Similar.Pictures

Find near duplicates on your computer

Home Demo Research Tools About

fedulov.vitali@gmail.com, 7 December 2021

Float N-dimensional vector hashing

The scope of the article is fast approximate vector comparison in N-dimensional spaces. A method to hash float vectors is presented. The method has been developed in the context of large scale image comparison.

Hash tables allow fast search. But not every value is suitable for hashing. For example how could you hash a person's height? There would be a near infinite number of hashes to find equal ones for comparison. Hashing in higher dimensions is even more challenging.

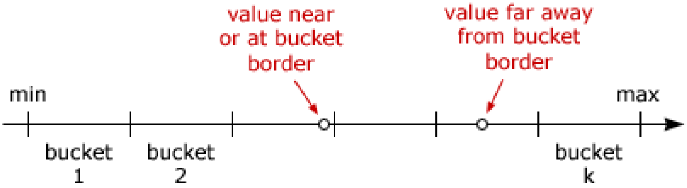
Here I demonstrate a method to hash numerical Float values and other values of high density, such as Int, by using binary bucket trees. In the method each component of a vector is projected into 1 or 2 buckets per dimension. Bucket numbers form hashes by appending them sequentially dimension by dimension.

In general, in each dimension, buckets can have variable widths. For example, if values come from interval (-Infinity, +Infinity), we could split this infinite space into finite K buckets with some kind of distribution reflecting the data (bell-curve or other). It might be good to analyze each dimension of your vectors to get the distribution curve, and split each axis so that each bucket contains approximately the same number of values.

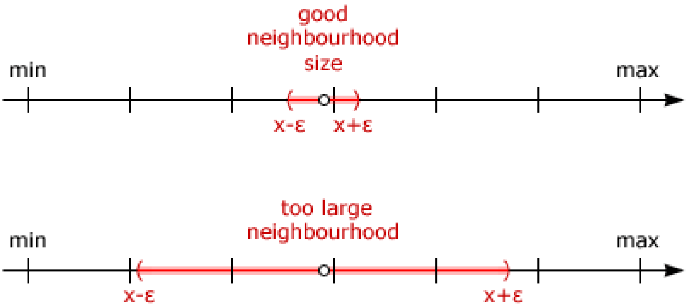
Value distribution is important, but in the method below I assume that buckets have equal widths and that we consider a fixed interval of float values [min, max] per dimension. For example, (0.0, 300.0) cm is a safe interval to measure humans, and 10 cm could be the bucket width.

To be able to hash/compare a float value, we need its bucket number. But it is not sufficient to check that a float value falls into a specific bucket. What if it falls exactly at a bucket split point? Or one value falls near the split on one side, and another similar value falls on the other side? Besides, those values typically contain measurement errors. Therefore we will introduce an uncertainty neighbourhood of a value: (x-ε, x+ε).

Let's start with 1-dimensional vector (figure below). The bucket number for the left point is uncertain:



If a float value falls near the center of a bucket, we would feel safe to allocate a single bucket for this value. But if the value finds itself near the bucket border, as in the picture above, the nearby bucket should also be included. By introducing the uncertainty interval of different sizes around the float value, we will get more than one bucket corresponding to such value:



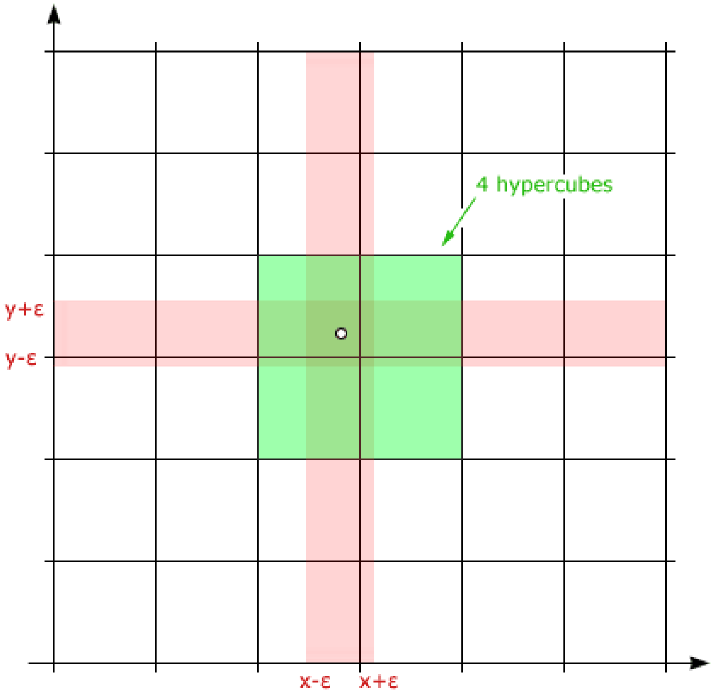
In the 1st case of the figure above 2 buckets are "activated" by the uncertainty interval. Therefore 2 bucket numbers will be necessary to represent one value.

In the 2nd case, the epsilon interval covers many buckets to represent one float value. This is impractical. An ideal bucket width must be larger than 2ε. Then a float value will fall into maximum 2 buckets. There is no "free lunch" to have all values in 1 bucket only - some will luckily do, but some will require 2 buckets. This is the price to be able to compare values, as explained above.

From now on we assume that bucket width is larger than 2ε.

As showed above, the uncertainty requires 2 buckets "reserved" in 1-dimensional space. If bucket width is much larger than ε, there will be few values with their uncertainty intervals overlaying 2 buckets, because probability of a value to be near a bucket border will be smaller. In such configuration most values will require only 1 bucket number to be correctly represented.

Now, let's consider higher dimensional space:



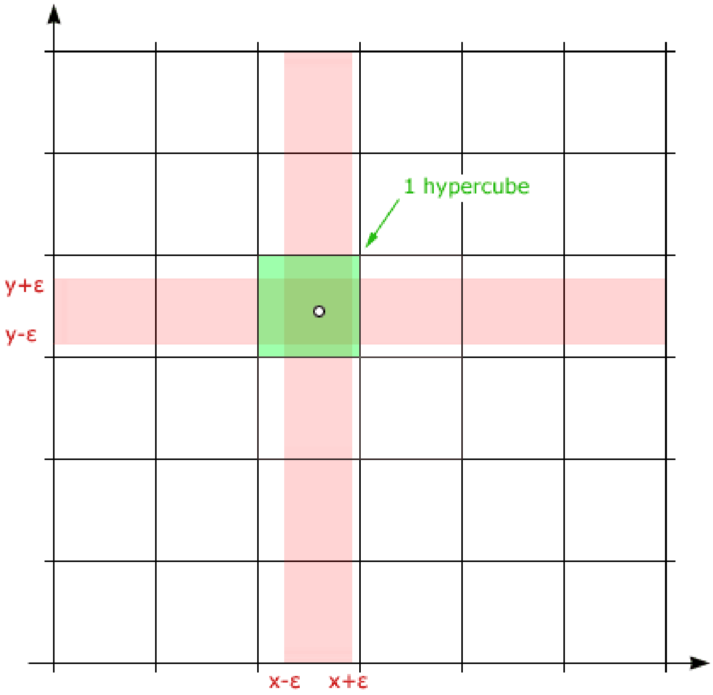
Buckets for 2 dimensions, value uncertainty intervals, and corresponding hypercubes

In N-space buckets are defined the same way as for one dimension, that is in relation to bucket borders. In 2 dimensions (figure above) buckets are vertical and horizontal stripes between black lines. The uncertainty regions are colored pink. In 3D buckets become infinite slices, and so on for higher dimensions.

Method essence: Bucket intersections form hypercubes, which correspond to the final hashes to be ultimately used to compare float vector values. The hashes are de-facto hypercube numbers.

Assuming that bucket width is larger than 2ε, for one dimension we will get 1-2 final hashes. For 2D we will get 1-4 hashes. For 3D we will get 1-8 hashes. And so on.

Below is shown the smallest number of buckets necessary in 2D case (one for each dimension), because each uncertainty interval falls inside one bucket.



The case of minimal number of buckets necessary in 2D

With each additional dimension the number of possible hypercubes will double. Therefore one N-dimensional vector will be represented by 1 to 2^N final hashes.

How to construct hashes

Let's assume that one character of a hash is a bucket number, and that we have 10 buckets per dimension numbered from 0 to 9.

In each dimension we always have 1 or 2 bucket numbers.

Let's consider a 2D case with some random bucket numbers. In the 1st dimension we have bucket number [2] or numbers [2, 3]. In the 2nd dimension [7] or [7, 8]. Now, to create full hashes, we append numbers of the 1st dimension to the numbers of the 2nd dimension, thus getting at minimum 1 hash: "27", and at maximum 4 hashes: "27", "28", "37", "38" (each to each number of the dimensions).

Such binary tree approach allows to continue with additional dimensions to construct larger hashes for N-dimensional float vectors. The final hash length is always N. For every additional dimension we combine hashes from previous dimensions with the current one. To continue from the previous example, if in the 3rd dimension there is only one bucket 5, the final hashes at maximum will be "275", "285", "375", "385".

Just to repeat: in the "most economic" case there will be only 1 hash in the final set, and in the worse case there will be 2^N hashes. Such hash set represents one N-vector for search and comparison.

How to search and compare

Every hash is de-facto a hypercube for nearest-neighbour search or comparison. Two vectors with same hash(es) are equal or near equal.

For hash lookup database, we do not need to store the whole set of hashes per N-vector. The only hash we need to store is the one corresponding to the central hypercube, which contains the value point.

But for a query we need all hashes from a set. Thus the query hash (set) is "fuzzy" because of uncertainty, but a recorded value hash is not.

It should be possible to inverse the approach and store the whole hash set of the vector, while only using the central hash for a lookup query. Anyway, no "free lunch" here: one-to-many or many-to-one principle must hold on search.

How the curse becomes a blessing

Since the hash set size may double with each additional dimension, it might seem impractical to search/compare for large N. Already for 10 dimensions we may get up to 2^{10} = 1024 hashes in the set! The curse of dimensionality?

To address the problem, as already mentioned for 1D case, we can choose larger bucket widths. Assuming very large bucket width and reasonably small ε, there will be only few cases of our value being near bucket borders, therefore only a few branches of the binary tree will be necessary. As a result we will typically get small hash sets with a few rare exceptions.

If bucket width is large, the similarity approximation is rough, but could be sufficient for initial grouping of similar vectors. After that Euclidean distance can be used to check for higher precision within the group. The Euclidean distance threshold can be set to ε.

To increase precision we could also add additional dimensions N+1, N+2 etc. As a result even the most basic dimension split into only 2 buckets may be enough. Then the curse of dimensionality becomes a blessing: number of hypercubes will grow fast and will discretize N-space quite granularly. For example having only 4 buckets per dimension in 10-dimensional space will give us 4^{10}> 1 million hashes. And for 8 buckets over 1 billion hashes.

Another way to fight the curse of dimensionality is to split N-dimensional vector to smaller vectors, and generate hash sets for each sub-vector. Then later intersect lookup results for those sub-sets.

The demonstrated bucket approach will work best when vector components are independent variables.

Algorithm code

Written in Golang float vector hash (Github).

Possible optimization

The demonstrated method uses uncertainty planes to identify additional hypercubes. A better method could be using an epsilon sphere to select the hypercubes. This should reduce the number of hashes in the query set.

This is my original research. Please treat it as such for references, provided you do not find an earlier similar research by others, which is possible. Sometimes solutions have to be rediscovered, because either papers are hard to find, or they are difficult to understand.