An introduction to the vector database.

@Milvus.io

Vector Database 101 - Scalar Quantization and Product Quantization

Introduction

Hey there - welcome back to Milvus tutorials. In the previous tutorial, went over a quick word embedding example to better understand the power of embeddings along with how they are stored and indexed in a vector database. This led to a brief overview of nearest neighbor search algorithms, a computing problem that involves finding the closest vector(s) to a query vector based on a selected distance metric.

In this tutorial, we'll build on top of that knowledge but diving deeper into quantization techniques - specifically scalar quantization (also called integer quantization) and product quantization. We'll implement our own scalar and product quantization algorithms in Python, but we'll also see how we can use *Facebook AI Similarity Search* (Faiss, pronounced like "face") to implement these indexing strategies more efficiently and with less effort. Towards the end, we'll also briefly discuss how Faiss is used as a subcomponent within the Milvus compute engine to power ANN search, keeping in mind the multitude of different layers atop the core indexing strategy that comprise a scalable vector database.

Let's dive in.

Scalar quantization

As mentioned in the previous tutorial, quantization is a technique for reducing the total size of the database by reducing the overall *precision* of the vectors. Note that this is very different from dimensionality reduction (PCA, LDA, etc), which attempts to reduce the *length* of the vectors:

```
>>> vector.size  # length of our original vector
128
>>> quantized_vector.size  # length of our quantized vector
128
>>> reduced_vector.size  # length of our dimensionality reduced vector
16
```

Dimensionality reduction methods such as PCA use linear algebra to project the input data into a lower dimensional space. Without getting too deep into the math here, just know that these methods generally aren't used as the primary indexing strategy because they tend to have limitations on the distribution of the data. PCA, for example, works best on data that can be separated into independent, Gaussian distributed components.

Quantization, on the other hand, makes no assumption about the distribution of the data - rather, it looks at each dimension (or group of dimensions) separately and attempts to “bin” each value into one of many discrete buckets. In particular, scalar quantization turns floating point values into low-dimensional integers:

```
>>> vector.dtype  # data type of our original vector
dtype('float64')
>>> quantized_vector.dtype
dtype('int8')
>>> reduced_vector.size
dtype('float64')
```

From the above example, we can see that scalar quantization has reduced the total size of our vector (and vector database) by a whopping 8x. Nice.

So how does scalar quantization work? Let’s first take a look at the forward-conversion process, i.e. turning floating point vectors into integer vectors. For each vector dimension, scalar quantization takes the maximum and minimum value of that particular dimension as seen across the entire database, and uniformly splits that dimension into bins across its entire range:

Scalar quantization for a single dimension.

Let’s try writing that in code. We’ll first generate a dataset of a thousand 128D floating point vectors sampled from a multivariate distribution. Since this is a toy example, I’ll be sampling from a Gaussian distribution; in practice, actual embeddings are rarely Gaussian distributed unless added as a constraint when training the model (such as in variational autoencoders):

```
>>> import numpy as np
>>> dataset = np.random.normal(size=(1000, 128))
```

This dataset serves as our dummy data for use in this scalar quantization implementation. Now, let’s determine the maximum and minimum values of each dimension of the vector and store it in a matrix called `ranges`:

```
>>> ranges = np.vstack((np.min(dataset, axis=1), np.max(dataset, axis
    =1)))
```

You’ll notice that the `mins` and `maxes` here are fairly uniform across all dimensions for the this toy example since the input data was sampled from zero-mean unit-variance Gaussians - don’t worry about that for now, as all the code here translates perfectly to real data as well. We now have the minimum and maximum value of each vector dimension in the entire dataset. With this, we can now determine the *start value* and *step size* for each dimension. The start value is simply the minimum value, and the step size is determined by the number of discrete bins in the integer type that we’ll be using. In this case, we’ll be using 8-bit unsigned integers `uint8_t` for a total of 256 bins:

```
>>> starts = ranges[0,:]
>>> steps =
```

Now let's put it all together in a

```
class ScalarQuantizer:

        def __init__(self, dataset):
            # determine quantization parameters
            pass

        def quantize(self, vector):
            # perform a quantization "forward pass"
            pass

>>> vectors = np.random.normal(size=(1000, 128))
>>> quantizer = ScalarQuantizer(vectors)
>>>
```

Scalar quantization

Product quantization

Product quantization is a much more powerful way to quantize vectors

Vector quantization techniques in Faiss

Now, let's try doing this at a larger scale with Facebook AI Similarity Search (Faiss). Faiss is a library for - you guessed it - scalable similarity search. Specifically, Faiss allows us to search across Specifically, Faiss allows us to rapidly search across a variety of

It assumes that the instances are represented as vectors and are identified by an integer, and that the vectors can be compared with L2 (Euclidean) distances or dot products. Vectors that are similar to a query vector are those that have the lowest L2 distance or the highest dot product with the query vector. It also supports cosine similarity, since this is a dot product on normalized vectors.

So, given a set of vectors, we can index them using Faiss — then using another vector (the query vector), we search for the most similar vectors within the index.

Now, Faiss not only allows us to build an index and search — but it also speeds up search times to ludicrous performance levels — something we will explore throughout this article.

Approximate nearest neighbor search

How Milvus uses Faiss

FAISS is one of the many components within Milvus used to

Wrapping up

In this tutorial, we took scalar quantization and product quantization, making our own implementation along with way. We also briefly took a look at how these can be implemented using Faiss before discussing how Faiss fits into Milvus

- A
- B
- C

Thanks for joining us for this tutorial! In the next couple of tutorials, we'll go over HNSW and ANNOY.