Learn | Article

Text-to-Image and Image-to-Image Search Using CLIP

Jump to section

Introduction

Industries today deal with ever increasing amounts of data. Especially in retail, fashion, and other industries where the image representati

In such a situation, we can often describ
perform accurate and least time-consur

Could I take advantage of state-of-the-art artificial intelligence solutions to tackle such a challenge?

This is where OpenAI's CLIP comes in handy. A deep learning algorithm that makes it easy to connect text and images.

After completing this conceptual blog, you will understand: (1) what CLIP is, (2) how it works and why you should adopt it, and finally, (3) how to implement it for your own use case using both local and cloud-based vector indexes.

What is CLIP?

Contrastive Language-Image Pre-training (CLIP for short) is a state-of-the-art model introduced by OpenAI in February 2021 [1].

CLIP is a neural network trained on about 400 million (text and image) pairs. Training uses a contrastive learning approach that aims to unify text and images, allowing tasks like image classification to be done with text-image similarity.

This means that CLIP can find whether a given image and textual description match without being trained for a specific domain. Making CLIP powerful for out-of-the-box text and image search, which is the main focus of this article.

Besides text and image search, we can apply CLIP to image classification, image generation, image similarity search, image ranking, object tracking, robotics control, image captioning, and more.

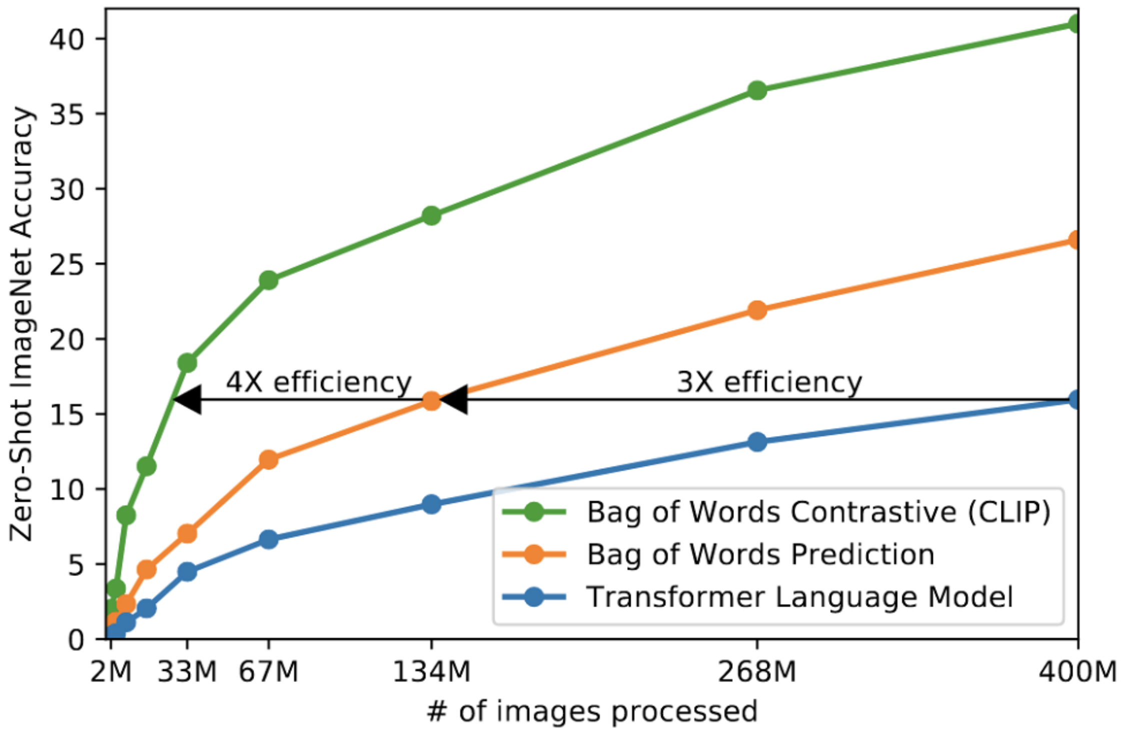
Why should you adopt the CLIP models?

Below are some reasons that increased

Efficiency

The use of the contrastive objective increased the efficiency of the CLIP model by 4-to-10x more at zero-shot ImageNet classification.

Also, the adoption of the Vision Transformer created an additional 3x gain in compute efficiency compared to the standard ResNet.

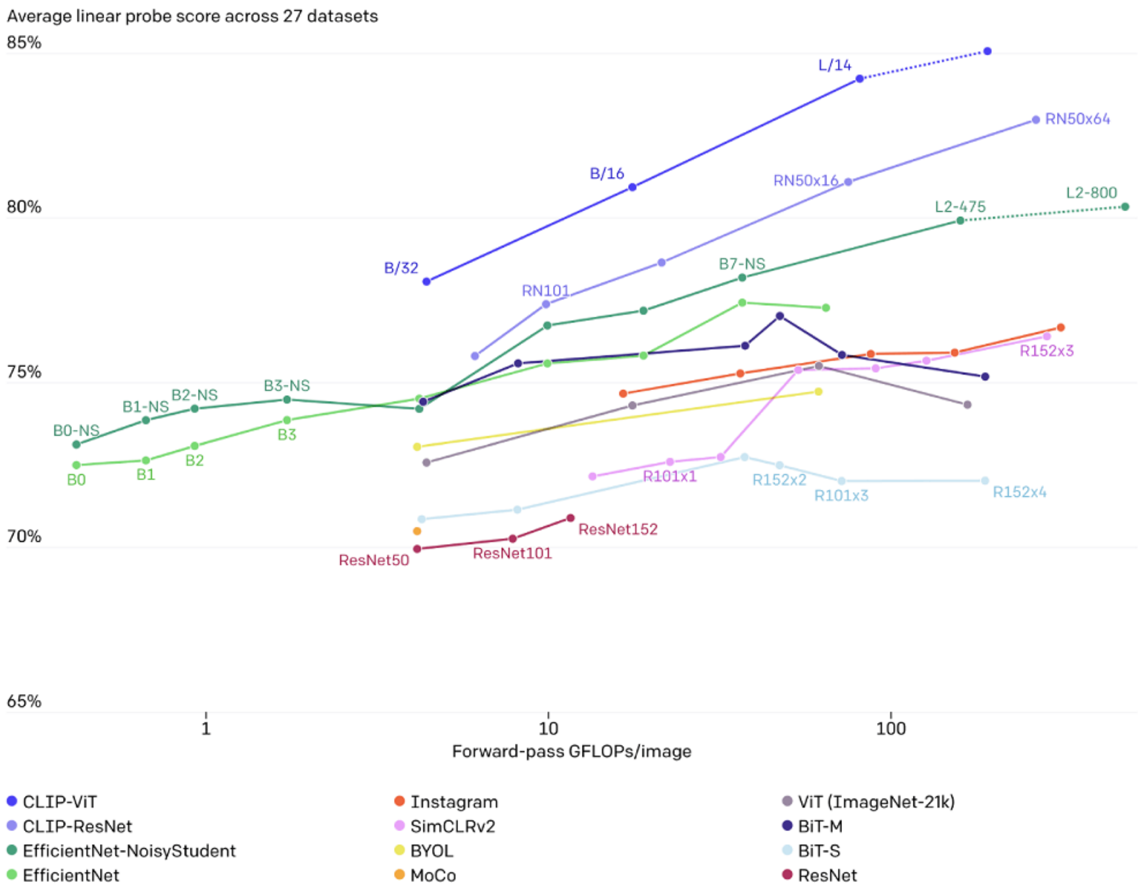


The efficiency of CLIP at zero-shot transfer
Source: arxiv

More general & flexible

CLIP outperforms existing ImageNet models in new domains because of its ability to learn a wide range of visual representations directly from natural language.

The following graphic highlights CLIP zero-shot performance compared to ResNet models few-shot linear probe performance on fine-grained object detection, geo-localization, action recognition, and optical character recognition tasks.



Average linear probe score across 27 datasets
Source: OpenAI

CLIP Architecture

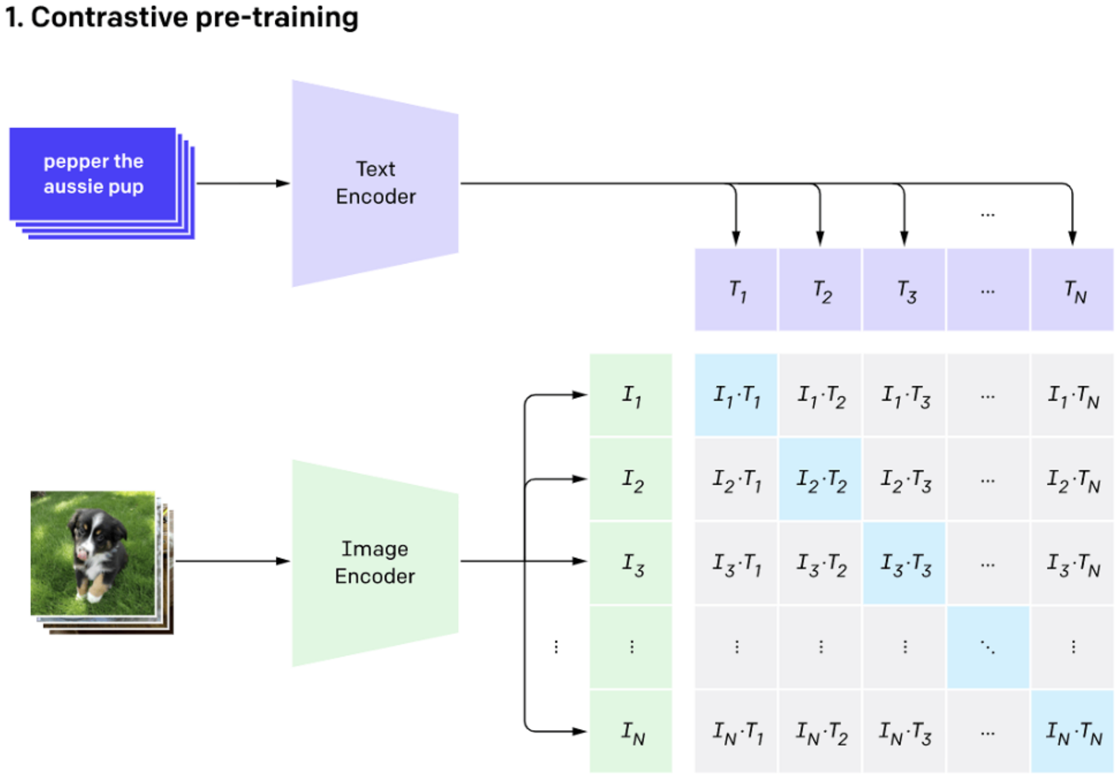
CLIP architecture consists of two main components: (1) a text encoder, and (2) an Image encoder. These two encoders are jointly trained to predict the correct pairings of a batch of training (image, text) examples.

- The *text encoder's* backbone is a transformer model [2], and the base size uses 63 millions-parameters, 12 layers, and a 512-wide model containing 8 attention heads.
- The *image encoder*, on the other hand, uses both a Vision Transformer (ViT) and a ResNet50 as its backbone, responsible for generating the feature representation of the image.

How does the CLIP a

We can answer this question by understanding these three approaches: (1) contrastive pre-training, (2) dataset classifier creation from labeled text, and finally, (3) application of the zero-shot technique for classification.

Let's explain each of these three concepts.



Contrastive pre-training
Source: OpenAI

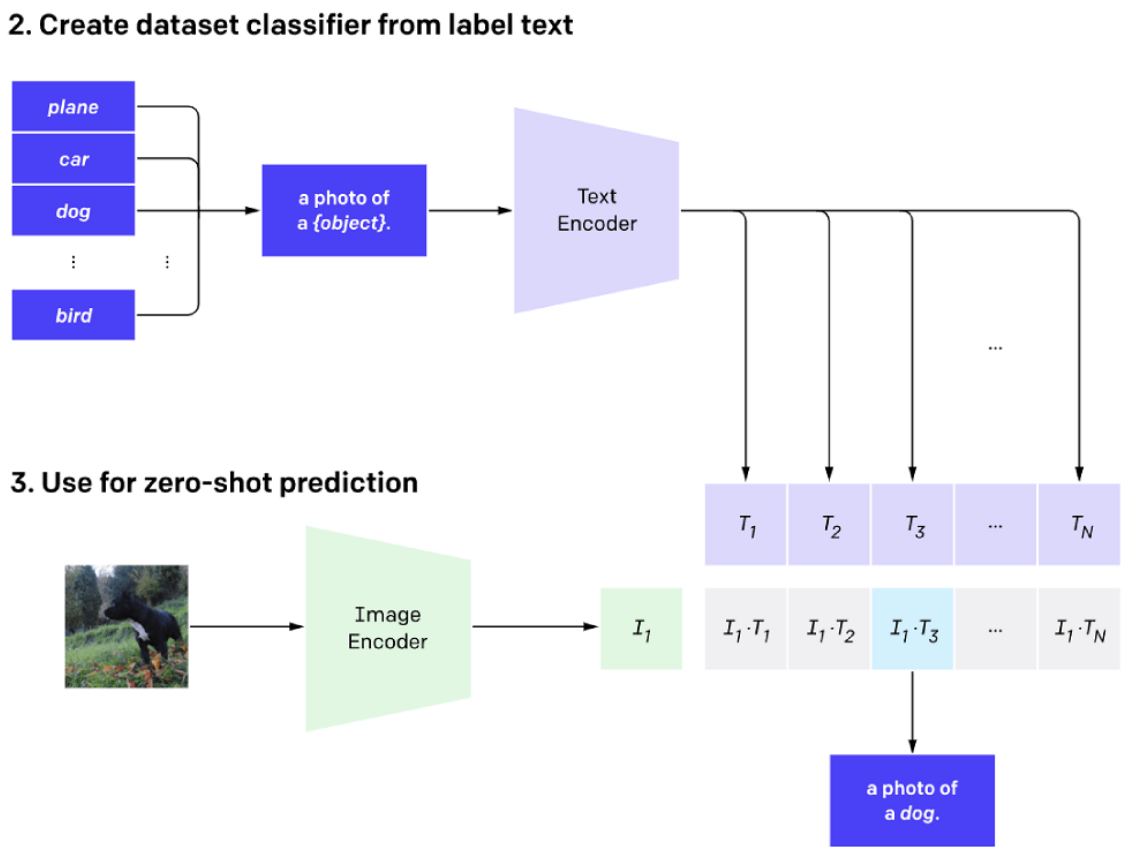
1. Contrastive pre-training

During this phase, a batch of 32,768 pairs of image and text is passed through the text and image encoders simultaneously to generate the vector representations of the text and the associated image, respectively.

The training is done by searching for each image, the closest text representation across the entire batch, which corresponds to maximizing cosine similarity between the actual N pairs that are maximally close.

Also, it makes the actual images far awa
similarity.

Finally, a symmetric cross-entropy loss is optimized over the previously computed similarity scores.



Classification dataset creation and zero-shot prediction
Source: OpenAI

2. Create dataset classifier from label text

This second step section encodes all the labels/objects in the following context format: "***a photo of a {object}***. The vector representation of each context is generated from the text encoder.

If we have *dog, car*, and *plane* as the classes of the dataset, we will output the following context representations:

- a photo of a dog
- a photo of a car
- a photo of a plane

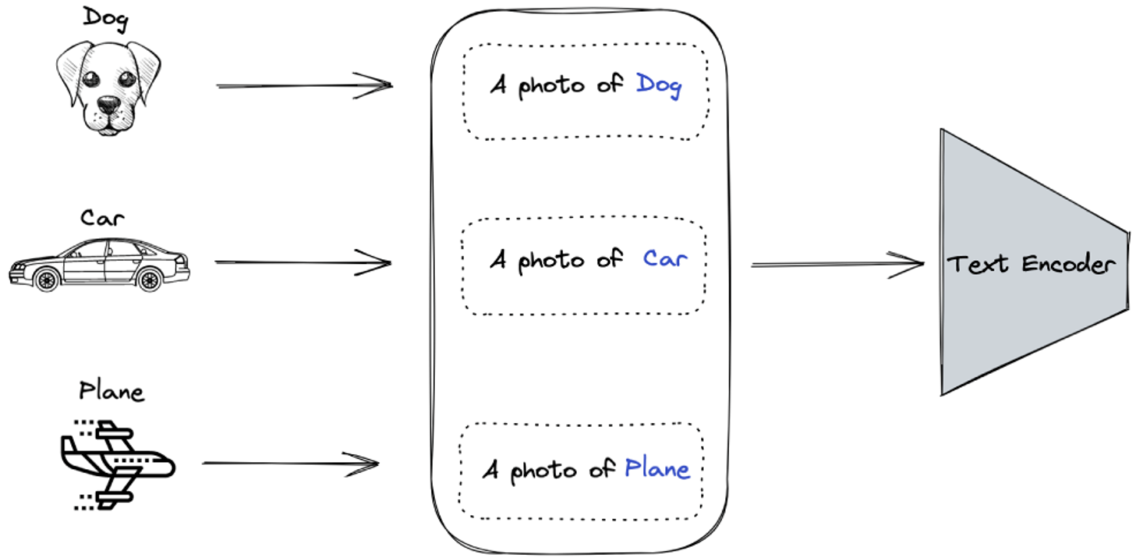


Image illustration of the context representations

3. Use of zero-shot prediction

We use the output of section 2 to predict which image vector corresponds to which context vector. The benefit of applying the zero-shot prediction approach is to make CLIP models generalize better on unseen data.

Implementation of CLIP With Python

Now that we know the architecture of CLIP and how it works, this section will walk you through all the steps to successfully implement two real-world scenarios. First, you will understand how to perform an image search in natural language. Also, you will be able to perform an image-to-image search using.

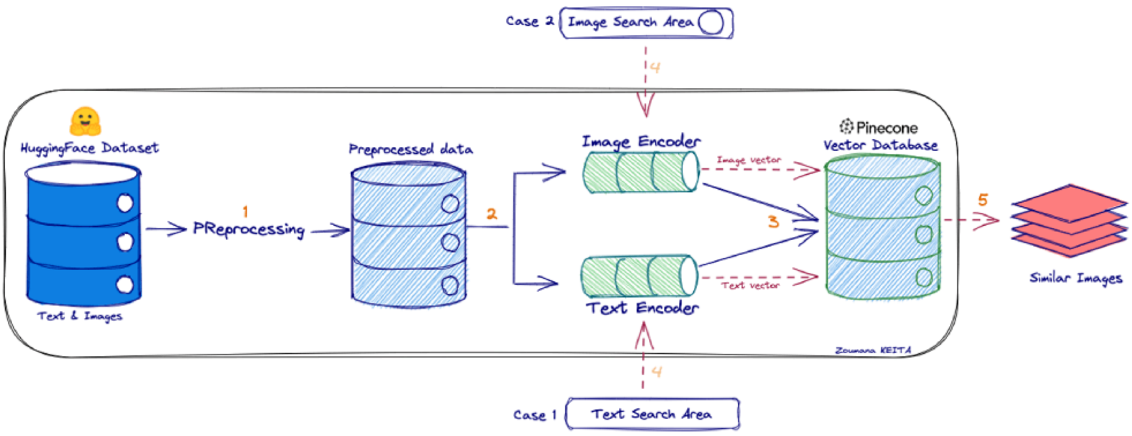
At the end of the process, you will understand the benefits of using a vector database for such a use case.

General workflow of

(Follow along with the Colab notebook!)

The end-to-end process is explained through the workflow below. We start by collecting data from the Hugging Face dataset, which is then processed to further generate vector index vectors through the Image and Text Encoders. Finally, the Pinecone client is used to insert them to a vector index.

The user will then be able to search images based on either text or another image.



General workflow for image search

Prerequisites

The following libraries are required to create the implementation.

Install the libraries

```
%%bash
# Uncomment this if using it for the first time. -qqq for ZERO-OUT
pip3 -qqq install transformers torch datasets

# The following two libraries avoid the UnidentifiedImageError
pip3 -qqq install gdcm
pip3 -qqq install pydicor
pip -qqq install faiss-gı
pip -qqq install pinecone
```


Import the libraries

```python
import os
import faiss
import torch
import skimage
import requests
import pinecone
import numpy as np
import pandas as pd
from PIL import Image
from io import BytesIO
import IPython.display
import matplotlib.pyplot as plt
from datasets import load_dataset
from collections import OrderedDict
from transformers import CLIPProcessor, CLIPModel, CLIPTokenizer
```

Data acquisition and exploration

The conceptual captions dataset consists of around 3.3M images with two main columns: the image URL and its caption. You can find more details from the corresponding huggingface link.

```python
# Get the dataset
image_data = load_dataset("conceptual_captions", split="train")
```

Data preprocessing

Not all URLs in the dataset are valid. We fix that by testing and removing all erroneous URL entries.

```python
def check_valid_URLs(ima
    try:
      response = requests
      Image.open(BytesIO(
      return True
    except:
      return False
```

```
def get_image(image_URL):
    response = requests.get(image_URL)
    image = Image.open(BytesIO(response.content)).convert("RGB")
    return image
```

The following expression creates a new dataframe with a new column "is_valid" which is True when the URL is valid or False otherwise.

```
# Transform dataframe
image_data_df["is_valid"] = image_data_df["image_url"].apply(check_va
# Get valid URLs
image_data_df = image_data_df[image_data_df["is_valid"]==True]
# Get image from URL
image_data_df["image"] = image_data_df["image_url"].apply(get_image)
```

The second step is to download the images from the URLs. This helps us avoid constant web requests.

Image and text embeddings implementation

The prerequisites to successfully implement the encoders are the model, the processor, and the tokenizer.

The following function fulfills those requirements from the model ID and the device used for the computation, either CPU or GPU.

```
def get_model_info(model_ID, device):
# Save the model to device
    model = CLIPModel.from_pretrained(model_ID).to(device)
    # Get the processor
    processor = CLIPProce
# Get the tokenizer
    tokenizer = CLIPToke
        # Return model, p
    return model, proces
# Set the device
```

```
device = "cuda" if torch.cuda.is_available() else "cpu"
# Define the model ID
model_ID = "openai/clip-vit-base-patch32"
# Get model, processor & tokenizer
model, processor, tokenizer = get_model_info(model_ID, device)
```

Text embeddings

We start by generating the embedding of a single text before applying the same function across the entire dataset.

```
def get_single_text_embedding(text):
inputs = tokenizer(text, return_tensors = "pt")
    text_embeddings = model.get_text_features(**inputs)
    # convert the embeddings to numpy array
    embedding_as_np = text_embeddings.cpu().detach().numpy()
return embedding_as_np
def get_all_text_embeddings(df, text_col):
df["text_embeddings"] = df[str(text_col)].apply(get_single_text_embe
return df
# Apply the functions to the dataset
image_data_df = get_all_text_embeddings(image_data_df, "caption")
```

The first five rows look like this:

	image_url	caption	is_valid	image	text_embeddings
0	http://lh6.ggpht.com/-IvRtNLNcG8o/TpFyrudaT6I/...	a very typical bus station	True	<PIL.Image.Image image mode=RGB size=800x534 a...	[[0.25922304, -0.08825898, 0.020317025, -0.127...
1	http://78.media.tumblr.com/3b133294bdc7c7784b7...	sierra looked stunning in this top and this sk...	True	<PIL.Image.Image image mode=RGB size=500x441 a...	[[0.0041467994, 0.18943565, -0.123970225, 0.30...
2	https://media.gettyimages.com/photos/young-con...	young confused girl standing in front of a war...	True	<PIL.Image.Image image mode=RGB size=490x612 a...	[[-0.28737983, -0.34814143, -0.04288538, 0.401...
3	https://thumb1.shutterstock.com/display_pic_wi...	interior design of modern living room with fir...	True	<PIL.Image.Image image mode=RGB size=450x470 a...	[[0.56064534, -0.15138063, -0.43740302, -0.339...
4	https://thumb1.shutterstock.com/display_pic_wi...	cybernetic scene isolated on white background .	True	<PIL.Image.Image image mode=RGB size=450x470 a...	[[0.035292536, 0.24262792, -0.12724756, -0.210...

Format of the vector ind

Image embeddings

The same process is used for image embeddings but with different functions.

```
def get_single_image_embedding(my_image):
image = processor(
        text = None,
        images = my_image,
        return_tensors="pt"
        )["pixel_values"].to(device)
embedding = model.get_image_features(image)
# convert the embeddings to numpy array
    embedding_as_np = embedding.cpu().detach().numpy()
    return embedding_as_np
 def get_all_images_embedding(df, img_column):
    df["img_embeddings"] = df[str(img_column)].apply(get_single_image
    return df
image_data_df = get_all_images_embedding(image_data_df, "image")
```

The final format of the text and image vector index looks like this:

	image_url	caption	is_valid	image	text_embeddings	img_embeddings
0	http://lh6.ggpht.com/-IvRtNLNcG8o/TpFyrudaT6I/...	a very typical bus station	True	<PIL.Image.Image image mode=RGB size=800x534 a...	[[0.25922304, -0.08825898, 0.020317025, -0.127...	[[-0.0034022853, -0.053583913, 0.35247508, 0.3...
1	http://78.media.tumblr.com/3b133294bdc7c7784b7...	sierra looked stunning in this top and this sk...	True	<PIL.Image.Image image mode=RGB size=500x441 a...	[[0.0041467994, 0.18943565, -0.123970225, 0.30...	[[-0.25019708, -0.1325763, 0.09706805, 0.97886...
2	https://media.gettyimages.com/photos/young-con...	young confused girl standing in front of a war...	True	<PIL.Image.Image image mode=RGB size=490x612 a...	[[-0.28737983, -0.34814143, -0.04288538, 0.401...	[[-0.36655784, 0.3118331, -0.13266361, 0.34909...
3	https://thumb1.shutterstock.com/display_pic_wi...	interior design of modern living room with fir...	True	<PIL.Image.Image image mode=RGB size=450x470 a...	[[0.56064534, -0.15138063, -0.43740302, -0.339...	[[-0.17221001, -0.29784596, -0.10141284, -0.06...
4	https://thumb1.shutterstock.com/display_pic_wi...	cybernetic scene isolated on white background .	True	<PIL.Image.Image image mode=RGB size=450x470 a...	[[0.035292536, 0.24262792, -0.12724756, -0.210...	[[0.18897031, -0.0012195408, -0.6513251, -0.12...

Vector index with image and captions embeddings (Image by Author)

Vector storage approach—Local vector index Vs. A cloud-based vector index

In this section, we will explore two differ for performing the searches: The first is Pinecone. Both approaches use the cos

Using local dataframe as vector index

The helper function ***get_top_N_images*** generates similar images for the two scenarios illustrated in the workflow above: text-to-image search or image-to-image search.

```
1  from sklearn.metrics.pairwise import cosine_similarity
2  def get_top_N_images(query, data, top_K=4, search_criterion="text"):
3      # Text to image Search
4      if(search_criterion.lower() == "text"):
5        query_vect = get_single_text_embedding(query)
6      # Image to image Search
7      else:
8        query_vect = get_single_image_embedding(query)
9      # Relevant columns
10     revevant_cols = ["caption", "image", "cos_sim"]
11     # Run similarity Search
12     data["cos_sim"] = data["img_embeddings"].apply(lambda x: cosine_s
13     data["cos_sim"] = data["cos_sim"].apply(lambda x: x[0][0])
14     """
15     Retrieve top_K (4 is default value) articles similar to the query
16     """
17    most_similar_articles = data.sort_values(by='cos_sim',  ascending=
18     return most_similar_articles[revevant_cols].reset_index()
```

Let's understand how we perform the recommendation.

→ The user provides either a text or an image as a search criterion, but the model performs a text-to-image search by default.

→ In line 17, a cosine similarity is performed between each image vector and the user's input vector.

→ Finally, in line 24, sort the result based on the similarity score in descending order, and we return the most similar images by excludi

Example of searches

This helper function makes it easy to ha
images. Each image will have the corres

```
1  def plot_images_by_side(top_images):
2   index_values = list(top_images.index.values)
3   list_images = [top_images.iloc[idx].image for idx in index_values]
4   list_captions = [top_images.iloc[idx].caption for idx in index_value
5   similarity_score = [top_images.iloc[idx].cos_sim for idx in index_va
6   n_row = n_col = 2
7   _, axs = plt.subplots(n_row, n_col, figsize=(12, 12))
8   axs = axs.flatten()
9   for img, ax, caption, sim_score in zip(list_images, axs, list_capti
10      ax.imshow(img)
11      sim_score = 100*float("{:.2f}".format(sim_score))
12      ax.title.set_text(f"Caption: {caption}\nSimilarity: {sim_score}
13  plt.show()
```

Text-to-image

→ First, the user provides the text that is used for the search.

→ Second, we run a similarity search.

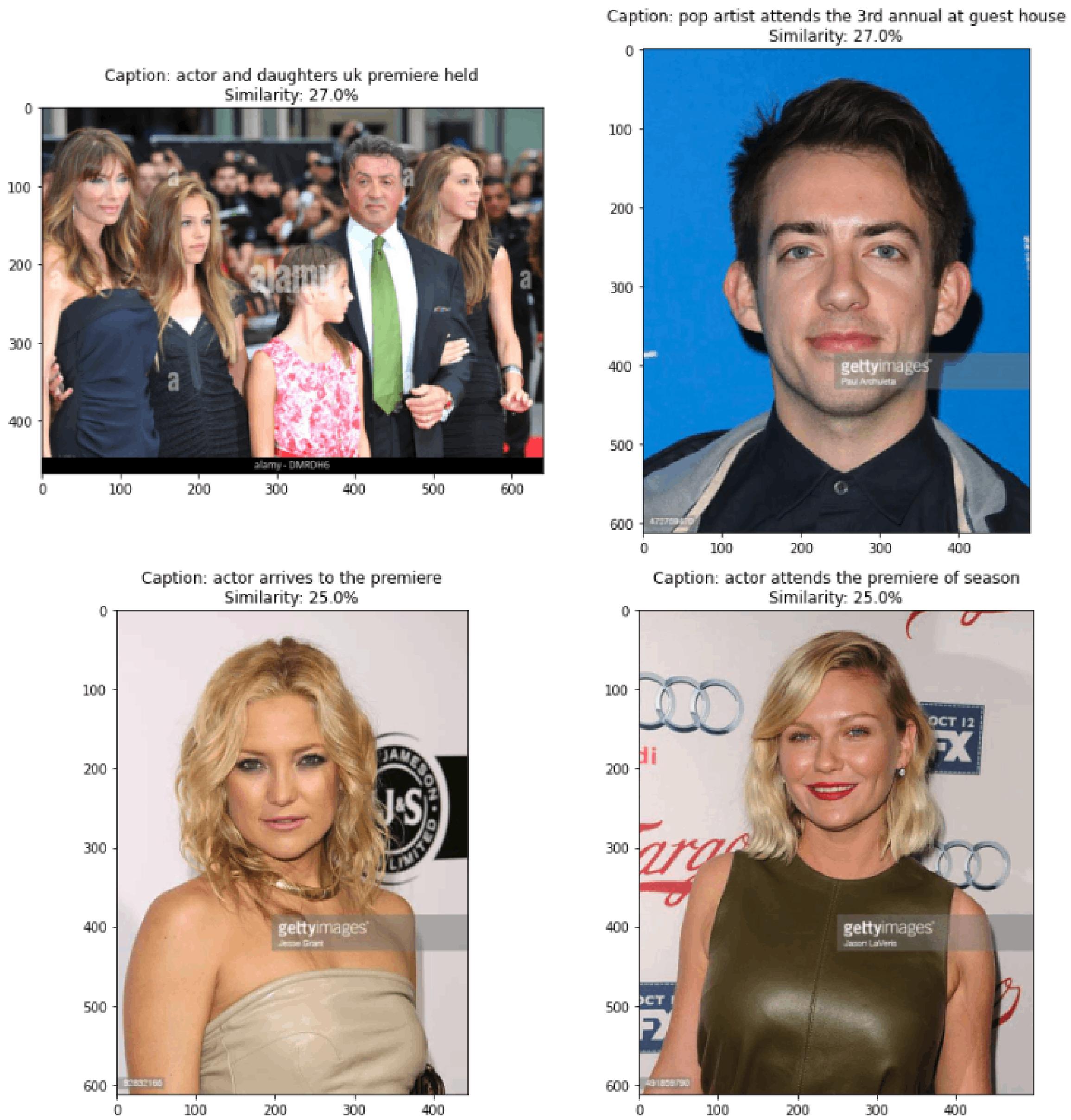
→ Third, we plot the images recommended by the algorithm.

```
1  query_caption = image_data_df.iloc[10].caption
2  # Print the original query text
3  print("Query: {}".format(query_caption))
4  # Run the similarity search
5  top_images = get_top_N_images(query_caption, image_data_df)
6  # Plot the recommended images
7  plot_images_by_side(top_images)
```

Line 3 generates the following text:

Query: actor arrives for the premiere of

Line 9 produces the plot below.

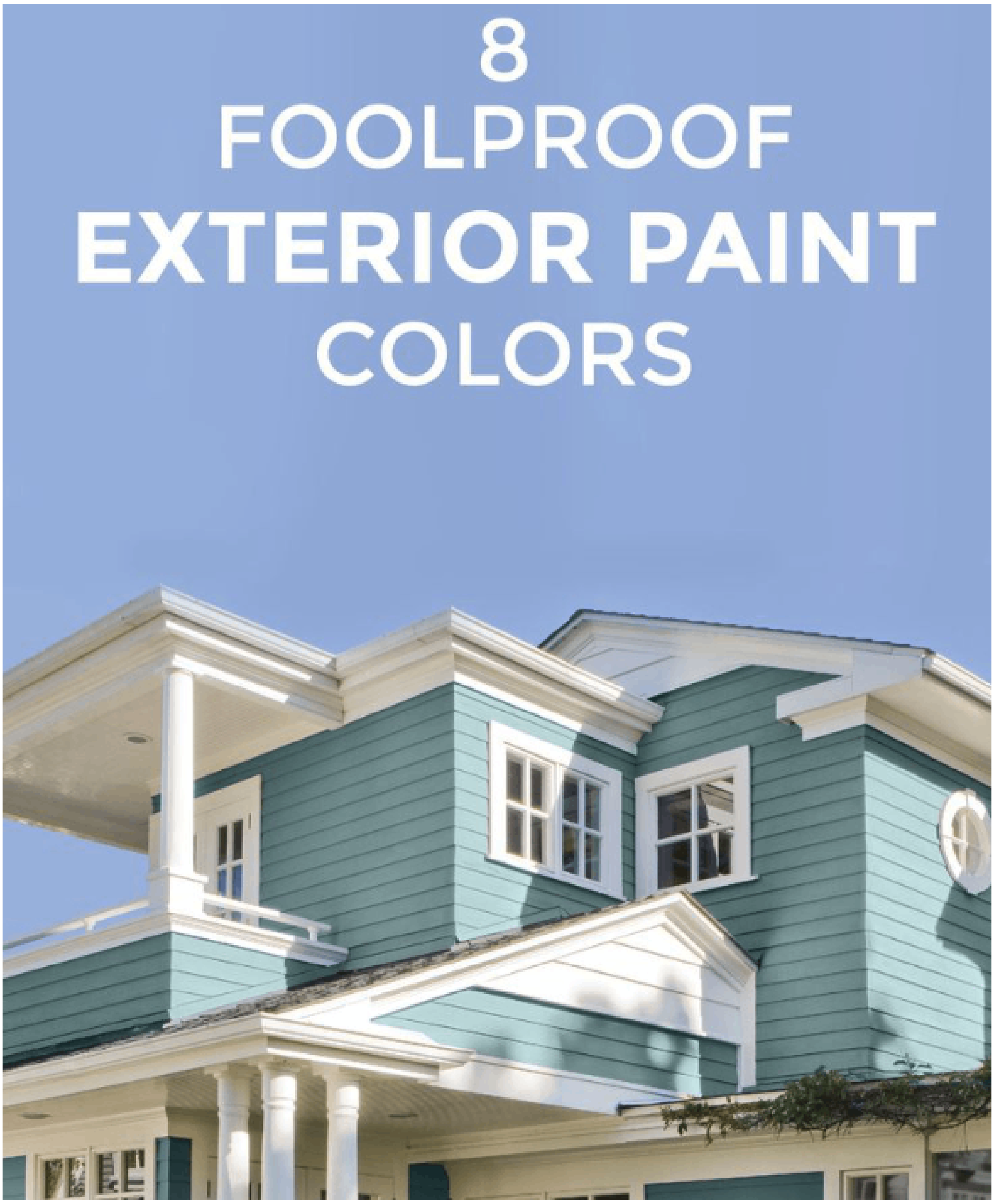


Images corresponding to the text: “actor arrives for the premiere of the film”

Image-to-image

The same process applies. The only difference this time is that the user provides an image instead of a caption.

```
# Get the query image an
query_image = image_data_
query_image
```



Original image of search (image at the index)

```
# Run the similarity search and plot the result
top_images = get_top_N_i
# Plot the result
plot_images_by_side(top_
```

We run the search by specifying the search_criterion which is “image” in line 2.

The final result is shown below.

Images corresponding to the image-to-image search (Image by Author)

We can observe that some of the images are less similar which introduces noise in the recommendation. We can reduce that noise by specifying a threshold level of similarity. For instance, consider all the images with at least 60% similarity.

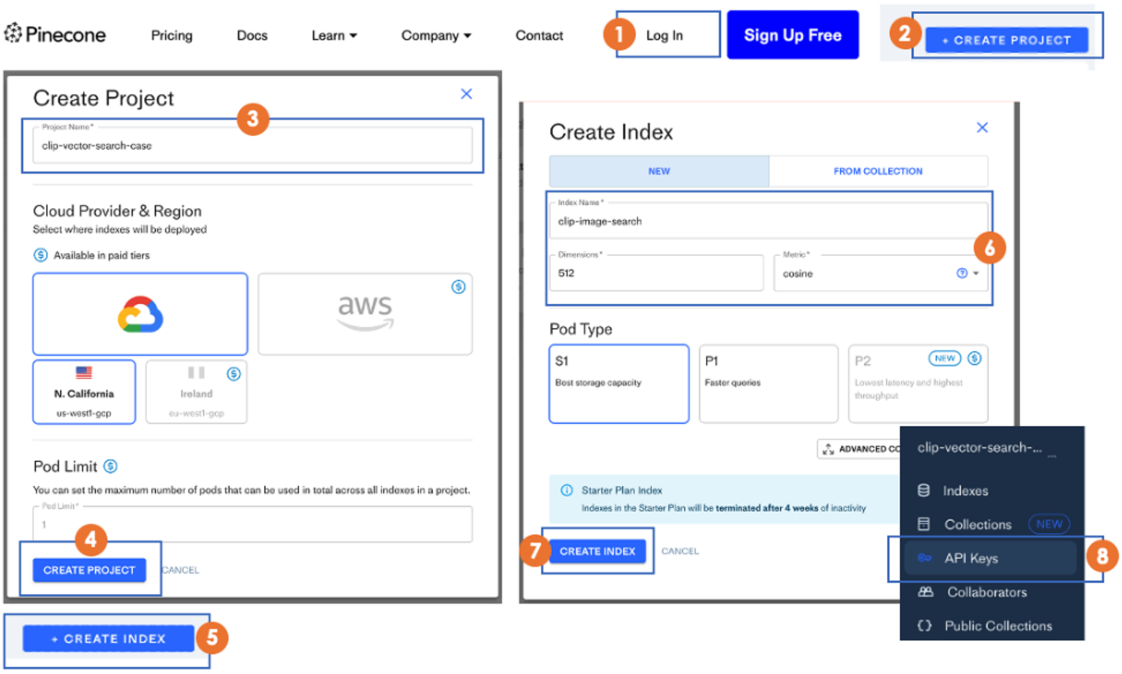
Leveraging the power of a managed vector index using Pinecone

Pinecone provides a fully-managed, eas high-performance vector search applica

This section will walk you through the steps from acquiring your API credentials to implementing the search engine.

Acquire your Pinecone API

Below are the eight steps to acquire your API credentials, starting from the Pinecone website.



Eight main steps to acquire your Pinecone Client API

Configure the vector index

From the API, we can create the index that allows us to perform all the create, update, delete, and insert actions.

```
pinecone.init(
    api_key = "YOUR_API_KEY",
    environment="YOUR_ENV"  # find next to API key in console
)
my_index_name = "clip-ima
vector_dim = image_data_c

if my_index_name not in p
  # Create the vectors di
  pinecone.create_index(na
```

```
11                         dimension=vector_dim,
12                         metric="cosine", shards=1,
13                         pod_type='s1.x1')
14 # Connect to the index
15 my_index = pinecone.Index(index_name = my_index_name)
```

- ***pinecone.init*** section initializes the pinecone workspace to allow future interactions.
- from lines 8 to 9 we specify the name we want for the vector index, and also the dimension of the vectors, which is 512 in our scenario.
- from lines 11 to 16 we create the index if it does not already exist.

The result of the following instruction shows that we have no data in the index.

```
my_index.describe_index_stats()
```

The only information we have is the dimension, which is 512.

```
{'dimension': 512,
           'index_fullness': 0.0,
            'namespaces': {},
'total_vector_count': 0}
```

Populate the database

Now that we have configured the Pinecone database, the next step is to populate it with the following code.


```
image_data_df["vector_id'
image_data_df["vector_id'
# Get all the metadata
final_metadata = []
```

```
5   for index in range(len(image_data_df)):
6    final_metadata.append({
7        'ID':  index,
8        'caption': image_data_df.iloc[index].caption,
9        'image': image_data_df.iloc[index].image_url
10   })
11  image_IDs = image_data_df.vector_id.tolist()
12  image_embeddings = [arr.tolist() for arr in image_data_df.img_embedd:
13  # Create the single list of dictionary format to insert
14  data_to_upsert = list(zip(image_IDs, image_embeddings, final_metadata
15  # Upload the final data
16  my_index.upsert(vectors = data_to_upsert)
17  # Check index size for each namespace
18  my_index.describe_index_stats()
```

Let's understand what is going on here.

The data to upsert requires three components: the unique identifiers (IDs) of each observation, the list of embeddings being stored, and the metadata containing additional information about the data to store.

→ From lines 5 to 12, the metadata is created by storing the "ID", "caption" and "URL" of each observation.

→ On lines 14 and 15, we generate a list of IDs, and convert the embeddings into a list of lists.

→ Then, we create a list of dictionaries mapping the IDs, embeddings, and metadata.

→ The final data is upserted to the index with the *.upsert()* function.

Similarly to the previous scenario, we can check that all vectors have been upserted via `my_index.describe_index_stats()` .

Start the query

All that remains is to query our index usi
Both will use the following syntax:

```
my_index.query(my_query_embedding, top_k=N, include_metadata=True)
```

→ *my_query_embedding* is the embedding (as a list) of the query (caption or image) provided by the user.

→ *N* corresponds to the top number of results to return.

→ *include_metadata=True* means that we want the query result to include metadata.

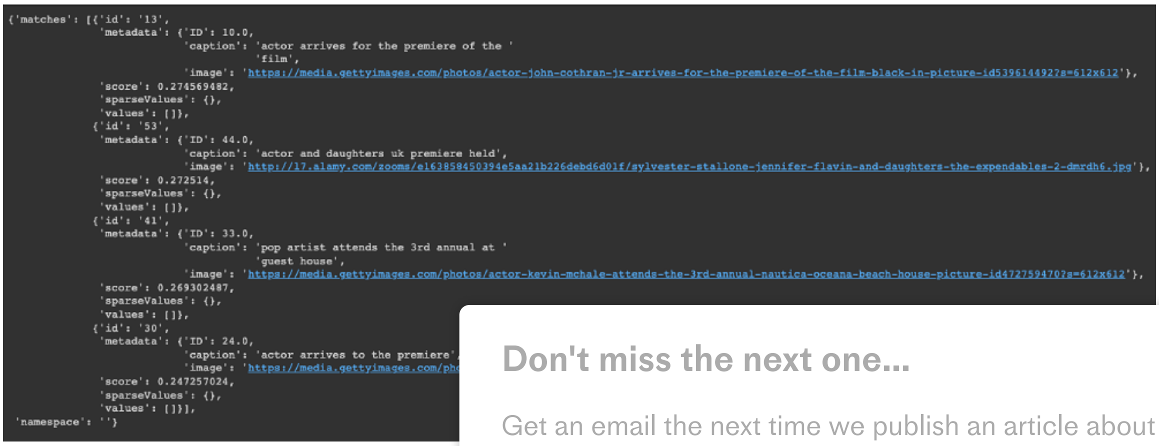
Text to image

```
# Get the query text
text_query = image_data_df.iloc[10].caption

# Get the caption embedding
query_embedding = get_single_text_embedding(text_query).tolist()

# Run the query
my_index.query(query_embedding, top_k=4, include_metadata=True)
```

Below is the JSON response returned from the query




```
{'matches': [{'id': '13',
              'metadata': {'ID': 10.0,
                           'caption': 'actor arrives for the premiere of the '
                                      'film',
                           'image': 'https://media.gettyimages.com/photos/actor-john-cothran-jr-arrives-for-the-premiere-of-the-film-black-in-picture-id539614492?s=612x612'},
              'score': 0.274569482,
              'sparseValues': {},
              'values': []},
             {'id': '53',
              'metadata': {'ID': 44.0,
                           'caption': 'actor and daughters uk premiere held',
                           'image': 'http://l7.alamy.com/zooms/e163858450394e5aa21b226debd6d01f/sylvester-stallone-jennifer-flavin-and-daughters-the-expendables-2-dmrdh6.jpg'},
              'score': 0.272514,
              'sparseValues': {},
              'values': []},
             {'id': '41',
              'metadata': {'ID': 33.0,
                           'caption': 'pop artist attends the 3rd annual at '
                                      'guest house',
                           'image': 'https://media.gettyimages.com/photos/actor-kevin-mchale-attends-the-3rd-annual-nautica-oceana-beach-house-picture-id472759470?s=612x612'},
              'score': 0.269302487,
              'sparseValues': {},
              'values': []},
             {'id': '30',
              'metadata': {'ID': 24.0,
                           'caption': 'actor arrives to the premiere',
                           'image': 'https://media.gettyimages.com/ph
              'score': 0.247257024,
              'sparseValues': {},
              'values': []}],
 'namespace': ''}
```


text-to-imag

From the "matches" attribute, we can observe the top four most similar images returned by the query.

Image-to-image

The same approach applies to image-to-image search.

```
image_query = image_data_df.iloc[43].image
```

This is the image provided by the user as the search criteria.

Query image

```
# Get the text embedding
query_embedding = get_si
```

```
# Run the query
my_index.query(query_embedding, top_k=4, include_metadata=True)
```


```
{'matches': [{'id': '52',
              'metadata': {'ID': 43.0,
                           'caption': 'a demonstration of a group of people '
                                      'practicing their rights',
                           'image': 'https://media.gettyimages.com/photos/demonstration-of-a-group-of-people-practicing-their-rights-picture-id172184947?s=612x612'},
              'score': 1.0,
              'sparseValues': {},
              'values': []},
                           'caption': 'red and white flag on the mast',
                           'image': 'https://ak6.picdn.net/shutterstock/videos/12263756/thumb/1.jpg'},
              'score': 0.577650547,
              'sparseValues': {},
              'values': []},
             {'id': '34',
              'metadata': {'ID': 27.0,
                           'caption': 'architectural details of a bridge',
                           'image': 'https://media.gettyimages.com/photos/architectural-details-of-a-bridge-picture-id511832249?s=612x612'},
              'score': 0.516087532,
              'sparseValues': {},
              'values': []},
             {'id': '80',
              'metadata': {'ID': 68.0,
                           'caption': 'farm tractor is moving on the field , '
                                      'cultivating land',
                           'image': 'https://ak7.picdn.net/shutterstock/videos/3682697/thumb/1.jpg'},
              'score': 0.494651496,
              'sparseValues': {},
              'values': []}],
 'namespace': ''}
```


image-to-image query result (Image by Author)

Once you've finished don't forget to delete your index to free up your resources with:

```
pinecone.delete_index(my_index)
```

What are the advantages to using a Pinecone over a local pandas dataframe?

This approach using Pinecone has seve

→ ***Simplicity***: the querying approach is r
the full responsibility of managing the ve

→ ***Speed***: Pinecone approach is faster, v

→ ***Scalability***: vector index hosted on Pinecone is scalable with little-to-no user effort from us. The first approach would become increasingly complex and slow as we scale.

→ ***Lower chance of information loss***: the vector index based on Pinecone is hosted in the cloud with backups and high information security. The first approach is too high risk for production use-cases.

→ ***Web-service friendly***: the result provided by the query is in JSON format and can be consumed by other applications, making it a better fit for web-based applications.

Conclusion

Congratulations, you have just learned how to fully implement an image search application using both image and natural language. I hope the benefits highlighted are valid enough to take your project to the next level using vector databases.

Multiple resources are available at our Learning Center to further your learning.

The source code for the article is available here.

References

Code Notebook

[1] A. Radford, J. W. Kim, et al., Learning Transferable Visual Models From Natural Language Supervision (2021)

[2] A. Vaswani, et al., Attention Is All You Need (2017), NeurIPS

Share via:

Zoumana Keita
Technical Writer