lecture 8 similarity search

wbg231

January 2023

1 Introduction

- what is a hash function? a function that maps some input to a key

finding items in a large collection

- search and recommendations rely on similarity calculations
- users provide a query could be a search string, example document etc
- and the system returns a list of matching documents for the database

example

- search engine: take a test string output related web documents
- recommender systems take in some representation of a user output item recommendations
- reverse image search put in a photo get out similar photos or where it came from

basic approach

- given a query q for each document in collection d compute $sim(q, d)$ abd return the top k documents
- this is linear in time can we do it more efficiently?

does this scale

- no it grows linear with the size of the connection and how we compute dimensionality make get more complex as the dimension of the representation grows
- so can we do better than a brute for search?

Approximate search

- so if we have n total documents we want to use some fast method to find $n << N$ candidate nearest neighbor pairs
- then we can use a true similarity match on the candidate set to discard any false positives
- this will require a data structure with a sub linear search time

min hash

similarity for sets

- items are represented as sets could be words in a document, movie a user has watched etc
- jaccard similarity is computed as $J(A, B) = \frac{A \cap B}{A \cup B}$
- and the jaccard distance is $D(A, B) = 1 - J(A, B)$

min hash

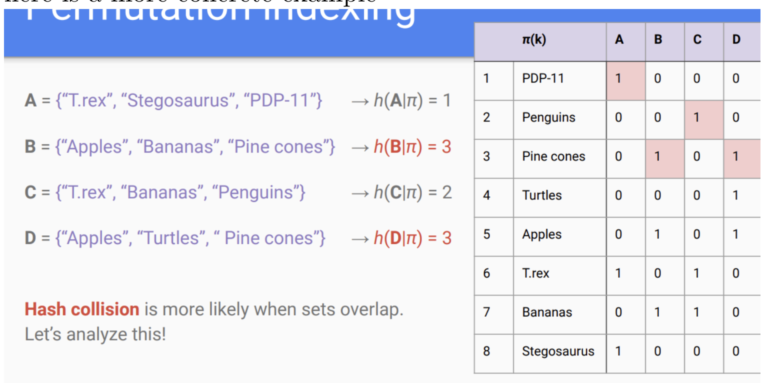
- fix a random ordering of the elements (a permutation) call if π
- imagine a table of set membership that is one hot encoded
- for each set S its hash is given as

$$h(s|\pi) = min(k|\pi(k) \in s)$$

- so that is the index of the first permuted item belonging to S **slido**: what is collision? it is when two values that are different has to the same key

permutation indexing

- here is a more concrete example



- hash collision is more likely to happen when sets overlap

jaccard similarity and hash collision

- for two set S_1 and S_2 there are three types of rows
  1. type 1: $\pi(k) \in S_1 \cap S_2$
  2. type 2: $\pi(k) \in S_1 \delta S_2$
  3. type 3: $\pi(k) \notin S_1 \cup S_2$
- note that a collision $\iff$ a type 1 row before all type 2 rows
- $P(\text{collision}) = \frac{\text{number of type 1 rows}}{\text{number of type 1 + the number of type 2 rows}} = \frac{S_1 \cap S_2}{S_1 \cup s_2} = J(S_1, S_2)$

monte carlo Approximate

- we want to get a good approximation of the probability of collision over potentially large sets, so we can just do monte carlo approximations and generate many random permutations and count there outcomes

searching with min hash

- a user provides a q
- initialize an empty dictionary $candidates \rightarrow \{\}$
- for each item π_i in the permutation π
- compute the hash $h(q|\pi_i)$

- $candidate+ = candidate + \{S : h(q_i|\pi_i) = h(S|\pi_i)\}$(that is documents that collide with the query)
- then we return the candidates ordered by $\#ofcollisions$ which is there Approximate similarity score (could also just take the full jaccard score of those candidate points)
- note that we do not need to compare the full collection to the query only those points that collide with it.
- bag an unordered group of objects with repeated elements

extending this to bags

- Ruzicka similarity is jaccard distance extended to bags
- idea reduce bags to sets by uniquely identifying each repetition

$$\begin{aligned}
&\{dog\} &&\rightarrow \{dog_1\}\\
&\{dog, dog\} &&\rightarrow \{dog_1, dog_2\}\\
&\{dog, dog, dog\} &&\rightarrow \{dog_1, dog_2, dog_3\}
\end{aligned}$$

- then we can calculate the Ruzicka similarity as

$$R(A, B) = \frac{\sum_i min(a[i], b[i])}{\sum_k (A[j], B[j])}$$

- this is not a perfect way to do it, but this broadly approximates jaccard similarity for bags

improving on word counts

- word $n - grams$ get permutations of n words in a row
- character shingles get n characters in a row

efficient approximation

- taking all possible permutations permutations can be expensive and would not scale
- instead we can replace permutation π_i with hash H_i
- a permutation is a perfect hash ie a reordering where distinct elements can not collide

- we can Approximate this with an imperfect has where distinct ellements may collide and as long as these collisions are unlikely this wil still work
- suppose we are trying to populate signature matrix initialized like this

	A	B	C	D
H_1	∞	∞	∞	∞
H_2	∞	∞	∞	∞

- and we have this table of hashes and signature matrix

x	$H_1(x)$	$H_2(x)$	A	B	C	D
PDP-11	0	0	1			
Penguins	1	2			1	
Pine cones	2	4		1		1
Turtles	3	0				1
Apples	0	1		1		1
T.rex	1	3	1		1	
Bananas	2	5		1	1	
Stegosaurus	3	1	1			

- the signature array is initialize to infinity for each entry
- in the first row both H_1, H_2 have A as once so update the A value for both hashes to be 0
- in row 2, the c column is where we look since it has the first 1 so the h_1 value of c gets set to 1, and the H_2 value of c gets set to 2
- in the third row we look at columns b and d both of them get updated to there coresponding h value since there orginal vlaue is infinity
- ir row 4 d is 1, h_2 is 0 so that vlaue updates to 0 H_1 is 3 which is greater than it current value so it does not update
- in teh next one b and d are looked at both values of updated and only the h1 value of d updates

	A	B	C	D
H_1	0	0	1	0
H_2	0	1	2	0

- and so on

when min hash fails

- permutation min hash, note that collisions are more likely when a small set of items are shared across many documents so stop words like "the" "and" "or" can be issues
- hashing approximations doe not fix this, collisons are possible and when we have a lot of collisions there is a large candidate set and slow retrieval
- what is recall? that is your ability to detect a true positive $r = \frac{TP}{TP+FN}$
- so our new question of interest becomes how can we reduce the size of the candidate set?

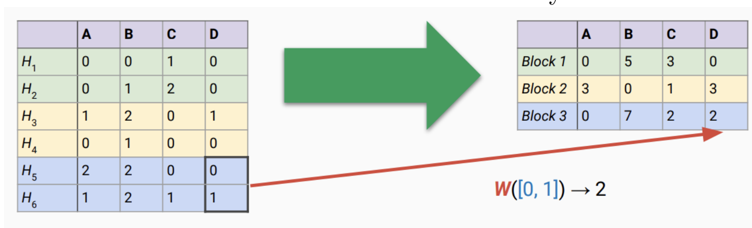
locality sensitive hashing

- traditional hashing scatters data as if random
- local sensitive hashing has a high probaility of collisions on input that are near each other
- LHS is a really wide topic and stuff

LHS + min hash

- care signature matrix into b blocks of R rows
- hash each sub column with a standard non local hashing function w. Pick W such that collisions are rare

- let the candidate set = items that collide in any row



	A	B	C	D
H_1	0	0	1	0
H_2	0	1	2	0
H_3	1	2	0	1
H_4	0	1	0	0
H_5	2	2	0	0
H_6	1	2	1	1

	A	B	C	D
Block 1	0	5	3	0
Block 2	3	0	1	3
Block 3	0	7	2	2

- what is the likelyhood that we had one block where all rows match
- if the likelyhood of a single row matching is j
- the liklylood hor all rows in a block cldiing would be j^r (so a lot less probible for collisions to happen)
- so collisions are more likely for high jaccard similarity rows and less likely for less
- if LHS and min hash has low recall (ie not getting many true positives) what could you do, change the hash function to have a higher chance of collisions

lhs for cosine similarity

- what if wae want to compare vectors $u, v \in \mathbb{R}^d$ by cosine similarity
- if we picked a vector at random and uniform from the unit sphere and hashed vectors as postive or negative if there dot product was postive or ngative what is the likelyhood of collisions
- is the likelyhood that it is more than 90 degrees away from one and less than 90 degrees away from the other
- that is not exactly $cos(\theta)$ but it is monotonically decreasing in $|\theta| \rightarrow$ same rank order as cosine similarity thus can be used to estimate cosine similarity

multiple projections

- then much like we did with multiple hashes in jaccard space, we can find the probability of collisions with multiple projections onto random vectors on the unit sphere

multi probe LSH

- random projections can isolate neighbors from each other. LHS uses multiple projections to minimize the chance of neighbors getting isolated but ti might take a lot of projections
- multi probe LSH explores neighboring in hash buckets to try to prevent this. bassically it just puts a query in the bucket if it is within a certain distance
- ends up with better recall and fewer hashes

spatial trees

recursive partitioning

- spatial trees recursively partition data into subsets
- we pick a direction w
- spit teh data set at the median of $\{w^t x_i\}$ ie split the data set in half based on the magnitude of each data points dot product with w
- recurse on teh left and right subsets
- stop when we are sufficiently small
- each split cuts the data in half so this is $O(log(n))$ splits to get small candidate Sets

KD trees

- the splitting section cycles through basis dimension
- this works in low dimensions but is bad for high dimension data
- can do a PCA type thing and split in the direction of max variance and that will likely work better
- split trees can also isolate data near descion boundarys
- carefull query can now land in mutple leaves