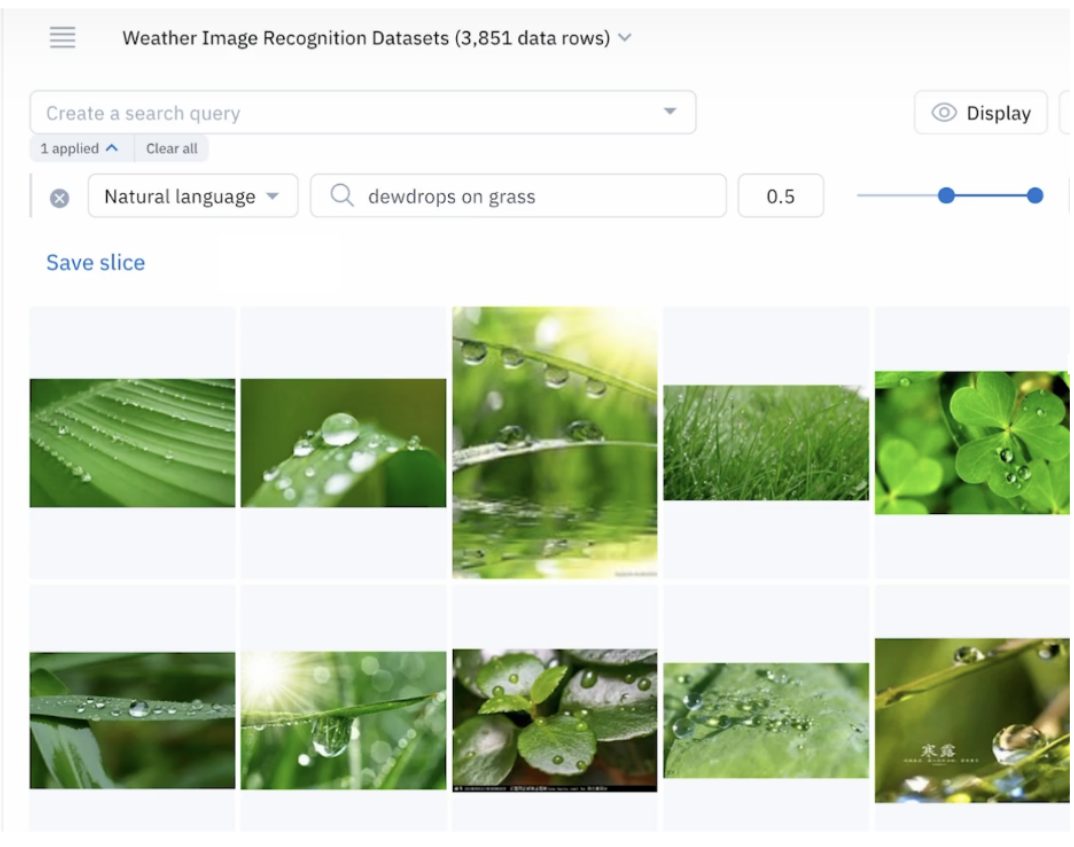
← All blog posts

Labelbox • April 5, 2023

How vector similarity search works



How vector similarity search works

A vector search database, also known as a vector similarity search engine or vector database, is a type of database that is designed to store, retrieve, and search for vectors based on their similarity given a query. Vector search databases are frequently used in applications such as image retrieval, natural language processing, recommendation systems, and more.

Given the increasing excitement and interest in vector search and vector databases, we wanted to provide a high-level introduction of how it works and how it helps teams access information faster. In this post, we'll take a look at what vector search is and why you might use it, while walking you through some of the main types of vector databases and use cases.

Vector search differs from the previous generation of search engines because these databases represented data primarily using the text that they contain. They performed various processing steps on that text to improve the performance of search against that text, which included steps like preprocessing, query expansion, synonym lists, etc. These approaches aimed to address the challenges of words that have similar meanings and create an accurate representation of the intended meaning in a given text. The issue at hand is how to identify and compare two texts that express the same idea, but do not match exactly. For instance, if a user searches for "tree," but a text uses the term "plant."

Since deep learning has become popular, we've started to represent things using embeddings (which are vector representations of high-dimensional data), rather than using explicit text features. The key idea behind vector search databases is to represent data items (e.g., images, documents, user profiles) as vectors in a high-dimensional space. Similarity between vectors is then measured using a distance metric, such as cosine similarity or Euclidean distance. The goal of a vector search database is to quickly find the most similar vectors to a given query vector.

Let's take a look at how vector search databases typically work:

1.**Vector embeddings generation**: Data items are first converted into vectors using a feature extraction or embedding technique. For example, images can be represented as vectors using convolutional neural networks (CNNs), and text documents can be represented as vectors using word embeddings or sentence embeddings.

2. Indexing & querying: The vectors are then indexed in the vector search database. Indexing is the process of organizing the vectors in a way that allows for efficient similarity search. Various indexing techniques and data structures, such as k-d trees, ball trees, and approximate nearest neighbor (ANN) algorithms, can be used to speed up the search process.

Given a query vector, the vector search database retrieves the most similar vectors from the indexed dataset. The query vector is typically generated using the same feature extraction or embedding technique used to create the indexed vectors. The similarity between the query vector and the indexed vectors is measured using a distance metric, and the most similar vectors are returned as the search results. The retrieved vectors are ranked based on their similarity scores, and the top-k most similar vectors are returned to the user.

Vector embeddings generation

The vectors used in vector search can represent various types of data, such as text, images, audio, or other data types. The process of creating embeddings for vector search depends on the type of data being represented and the specific use case. Below, we'll describe how embeddings are created for vector search in the specific context of both text data and image data:

1. Text data: For text data, embeddings can be created using methods such as Word2Vec, GloVe or BERT. These methods create vector representations of words, phrases, or sentences based on the semantic and syntactic relationships between them. The embeddings are typically generated by training neural network models on large collections of text. Some popular methods for creating text embeddings include:

- Bag-of-words (BoW) model
- Word embeddings (Word2Vec, GloVe)
- Pre-trained language models (BERT, GPT)

2. Image data: For image data, embeddings can be created using convolutional neural networks (CNNs). CNNs are trained on large datasets of labeled images to perform tasks such as image classification or object detection. The intermediate layers of the CNN can be used to extract feature vectors (embeddings) that represent the content of the images. These embeddings can then be used for vector search to find similar images. Some popular methods for creating image embeddings include:

- Pre-trained language models: Such as VGG, ResNet, Inception, and MobileNet can be used as feature extractors to create image embeddings. The output of a specific layer or a combination of layer outputs is used as the embedding.
- Autoencoders: These unsupervised models learn to compress and reconstruct images. The compressed representation (latent space) serves as the embedding.

3. Audio data: For audio data, embeddings can be created using methods such as spectrogram analysis or deep learning models like recurrent neural networks (RNNs) or CNNs. These models can be trained on audio data to extract meaningful features and create embeddings that capture the characteristics of the audio signals.

- Mel-Frequency Cepstral Coefficients (MFCCs): A widely used feature extraction technique for audio signals, which captures the spectral shape of the signal.
- Spectrogram-based embeddings: Convert audio signals into spectrograms and use them as input to CNNs or other models to learn embeddings.
- Recurrent Neural Networks (RNNs): Audio signals can be represented as sequences and processed by RNNs (e.g., LSTMs or GRUs) to learn embeddings.

4. Multimodal embeddings Multi-modal embeddings refer to the process of creating vector representations (embeddings) for data that comes from multiple modalities, such as text, images, audio, and video. The goal of multi-modal embeddings is to create a shared embedding space where similar items from different modalities are close to each other, regardless of the modality they originate from. This shared space enables tasks such as cross-modal retrieval, multi-modal classification, and multi-modal generation. Examples include: OpenAI CLIP and BLIP.

5. Other data types: For other types of data, such as tabular data or time-series data, embeddings can be created using various machine learning techniques, including autoencoders, clustering algorithms, or other neural network architectures. The goal is to create embeddings that capture the relevant features and patterns in your data.

Once embeddings are created for different modalities, they can be indexed using approximate nearest neighbor (ANN) algorithms like Annoy, HNSW, or Faiss, allowing efficient similarity search in high-dimensional vector spaces. Let’s take a look at how indexing and querying works by utilizing these types of algorithms.

How indexing and querying works

Vector search finds similar data using approximate nearing neighbor (ANN) algorithms. Compared to traditional keyword search, vector search yields more relevant results and executes faster.

Approximate Nearest Neighbors (ANN) is a class of algorithms used to find the nearest neighbors of a query point in a high-dimensional dataset. These algorithms are called "approximate" because they trade off a small amount of accuracy for a significant speedup compared to exact nearest neighbor search algorithms. ANN algorithms are commonly used in applications such as recommendation systems, image retrieval, natural language processing, and more.

The general idea behind ANN algorithms is to preprocess the dataset to create an index or data structure that allows for efficient querying. When a query point is provided, the algorithm uses the index to quickly identify a set of candidate points that are likely to be close to the query point. This way, when querying the vector database to

points around it.

There are several popular ANN algorithms, each with its own approach to building the index and searching for nearest neighbors. Here are brief descriptions of a few common ANN algorithms:

1. Brute force: Whilst not technically an ANN algorithm it provides the most intuitive solution and a baseline to evaluate all other models. Also known as the exact nearest neighbor search, this is a straightforward method for finding the nearest neighbors of a query vector in a dataset. Unlike approximate methods, the brute force algorithm guarantees finding the exact nearest neighbors by exhaustively computing the distances between the query vector and every vector in the dataset. Due to its simplicity, the brute force algorithm is often used as a baseline for evaluating the performance of more sophisticated ANN algorithms.

2. Locality-Sensitive Hashing (LSH): LSH is based on the idea of hashing similar points to the same hash bucket. The dataset is hashed multiple times using different hash functions, each of which is designed to ensure that similar points are likely to collide. It differs from conventional hashing techniques in that hash collisions are maximized, not minimized. During the query phase, the query point is hashed using the same hash functions, and the algorithm retrieves the points in the corresponding hash buckets as candidates.

3. k-d Trees: k-d trees are binary search trees that partition the data along different dimensions at each level of the tree. During the construction of the k-d tree, the algorithm selects a dimension and a splitting value to partition the data into two subsets. The process is recursively applied to each subset until the tree is fully constructed. During the query phase, the algorithm traverses the tree to find the nearest neighbors. k-d trees work well for low-dimensional data but become less efficient as the dimensionality increases due to the curse of dimensionality.

4. Annoy (Approximate Nearest Neighbors Oh Yeah): Annoy is an open-source library by Spotify that builds a forest of binary search trees for approximate nearest neighbor search.Each tree is constructed by recursively splitting the data using random hyperplanes. During the query phase, the algorithm traverses multiple trees to find the nearest neighbors. It has the ability to use static files as indexes. In particular, this means one can share indexes across processes. Annoy also decouples creating indexes from loading them, so one can pass around indexes as files and map them into memory quickly. Annoy is designed to work efficiently with high-dimensional data and is currently used by Spotify for their music recommendation engine.

5. Hierarchical Navigable Small World (HNSW) Graphs:
The Hierarchical Navigable Small World (HNSW) algorithm is an approximate nearest neighbor search method used for vector search in high-dimensional spaces. It constructs a hierarchical graph where each node represents a data point, and edges connect nearby points. The graph has multiple layers, with each layer representing a different level of granularity. The algorithm allows for efficient nearest neighbor searches by navigating the graph's layers. HNSW is known for its high search speed and accuracy.

6. ScaNN (Scalable Nearest Neighbors)
The ScaNN (Scalable Nearest Neighbors) algorithm is an approximate nearest neighbor search method developed by Google Research. It is designed to efficiently search for nearest neighbors in large-scale, high-dimensional datasets. ScaNN achieves high search accuracy and speed by combining several techniques, including quantization, vector decomposition, and graph-based search.

7. Hybrid, as the name suggests, is some form of a combination of the above implementations.There’s many more algorithms currently being researched and developed. This field is progressing rapidly given the rise of foundation models and importance of vector search in this context.

Using vector databases

There are a number of options when it comes to vector databases. Each of them has its unique advantages. Depending on the nature of your application, and whether you’re trying to build the infrastructure from scratch, one of these may be a right option for you.

Vector database options include:

- Vertex matching engine by Google
- Pinecone, a fully managed vector database
- Weaviate, an open-source vector search engine
- Redis as a vector database
- Qdrant, a vector search engine
- Milvus, a vector database built for scalable similarity search

- Chroma, an open-source embeddings store
- Elastic, an open source and hosted database

Utilizing Labelbox Catalog for vector search applications

Labelbox Catalog is designed with vector search capabilities to help you better organize, enrich and make useful applications with unstructured data. With Catalog, teams can easily upload text snippets, conversations or PDF documents within the UI or via the Python SDK. Labelbox will automatically generate embeddings with the uploaded data and allow you to start capitalizing on the power of vector search.

Natural language search

Natural language search is powered by *vector embeddings*. A vector embedding is a numerical representation of a piece of data (e.g., an image, text, document, or video) that translates the raw data into a lower-dimensional space.

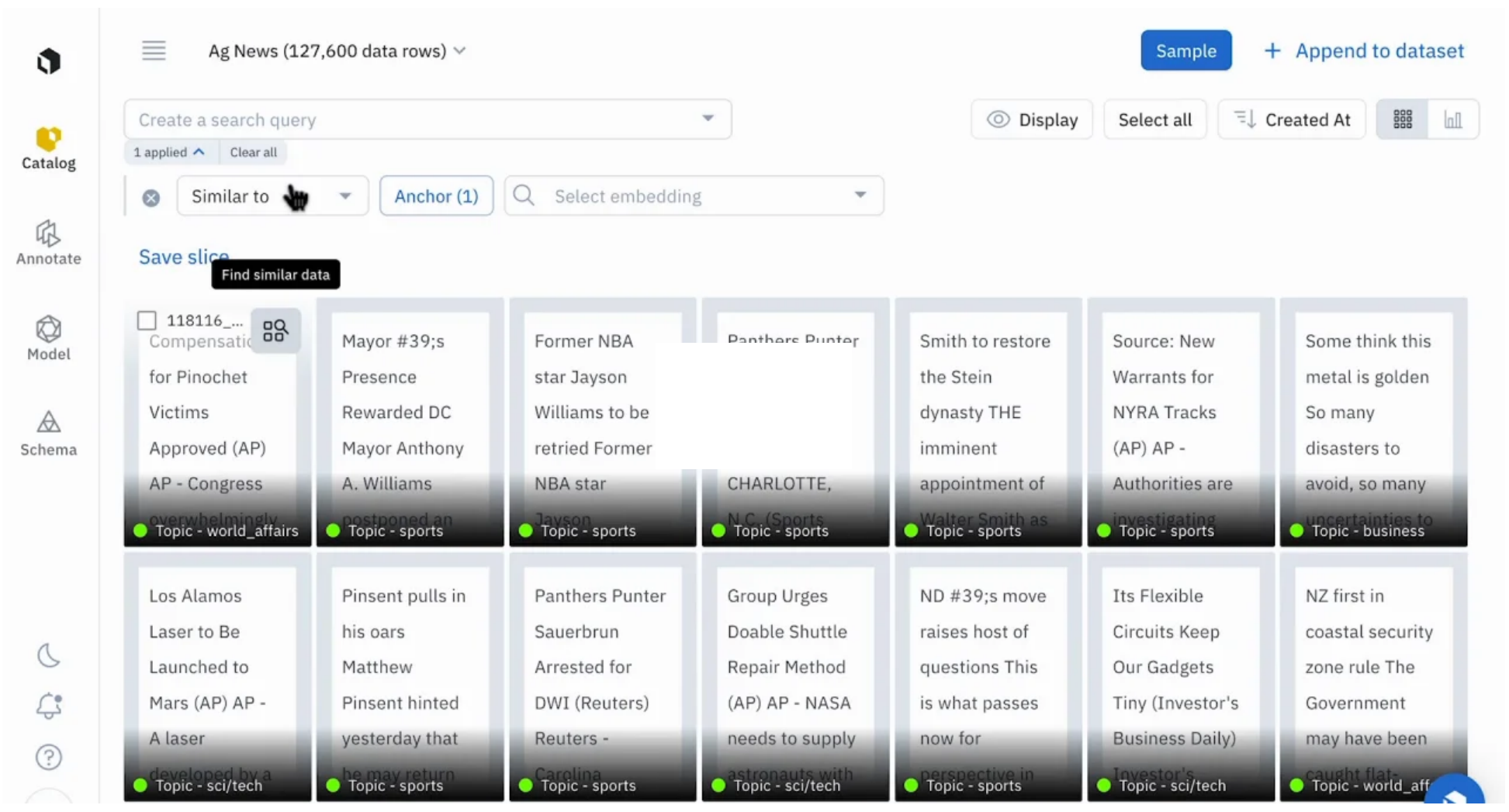
Recent advances in the machine learning field enable some neural networks (e.g., CLIP by OpenAI) to recognize a wide variety of visual concepts in images and associate them with keywords.

You can now surface images in Catalog by describing them in natural language. For example, type in *"a photo of birds in the sunset"* to surface images of birds in the sunset.

Similarity search

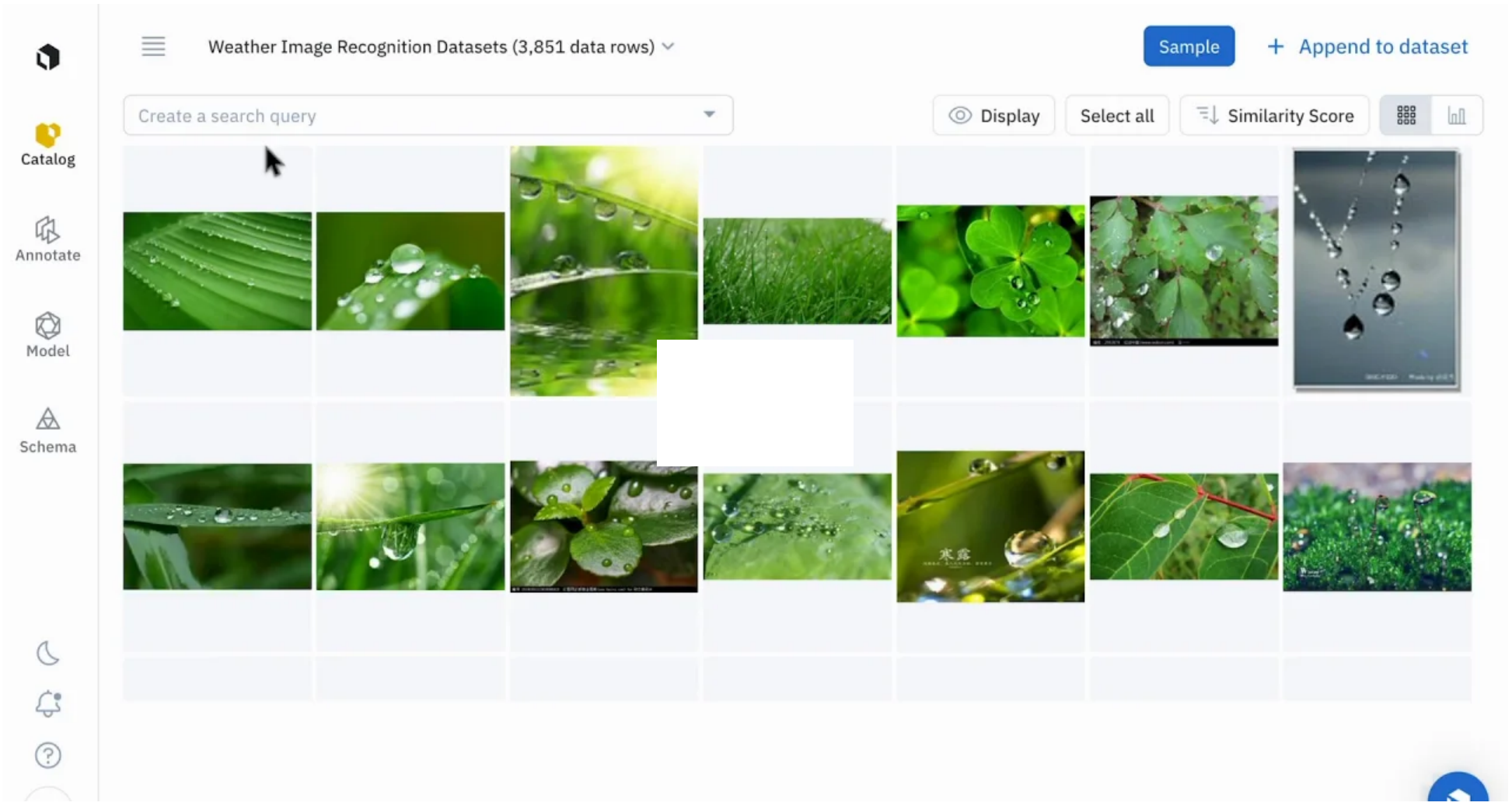
Labelbox's similarity search tool is designed to help you programmatically identify all of your data that is similar or dissimilar. You can utilize this tool to mine data and look for examples of rare assets or edge cases that will dramatically improve your model's performance. This similarity search engine also gives your team an advantage by helping you surface high-impact data rows in an ocean of data.

Similarity search on text data using embeddings:



0:19

Similarity search on image data using embeddings:



Learn more about ways to better explore and organize your data in Catalog

- **Make your videos queryable using foundation models**
- Utilizing Few-shot and Zero-shot learning with OpenAI embeddings

---

Sources referenced:

- https://en.wikipedia.org/wiki/Locality-sensitive_hashing
- https://en.wikipedia.org/wiki/K-d_tree
- https://github.com/spotify/annoy
- https://github.com/google-research/google-research/tree/master/scann
- https://towardsdatascience.com/tree-algorithms-explained-ball-tree-algorithm-vs-kd-tree-vs-brute-force-9746debcd940
- https://cloud.google.com/blog/topics/developers-practitioners/meet-ais-multitool-vector-embeddings
- https://towardsdatascience.com/illustrated-guide-to-siamese-network-3939da1b0c9d

Continue reading

Labelbox • December 15, 2023

How to leverage Google’s Gemini models in Labelbox Foundry for building AI

Learn how you can easily evaluate Gemini models, compare them to other powerful foundation models like Open AI’s GPT-4, and choose the best model for your use case with model foundry in Labelbox.

Labelbox • October 26, 2023

10X faster uploads: Labelbox’s data ingestion upgrades and how to use them

Unleash the full potential of your data in Labelbox with 10x faster performance on large-scale data ingestion uploads.

Labelbox • October 23, 2023

RLHF vs RLAIF: Choosing the right approach for fine-tuning your LLM

Learn how these two methods differ, the pros and cons of each, and how to choose the right method for your fine-tuning project.

Try Labelbox today

Get started for free or see how Labelbox can fit your specific needs by requesting a demo

Start for free