Lecture 12: Implementing MinHash

Sam McCauley

April 27, 2020

Williams College

Where Are We Now?

MinHash:

- Hash all items using k concatenated indices

Where Are We Now?

MinHash:

- Hash all items using k concatenated indices
- For any two items with the same hash value, calculate their similarity

Where Are We Now?

MinHash:

- Hash all items using k concatenated indices
- For any two items with the same hash value, calculate their similarity
- If two items have similarity over threshold, return those items.

Where Are We Now?

MinHash:

- Hash all items using k concatenated indices
- For any two items with the same hash value, calculate their similarity
- If two items have similarity over threshold, return those items.
- If two such items never found, repeat with new permutations

Where Are We Now?

MinHash:

- Hash all items using k concatenated indices
- For any two items with the same hash value, calculate their similarity
- If two items have similarity over threshold, return those items.
- If two such items never found, repeat with new permutations

How can we do this?

Assignment Parameters

- 128 bit integers (stored as two unsigned 64 bit ints "Pair")

Assignment Parameters

- 128 bit integers (stored as two unsigned 64 bit ints "Pair")
- Universe: $\{0, \ldots, 127\}$. (You can pretend that these are images, each of which is labelled with a subset of 128 possible tags.)

Assignment Parameters

- 128 bit integers (stored as two unsigned 64 bit ints "Pair")
- Universe: $\{0, \ldots, 127\}$. (You can pretend that these are images, each of which is labelled with a subset of 128 possible tags.)
- Each bit is a 0 or 1 at random

Assignment Parameters

- 128 bit integers (stored as two unsigned 64 bit ints "Pair")
- Universe: $\{0, \ldots, 127\}$. (You can pretend that these are images, each of which is labelled with a subset of 128 possible tags.)
- Each bit is a 0 or 1 at random
- (Not realistic case, but hard case!)

What About Hashing?

- MinHash: go through each index in the permutation

What About Hashing?

- MinHash: go through each index in the permutation
- See if the corresponding bit is a 1 in the element we're hashing.

What About Hashing?

- MinHash: go through each index in the permutation
- See if the corresponding bit is a 1 in the element we're hashing.
- How can we do this?

What About Hashing?

- MinHash: go through each index in the permutation
- See if the corresponding bit is a 1 in the element we're hashing.
- How can we do this?
- Most efficient way I know is not clever. Just go through each index, and check to see if that bit is set (say by calculating `x & (1 << index)` —but remember that these are 128 bits)

Concatenating Indices

- Each time you hash you'll get k indices

Concatenating Indices

- Each time you hash you'll get k indices
- Each is a number from 0 to 127

Concatenating Indices

- Each time you hash you'll get k indices
- Each is a number from 0 to 127
- How can these get concatenated together?

Concatenating Indices

- Each time you hash you'll get k indices
- Each is a number from 0 to 127
- How can these get concatenated together?
- Option 1: convert to strings, call `strcat`

Concatenating Indices

- Each time you hash you'll get k indices
- Each is a number from 0 to 127
- How can these get concatenated together?
- Option 1: convert to strings, call `strcat`
- Note: need to make sure to convert to *three-digit* strings! Otherwise hashing to 12 and then 1 will look the same as hashing to 1 and then 21. (012 and 001 instead)

Concatenating Indices

- Each time you hash you'll get k indices
- Each is a number from 0 to 127
- How can these get concatenated together?
- Option 1: convert to strings, call `strcat`
- Note: need to make sure to convert to *three-digit* strings! Otherwise hashing to 12 and then 1 will look the same as hashing to 1 and then 21. (012 and 001 instead)
- Option 2: Treat as bits. 0 to 127 can be stored in 7 bits. Store the hash as a sequence of k 8-bit chunks.

Getting a Good k

- In theory we want buckets of size 1.

Getting a Good k

- In theory we want buckets of size 1.
- In practice, we want *slightly* bigger.

Getting a Good k

- In theory we want buckets of size 1.
- In practice, we want *slightly* bigger.
- Why? Lots of buckets and lots of repetitions have bad constants.

Getting a Good k

- In theory we want buckets of size 1.
- In practice, we want *slightly* bigger.
- Why? Lots of buckets and lots of repetitions have bad constants.
- Smaller k means fewer buckets, fewer repetitions (but bigger buckets and more comparisons)

Getting a Good k

- In theory we want buckets of size 1.
- In practice, we want *slightly* bigger.
- Why? Lots of buckets and lots of repetitions have bad constants.
- Smaller k means fewer buckets, fewer repetitions (but bigger buckets and more comparisons)
- Start with $k \approx \log_3 n$, but experiment with slightly smaller values.

Repetitions?

- You're guaranteed that there exists a close pair in the dataset

Repetitions?

- You're guaranteed that there exists a close pair in the dataset
- My implementation just keeps repeating until the pair is found (no maximum number of repetitions)

Repetitions?

- You're guaranteed that there exists a close pair in the dataset
- My implementation just keeps repeating until the pair is found (no maximum number of repetitions)
- The discussion of repetitions in the lecture is for two reasons: 1. analysis, 2. give intuition for the tradeoff by varying k

How to Deal with Buckets?

- Each time we hash, (i.e. build a new "hash table") need to figure out what hashes where so that we can compare elements with the same hash

How to Deal with Buckets?

- Each time we hash, (i.e. build a new “hash table”) need to figure out what hashes where so that we can compare elements with the same hash
- Unfortunately, we’re not hashing to a number from (say) 0 to $n-1$. We’re instead concatenating indices

How to Deal with Buckets?

- Each time we hash, (i.e. build a new "hash table") need to figure out what hashes where so that we can compare elements with the same hash
- Unfortunately, we're not hashing to a number from (say) 0 to $n - 1$. We're instead concatenating indices
- How to keep track of buckets?

How to Deal with Buckets?

- Each time we hash, (i.e. build a new "hash table") need to figure out what hashes where so that we can compare elements with the same hash
- Unfortunately, we're not hashing to a number from (say) 0 to $n - 1$. We're instead concatenating indices
- How to keep track of buckets?
- I'll give three options. I believe one is likely best but I'm not sure.

Option 1: Sorting

- This I got from a student's midterm solution (thanks if it was you!)

Option 1: Sorting

- This I got from a student's midterm solution (thanks if it was you!)
- For each item, store a struct with both the item and its hash value. Store these structs all in an array

Option 1: Sorting

- This I got from a student's midterm solution (thanks if it was you!)
- For each item, store a struct with both the item and its hash value. Store these structs all in an array
- Sort the array by hash value. Then all items with the same hash value will be adjacent!

Option 1: Sorting

- This I got from a student's midterm solution (thanks if it was you!)
- For each item, store a struct with both the item and its hash value. Store these structs all in an array
- Sort the array by hash value. Then all items with the same hash value will be adjacent!
- Then: scan array left to right. Call an all-compare-all function on each sequence of array indices that have the same hash value.

Option 1: Sorting

Pros of this method?

- Easy to implement; just need an array and a sort function

Option 1: Sorting

Pros of this method?

- Easy to implement; just need an array and a sort function
- Cache-efficient

Option 1: Sorting

Pros of this method?

- Easy to implement; just need an array and a sort function
- Cache-efficient
- Space-efficient

Cons?

Option 1: Sorting

Pros of this method?

- Easy to implement; just need an array and a sort function
- Cache-efficient
- Space-efficient

Cons?

- $O(n \log n)$ time, where only $O(n)$ time is required

Option 1: Sorting

Pros of this method?

- Easy to implement; just need an array and a sort function
- Cache-efficient
- Space-efficient

Cons?

- $O(n \log n)$ time, where only $O(n)$ time is required
- Need to make the structs and copy over the data

Option 2: Hash table

- Create a hash table of size $N = O(n)$

Option 2: Hash table

- Create a hash table of size $N = O(n)$
- Once you get the hash value, use murmurhash to get a random 32-bit number. Mod that to get a number from 0 to $N - 1$

Option 2: Hash table

- Create a hash table of size $N = O(n)$
- Once you get the hash value, use murmurhash to get a random 32-bit number. Mod that to get a number from 0 to $N - 1$
- Use chaining to resolve collisions

Option 2: Hash table

- Create a hash table of size $N = O(n)$
- Once you get the hash value, use murmurhash to get a random 32-bit number. Mod that to get a number from 0 to $N - 1$
- Use chaining to resolve collisions
- This does increase bucket size (as multiple buckets may wind up in the same place in the table)

Option 2: Hash table

Pros?

- $O(n)$, easy to pass buckets

Cons?

- Need to make a whole hash table

Option 2: Hash table

Pros?

- $O(n)$, easy to pass buckets

Cons?

- Need to make a whole hash table
- (very) cache-inefficient

Option 3: Array for Each Bucket

- Scan items to get the size of each bucket

Option 3: Array for Each Bucket

- Scan items to get the size of each bucket
- Then, make an array for each bucket

Option 3: Array for Each Bucket

- Scan items to get the size of each bucket
- Then, make an array for each bucket
- Pros: $O(n)$ time, optimal space, easy to pass around

Option 3: Array for Each Bucket

- Scan items to get the size of each bucket
- Then, make an array for each bucket
- Pros: $O(n)$ time, optimal space, easy to pass around
- Cons: Seems difficult, and perhaps bad constants

Storing a Hash

- Just need a permutation on $\{0, \ldots, 127\}$

Storing a Hash

- Just need a permutation on $\{0, \ldots, 127\}$
- How can we store that?

Storing a Hash

- Just need a permutation on $\{0, \ldots, 127\}$
- How can we store that?
- First key observation: we (basically) *never* make it through the whole permutation (we'll always see at least one 1 first)

Storing a Hash

- Just need a permutation on $\{0, \ldots, 127\}$
- How can we store that?
- First key observation: we (basically) *never* make it through the whole permutation (we'll always see at least one 1 first)
- Taking that a bit further: we only really need the first few indices. If we're using $\hat{k}$ indices from one ordering, something like $4\hat{k}$ or $8\hat{k}$ will almost certainly suffice.

Storing a Hash

- Just need a permutation on $\{0, \ldots, 127\}$
- How can we store that?
- First key observation: we (basically) *never* make it through the whole permutation (we'll always see at least one 1 first)
- Taking that a bit further: we only really need the first few indices. If we're using $\hat{k}$ indices from one ordering, something like $4\hat{k}$ or $8\hat{k}$ will almost certainly suffice.
- What about elements that hash further? Answer: just give them the value of the last index in the ordering.

Truncating Hash Example

- Let's say our permutation is $\{47, 11, 85, 64, 13, 74, 70, 107, 112, 103, 7, 95, 3, \ldots\}$ and $\hat{k} = 2$.

Truncating Hash Example

- Let's say our permutation is $\{47, 11, 85, 64, 13, 74, 70, 107, 112, 103, 7, 95, 3, \ldots\}$ and $\hat{k} = 2$.
- I only store $\{47, 11, 85, 64, 13, 74, 107, 112\}$. If we go past 112 for some x, and we have not seen $\hat{k}$ indices that are a 1 in x, I just write 112 until I get $\hat{k}$ numbers.

Takeaway from Truncating Hashes

- This means we can store fewer bits, fewer random numbers

Takeaway from Truncating Hashes

- This means we can store fewer bits, fewer random numbers
- Might be easier to handle. (Arrays of size 16-20 are nicer than arrays of size 128.)