pip install -U h5py --user
! pip install proto-plus==1.24.0.dev1
```

7. After you install the additional packages, you need to restart the notebook kernel so it can find the packages.

```
import os

if not os.getenv("IS_TESTING"):
    # Automatically restart kernel after installs
    import IPython
```

```
import IPython

app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```

Output:

Task 3. Create a Cloud Storage Bucket

1. Run the following code snippet to create a Cloud Storage Bucket in [Lab GCP Region] region.

```
BUCKET_NAME = "gs://GCP Project ID-aip"
REGION = "Lab GCP Region"
PROJECT_ID = "GCP Project ID"
NETWORK_NAME = "Network Name"
```

```
! gsutil mb -l $REGION -p $PROJECT_ID $BUCKET_NAME
```

2. Finally, validate access to your Cloud Storage bucket by examining its contents:

```
! gsutil ls -al $BUCKET_NAME
```

Task 4. Import Libraries

Import the Vertex AI (unified) client library into your Python environment.

1. Run the following code snippet to import Vertex AI client library.

```
# Upgrade protobuf to the latest version
!pip install --upgrade protobuf

import time
import grpc
import h5py
from google.cloud import aiplatform_v1
from google.protobuf import struct_pb2
```

```
REGION = "Lab GCP Region"
ENDPOINT = "{}-aiplatform.googleapis.com".format(REGION)
NETWORK_NAME = "Network Name"


AUTH_TOKEN = !gcloud auth print-access-token
PROJECT_NUMBER = !gcloud projects list --
filter="PROJECT_ID:'{PROJECT_ID}'" --
format='value(PROJECT_NUMBER)'
PROJECT_NUMBER = PROJECT_NUMBER[0]

PARENT = "projects/{}/locations/{}".format(PROJECT_ID, REGION)

print("ENDPOINT: {}".format(ENDPOINT))
print("PROJECT_ID: {}".format(PROJECT_ID))
print("REGION: {}".format(REGION))

!gcloud config set project {PROJECT_ID} --quiet
!gcloud config set ai_platform/region {REGION} --quiet
```

Output:

```
PROJECT_ID: qwiklabs-gcp-04-0a0c36345736
REGION: us-west1
```

```
Updated property [core/project].
Updated property [ai_platform/region].
```

Task 5. Prepare the Data

The GloVe dataset consists of a set of pre-trained embeddings. The embeddings are split into a "train" split, and a "test" split. We will create a vector search index from the "train" split, and use the embedding vectors in the "test" split as query vectors to test the vector search index.

Note: While the data split uses the term "train", these are pre-trained embeddings and thus are ready to be indexed for search. The terms "train" and "test" split are used just to be consistent with usual machine learning terminology.

```
! gsutil cp gs://cloud-samples-data/vertex-
ai/matching_engine/glove-100-angular.hdf5 .
```

Output:

```
Copying gs://cloud-samples-data/vertex-ai/matching_engine/glove-100-angular.hdf5...
- [1 files][462.9 MiB/462.9 MiB]
Operation completed over 1 objects/462.9 MiB.
```

2. Read the data into memory.

```
# The number of nearest neighbors to be retrieved from database
for each query.
k = 10

h5 = h5py.File("glove-100-angular.hdf5", "r")
train = h5["train"]
test = h5["test"]
```

Output:

```
array([-0.11333  ,  0.48402  ,  0.090771 , -0.22439  ,  0.034206 ,
       -0.55831  ,  0.041849 , -0.53573  ,  0.18809  , -0.58722  ,
        0.015313 , -0.014555 ,  0.80842  , -0.038519 ,  0.75348  ,
        0.70502  , -0.17863  ,  0.3222   ,  0.67575  ,  0.67198  ,
        0.26044  ,  0.4187   , -0.34122  ,  0.2286   , -0.53529  ,
        1.2582   , -0.091543 ,  0.19716  , -0.037454 , -0.3336   ,
        0.31399  ,  0.36488  ,  0.71263  ,  0.1307   , -0.24654  ,
       -0.52445  , -0.036091 ,  0.55068  ,  0.10017  ,  0.48095  ,
        0.71104  , -0.053462 ,  0.22325  ,  0.30917  , -0.39926  ,
        0.036634 , -0.35431  , -0.42795  ,  0.46444  ,  0.25586  ,
        0.68257  , -0.20821  ,  0.38433  ,  0.055773 , -0.2539   ,
       -0.20804  ,  0.52522  , -0.11399  , -0.3253   , -0.44104  ,
        0.17528  ,  0.62255  ,  0.50237  , -0.7607   , -0.071786 ,
        0.0080131, -0.13286  ,  0.50097  ,  0.18824  , -0.54722  ,
       -0.42664  ,  0.4292   ,  0.14877  , -0.0072514, -0.16484  ,
       -0.059798 ,  0.9895   , -0.61738  ,  0.054169 ,  0.48424  ,
       -0.35084  , -0.27053  ,  0.37829  ,  0.11503  , -0.39613  ,
        0.24266  ,  0.39147  , -0.075256 ,  0.65093  , -0.20822  ,
       -0.17456  ,  0.53571  , -0.16537  ,  0.13582  , -0.56016  ,
        0.016964 ,  0.1277   ,  0.94071  , -0.22608  , -0.021106 ],
      dtype=float32)
```

3. Save the train split in JSONL format.

```
# Add restricts to each data point, in this demo, we only add one
namespace and the allowlist is set to be the same as the id.
# Split datapoins into two groups 'a' and 'b'. The datapoint
whose ids are even are in group 'a', otherwise are in group 'b'
# We will demo how to configure the query to return up to k data
points for each group.
with open("glove100.json", "w") as f:
    for i in range(len(train)):
        f.write('{"id":"' + str(i) + '",')
        f.write('"embedding":[' + ",".join(str(x) for x in
train[i]) + "],")
        f.write('"restricts":[{"namespace": "class", "allow": ["'
+ str(i) + '"]}],')
        f.write('"crowding_tag":' + ('"a"' if i % 2 == 0 else
'"b"') + "}")
        f.write("\n")
        if i >= 100:
            break
```

4. Upload the training data to Google Cloud Storage Bucket created earlier.

```
# NOTE: Everything in this GCS DIR will be DELETED before
uploading the data.
```

```
! gsutil cp glove100.json {BUCKET_NAME}/glove100.json
```

Output:

```
Copying file://glove100.json [Content-Type=application/json]...
/ [1 files][ 93.1 KiB/ 93.1 KiB]
Operation completed over 1 objects/93.1 KiB.
```

5. List the newly added contents in the bucket.

```
! gsutil ls {BUCKET_NAME}
```

Output:

```
gs://qwiklabs-gcp-04-da8c38545758-aip/glove100.json
```

Task 6. Create Stream Update Index

1. Run the following code snippet in the next cells to create a instance of the `IndexServiceClient` from the AI Platform (Unified) Python client library. This is used for interacting with AI Platform services related to indexes, such as creating and managing Approximate Nearest Neighbor (ANN)

```
index_client = aiplatform_v1.IndexServiceClient(
    client_options=dict(api_endpoint=ENDPOINT)
)
```

```
DIMENSIONS = 100
DISPLAY_NAME = "glove_100_1"
```

2. Let's define the configuration for creating an Approximate Nearest Neighbor (ANN) index.

```
treeAhConfig = struct_pb2.Struct(
    fields={
        "leafNodeEmbeddingCount":
struct_pb2.Value(number_value=500),
        "leafNodesToSearchPercent":
struct_pb2.Value(number_value=7),
    }
)

algorithmConfig = struct_pb2.Struct(
    fields={"treeAhConfig":
struct_pb2.Value(struct_value=treeAhConfig)}
)

config = struct_pb2.Struct(
    fields={
        "dimensions": struct_pb2.Value(number_value=DIMENSIONS),
        "approximateNeighborsCount":
struct_pb2.Value(number_value=150),
        "distanceMeasureType":
struct_pb2.Value(string_value="DOT_PRODUCT_DISTANCE"),
        "algorithmConfig":
)

metadata = struct_pb2.Struct(
    fields={
        "config": struct_pb2.Value(struct_value=config),
        "contentsDeltaUri":
struct_pb2.Value(string_value=BUCKET_NAME),
    }
)

ann_index = {
    "display_name": DISPLAY_NAME,
    "description": "Glove 100 ANN index",
    "metadata": struct_pb2.Value(struct_value=metadata),
    "index_update_method":
aiplatform_v1.Index.IndexUpdateMethod.STREAM_UPDATE,
}
```

3. Now, let's create an Approximate Nearest Neighbor (ANN) index using the

specified configuration.

```
index=ann_index)
```

4. Retrieve the result of previously executed `ann_index`.

```
ann_index.result()
```

Output:

```
name: "projects/1032193649974/locations/us-west1/indexes/1498203340100599808"
```

Note It can take up to 15 minutes to be available.

5. Finally, retrieve the name of the created index (`ann_index`) and store it in the variable `INDEX_RESOURCE_NAME`.

```
INDEX_RESOURCE_NAME = ann_index.result().name
INDEX_RESOURCE_NAME
```

Task 7. Create an IndexEndpoint with VPC Network

1. Run the following code snippet to initialize an AI Platform Index Endpoint Service client.

```
index_endpoint_client = aiplatform_v1.IndexEndpointServiceClient(
    client_options=dict(api_endpoint=ENDPOINT)
)
```

2. Define a configuration for an AI Platform Index Endpoint.

```
VPC_NETWORK_NAME =
```

```
VPC_NETWORK_NAME
```

```
index_endpoint = {
    "display_name": "index_endpoint_for_demo",
    "network": VPC_NETWORK_NAME,
}
```

3. Create an AI Platform Index Endpoint.

```
r = index_endpoint_client.create_index_endpoint(
    parent=PARENT, index_endpoint=index_endpoint
)
```

4. Retrieve the result of previous operation.

```
r.result()
```

```
name: "projects/1032193649974/locations/us-west1/indexEndpoints/3882956099816521728"
```

5. Finally, retrieve the name of the created index endpoint and store it in the variable `INDEX_ENDPOINT_NAME`.

```
INDEX_ENDPOINT_NAME = r.result().name
INDEX_ENDPOINT_NAME
```

Task 8. Deploy Stream Update Index

Task 8. Deploy Stream Update Index

1. Run the following code snippet in the next cells to deploy the index to an index endpoint.

```
deploy_ann_index = {
    "id": DEPLOYED_INDEX_ID,
    "display_name": DEPLOYED_INDEX_ID,
    "index": INDEX_RESOURCE_NAME,
}
```

```
r = index_endpoint_client.deploy_index(
    index_endpoint=INDEX_ENDPOINT_NAME,
deployed_index=deploy_ann_index
)
```

2. Wait for the operation to complete.

```
# Poll the operation until it's done successfullly.

while True:
    if r.done():
        time.sleep(60)
```

Output:

```
Poll the operation to deploy index...
Poll the operation to deploy index...
Poll the operation to deploy index...
Poll the operation to deploy index...
Poll the operation to deploy index...
Poll the operation to deploy index...
Poll the operation to deploy index...
Poll the operation to deploy index...
Poll the operation to deploy index...
Poll the operation to deploy index...
```

Note: This operation might take upto 15 minutes to complete.

3. Retrieve the result of the operation.

```
r.result()
```

```
deployed_index {
  id: "stream_update_glove_deployed"
}
```

Task 9. Create Online Queries

After you built your indexes, you may query against the deployed index through the online querying gRPC API (Match service) within the virtual machine instances from the same region.

1. Run the following code snippet to create write `match_service.proto` locally. This is a Protocol Buffer (protobuf) definition for a service called `MatchService`.

```
%%writefile match_service.proto
```

```
package google.cloud.aiplatform.container.v1;

// MatchService is a Google managed service for efficient vector
similarity
// search at scale.
service MatchService {
  // Returns the nearest neighbors for the query. If it is a
sharded
  // deployment, calls the other shards and aggregates the
responses.
  rpc Match(MatchRequest) returns (MatchResponse) {}
}
```

```
// Parameters for a match query.
message MatchRequest {
  // The ID of the DeployedIndex that will serve the request.
  // This MatchRequest is sent to a specific IndexEndpoint of the
Control API,
  // as per the IndexEndpoint.network. That IndexEndpoint also
has
  // IndexEndpoint.deployed_indexes, and each such index has an
```

```
DeployedIndex.id
  // fields of the IndexEndpoint that is being called for this
request.
  string deployed_index_id = 1;

  // The embedding values.
  repeated float float_val = 2;

  // The number of nearest neighbors to be retrieved from
database for
  // each query. If not set, will use the default from
  // the service configuration.
  int32 num_neighbors = 3;

  // The list of restricts.
  repeated Namespace restricts = 4;

  // Crowding is a constraint on a neighbor list produced by
nearest neighbor
  // search requiring that no more than some value k' of the k
neighbors
  // returned have the same value of crowding_attribute.
```

```
crowding tag.
  int32 per_crowding_attribute_num_neighbors = 5;

  // The number of neighbors to find via approximate search
before
  // exact reordering is performed. If not set, the default value
from scam
  // config is used; if set, this value must be > 0.
  int32 approx_num_neighbors = 6;

  // The fraction of the number of leaves to search, set at query
time allows
  // user to tune search performance. This value increase result
in both search
  // accuracy and latency increase. The value should be between
0.0 and 1.0. If
  // not set or set to 0.0, query uses the default value
specified in
  //
NearestNeighborSearchConfig.TreeAHConfig.leaf_nodes_to_search_perce
  int32 leaf_nodes_to_search_percent_override = 7;
}
```

```
message MatchResponse {
  message Neighbor {
    // The ids of the matches.
    string id = 1;

    // The distances of the matches.
    double distance = 2;
  }
  // All its neighbors.
  repeated Neighbor neighbor = 1;
}

// Namespace specifies the rules for determining the datapoints
that are
// eligible for each matching query, overall query is an AND
across namespaces.
message Namespace {
  // The string name of the namespace that this proto is
specifying,
  // such as "color", "shape", "geo", or "tags".
  string name = 1;

  // The allowed tokens in the namespace.
```

```
  // The denied tokens in the namespace.
  // The denied tokens have exactly the same format as the token
fields, but
  // represents a negation. When a token is denied, then matches
will be
  // excluded whenever the other datapoint has that token.
  //
  // For example, if a query specifies {color: red, blue,
!purple}, then that
  // query will match datapoints that are red or blue, but if
```

```
those points are
  // also purple, then they will be excluded even if they are
red/blue.
  repeated string deny_tokens = 3;
}
```

2. Clone the repository that contains the dependencies of `match_service.proto`.

```
! git clone https://github.com/googleapis/googleapis.git
```

```
Cloning into 'googleapis'...
remote: Enumerating objects: 216997, done.
remote: Counting objects: 100% (12949/12949), done.
remote: Compressing objects: 100% (726/726), done.
remote: Total 216997 (delta 12488), reused 12251 (delta 12220), pack-reused 204048
Receiving objects: 100% (216997/216997), 192.43 MiB | 21.06 MiB/s, done.
Resolving deltas: 100% (184620/184620), done.
```

3. Compile the protocol buffer, that generates the following files: `match_service_pb2.py` and `match_service_pb2_grpc.py`.

```
! python -m grpc_tools.protoc -I=. --proto_path=./googleapis --
python_out=. --grpc_python_out=. match_service.proto
```

4. Obtain the private Endpoint.

```
DEPLOYED_INDEX_SERVER_IP = (

list(index_endpoint_client.list_index_endpoints(parent=PARENT))
[0]
    .deployed_indexes[0]
```

```
DEPLOYED_INDEX_SERVER_IP
```

5. Test your query.

```
import match_service_pb2
import match_service_pb2_grpc

channel = grpc.insecure_channel("
{}:10000".format(DEPLOYED_INDEX_SERVER_IP))
stub = match_service_pb2_grpc.MatchServiceStub(channel)
```

```
# Test query
query = [
    -0.11333,
    0.48402,
    0.090771,
    -0.22439,
    0.034206,
    -0.55831,
```

```
    0.18889,
    -0.58722,
    0.015313,
    -0.014555,
    0.80842,
    -0.038519,
    0.75348,
    0.70502,
    -0.17863,
    0.3222,
    0.67575,
    0.67198,
    0.26044,
    0.4187,
    -0.34122,
    0.2286,
    -0.53529,
    1.2582,
    -0.091543,
    0.19716,
    -0.037454,
    -0.3336,
```

```
    0.1307,
    -0.24654,
    -0.52445,
    -0.036091,
    0.55068,
    0.10017,
    0.48095
```

```
    0.71104,
    -0.053462,
    0.22325,
    0.30917,
    -0.39926,
    0.036634,
    -0.35431,
    -0.42795,
    0.46444,
    0.25586,
    0.68257,
    -0.20821,
    0.38433,
    0.055773,
    -0.2539
```

```
    -0.11399,
    -0.3253,
    -0.44104,
    0.17528,
    0.62255,
    0.50237,
    -0.7607,
    -0.071786,
    0.0080131,
    -0.13286,
    0.50097,
    0.18824,
    -0.54722,
    -0.42664,
    0.4292,
    0.14877,
    -0.0072514,
    -0.16484,
    -0.059798,
    0.9895,
    -0.61738,
    0.054169,
```

```
    -0.27053,
    0.37829,
    0.11503,
    -0.39613,
    0.24266,
    0.39147,
    -0.075256,
    0.65093,
    -0.20822,
    -0.17456,
    0.53571,
    -0.16537,
    0.13582,
    -0.56016,
    0.016964,
    0.1277,
    0.94071,
    -0.22608,
    -0.021106,
]
```

```
request.deployed_index_id = DEPLOYED_INDEX_ID
for val in query:
    request.float_val.append(val)

# The output before stream update
response = stub.Match(request)
response
```

Output:

```
neighbor {
  id: "0"
  distance: 17.592369079589844
}
neighbor {
  id: "31"
  distance: 14.614908218383789
}
neighbor {
  id: "50"
  distance: 11.242000579833984
}
neighbor {
  id: "42"
  distance: 10.925321578979492
}
```

```
}
neighbor {
  id: "100"
  distance: 10.031323432922363
}
neighbor {
```

```
  id: "71"
  distance: 9.4601297378540039
}
neighbor {
  id: "64"
  distance: 9.3296346664428711
}
neighbor {
  id: "54"
  distance: 9.25944709777832
}
neighbor {
  id: "98"
  distance: 8.94312858581543
}
```

6. Insert datapoints.

```
insert_datapoints_payload = aiplatform_v1.IndexDatapoint(
    datapoint_id="101",
    feature_vector=query,
crowding_tag=aiplatform_v1.IndexDatapoint.CrowdingTag(crowding_attr
)

upsert_request = aiplatform_v1.UpsertDatapointsRequest(
    index=INDEX_RESOURCE_NAME, datapoints=
[insert_datapoints_payload]
)

index_client.upsert_datapoints(request=upsert_request)

request = match_service_pb2.MatchRequest()
request.deployed_index_id = DEPLOYED_INDEX_ID
for val in query:
    request.float_val.append(val)

# The new inserted datapoint with id 101 will show up in the
output
response = stub.Match(request)
response
```

Output:

```
  id: "0"
  distance: 17.592369079589844
}
neighbor {
  id: "101"
  distance: 17.592369079589844
}
neighbor {
  id: "31"
  distance: 14.614908218383789
}
neighbor {
  id: "50"
  distance: 11.242000579833984
}
neighbor {
  id: "42"
  distance: 10.925321578979492
}
neighbor {
  id: "46"
  distance: 10.185911178588867
}
neighbor {
  id: "100"
  distance: 10.031323432922363
}
neighbor {
  id: "71"
  distance: 9.4601297378540039
}
```

```
}
neighbor {
  id: "54"
  distance: 9.25944709777832
}
```

7. Add filtering.

```
request = match_service_pb2.MatchRequest()
request.deployed_index_id = DEPLOYED_INDEX_ID
for val in query:
    request.float_val.append(val)

# Only the datapoints whose id is 1 and 101 will show up in the
output
restrict = match_service_pb2.Namespace()
restrict.name = "class"
restrict.allow_tokens.append("1")
restrict.allow_tokens.append("101")

request.restricts.append(restrict)
```

Update datapoint filtering:

```
update_datapoints_payload = aiplatform_v1.IndexDatapoint(
    datapoint_id="101",
    feature_vector=query,
    restricts=[{"namespace": "class", "allow_list": ["102"]}],

crowding_tag=aiplatform_v1.IndexDatapoint.CrowdingTag(crowding_attr
)

upsert_request = aiplatform_v1.UpsertDatapointsRequest(
    index=INDEX_RESOURCE_NAME, datapoints=
[update_datapoints_payload]
)

index_client.upsert_datapoints(request=upsert_request)

response = stub.Match(request)
response
```

8. Add crowding.

```
request = match_service_pb2.MatchRequest()
request.deployed_index_id = DEPLOYED_INDEX_ID
for val in query:
    request.float_val.append(val)

# Set the limit of the number of neighbors in each crowding to 1
# So no more than one neighbor of each crowding group will appear
in the output
request.per_crowding_attribute_num_neighbors = 1

response = stub.Match(request)
response
```

9. Update datapoint crowding.

```
# Change the crowding_attribute from 'b' to 'a' for the datapoint
with id '101' by using stream update
update_datapoints_payload = aiplatform_v1.IndexDatapoint(
    restricts=[{"namespace": "class", "allow_list": ["101"]}],

crowding_tag=aiplatform_v1.IndexDatapoint.CrowdingTag(crowding_attr
)

upsert_request = aiplatform_v1.UpsertDatapointsRequest(
    index=INDEX_RESOURCE_NAME, datapoints=
[update_datapoints_payload]
)

index_client.upsert_datapoints(request=upsert_request)

response = stub.Match(request)
response
```

10. Remove datapoints.

```
# Remove the datapoint with id '101' from the index
remove_request = aiplatform_v1.RemoveDatapointsRequest(
    index=INDEX_RESOURCE_NAME, datapoint_ids=["101"]

index_client.remove_datapoints(request=remove_request)

request = match_service_pb2.MatchRequest()
request.deployed_index_id = DEPLOYED_INDEX_ID
for val in query:
    request.float_val.append(val)

response = stub.Match(request)
response
```

Output:

```
neighbor {
  id: "0"
  distance: 17.592369079589844
}
neighbor {
  id: "31"
  distance: 14.614908218383789
}
neighbor {
  id: "50"
  distance: 11.242000579833984
}
neighbor {
```

```
}
neighbor {
  id: "46"
  distance: 10.185911178588867
}
neighbor {
  id: "100"
  distance: 10.031323432922363
}
neighbor {
  id: "71"
  distance: 9.4601297378540039
}
neighbor {
  id: "64"
  distance: 9.3296346664428711
}
neighbor {
  id: "54"
  distance: 9.25944709777832
}
neighbor {
  id: "98"
  distance: 8.94312858581543
}
```

Congratulations!

At the end of this lab, you have successfully built and managed ANN Indexes. You have not only acquired the knowledge to create and deploy the index but also learned how to dynamically update it through stream updates. The ability to perform online queries and compute recall adds a practical dimension, emphasizing the real-world applications of ANN Indexes in similarity searches.

Manual Last Updated November 11, 2024

Lab Last Tested November 11, 2024