Hashing for Data with some Distance Structure

(Near Neighbor Search and Locality Sensitive Hashing)

Often data has structure and hashing tends to remove it—
we want that changing a single letter in a word causes it
to hash to a very different place — look independent!

But at other times we may want to preserve some structure
- we're aiming for compression without losing the notion of
  similarity in the original space

Example: we're given strings and want to find pairs of "similar strings"
  e.g. in document de-duplication.
  or given a corpus of strings, and a query q,
  find strings in S that are "close to" q.
  e.g. in web searching.

Note: exact problem is easier, using just standard hashing.
- hash all strings in the corpus into a table
  when query comes, look at strings that hash to location h(q)
  and explicitly compare to them.

We're interested in approximate similarity.
Let's look at an example.

Suppose data are strings (maybe files of text)
need a notion of similarity (metric space)

(a) could imagine the underlying points belong to \( {\text{A-Za-z0-9}} \)
and distance is Edit distance.

+ Natural
  - difficult to manipulate, do searches.

(b) Strings are bit vectors (say in ASCII representation).
  distance is Euclidean or Hamming distance

+ Easier to manipulate / understand
  - no more intuition for what is similar or different.
  - different length strings make it awkward

(c) Convert document into a "bag of words".

  - Say \( 10^5 \) words in English; keep count of how many times
    each word appears in document / string.

    + each document \( \in \mathbb{Z}^{10^5} \)
      - lost order information / proximity information.
    - taking two copies of same doc gives very different point.
    - stop words can overwhelm all else.

  Normalize, say by total # terms in doc (TF/IDF)
  
  \[ \text{term frequency} / \text{inverse doc. freq.} \]
  
  word is in the corpus.

  \[ \text{Say } \left( \frac{\text{freq of } w}{\text{total # words in doc}} \right) \times \log \left( \frac{\# \text{docs in corpus}}{\# \text{docs into word } w} \right) \]
Also to keep some local structure:

**Shingling** — take k consecutive words in doc, call that a shingle
(e.g. 3-shingle)

Now do some things for these shingles. ← also do for images
Maybe drop stop words.
+ hundreds of useful cool tricks.

But eventually have say
← a bag of words
(a) a set of *items*, want to see if they are similar, or
(b) a *vectors* ← bag put in some geometric representation.

What's the notion of similarity? or distance?

Let's look at the vector case, say.

- Each item is a vector \( V_i \)
- Distance is Hamming distance: \( \text{dist}(V_i, V_j) = \# \text{coordinates where they differ} \)
  
  or Euclidean distance: \( \| V_i - V_j \|_2 = \sqrt{\sum_{k=1}^{d} (V_{ik} - V_{jk})^2} \)
  
  or \( L_1 \) distance
  (Manhattan taxicab distance)
  \( \| V_i - V_j \|_1 = \sum_{k=1}^{d} |V_{ik} - V_{jk}| \)
  
  or millions of others.
Let's try to solve the (approximate) near neighbors problem.

Given a metric space (distances, say \( \mathbb{R}^d \)) and a corpus \( S \subseteq \mathbb{R}^d \) of points, create a data structure such that it answers queries of the form "return a near-neighbor of point \( q \)."

I.e., return a point \( a \in S \) such that \( d(q, a) \leq c \times \min_{a^* \in S} d(q, a^*) \)

\( 1 \) approx factor. say \( z \) \( \leq \) today's lecture

\( 2 \) nearest neighbor distance

\( \text{Challenge: } S \text{ is large. Say } n = |S| = 10^{10} \text{ points} \)

\( \text{Strawman solution: } \)

no data structure. Brute force search each time.

So \( O(nd) \) time per query

\( \# \text{ points} \) \( \approx O(d) \) to compute distance, say \( 10^6 \) ms

\( \text{Should do better. } \)

Indeed, will use locality sensitive hashing.

- Similar items fall "close" to each other.

- Dissimilar items do not.
Want: data structure with small storage space, small preprocessing time and small query time

Solution #2 (Strassen ++)
Use k-d tree (or variant)
Partition bounding box for all points in S
into half along each coordinate
and so on, until each smallest box contains a single point.
For each box keep a representative point
Given query, go down the hierarchy, keeping closest representative
at each level look at all neighboring boxes as well.

\[ \begin{array}{c|c}
\checkmark & \checkmark \\
\checkmark & \checkmark \\
\end{array} \rightarrow \begin{array}{c}
\checkmark \\
\checkmark \\
\checkmark \\
\checkmark \\
\end{array} \]

So takes time \(2^d \cdot \log \left( \frac{d_{\max}}{d_{\min}} \right)\) for queries
\# \# neighboring boxes, \# levels of recursion

Curse of dimensionality!
- high dimensional point sets take \(\exp(\text{dimension})\) query time
Want to do better than that.

Note: have gone from \(O(nd)\) query time, no preprocessing

\[ O(2^d \cdot \log(\frac{d_{\max}}{d_{\min}})) \quad O(n \cdot \log(\frac{d_{\max}}{d_{\min}})) \]\space 2 preprocessing.
Curse of dimensionality:

if \( n \) items in \( d \)-dimensional space, searching for near neighbors in this space typically takes \( 2^{o(d)} \) time

Exponential dependence. ☹ if we do it naively.

(E.g., if we do it via quad-trees).

So let's be smarter.

Idea #1: Doubling Search.

Given \( S \) and parameter \( r \), solve the problem

Ball search

- given query \( q \), if \( \exists a \in S \) at distance \( (q,a) \leq r \)
  - return some \( a' \) at distance \( \leq (1+\varepsilon)r \).

if no \( a \in S \) at distance \( \leq r \) from query \( q \),
  - return whatever you want.

Now can try all values of \( r \), get an answer \( a_r \), format value

- powers of \( (1+\varepsilon) \)
  - \( r_r = (1+\varepsilon)^r \)

  - check the distance \( (q,a_r) \)

  - return the best of those.

If actual closest point is \( a* \in S \) at distance \( e \in [(1+\varepsilon)^2, (1+\varepsilon)^{2+}] \)

then will get a good answer for \( r = (1+\varepsilon)^{2+} \)

that has distance from \( q \), being \( \leq (1+\varepsilon)^{2+} \)

\[ \leq (1+\varepsilon)^2 \cdot d(q_*a) \]

\[ \leq (1+2\varepsilon) \cdot \text{dist}(q,a) \]
OK: how to solve the doubling search version?
Given S, radius r, answer (approx) queries:
if $\exists$ a point at distance $\leq r$ from q
return a point at dist $\leq 2r$ (say)

Here's idea:

Use hash function $h: \text{metric space} \rightarrow \text{buckets}$
such that

\[
\begin{align*}
\text{if dist}(x, y) \leq r & \Rightarrow P[h(x) = h(y)] \geq P_{\text{close}} \\
\text{dist}(x, y) > 2r & \Rightarrow P[h(x) = h(y)] \leq P_{\text{far}}
\end{align*}
\]

Ideally: $P_{\text{close}} \gg P_{\text{far}}$

Use these to hash into buckets.

\[
\mathbb{E}[\text{# junk points in } h(q) \text{ bucket}] \leq \sum_{x : \text{dist}(q, x) \geq 2r} P_{\text{far}} \cdot n.
\]

But then maybe $P_{\text{close}}$ is small as well.

So repeat $\approx \frac{1}{P_{\text{close}}}$ times, i.e., use $\frac{1}{P_{\text{close}}}$ independent many hash functions

to increase probability we see a close point.
Let's do a concrete example:

Want to distinguish points in $\mathbb{Z}_d$ with Hamming distance.

that are at distance $\leq r$ or at distance $\geq 2r$.

Try #0: $h(x) = \text{pick a random coordinate } i$ and output $x_i$.

if $\text{dist}(x, y) \leq r$ \quad $\Rightarrow$ \quad $\mathbb{P}_r[h(x) = h(y)] \geq 1 - \frac{r}{d} = P_{\text{close}}$

$\text{dist}(x, y) > 2r$ \quad $\Rightarrow$ \quad $\mathbb{P}_r[h(x) = h(y)] \leq 1 - \frac{2r}{d} = P_{\text{far}}$

Very small gap between $P_{\text{close}}$ & $P_{\text{far}}$.

Try #1: Parallel Repetition

$h(x) = \text{pick } l \text{ random coordinates with repetition}$

$\text{output those } l \text{ coordinates values}$

if $\text{dist}(x, y) \leq r$ \quad $\Rightarrow$ \quad $\mathbb{P}_r[h(x) = h(y)] \geq (1 - \frac{r}{d})^l = P_{\text{close}}$

$> 2r$ \quad $\Rightarrow$ \quad $\leq (1 - 2\frac{r}{d})^l = P_{\text{far}}$

As always: if $r << d$ (say) then $1 - \frac{r}{d} \approx e^{-\frac{r}{d}}$ etc.

$\Rightarrow P_{\text{close}} \approx e^{-\frac{r}{d} \cdot l}$

$P_{\text{far}} \leq e^{-2\frac{r}{d} \cdot l} = \frac{1}{n}$ (say)

$\Rightarrow 2r \cdot l = \log_2 n \Rightarrow l = \frac{d \cdot \log_2 n}{2r}$.
\[ P_{\text{close}} = e^{-d x^2} = e^{-\frac{1}{2} \log_2 n} = \frac{1}{\sqrt{n}}. \]

\[ \Rightarrow \text{if } \alpha \text{ and } \beta \text{ are indeed at distance } \leq r \]
\[ \text{then } P[h(x) = h(\beta)] \geq \frac{1}{\sqrt{n}} \]
\[ \text{small probability, but much better than } P_{\text{far}}. \]

So final ingredient: Serial Repetition

Use \( L \) independent hash functions

(i.e. keep \( L \) copies of data structures, probe in each and return the best)

If we set \( L = \sqrt{n} \log n \)

\[ \Rightarrow \exists P_{h[i]} \text{ s.t. } h[\beta] = h(x) \geq 1 - \left(1 - \frac{1}{\sqrt{n}}\right)^L \]
\[ \approx 1 - (e^{-\frac{L}{\sqrt{n}}}) \]
\[ = 1 - e^{-\log n} = 1 - \frac{1}{n}. \]

So in summary:

- used a weak hash function \( h \) with \( P_{\text{close}}, P_{\text{far}} \) not very different
- then reduced \( P_{\text{far}} \) by parallel repetition
- and handled the "small \( P_{\text{close}} \) value" problem
  by serial repetition.
Note: if $S = \Pr[b(x) = h(q)]$ for a single hash function.

Then $\Pr[h(x) = h^e(q)]$ for the function after parallel rep

$$= S^t.$$  

$$\Rightarrow \Pr[h^e(x) \neq h^e(q)] = (1-S^t).$$  

$$\Rightarrow \Pr[x \text{ is not seen in all } t \text{ copies of the data structure}] = (1-S^t)^t$$  

$$\Rightarrow \Pr[x \text{ is one of the candidates for query } q] = 1 - (1-S^t)^t$$

\[ \Pr(\text{becoming candidate}) \]

$$1 - (1-S^t)^t \leq \frac{1}{2} \text{ at } (1-S^t)^t \leq \frac{1}{2} \text{ or } S \approx (\frac{1}{t})^{\frac{1}{t}}$$

by choosing $t,e$ carefully

- can get space usage $n^{2-e}$ and query time $n^{1-e}$ in worst case
- in practice does quite a bit better.
Observe: steps II (parallel repetition) and III (serial repetition) are completely generic.

So: if we can get a small gap between collision probability in the close & far cases \( \Rightarrow \) can amplify the success.

Saw Step I for bit-strings and Hamming distance.

Can do this for other metric spaces:

- **Unit vectors and cosine distances**

  Elements are unit vectors.

  \[ d(x, y) = \cos \theta \quad \theta \text{ is angle between } x, y \]

  hash function: pick a random hyperplane.

  defines a halfspace. \( x \in \text{Halfspace} \Rightarrow h(x) = 1 \)

  \( \text{else } h(x) = 0. \)

  \[ \Pr[h(x) \neq h(y)] = \frac{\theta}{\pi} \]

- **Sets and Jaccard distance**

  \[ d(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|} \]

  hash function: pick a random permutation on elements.

  \( h(S) = \text{first element (in this permutation) in } S. \)

  \[ \Pr[h(A) = h(B)] = \frac{|A \cap B|}{|A \cup B|} \quad \Rightarrow \quad \Pr[h(A) \neq h(B)] = d(A, B). \]
Takeaways

- Can easily extend the basic ideas behind hashing to structured data.

\[ d(x, y) \leq r \quad \text{similar} \]
\[ d(x, y) > cr \quad \text{dissimilar} \]

Grey zone in the middle (don't care either way)

- Then can ask for

P [similar items collide] \geq \text{large}

P [dissimilar items collide] \leq \text{small}

And then amplify correctness by repetition.

- Can do this for many metric spaces
  - Hamming Cube (intuition)
  - Euclidean distances (embed into Ham Cube)
  - Cosine distances
  - Jaccard distances

- Popular and commonly used heuristic

(good in theory, even better in practice)

Next time: Reduce the ambient dimension of your data.