CSE 312

Foundations of Computing II

Lecture 9: Variance and Independence of RVs (continued)

Lecture 10: Bloom Filters

Announcements

- PSet 3 due today
- PSet 2 returned yesterday
- PSet 4 posted this evening
  - Last PSet prior to midterm (midterm is in exactly two weeks from now)
  - Midterm info will follow soon
  - PSet 5 will only come after the midterm in two weeks

Recap Variance – Properties

Definition. The **variance** of a (discrete) RV X is

$$\mathrm{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2] = \sum_x p_X(x) \cdot (x - \mathbb{E}[X])^2$$

Theorem. For any $a, b \in \mathbb{R}$, $\mathrm{Var}(a \cdot X + b) = a^2 \cdot \mathrm{Var}(X)$

Theorem. $\mathrm{Var}(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2$

Variance

Theorem. $\mathrm{Var}(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2$

Proof: $\mathrm{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2]$

Recall $\mathbb{E}[X]$ is a **constant**

$$= \mathbb{E}[X^2 - 2\mathbb{E}[X] \cdot X + \mathbb{E}[X]^2]$$

$$= \mathbb{E}(X^2) - 2\mathbb{E}[X]\mathbb{E}[X] + \mathbb{E}[X]^2$$

(linearity of expectation!)

$$= \mathbb{E}[X^2] - \mathbb{E}[X]^2$$

$\mathbb{E}[X^2]$ and $\mathbb{E}[X]^2$ are different !

Variance of Indicator Random Variables

Suppose that X_A is an indicator RV for event A with $P(A) = p$ so

$$\mathbb{E}[X_A] = P(A) = p$$

Since X_A only takes on values 0 and 1, we always have $X_A^2 = X_A$ so

$$\text{Var}(X_A) = \mathbb{E}[X_A^2] - \mathbb{E}[X_A]^2 = \mathbb{E}[X_A] - \mathbb{E}[X_A]^2 = p - p^2 = p(1 - p)$$

In General, $\mathrm{Var}(X + Y) \neq \mathrm{Var}(X) + \mathrm{Var}(Y)$

Proof by counter-example:

- Let X be a r.v. with pmf $P(X = 1) = P(X = -1) = 1/2$
  - What is $\mathbb{E}[X]$ and $\mathrm{Var}(X)$?
- Let $Y = -X$
  - What is $\mathbb{E}[Y]$ and $\mathrm{Var}(Y)$?

What is $\mathrm{Var}(X + Y)$?

Agenda

- Variance
- Properties of Variance
- Independent Random Variables
- Properties of Independent Random Variables
- An Application: Bloom Filters!

Random Variables and Independence

Comma is shorthand for AND

Definition. Two random variables X, Y are **(mutually) independent** if for all $x, y,$

$$P(X = x, Y = y) = P(X = x) \cdot P(Y = y)$$

Intuition: Knowing X doesn't help you guess Y and vice versa

Definition. The random variables $X_1, \dots, X_n$ are **(mutually) independent** if for all $x_1, \dots, x_n,$

$$P(X_1 = x_1, \dots, X_n = x_n) = P(X_1 = x_1) \cdots P(X_n = x_n)$$

Note: No need to check for all subsets, but need to check <u>for all outcomes</u>!

Example

Let X be the number of heads in n *independent* coin flips of the same coin. Let $Y = X \bmod 2$ be the parity (even/odd) of X.

Are X and Y independent?

Poll:
pollev.com/paulbeame028

A. Yes
B. No

Example

Make $2n$ *independent* coin flips of the same coin.

Let X be the number of heads in the first n flips and Y be the number of heads in the last n flips.

Are X and Y independent?

Poll:
pollev.com/paulbeame028

A. Yes
B. No

Agenda

- Variance
- Properties of Variance
- Independent Random Variables
- Properties of Independent Random Variables
- An Application: Bloom Filters!

Important Facts about Independent Random Variables

Theorem. If X, Y independent, $\mathbb{E}[X \cdot Y] = \mathbb{E}[X] \cdot \mathbb{E}[Y]$

Theorem. If X, Y independent, $\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y)$

Corollary. If $X_1, X_2, \ldots, X_n$ mutually independent,

$$\mathrm{Var}\left(\sum_{i=1}^{n} X_i\right) = \sum_{i}^{n} \mathrm{Var}(X_i)$$

Example – Coin Tosses

We flip n *independent* coins, each one heads with probability p

- $X_i = \begin{cases} 1, & i^{\text{th}} \text{ outcome is heads} \\ 0, & i^{\text{th}} \text{ outcome is tails.} \end{cases}$
- $Z =$ number of heads

Fact. $Z = \sum_{i=1}^{n} X_i$

$P(X_i = 1) = p$
$P(X_i = 0) = 1 - p$

$P(Z = k) = \binom{n}{k} p^k (1-p)^{n-k}$

What is $\mathbb{E}[Z]$? What is $\mathrm{Var}(Z)$?

Note: $X_1, \dots, X_n$ are <u>mutually</u> independent! [Verify it formally!]

$$\mathrm{Var}(Z) = \sum_{i=1}^{n} \mathrm{Var}(X_i) = n \cdot p(1-p)$$

Note $\mathrm{Var}(X_i) = p(1-p)$

(Not Covered) Proof of $\mathbb{E}[X \cdot Y] = \mathbb{E}[X] \cdot \mathbb{E}[Y]$

Theorem. If X, Y independent, $\mathbb{E}[X \cdot Y] = \mathbb{E}[X] \cdot \mathbb{E}[Y]$

Proof

Let $x_i, y_i, i = 1, 2, \ldots$ be the possible values of X, Y.

$$\mathbb{E}[X \cdot Y] = \sum_i \sum_j x_i \cdot y_j \cdot P(X = x_i \wedge Y = y_j)$$

independence

$$= \sum_i \sum_j x_i \cdot y_i \cdot P(X = x_i) \cdot P(Y = y_j)$$

$$= \sum_i x_i \cdot P(X = x_i) \cdot \left(\sum_j y_j \cdot P(Y = y_j) \right)$$

$$= \mathbb{E}[X] \cdot \mathbb{E}[Y]$$

Note: *NOT* true in general; see earlier example $\mathbb{E}[X^2] \neq \mathbb{E}[X]^2$

(Not Covered) Proof of $\mathrm{Var}(X+Y) = \mathrm{Var}(X) + \mathrm{Var}(Y)$

Theorem. If X, Y independent, $\mathrm{Var}(X+Y) = \mathrm{Var}(X) + \mathrm{Var}(Y)$

Proof

$Var(X+Y)$

$= \mathbb{E}[(X+Y)^2] - (\mathbb{E}[X+Y])^2$

$= \mathbb{E}[X^2 + 2XY + Y^2] - (\mathbb{E}[X] + \mathbb{E}[Y])^2$

$= \mathbb{E}[X^2] + 2\,\mathbb{E}[XY] + \mathbb{E}[Y^2] - (\mathbb{E}[X]^2 + 2\,\mathbb{E}[X]\,\mathbb{E}[Y] + \mathbb{E}[Y]^2)$

$= \mathbb{E}[X^2] - \mathbb{E}[X]^2 + \mathbb{E}[Y^2] - \mathbb{E}[Y]^2 + 2\,\mathbb{E}[XY] - 2\,\mathbb{E}[X]\,\mathbb{E}[Y]$

$= Var(X) + Var(Y) + 2\,\mathbb{E}[XY] - 2\,\mathbb{E}[X]\,\mathbb{E}[Y]$

$= Var(X) + Var(Y)$

linearity

equal by independence

Brain Break

Agenda

- Variance
- Properties of Variance
- Independent Random Variables
- Properties of Independent Random Variables
- An Application: Bloom Filters!

Basic Problem

Problem: Store a subset S of a large set U.

Example. U = set of 128 bit strings $|U| \approx 2^{128}$
S = subset of strings of interest $|S| \approx 1000$

Two goals:

1. **Very fast** (ideally constant time) answers to queries "Is $x \in S$?" for any $x \in U$.
2. **Minimal storage** requirements.

Naïve Solution I – Constant Time

Idea: Represent S as an array A with 2^{128} entries.

$$\mathrm{A}[x] = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases}$$

$S = \{0,2, \dots, K\}$ →

0	1	2	…	K	…		
1	0	1	0	1	…	0	0

Membership test: To check. $x \in S$ just check whether $\mathrm{A}[x] = 1$.

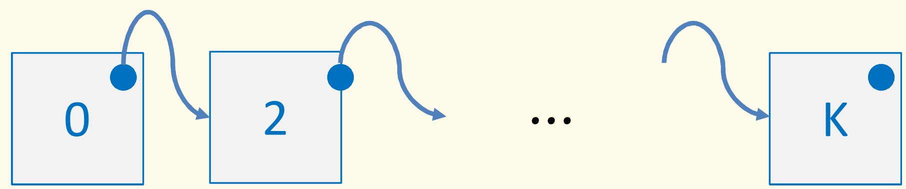
→ **constant time!**

Storage: Require storing 2^{128} bits, even for small S.

Naïve Solution II – Small Storage

Idea: Represent S as a list with $|S|$ entries.

$S = \{0,2, \dots, K\}$



Storage: Grows with $|S|$ <u>only</u>

Membership test: Check $x \in S$ requires time linear in $|S|$

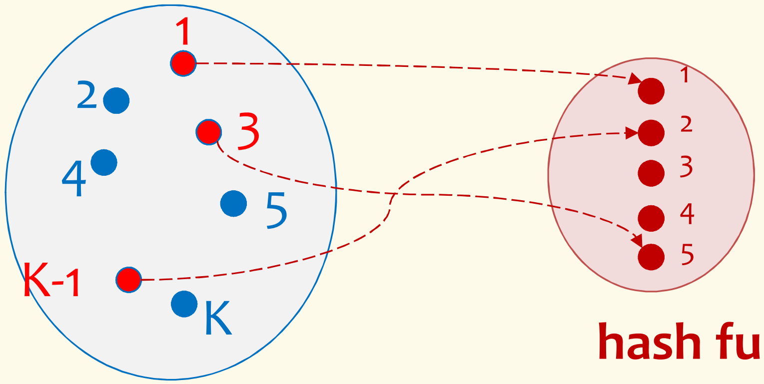
(Can be made logarithmic by using a tree)

Hash Table

Idea: Map elements in S into an array A of size m using a hash function $\mathbf{h}$

Membership test: To check $x \in S$ just check whether $A[\mathbf{h}(x)] = x$

Storage: m elements (size of array)



hash function h: $U \to [m]$

Hash Table

Idea: Map elements in S into an array A of size m using a hash function $\mathbf{h}$

Membership test: To check $x \in S$ just check whether $A[\mathbf{h}(x)] = x$

Storage: m elements (size of array)

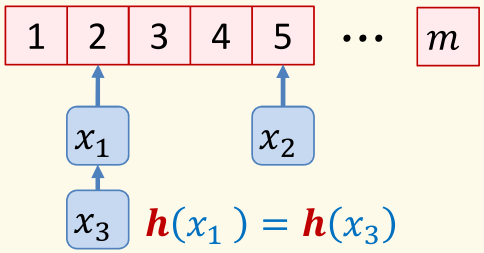
Challenge 1: Ensure $\boldsymbol{h}(x) \neq \boldsymbol{h}(y)$ *for most* $x, y \in S$

Challenge 2: Ensure $m = O(|S|)$

Hashing: collisions

Collisions occur when $\boldsymbol{h}(x) = \boldsymbol{h}(y)$ for some distinct $x, y \in S$, i.e., two elements of set map to the same location

- Common solution: <u>chaining</u> – at each location (bucket) in the table, keep linked list of all elements that hash there.



Good hash functions to keep collisions low

- The hash function h is good iff it
  - distributes elements uniformly across the m array locations so that
  - pairs of elements are mapped independently

"Universal Hash Functions" – see CSE 332

Hashing: summary

Hash Tables

- They store the data itself
- With a good hash function, the data is well distributed in the table and lookup times are small.
- However, they need at least as much space as all the data being stored, i.e., $m = \Omega(|S|)$

In some cases, $|S|$ is huge, or not known a-priori …

Can we do better!?

Bloom Filters
to the rescue
(Named after Burton Howard Bloom)

Bloom Filters – Main points

- Probabilistic data structure.
- Close cousins of hash tables.
  - But: Ridiculously space efficient
- Occasional errors, specifically false positives.

Bloom Filters

- Stores information about a set of elements $S \subseteq U$.
- Supports two operations:
  1. **`add`**(x) - adds $x \in U$ to the set S
  2. **`contains`**(x) – ideally: true if $x \in S$, false otherwise

Instead, relaxed guarantees:

- False $\rightarrow$ **definitely** not in S
- True $\rightarrow$ **possibly** in S

  [i.e. we could have *false positives*]

Bloom Filters – Why Accept False Positives?

- **Speed** – both `add` and `contains` very very fast.
- **Space** – requires a miniscule amount of space relative to storing all the actual items that have been added.
  - Often just 8 bits per inserted item!
- **Fallback mechanism** – can distinguish false positives from true positives with extra cost
  - Ok if mostly negatives expected + low false positive rate

Bloom Filters: Application

- Google Chrome has a database of malicious URLs, but it takes a long time to query.
- Want an in-browser structure, so needs to be efficient and be space-efficient
- Want it so that can check if a URL is in structure:
  - If return False, then definitely not in the structure (don’t need to do expensive database lookup, website is safe)
  - If return True, the URL may or may not be in the structure. Have to perform expensive lookup in this rare case.

Bloom Filters – More Applications

- Any scenario where space and efficiency are important.
- Used a lot in networking
- In distributed systems when want to check consistency of data across different locations, might send a Bloom filter rather than the full set of data being stored.
- Google BigTable uses Bloom filters to reduce disk lookups
- Internet routers often use Bloom filters to track blocked IP addresses.
- And on and on…

What you can’t do with Bloom filters

- There is no delete operation
  - Once you have added something to a Bloom filter for S, it stays

- You can’t use a Bloom filter to name any element of S

But what you ***can*** do makes them very effective!

Bloom Filters – Ingredients

Basic data structure is a $k \times m$ binary array
"the Bloom filter"

- k rows $t_1, \dots, t_k$, each of size m
- Think of each row as an m-bit vector

k different hash functions $\mathbf{h}_1, \dots, \mathbf{h}_k : U \to [m]$

Bloom Filters – Three operations

- Set up Bloom filter for $S = \emptyset$

$$\begin{array}{l}\textbf{function } \text{INITIALIZE}(k, m)\\ \quad \textbf{for } i = 1, \ldots, k\text{: } \textbf{do}\\ \quad\quad t_i = \text{new bit vector of } m \text{ 0s}\end{array}$$

- Update Bloom filter for $S \leftarrow S \cup \{x\}$

$$\begin{array}{l}\textbf{function } \text{ADD}(x)\\ \quad \textbf{for } i = 1, \ldots, k\text{: } \textbf{do}\\ \quad\quad t_i[h_i(x)] = 1\end{array}$$

- Check if $x \in S$

$$\begin{array}{l}\textbf{function } \text{CONTAINS}(x)\\ \quad \textbf{return } t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1\end{array}$$

Bloom Filters - Initialization

Number of hash functions

Size of array associated to each hash function.

function INITIALIZE(k, m)
    for $i = 1, \ldots, k$: **do**
        t_i = new bit vector of m 0s

for each hash function, initialize an empty bit vector of size m

Bloom Filters: Example

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

function INITIALIZE(k, m)
for $i = 1, \ldots, k$: **do**
t_i = new bit vector of m 0s

Index →	0	1	2	3	4
t_1	0	0	0	0	0
t_2	0	0	0	0	0
t_3	0	0	0	0	0

Bloom Filters: Add

function ADD(x)
  for $i = 1, \ldots, k$: **do**
    $t_i[h_i(x)] = 1$

→ for each hash function $\mathbf{h}_i$

Index into i-th bit-vector, at index produced by hash function and set to 1

$\mathbf{h}_i(x) \rightarrow$ result of hash function $\mathbf{h}_i$ on x

Bloom Filters: Example

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

function ADD(x)
  for $i = 1, \ldots, k$: **do**
    $t_i[h_i(x)] = 1$

add("thisisavirus.com")

h_1("thisisavirus.com") $\rightarrow$ 2

Index $\rightarrow$	0	1	2	3	4
t_1	0	0	0	0	0
t_2	0	0	0	0	0
t_3	0	0	0	0	0

Bloom Filters: Example

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

function ADD(x)
  for $i = 1, \dots, k$: **do**
    $t_i[h_i(x)] = 1$

add("thisisavirus.com")

h_1("thisisavirus.com") → 2

h_2("thisisavirus.com") → 1

Index →	0	1	2	3	4
t_1	0	0	1	0	0
t_2	0	0	0	0	0
t_3	0	0	0	0	0

Bloom Filters: Example

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

function ADD(x)
 for $i = 1, \ldots, k$: **do**
  $t_i[h_i(x)] = 1$

add("thisisavirus.com")

h_1("thisisavirus.com") → 2

h_2("thisisavirus.com") → 1

h_3("thisisavirus.com") → 4

Index →	0	1	2	3	4
t_1	0	0	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	0

Bloom Filters: Example

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

function ADD(x)
for $i = 1, \ldots, k$: **do**
$t_i[h_i(x)] = 1$

add("thisisavirus.com")

h_1("thisisavirus.com") → 2

h_2("thisisavirus.com") → 1

h_3("thisisavirus.com") → 4

Index →	0	1	2	3	4
t_1	0	0	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: Contains

function CONTAINS(x)
    return $t_1[h_1(x)] == 1 \land t_2[h_2(x)] == 1 \land \cdots \land t_k[h_k(x)] == 1$

Returns True if the bit vector t_i for each hash function has bit 1 at index determined by $h_i(x)$,
Returns False otherwise

Bloom Filters: Example

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

function CONTAINS(x)
return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

contains("thisisavirus.com")

Index →	0	1	2	3	4
t_1	0	0	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: Example

Bloom filter t of length $\boldsymbol{m} = 5$ that uses $\boldsymbol{k} = 3$ hash functions

function CONTAINS(x)
return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

True

contains("thisisavirus.com")

h_1("thisisavirus.com") → 2

Index →	0	1	2	3	4
t_1	0	0	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: Example

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

function CONTAINS(x)
    return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

True   True

contains("thisisavirus.com")

h_1("thisisavirus.com") → 2

h_2("thisisavirus.com") → 1

Index →	0	1	2	3	4
t_1	0	0	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: Example

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

function CONTAINS(x)
  return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

True True True

contains("thisisavirus.com")

h_1("thisisavirus.com") → 2

h_2("thisisavirus.com") → 1

h_3("thisisavirus.com") → 4

Index →	0	1	2	3	4
t_1	0	0	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: Example

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

function CONTAINS(x)
  return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

True True True

contains("thisisavirus.com")

h_1("thisisavirus.com") → 2

h_2("thisisavirus.com") → 1

h_3("thisisavirus.com") → 4

Index →	0	1	2	3	4
t_1	0	0	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	1

Since all conditions satisfied, returns True (correctly)

Bloom Filters: False Positives

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

add("totallynotsuspicious.com")

function ADD(x)
    for $i = 1, \ldots, k$: **do**
        $t_i[h_i(x)] = 1$

Index →	0	1	2	3	4
t_1	0	0	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: False Positives

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

add("totallynotsuspicious.com")

h_1("totallynotsuspicious.com") $\rightarrow$ 1

function ADD(x)
  for $i = 1, \ldots, k$: **do**
    $t_i[h_i(x)] = 1$

Index $\rightarrow$	0	1	2	3	4
t_1	0	0	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: False Positives

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

function ADD(x)
  for $i = 1, \ldots, k$: **do**
    $t_i[h_i(x)] = 1$

add("totallynotsuspicious.com")

h_1("totallynotsuspicious.com") → 1

h_2("totallynotsuspicious.com") → 0

Index →	0	1	2	3	4
t_1	0	1	1	0	0
t_2	0	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: False Positives

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

function ADD(x)
  for $i = 1, \ldots, k$: **do**
    $t_i[h_i(x)] = 1$

add("totallynotsuspicious.com")

h_1("totallynotsuspicious.com") → 1

h_2("totallynotsuspicious.com") → 0

h_3("totallynotsuspicious.com") → 4

Index →	0	1	2	3	4
t_1	0	1	1	0	0
t_2	1	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: False Positives

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

function ADD(x)

for $i = 1, \ldots, k$: **do**

$t_i[h_i(x)] = 1$

add(“totallynotsuspicious.com”)

h_1(“totallynotsuspicious.com”) → 1

h_2(“totallynotsuspicious.com”) → 0

h_3(“totallynotsuspicious.com”) → 4

Index →	0	1	2	3	4
t_1	0	1	1	0	0
t_2	1	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: False Positives

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

function CONTAINS(x)
    return $t_1[h_1(x)] == 1 \land t_2[h_2(x)] == 1 \land \cdots \land t_k[h_k(x)] == 1$

contains(“verynormalsite.com”)

Index →	0	1	2	3	4
t_1	0	1	1	0	0
t_2	1	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: False Positives

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

function CONTAINS(x)
  return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

True

contains("verynormalsite.com")

h_1("verynormalsite.com") $\rightarrow$ 2

Index $\rightarrow$	0	1	2	3	4
t_1	0	1	1	0	0
t_2	1	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: False Positives

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

function CONTAINS(x)
    return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

True   True

contains("verynormalsite.com")

h_1("verynormalsite.com") → 2

h_2("verynormalsite.com") → 0

Index →	0	1	2	3	4
t_1	0	1	1	0	0
t_2	1	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: False Positives

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

function CONTAINS(x)
    return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

True True True

contains("verynormalsite.com")

h_1("verynormalsite.com") → 2

h_2("verynormalsite.com") → 0

h_3("verynormalsite.com") → 4

Index →	0	1	2	3	4
t_1	0	1	1	0	0
t_2	1	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: False Positives

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

function CONTAINS(x)
  return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$

True True True

contains("verynormalsite.com")

h_1("verynormalsite.com") → 2
h_2("verynormalsite.com") → 0
h_3("verynormalsite.com") → 4

Index →	0	1	2	3	4
t_1	0	1	1	0	0
t_2	1	1	0	0	0
t_3	0	0	0	0	1

Since all conditions satisfied, returns True (incorrectly)


Brain Break

Analysis: False positive probability

Question: For an element $x \in U$, what is the probability that **`contains`**(x) returns true if **`add`**(x) was never executed before?

Probability over what?! Over the choice of the $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_k$

Assumptions for the analysis (somewhat stronger than for ordinary hashing):

- Each $\mathbf{h}_i(x)$ is uniformly distributed in $[m]$ for all x and i
- Hash function outputs for each $\mathbf{h}_i$ are mutually independent (not just in pairs)
- Different hash functions are independent of each other

False positive probability – Events

Assume we perform $\texttt{add}(x_1), \dots, \texttt{add}(x_n)$
+ $\texttt{contains}(x)$ for $x \notin \{x_1, \dots, x_n\}$

Event E_i holds iff $\mathbf{h}_i(x) \in \{\mathbf{h}_i(x_1), \dots, \mathbf{h}_i(x_n)\}$

$$P(\text{false positive}) = P(E_1 \cap E_2 \cap \cdots \cap E_k) = \prod_{i=1}^{k} P(E_i)$$

$\mathbf{h}_1, \dots, \mathbf{h}_k$ independent

False positive probability – Events

Event E_i holds iff $\mathbf{h}_i(x) \in \{\mathbf{h}_i(x_1), \ldots, \mathbf{h}_i(x_n)\}$

Event E_i^c holds iff $\mathbf{h}_i(x) \neq \mathbf{h}_i(x_1)$ and … and $\mathbf{h}_i(x) \neq \mathbf{h}_i(x_n)$

$$P(E_i^c) = \sum_{z=1}^{m} P(\mathbf{h}_i(x) = z) \cdot P(E_i^c \mid \mathbf{h}_i(x) = \mathrm{z})$$

LTP

False positive probability – Events

Event E_i^c holds iff $\mathbf{h}_i(x) \neq \mathbf{h}_i(x_1)$ and … and $\mathbf{h}_i(x) \neq \mathbf{h}_i(x_n)$

$$P(E_i^c \mid \mathbf{h}_i(x) = z) = P(\mathbf{h}_i(x_1) \neq z, \dots, \mathbf{h}_i(x_n) \neq z \mid \mathbf{h}_i(x) = z)$$

Independence of values of $\boldsymbol{h}_i$ on different inputs

$$= P(\mathbf{h}_i(x_1) \neq z, \dots, \mathbf{h}_i(x_n) \neq z)$$

$$= \prod_{j=1}^{n} P(\mathbf{h}_i(x_j) \neq z)$$

Outputs of $\boldsymbol{h}_i$ uniformly spread

$$= \prod_{j=1}^{n} \left(1 - \frac{1}{m}\right) = \left(1 - \frac{1}{m}\right)^n$$

$$\Rightarrow P(E_i^c) = \sum_{z=1}^{m} P(\mathbf{h}_i(x) = z) \cdot P(E_i^c \mid \mathbf{h}_i(x) = z) = \left(1 - \frac{1}{m}\right)^n$$

False positive probability – Events

Event E_i holds iff $\mathbf{h}_i(x) \in \{\mathbf{h}_i(x_1), \dots, \mathbf{h}_i(x_n)\}$

Event E_i^c holds iff $\mathbf{h}_i(x) \neq \mathbf{h}_i(x_1)$ and … and $\mathbf{h}_i(x) \neq \mathbf{h}_i(x_n)$

$$P(E_i^c) = \left(1 - \frac{1}{m}\right)^n$$

$$\Rightarrow \quad \text{FPR} = \prod_{i=1}^{k} \left(1 - P(E_i^c)\right) = \left(1 - \left(1 - \frac{1}{m}\right)^n\right)^k$$

False Positivity Rate – Example

$$\text{FPR} = \left(1 - \left(1 - \frac{1}{m}\right)^{n}\right)^{k}$$

e.g., $n = 5{,}000{,}000$

$k = 30$

$m = 2{,}500{,}000$

→ FPR = 1.28%

Comparison with Hash Tables - Space

- Google storing 5 million URLs, each URL 40 bytes.
- Bloom filter with $k = 30$ and $m = 2{,}500{,}000$

Hash Table

(optimistic)

$5{,}000{,}000 \times 40B = 200\text{MB}$

Bloom Filter

$2{,}500{,}000 \times 30 = 75{,}000{,}000$ bits

< 10 MB

Time

- Say avg user visits 102,000 URLs in a year, of which 2,000 are malicious.
- 0.5 seconds to do lookup in the database, 1ms for lookup in Bloom filter.
- Suppose the false positive rate is 3%

$$1\text{ms} + \frac{100000 \times 0.03 \times 500\text{ms} + 2000 \times 500\text{ ms}}{102000} \approx 25.51\text{ms}$$

false positives

0.5 seconds DB lookup

total URLs

malicious URLs

Bloom filter lookup

Bloom Filters typical of….

… randomized algorithms and randomized data structures.

- **Simple**
- **Fast**
- **Efficient**
- **Elegant**
- **Useful!**