Simple Simhashing

Clustering in linear time.

JANUARY 21, 2010

AUTHORS

Moulton Ryan

ABSTRACT

Suppose you have a huge number of items that you would like to group together by a fuzzy notion of similarity. Suppose the only tool available to you is a key-value store. Suppose you only have the resources to consider each object once. Never fear, simhashing is here!

Well, there goes the neighborhood…

Most clustering algorithms are frustratingly non-local, and what is frustrating at small scale becomes intractable at large scale. Limiting your scope to a neighborhood of items usually requires heuristics that are clustering algorithms in their own right (Yo dawg, I put some clustering in your clustering.) Any algorithm that requires a notion of pairwise similarity at best requires fetching many items from your data store, and at worse requires n^2 time and space.

Wouldn't it be nice if you could look at an item once, and determine its cluster immediately without consulting any other data? Wouldn't it be nice if the clusters were stable between runs, so that the existence of one item would never change the cluster of another? Simhashing does exactly this. There are many approaches to simhashing, in this document I'm going to talk only about my favorite. It's simple to implement, mathematically elegant, works on anything with many binary features, and produces high quality results. It's also simple to analyze, so don't let the notation scare you off.

Comparing Two Sets

Suppose you have two sets, A and B, and you would like to know how similar they are. First you might ask, how big is their intersection?

$$|A \cap B|$$

That's nice, but isn't comparable across different sizes of sets, so let's normalize it by the union of the two sizes.

$$\frac{|A \cap B|}{|A \cup B|}$$

This is called the Jaccard Index, and is a common measure of set similarity. It has the nice property of being 0 when the sets are disjoint, and 1 when they are identical.

Hashing and Sorting

Suppose you have a uniform pseudo-random hash function H from elements in your set to the range $[0,1]$.

For simplicity, assume that the output of H is unique for each input. I'll use $H(A)$ to denote the set of hashes produced by applying H to each element of A, i.e. $\{H(a_1), H(a_2), H(a_3), \cdots, H(a_n)\}$.

Consider $\min(H(A))$. When you insert and delete elements from A, how often does $\min(H(A))$ change?

If you delete x from A then $\min(H(A))$ will only change if $H(x) = \min(H(A))$. Since any element has an equal chance of having the minimum hash value, the probability of this is $\frac{1}{|A|}$.

If you insert x into A then $\min(H(A))$ will only change if $H(x) < \min(H(A))$. Again, since any element has an equal chance of having the minimum hash value, the probability of this is $\frac{1}{|A|+1}$.

For our purposes, this means that $\min(H(A))$ is useful as a stable description of A.

Probability of a Match

What is the probability that $\min(H(A)) = \min(H(B))$?

If an element produces the minimum hash in both sets on their own, it also produces the minimum hash in their union.

$\min(H(A)) = \min(H(B))$ if and only if $\min(H(A \cup B)) = \min(H(A)) = \min(H(B))$.

Let x be the member of $A \cup B$ that produces the minimum hash value. The probability that A and B share the minimum hash is equivalent to the probability that x is in both A and B. Since any element of $A \cup B$ has an equal chance of having the minimum hash value, this becomes

$$\frac{|A \cap B|}{|A \cup B|}$$

Look familiar? Presto, we now have a simhash.

Tuning for Precision

This may be too generous for your purposes, but it is easy to make it more restrictive. One approach is to repeat the whole process with n independent hash functions, and concatenate the results. This makes the probability of a match

$$\left(\frac{|A \cap B|}{|A \cup B|}\right)^n$$

I prefer an alternate approach. Use only one hash function, but instead of selecting only the minimum value as the simhash, select the least n values. The probability of a match then becomes

$$\frac{\binom{|A \cap B|}{n}}{\binom{|A \cup B|}{n}}$$

and if $n \ll |A \cap B|$,

$$\frac{\binom{|A \cap B|}{n}}{\binom{|A \cup B|}{n}} \approx \left(\frac{|A \cap B|}{|A \cup B|}\right)^n$$

The advantage of this over independent hash functions is that it sets a minimum on the number of members that the two sets must share in order to match. This mitigates the effect of extremely common set members on your clusters. With several independent hash functions, a very common set member that produces low values in a small number of hash functions can cause a huge blowup of the resulting clusters. Selecting n from a single hash function ensures that it can only effect one term. It is for this reason that many simhash implementations unrelated to this one take into account the global frequency of each feature, but this complicates their implementation.

Turning Anything Into a Set

This algorithm works on a set, but the things we'd like to cluster usually aren't sets. Mapping from one to the other is straightforward if each item has many binary features, but can require some experimentation to get good results. If your items are text documents, you can produce a set using a sliding window of n-grams. I've found 3-grams to work well on lyrics, but YMMV. Since there's no order to the members of the set, it's important to make them long enough to preserve some of the local structure of the thing you'd like to cluster.

Pseudocode

```
int SimHash(Item item, int restrictiveness)
    Set set = SplitItemToSet(item)
    PriorityQueue queue
    for x in set
        queue.Insert(Hash(x))
    simhash = 0
    for x in [0 : restrictiveness]
        simhash ^= queue.PopMin()
    return simhash
```

Further Reading

This specific technique is often referred to as "Min Hashing" in the literature, so that's a good query to start looking for specific applications to your problem. It is a member of a general class of techniques called "Locality Sensitive Hashing," often abbreviated as LSH. Google has patented an application of another simhashing technique that is generally unrelated to this one. The paper that introduces it is also available, as is Google's paper on their implementation.

REFERENCES

1. Syntactic clustering of the Web
Reference Link

2. Similarity Estimation Techniques from Rounding Algorithms
Reference Link

Recent Articles Comments

A Short, Simple Introduction to Information Theory

How to Backpack

Arranging Music for A Cappella

Simple Simhashing

Great First Backpacking Trips

Meta

- Register
- Log in
- Entries RSS
- Comments RSS
- WordPress.com