An introduction to vector similarity search.

@Milvus.io

Vector Database 101 - Introduction to Vector Similarity Search

Introduction

Hey there - welcome back to Milvus tutorials. In the previous tutorials, we took a look at unstructured data, vector databases, and Milvus - the world's most popular open-source vector database. We also briefly touched upon the idea of *embeddings*, high-dimensional vectors which serve as awesome semantic representations of unstructured data. One key note to remember - embeddings which are "close" to one another represent semantically similar pieces of data.

In this tutorial, we'll build on that knowledge by going over a word embedding example and seeing how semantically similar pieces of unstructured data are "near" one another while dissimlar pieces of unstructured data are "far" from one another. This will lead into a high-level over view of *nearest neighbor search*, a computing problem that involves finding the closest vector(s) to a query vector based on a unified *distance metric*. We'll go over some well-known methods for nearest neighbor search (including my favorite - ANNOY) in addition to commonly used *distance metrics*.

Let's dive in.

Comparing embeddings

Some prep work

Before beginning, we'll need to install the `gensim` library and load a `word2vec` model.

```
1  % pip install gensim --disable-pip-version-check
2  % wget https://s3.amazonaws.com/dl4j-distribution/GoogleNews-vectors-
     negative300.bin.gz
3  % gunzip GoogleNews-vectors-negative300.bin
```

```
1  Requirement already satisfied: gensim in /Users/fzliu/.pyenv/lib/
     python3.8/site-packages (4.1.2)
2  Requirement already satisfied: smart-open>=1.8.1 in /Users/fzliu/.pyenv
     /lib/python3.8/site-packages (from gensim) (5.2.1)
3  Requirement already satisfied: numpy>=1.17.0 in /Users/fzliu/.pyenv/lib
     /python3.8/site-packages (from gensim) (1.19.5)
4  Requirement already satisfied: scipy>=0.18.1 in /Users/fzliu/.pyenv/lib
     /python3.8/site-packages (from gensim) (1.7.3)
5  --2022-02-22 00:30:34--  https://s3.amazonaws.com/dl4j-distribution/
     GoogleNews-vectors-negative300.bin.gz
6  Resolving s3.amazonaws.com (s3.amazonaws.com)... 52.216.20.165
```

```
 7  Connecting to s3.amazonaws.com (s3.amazonaws.com)|52.216.20.165|:443...
       connected.
 8  HTTP request sent, awaiting response... 200 OK
 9  Length: 1647046227 (1.5G) [application/x-gzip]
10  Saving to: GoogleNews-vectors-negative300.bin.gz
11
12  GoogleNews-vectors- 100%[===================>]   1.53G  2.66MB/s    in
       11m 23s
13
14  2022-02-22 00:41:57 (2.30 MB/s) - GoogleNews-vectors-negative300.bin.gz
       saved [1647046227/1647046227]
15
16  gunzip: GoogleNews-vectors-negative300.bin: unknown suffix -- ignored
```

Now that we've done all the prep work required to generate word-to-vector embeddings, let's load the trained `word2vec` model.

```
1  >>> from gensim.models import KeyedVectors
2  >>> model = KeyedVectors.load_word2vec_format('GoogleNews-vectors-
       negative300.bin', binary=True)
```

Example 0: Marlon Brando

Let's take a look at how `word2vec` interprets the famous actor Marlon Brando.

```
1  >>> print(model.most_similar(positive=['Marlon_Brando']))
```

```
1  [('Brando', 0.757453978061676), ('Humphrey_Bogart', 0.6143958568572998)
       , ('actor_Marlon_Brando', 0.6016287207603455), ('Al_Pacino',
       0.5675410032272339), ('Elia_Kazan', 0.5594002604484558), ('
       Steve_McQueen', 0.5539456605911255), ('Marilyn_Monroe',
       0.5512186884880066), ('Jack_Nicholson', 0.5440199375152588), ('
       Shelley_Winters', 0.5432392954826355), ('Apocalypse_Now',
       0.5306933522224426)]
```

Marlon Brando worked with Al Pacino in The Godfather and Elia Kazan in A Streetcar Named Desire. He also starred in Apocalypse Now.

Example 1: If all of the kings had their queens on the throne

Vectors can be added and subtracted from each other to demo underlying semantic changes.

```
1  >>> print(model.most_similar(positive=['king', 'woman'], negative=['man
       '], topn=1))
```

```
1  [('queen', 0.7118193507194519)]
```

Who says engineers can't enjoy a bit of dance-pop now and then?

Example 2: Apple, the company, the fruit, ... or both?

The word "apple" can refer to both the company as well as the delicious red fruit. In this example, we can see that Word2Vec retains both meanings.

```
1  >>> print(model.most_similar(positive=['samsung', 'iphone'], negative=[
       'apple'], topn=1))
2  >>> print(model.most_similar(positive=['fruit'], topn=10)[9:])
```

```
1  [('droid_x', 0.6324754953384399)]
2  [('apple', 0.6410146951675415)]
```

"Droid" refers to Samsung's first 4G LTE smartphone ("Samsung" + "iPhone" - "Apple" = "Droid"), while "apple" is the 10th closest word to "fruit".

Vector search strategies

Now that we've seen the power of embeddings, let's briefly take a look at some of the ways we can speed up nearest neighbor search. This is not a comprehensive list; we'll just breifly go over some common methods in order to provide a high-level overview of how vector search is conducted at scale. Note that some of these methods are not exclusive to each other - it's possible, for example, to use quantization in conjunction with space partitioning.

Linear search

The simplest but most naïve nearest neighbor search algorithm is good old linear search: computing the distance from a query vector to all other vectors in the vector database. For obvious reasons, naïve search does not work when trying to scale our vector database to tens or hundreds of millions of vectors, but when the total number of elements in the database is small, this can actually be the most efficient way to perform vector search since a separate data structure for the index is not required, while inserts and deletes can be implemented fairly easily. Due to the lack of space complexity as well as constant space overhead associated with naïve search, this method can often outperform space partitioning even when querying across a moderate number of vectors.

Space partitioning

Space partitioning is not a single algorithm, but rather a family of algorithms that all use the same concept. K-dimensional trees (kd-trees) are perhaps the most well-known in this family, and work by continuously bisecting the search space (splitting the vectors into "left" and "right" buckets) in a manner similar to binary search trees. Inverted file index (IVF) can also be considered space partitioning, and works by assigning each vector to its nearest centroid - searches are then conducted by first determining the query vector's closest centroid and conducting the search around there, significantly reducing the total number of vectors that need to be searched. IVF is a popular indexing strategy that we'll take a closer look at in the next tutorial, so stay tuned.

Quantization

Quantization is a technique for reducing the total size of the database by reducing the precision of the vectors. Scalar quantization (SQ), for example, works by multiplying high-precision floating point vectors with a scalar value, then casting the elements of the resultant vector to their nearest integers. This not only reduces the effective size of the entire database (e.g. by a factor of eight for conversion from `float64_t` to `int8_t`), but also has the positive side-effect of speeding up vector-to-vector distance computations. Product quantization (PQ) is another quantization technique that works similar to dictionary compression. In PQ, all vectors are split into equally-sized subvectors, and each subvector is then replaced with a centroid. We'll go over both SQ and PQ in an upcoming tutorial.

Hierarchical Navigable Small Worlds

Hierarchical Navigable Small Worlds (HNSW) is a graph-based indexing and retrieval algorithm. This works differently from product quantization: instead of improving the searchability of the database by reducing its effective size, HNSW creates a multi-layer graph from the original data. Upper layers contain only "long connections" while lower layers contain only "short connections" between vectors in the database (see the next section for an overview of vector distance metrics). Individual graph connections are created a-la skip lists. With this architecture in place, searching becomes fairly straightforward – we greedily traverse the uppermost graph (the one with the longest inter-vector connections) for the vector closest to our query vector. We then do the same for the second layer, using the result of the first layer search as the starting point. This continues until we complete the search at the bottommost layer, the result of which becomes the nearest neighbor of the query vector.

Approximate Nearest Neighbors Oh Yeah

This is probably my favorite ANN algorithm simply due to its playful and unintunitive name. Approximate Nearest Neighbors Oh Yeah (ANNOY) is a tree-based algorithm popularized by Spotify (it's used in their music recommendation system). Despite the strange name, the underlying concept behind ANNOY is actually fairly simple – binary trees. ANNOY works by first randomly selecting two vectors in the database and bisecting the search space along the hyperplane separating those two vectors. This is done iteratively until there are fewer than some predefined parameter `NUM_MAX_ELEMS` per node. Since the resulting index is essentially a binary tree, this allows us to do our search on O(log n) complexity.

Commonly used similarity metrics

The very best vector databases are useless without similarity metrics – methods for computing the distance between two vectors. Numerous metrics exist, so we will discuss only the most commonly used subset here.

Floating point vector similarity metrics

The most common floating point vector similarity metrics are, in no particular order, *L1 distance*, *L2 distance*, and *cosine similarity*. The first two values are *distance metrics* (lower values imply more similarity while higher values imply lower similarity), while cosine similarity is a *similarity metric* (higher values imply more simlarity).

1.

$$d_{l1}(\mathbf{a}, \mathbf{b}) = \sum_{i=1}^{N} |\mathbf{a}_i - \mathbf{b}_i|$$

2.

$$d_{l2}(\mathbf{a}, \mathbf{b}) = \sqrt{\sum_{i=1}^{N} (\mathbf{a}_i - \mathbf{b}_i)^2}$$

3.

$$d_{cos}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|}$$

L1 distance is also commonly referred to as Manhattan distance, aptly named after the fact that getting from point A to point B in Manhattan requires moving along one of two fixed directions. The second equation, L2 distance, is simply the distance between two vectors in Euclidean space. The third and final equation is cosine distance, equivalent to the cosine of the angle between two vectors. Note the equation for cosine similarity works out to be the dot product between normalized versions of input vectors **a** and **b**.

With a bit of math, we can also show that L2 distance and cosine similarity are effectively equivalent when it comes to similarity ranking:

$$d_{l2}(\mathbf{a}, \mathbf{b}) = (\mathbf{a} - \mathbf{b})^T (\mathbf{a} - \mathbf{b})$$

$$= \mathbf{a}^T \mathbf{a} - 2\mathbf{a}^T \mathbf{b} + \mathbf{b}^T \mathbf{b}$$

$$= 2 - 2\mathbf{a}^T \mathbf{b}$$

$$= 2(1 - d_{cos}(\mathbf{a}, \mathbf{b}))$$

Essentially, if you have unit norm vectors

$$|\mathbf{a}| = |\mathbf{b}| = 1$$

, L2 distance and cosine similarity are functionally equivalent! Always remember to normalize your

embeddings.

Binary vector similarity metrics

Binary vectors, as their name suggest, do not have metrics based in arithmetics a-la floating point vectors. Similarity metrics for binary vectors instead rely on either set mathematics, bit manipulation, or a combination of both (it's okay, I also hate discrete math). Here are the formulas for two commonly used binary vector similarity metrics:

1.

$$d_J(\mathbf{a}, \mathbf{b}) = 1 - \frac{\mathbf{a} \cdot \mathbf{b}}{|a|^2 + |b|^2 - \mathbf{a} \cdot \mathbf{b}}$$

2.

$$d_J(\mathbf{a}, \mathbf{b}) = \sum_{i=1}^{N} \mathbf{a}_i \oplus \mathbf{b}_i$$

The first equation is called Tanimoto/Jaccard distance, and is essentially a measure of the amount of overlap between two binary vectors. The second equation is Hamming distance, and is a count of the number of vector elements in a and b which differ from each other.

You can most likely safely ignore these similarity metrics, since the majority of applications use cosine similarity over floating point embeddings.

Wrapping up

In this tutorial, we took a look at vector similarity search, along with some common vector search algorithms and distance metrics. Here are some key takeaways:

- Embedding vectors are powerful representations, both in terms of distance between the vectors and in terms of vector arithmetic. By applying a liberal quantity of vector algebra to embeddings, we can perform scalable semantic analysis using just basic mathematical operators.
- There are a wide variety of approximate nearest neighbor search algorithms and/or index types to choose from. The most commonly one used today is HNSW, but a different indexing algorithm may work better for your particular application, depending on the total number of embeddings you have in addition to the length of each individual vector.
- The two primary distance metrics used today are L2/Euclidean distance and cosine distance. These two metrics, when used on normalized embeddings, are functionally equivalent.

Thanks for joining us for this tutorial! Vector search is a core part of Milvus, aand it will continue to be In future tutorials, we'll be doing some deeper dives into the most commonly used ANNS algorithms - HNSW and ScaNN.

See you in the next tutorial.