Computational Tools for Data Science
02807, E 2018

Filtering Streams

Paul Fischer

Institut for Matematik og Computer Science
Danmarks Tekniske Universitet

Efterår 2018

Overview

- What are streams and what is mined from them?
- Hashing.
- The Bloom Filter.
- Majority Element.
- Heavy hitters and Count-Min Sketch

Example

Hashing is a technique which maps elements from a large space to elements in a smaller space.

The effect is to save space and often also time.

Example

Consider the space of strings of at most 20 letters, where the alphabet is {A,B,. . . ,Z} (26 letters).
There are $207252748510177855184338052 71 \approx 2.07 \cdot 10^{28}$ such strings.
Suppose the streams consist of such strings and we want to remember which strings have been in the stream.

Example

Hashing is a technique which maps elements from a large space to elements in a smaller space.

The effect is to save space and often also time.

Example

Consider the space of strings of at most 20 letters, where the alphabet is {A,B,. . . ,Z} (26 letters). There are $207252748510177855184338052 71 \approx 2.07 \cdot 10^{28}$ such strings.
Suppose the streams consist of such strings and we want to remember which strings have been in the stream.

Version 1: We make a list of all such strings an mark those we have seen. Impossible, we would need more than 10^{16} TB.

Example

Hashing is a technique which maps elements from a large space to elements in a smaller space.

The effect is to save space and often also time.

Example

Consider the space of strings of at most 20 letters, where the alphabet is {A,B,. . . ,Z} (26 letters). There are $207252748510177855184338052 71 \approx 2.07 \cdot 10^{28}$ such strings.
Suppose the streams consist of such strings and we want to remember which strings have been in the stream.

Version 1: We make a list of all such strings an mark those we have seen. Impossible, we would need more than 10^{16} TB.

Version 2: We make a list of one million integers, say $[0, 1, 2, \ldots, 999\ 999]$. From each string S which we see, we compute a number $h(S)$ between 0 and $999; 999$ and mark this number.

More Formal

In general: A *hash function* $h: \mathcal{U} \mapsto \mathcal{T}$ maps elements form a large universe $\mathcal{U}$ to a small *hash table* $\mathcal{T}$.

In our case $h: \{A, B, \ldots, Z\}^{\leq 20} \mapsto [0, 999\ 999]$

More Formal

In general: A *hash function* $h: \mathcal{U} \mapsto \mathcal{T}$ maps elements form a large universe $\mathcal{U}$ to a small *hash table* $\mathcal{T}$.

In our case $h: \{A, B, \ldots, Z\}^{\leq 20} \mapsto [0, 999\,999]$

There are many ways to define h. For example we could sum the ASCII codes for the letters (and take the remainder modulo $1\,000\,000$). ASCII(A) = 65, ASCII(B) = 66, so $h(PAUL) = 306$.

More Formal

In general: A *hash function* $h: \mathcal{U} \mapsto \mathcal{T}$ maps elements form a large universe $\mathcal{U}$ to a small *hash table* $\mathcal{T}$.

In our case $h: \{A, B, \ldots, Z\}^{\leq 20} \mapsto [0, 999\,999]$

There are many ways to define h. For example we could sum the ASCII codes for the letters (and take the remainder modulo $1\,000\,000$). ASCII(A) = 65, ASCII(B) = 66, so $h(PAUL) = 306$.

Advantage: MUCH less space.

Disadvantage Not correct. Note that $h(PAUL) = 306$ and $h(AUPL) = 306$. So, if 306 is marked in our list, have we seen PAUL or AUPL or something different?

More Formal

In general: A *hash function* $h: \mathcal{U} \mapsto \mathcal{T}$ maps elements form a large universe $\mathcal{U}$ to a small *hash table* $\mathcal{T}$.

In our case $h: \{A, B, \ldots, Z\}^{\leq 20} \mapsto [0, 999\,999]$

There are many ways to define h. For example we could sum the ASCII codes for the letters (and take the remainder modulo $1\,000\,000$). ASCII(A) = 65, ASCII(B) = 66, so $h(PAUL) = 306$.

Advantage: MUCH less space.

Disadvantage Not correct. Note that $h(PAUL) = 306$ and $h(AUPL) = 306$. So, if 306 is marked in our list, have we seen PAUL or AUPL or something different?

Regardless which hash one chooses, this effect cannot be avoided because $|\mathcal{T}| < |\mathcal{U}|$. However there are much smarter hash functions than the one we used.

More Formal

In general: A *hash function* $h: \mathcal{U} \mapsto \mathcal{T}$ maps elements form a large universe $\mathcal{U}$ to a small *hash table* $\mathcal{T}$.

In our case $h: \{A, B, \ldots, Z\}^{\leq 20} \mapsto [0, 999\,999]$

There are many ways to define h. For example we could sum the ASCII codes for the letters (and take the remainder modulo $1\,000\,000$). ASCII(A) = 65, ASCII(B) = 66, so $h(PAUL) = 306$.

Advantage: MUCH less space.

Disadvantage Not correct. Note that $h(PAUL) = 306$ and $h(AUPL) = 306$. So, if 306 is marked in our list, have we seen PAUL or AUPL or something different?

Regardless which hash one chooses, this effect cannot be avoided because $|\mathcal{T}| < |\mathcal{U}|$. However there are much smarter hash functions than the one we used.

Why use Hashing nevertheless?

Assume that we expect that our stream does not contain many random strings, but most of the strings are word from the English language. However some strings might be random. Again, we cannot make a list of all possible words, because we (probaly) do not know all English words.

Why use Hashing nevertheless?

Assume that we expect that our stream does not contain many random strings, but most of the strings are word from the English language. However some strings might be random. Again, we cannot make a list of all possible words, because we (probaly) do not know all English words.

But we should expect, that there will be less than one million different strings.

Why use Hashing nevertheless?

Assume that we expect that our stream does not contain many random strings, but most of the strings are word from the English language. However some strings might be random. Again, we cannot make a list of all possible words, because we (probaly) do not know all English words.

But we should expect, that there will be less than one million different strings.

If we have a hash function h which "scatters nicely" then using hashing should be quite precise.

Here "scatters nicely" means that all values in the table $\mathcal{T}$ will be hit almost equally often when one computes $h(u)$ for all $u \in \mathcal{U}$.

It is also desirable that the hash function comes from a "universal family" $\mathcal{H}$ of functions. That is, with $m = |\mathcal{T}|$

$$\forall x, y \in \mathcal{U}, x \neq y : \; Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}$$

Streams

A *stream* is a sequence of objects which appear one after the other in time. One often assumes that the objects are of the same type, e.g, strings integers.

Streams

A *stream* is a sequence of objects which appear one after the other in time. One often assumes that the objects are of the same type, e.g, strings integers.

A stream has no predefined or know end.

Streams

A *stream* is a sequence of objects which appear one after the other in time. One often assumes that the objects are of the same type, e.g, strings integers.

A stream has no predefined or know end.

The task it be always able to answer question on the part of the stream seen so far. Thus some information has to be updated when a new element appears in the stream.

Streams

A *stream* is a sequence of objects which appear one after the other in time. One often assumes that the objects are of the same type, e.g, strings integers.

A stream has no predefined or know end.

The task it be always able to answer question on the part of the stream seen so far. Thus some information has to be updated when a new element appears in the stream.

Information asked about a stream (mined from it) could be:

- Did a specific object occur in the stream by now?
- How many times did a specific object occur in the stream by now?
- Does the last element we saw have a certain property?

Filtering Streams

A frequent problem for analysing streams is *selection*, or *filtering*. One wants to identify elements in the stream which meet a certain criterion. These elements are treated/stored, while the other elements are discarded.

An example is a stream of URLs which are considered safe or unsafe. We introduce the **Bloom filter** for handling such tasks.

Filtering Streams

A frequent problem for analysing streams is *selection*, or *filtering*. One wants to identify elements in the stream which meet a certain criterion. These elements are treated/stored, while the other elements are discarded.

An example is a stream of URLs which are considered safe or unsafe. We introduce the **Bloom filter** for handling such tasks.

Elements of a Bloom filter:

1. A set $\mathcal{S}$ of m key values which are all considerer save.
2. An array A of n bits, initially all 0's.
3. A collection of hash functions $h_1, h_2, \ldots, h_k$, such that $h_i : \mathcal{U} \mapsto \{1, 2, \ldots, n\}$, where $\mathcal{U} \supseteq \mathcal{S}$

The purpose of the Bloom filter is to allow through **all** stream elements whose keys are in $\mathcal{S}$, while rejecting **most** of the stream elements whose keys are not in $\mathcal{S}$.

Again we want to avoid storing all of $\mathcal{S}$.

Training the Bloom Filter

In the training phase, we look at all values in $\mathcal{S}$ and compute their hash values:

for $(i = 1, 2, \ldots n)$ **do**
  $A[i] \leftarrow 0$;
end
for $(s \in \mathcal{S})$ **do**
  for $(i = 1, 2, \ldots, k)$ **do**
    $j \leftarrow h_i(s)$;
    $A[j] \leftarrow 1$
  end
end

Algorithm 1: Training the Bloom filter.

Using the Bloom Filter

When a new, unclassified key t arrives, we want to check whether it is in the set $\mathcal{S}$ of safe keys. We do so by checking whether all hash values of t point to a 1.

for $(i = 1, 2, \ldots, k)$ **do**
    $j \leftarrow h_i(t)$;
    if $(A[j] = 0)$ **then**
        return *UNSAFE*;
    end
end
return *SAFE*;

Algorithm 2: Using the Bloom filter.

Using the Bloom Filter

When a new, unclassified key t arrives, we want to check whether it is in the set $\mathcal{S}$ of safe keys. We do so by checking whether all hash values of t point to a 1.

for *($i = 1, 2, \ldots, k$)* **do**
  $j \leftarrow h_i(t)$;
  if *($A[j] = 0$)* **then**
    return *UNSAFE*;
  end
end
return *SAFE*;

Algorithm 3: Using the Bloom filter.

If the value t is in $\mathcal{S}$, i.e., it is safe, then the filter will always return SAFE. If t is not in $\mathcal{S}$, i.e., it is unsafe, then the filter might return UNSAFE or SAFE. The latter case is called a *false positive*. We want to make the probability for false positives small.

Analysis the Bloom Filter

We give some intuition why the Bloom filter is constructed as it is and refer to the book for details.

Analysis the Bloom Filter

We give some intuition why the Bloom filter is constructed as it is and refer to the book for details.

The use of more than one hash function tries to lessen the probability for false positives. Intuitively, if the hash functions “map differently” then the chances that **all** functions map a $t \notin \mathcal{S}$ to 1 is smaller than the probability that a single function does this.

Analysis the Bloom Filter

We give some intuition why the Bloom filter is constructed as it is and refer to the book for details.

The use of more than one hash function tries to lessen the probability for false positives. Intuitively, if the hash functions “map differently” then the chances that **all** functions map a $t \notin \mathcal{S}$ to 1 is smaller than the probability that a single function does this.

Also $n = |A|$ should be larger than $m = |\mathcal{S}|$ so that “there is enough space for zeros”.

Analysis the Bloom Filter

We give some intuition why the Bloom filter is constructed as it is and refer to the book for details.

The use of more than one hash function tries to lessen the probability for false positives. Intuitively, if the hash functions "map differently" then the chances that **all** functions map a $t \notin \mathcal{S}$ to 1 is smaller than the probability that a single function does this.

Also $n = |A|$ should be larger than $m = |\mathcal{S}|$ so that "there is enough space for zeros".

With $m = |\mathcal{S}|$, $n = |A|$, and k ($k = n/m$ is often used) hash functions, the probability for a false positive is

$$\left(1 - e^{-km/n}\right)^k$$

For $m = 10^9$, $n = 8 \cdot 10^9$ and $k = 8$ the the probability for a false positive is 0.02549, i.e., ca. 2.5%.

Analysis the Bloom Filter

We give some intuition why the Bloom filter is constructed as it is and refer to the book for details.

The use of more than one hash function tries to lessen the probability for false positives. Intuitively, if the hash functions "map differently" then the chances that **all** functions map a $t \notin \mathcal{S}$ to 1 is smaller than the probability that a single function does this.

Also $n = |A|$ should be larger than $m = |\mathcal{S}|$ so that "there is enough space for zeros".

With $m = |\mathcal{S}|$, $n = |A|$, and k ($k = n/m$ is often used) hash functions, the probability for a false positive is

$$\left(1 - e^{-km/n}\right)^k$$

For $m = 10^9$, $n = 8 \cdot 10^9$ and $k = 8$ the the probability for a false positive is 0.02549, i.e., ca. 2.5%.
Exercise: Implement a Bloom filter, use packages for bit vector and hash.

Finding the Majority Element

- Given an array A of length n.
- We know that there is an element which appears strictly more than $n/2$ times in the array.
- Find the element.

Finding the Majority Element

- Given an array A of length n.
- We know that there is an element which appears strictly more than $n/2$ times in the array.
- Find the element.

Possible solutions

- Sort the array, run through and count how often you fin the same element in a row. Time $O(n \log n)$ for sorting.
- Find the median; this is the wanted element. Time $O(n)$ with large constants.
- Use the one-pass algorithm described in a moment.

Finding the Majority Element

The one-pass algorithm:

```
counter ← 0;
current ← NULL;
for (i = 1, ..., n) do
    if (counter = 0) then
        current ← A[i];
        counter ← counter + 1;
    else
        if (current = A[i]) then
            counter ← counter + 1;
        else
            counter ← counter − 1;
        end
    end
end
return current
```

Idea: Each entry of A which contains a non-majority-value can only "cancel out" one copy of the majority value. The algorithms uses time $O(n)$ and constant auxiliary space.

The Heavy Hitters Problem

The Heavy Hitters problem generalizes the majority problem.

- Given an array A of length n.
- A positive integer k (much) smaller than n.
- Find all elements in A which appear more than n/k times. There are at most k such elements.

The problem is harder than the majority problem: There is **no** algorithm that solves the Heavy Hitters problems in one pass while using a sublinear amount of auxiliary space.

The Heavy Hitters Problem

The Heavy Hitters problem generalizes the majority problem.

- Given an array A of length n.
- A positive integer k (much) smaller than n.
- Find all elements in A which appear more than n/k times. There are at most k such elements.

The problem is harder than the majority problem: There is **no** algorithm that solves the Heavy Hitters problems in one pass while using a sublinear amount of auxiliary space.

Question: When considering streams, why do we use n/k and not a fixed number like “an element is heavy if it occurs at least 1000 times”.

The Heavy Hitters Problem

Let us relax the problem to allow a fast and space efficient solution.

- Given an array A of length n.
- A positive integer k (much) smaller than n.
- Find a list L of values such that
  - Every value that occurs at least n/k times in A is in L.
  - Every value L occurs at least $n/k - \varepsilon n$ times in A.

Here $\varepsilon > 0$ is a user-defined value. The resulting problem is called *ε-approximate heavy hitters* (ε–HH).

The Count-Min Sketch Algorithm

Parameters A (small) number ℓ of hash functions. A number b of bucktes, b medium sized, but $b \ll n$.

The data structure A $\ell \times b$ array CSM of non-negative integer counters, initially all 0.

Increment operation Given an object x, increment one counter per row

for $i = 1, 2, \ldots, \ell$ **do**
  $CMS[i][h_i(x)] \leftarrow CMS[i][h_i(x)] + 1$
end

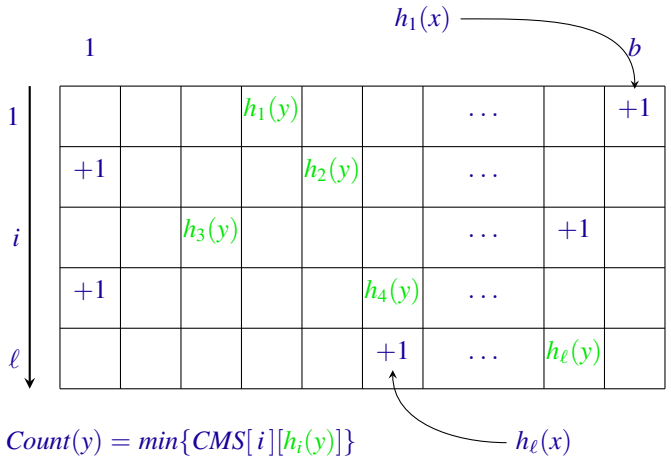
Algorithm 4: INC(x)

Count operation

return $min\{CMS[i][h_i(x)] \mid i = 1, 2, \ldots, \ell\}$

Algorithm 5: COUNT(x)

CSM Data Structure



Properties of CMS

- Let f_x be the true number of occurrences of x in the data, at the current time.
- It always holds that $Count(x) \geq f_x$. Reason: For all occurrences of x, we add 1 to $CSM[\,i\,][h_i(x)]$, $i = 1, \ldots, \ell$. There might be $y \neq x$ such that for some i we have $h_i(x) = h_i(y)$, resulting in an overcount.
- The data structure guarantees one-sided error: any heavy element will be identified.
- One has to control that non "ε-heavy" elements ($f_x < n/k - \varepsilon n$) do not appear in the list. This can only be achieved with a certain probability δ.

Choosing and Setting Parameters

User's choices:

- k, the fraction which makes an object x heavy ($f_x \geq n/k$).
- ε, the tolerance allowed for near-heavy objects x ($f_x \geq n/k - \varepsilon n$), often $\varepsilon = 1/(2k)$.
- δ, the allowed failure probability: $Prob\left[\min_i CMS[i][h_i(x)] > f_x + \varepsilon n\right] \leq \delta$, often 0.01.

Derived parameter settings:

- $b = e/\varepsilon$ (Note: independent of n, if ε is, $e = 2.71\ldots$ Euler's constants.)
- $\ell \geq \ln(1/\delta)$ (For $\delta = 0.01$, it will suffice $\ell = 5$).