THE BLOOM FILTER

3. 布隆过滤器 BloomFilter

3.1 介绍

BloomFilter，布隆过滤器，是一种用来减少 hash table 存储空间的技术。传统的 hash table 会维护一段地址空间，其最小单元暂且命名为 cell，cell 用来存储数据。每个 cell 有一个地址——hash address（通过对插入数据做迭代计算得到）。在插入一个数据 data 时，比如一个网站的登录名 username，首先计算其 hash address，然后将该 username 插入到 hash address 对应的 cell 中。查询的时候，也是先计算出要查询的 username 对应的 hash address，然后到对应的 cell 中对比其内容，如果 cell 中有数据，并且存储的 username 值和要查询的 username 一致，则认为查询到了。

假如现在将 username 替换成 email 地址，思考如下应用场景：过滤垃圾邮件。对于像 gmail 和 hotmail 这样的公共 email 提供商，其垃圾邮件地址可能是上亿，几十亿条。假设存储一条 email 占用空间 8B，那么每存储一亿条 email 到 hash table 中，耗费至少就需要 1.49GB 的内存（用哈希表实现的具体办法是将每一个 email 地址对应成一个八字节的信息指纹，然后将这些信息指纹存入哈希表，由于哈希表的存储效率一般只有 50%，因此存储一亿个地址大约要 1.49GB 内存）[6] 。假设有 40 亿条垃圾邮件地址，按照传统 hash 存储，至少需要 60GB 内存，对于一般的服务器而言，很难将这些数据都存储在内存中。如果将一部分存于磁盘，又会导致查询速度严重降低。

因此需要能够有一种缩小 hash table 空间的方法。Burton Howard Bloom 在 1970 发表的论文 [2] 中，首次提出 BloomFilter。它不同于传统的 hash table，BloomFilter 通过允许一定的 false positive 错误 (即数据并没有被插入到 BloomFilter 中，但是查询的时候，BloomFilter 告诉你查询到了)，相对于传统的 hash table，减少了非常多的存储空间。

Burton H Bloom 在 1963 - 1965 年期间在 MIT 学习 AI 和数值方法。他是 MIT 1966 届的数学系校友，但是并没有取得硕士学位。

BloomFilter 如果使用 6 个 hash function，保证数据插入率 n/m 在 1/9 的前提

下，false positive rate 约为 1.3%，在前文存储 40 亿条垃圾邮件例子中，占用存储空间仅仅为 4.19GB（对于误判的邮件，可以使用一个白名单去过滤，后续会解释这里提到的名词）。

Burton 在他的 BloomFilter 原型中，提出了两种减少存储空间的方法：

1. Method 1：让 cell 里存储由数据 data 产生的一种 code，code 的 size 由 false positive rate 控制：false positive rate 越低，code size 越大，当 false positive rate 低到一定程度，那么 code size 就会等于 data size。
2. Method 2：hash cell 大小退化成为一个 bit，整个 hash table 由是一个 N bit 长的数组。这个以 bit 为单位的数组，每个 bit 初始化为 0。当插入数据 data 到 BloomFilter 中，首先会根据多个 hash 函数，计算出多个 hash address（假设为 a_1，a_2，a_3 ... a_d）；然后将 hash address 对应的 BloomFilter 数组位置都置 1，这样就完成了一个数据 data 的插入。查询的时候，只有当所有之前的 hash address a_1，a_2，a_3 ... a_d 都为 1，才认为查询的数据 data 已经在 BloomFilter 中了。只要有一个不是 1，那么就可以认为数据 data 并不在 BloomFilter 中。（从这里应该可以想象出，数据不在 BloomFilter 的查询结果，总是可信的；但是数据在 BloomFilter 里的查询结果，就可能产生 false positive）。

Burton 的论文以及如今的 BloomFilter，采用的是 Method 2 实现方法。

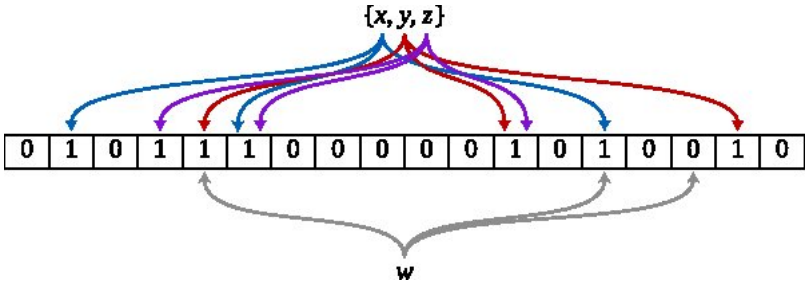

Figure 3.1: BloomFilter 的一个示例

在 Figure 3.1 中，有一个数据集 {x，y，z } 要添加到 BloomFilter 中。可以看到，这里应该有三个 hash 函数。每个数据都对应了三个箭头，对应其映射到 BloomFilter 数组中的位置。当然，不同的数据可能会被 hash 映射到同一个位置，比如 x 的第二个蓝色箭头和 z 的第二个紫色箭头指向数组的同一位置，但这并不影响 BloomFilter 的正常工作 (这里也许会产生疑问，如果删除某个数据，那岂不是会影响到之前插入的数据？确实，基本的 BloomFilter 是不支持删除操作的，后面会介绍支持删除操作的改进 BloomFilter TODO)。 ■

虽然 BloomFilter 减少了空间，但是带来的影响是 false positive。False positive rate 与 BloomFilter 的大小，以及插入数据量都相关，后面我们会继续进行分析。

3.2 数学原理

下面我们来看一下 BloomFilter 涉及的数学原理 [4]。

在 Burton 的论文中，提到了三个计算因子。reject time，space，allowable fraction of errors。第三个参数可以令 BloomFilter 在不增加 reject time 的前提下，减少存

储空间 space。简而言之，就是短小精悍速度快。但是原著的计算分析过程有些复杂，下面将采用另外一种相对简单明了的方式进行原理解释。

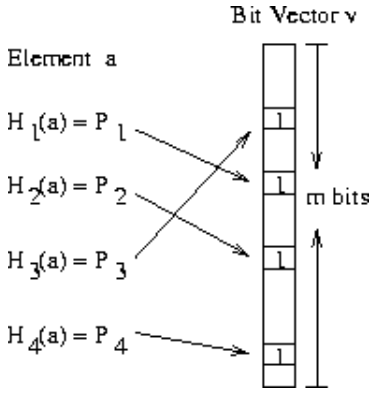

Figure 3.2: BloomFilter 的一个示例

在 Figure 3.2 中，右边是 BloomFilter 的存储空间：一个基本 cell 为 1 个 bit，长度为 m 的数组。BloomFilter 会选择 k 个独立的 hash 函数 $h_1, h_2, ..., h_k$，每个产生的 hash address 的范围是 [1, m]。当插入一个数据（这里用 a 表示），k 个 hash address 被计算出来，$h_1(a), h_2(a), ..., h_k(a)$，这些位置的 bit 位都被置 1。

BloomFilter 的主要特点，就是通过控制 k 和 m 的大小，来控制其 false positive rate。下面来看在给定 k，m 的前提下，false positive rate 是怎么计算出来的。

假设有一个数据集 S，共有 n 个数据要插入到 BloomFilter 中。在插入某一个数据 data 时候，k 个 hash 函数中的某一个函数（h_i，其中 $0 < i \leq k$），考虑某个特定位置的 cell，其不被 h_i 置 1 的概率是 $1 - \frac{1}{m}$（因为被置 1 的概率是 $\frac{1}{m}$，根据这两个事件互斥，可以得到）。那么该位置在这次数据 data 插入的过程中，不被任何 k 个 hash 函数置 1 的概率就是 $(1 - \frac{1}{m})^k$。

所以，在插入 n 个数据以后（当 hash 函数之间相互独立，可以认为这 n 次插入也是独立事件，根据独立事件的概率计算规则），该 bit 位仍然是 0 的概率就是：

对于两个独立事件 A 与 B，那么 P(A|B)=P(A)，P(A|B)=P(B)，P(A∩B)=P(A)P(B)

bit 位为 0 概率.

$$(1 - \frac{1}{m})^{kn} \tag{3.1}$$

极限定理.

$$\lim_{x \to \infty} (1 - \frac{1}{x})^x = \frac{1}{e} \tag{3.2}$$

于是根据公式 3.2，公式 3.1 可以转化为：

约等于.

$$e^{-\frac{kn}{m}} \tag{3.3}$$

那么该 bit 位是 1 的概率就是：

bit 位为 1 概率.

$$1-(1-\frac{1}{m})^{kn} \approx 1-e^{-\frac{kn}{m}} \tag{3.4}$$

现在测试一个不在数据集 S 中的数据 fdata (假设 S 中的数据已经都被添加到 BloomFilter 中)。根据 k 个 hash 函数，会生成 k 个 hash address：$h_1(fdata)$, $h_2(fdata)$, ... , $h_k(fdata)$。那么根据公式 3.4 的概率公式，每个 hash address 对应的 bit 位是 1 的概率都等于公式 3.4 算得的值，那么这 k 个位置都是 1 的概率（产生 false positive 的概率）为：

false positive rate.

$$(1-(1-\frac{1}{m})^{kn})^k \approx (1-e^{-\frac{kn}{m}})^k \tag{3.5}$$

从公式 3.5 来看，hash 函数个数 k，BloomFilter 大小 m，以及已经插入的数据总量 n 都会影响到 false positive rate 值。公式 3.5 右边是简化的结果，之所以这么做是为了方便求偏导，求出极值。

在实际应用中，n 是不可控的，只能估计出 n 的范围（即大概会有多少数据被插入 BloomFilter 中），那么能控制的参数就只有 k 和 m。接下来，看一下这些参数变化对 false positive rate 都有什么样的影响。

首先看 k 的变化，也就是 hash 函数个数选取对 false positive rate 的影响，这里控制 n 和 m 的比例保持恒定。

下面绘制两幅以 k 作为变量的曲线图:

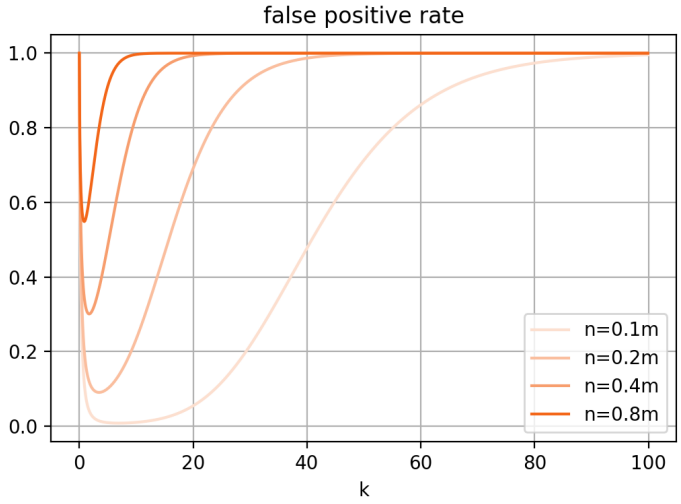

Figure 3.3: 未化简函数相对 k 变化曲线

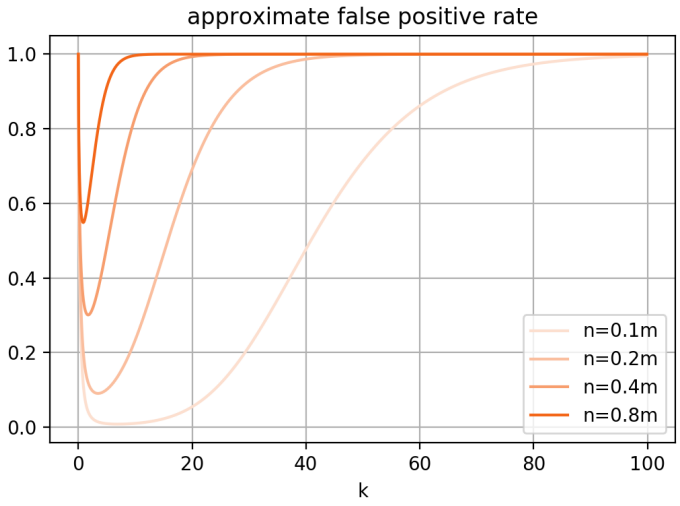

Figure 3.4: 化简函数相对 k 变化曲线

这里 n=0.1m 表示插入的数据总量 n 是 BloomFilter 长度 m 的十分之一，其他以此类堆。可以看出简化和未简化的函数曲线相对 k 值变化几乎相同。这里我们可以发现一个有趣的规律，就是当 k 值从 1 变化到大约 7 左右，false positive rate 急剧减小，之后 false positive rate 反而会随着 k 增大而增长。这个规律可以感官上分析出来：想象一下如果 hash 函数个数 k 很大，那么一次数据的插入就会有很多的 bit 位被置 1，如果 k 过大，会导致 BloomFilter 的 cell 提早被填满（即大部分都置为 1），这样的话就很容易导致 false positive（可以想象如果 BloomFilter 每个 cell 都是 1，那么无论什么样的数据进行查询，都会认为在 BloomFilter 里）。 ■

为了达到最低的 false positive rate，需要对公式 3.5 右边的化简形式求 k 的偏导数，然后令其等于 0 求其极小值 [3]。

对 k 求偏导.

$$f(k) = (1-(1-\frac{1}{m})^{kn})^k \approx (1-e^{-\frac{kn}{m}})^k \quad \text{false positive rate} \tag{3.6}$$

$$a^{\ln b} = b^{\ln a} \tag{3.7}$$

根据性质 3.7，公式 3.6 很容易变成下面的公式：

目标函数形式变换.

$$\begin{aligned} f(k) &= (1-e^{-\frac{kn}{m}})^k \\ &= (1-e^{-\frac{kn}{m}})^{(\ln e)*k} \\ &= (e^{\ln(1-e^{-\frac{kn}{m}})})^k \\ &= e^{k*\ln(1-e^{-\frac{kn}{m}})} \end{aligned} \tag{3.8}$$

因为 e^x 是单调递增函数，所以我们只需要对：

新目标函数.

$$g(k) = k * \ln(1 - e^{-\frac{kn}{m}}) \tag{3.9}$$

求公式 3.9 的最小值，对它求 k 的偏导数即可。

k 的偏导数.

$$\frac{\partial g}{\partial k} = \ln(1 - e^{-\frac{kn}{m}}) + \frac{kn}{m} * \frac{e^{-\frac{kn}{m}}}{1 - e^{-\frac{kn}{m}}} \tag{3.10}$$

式 3.10 等于 0 ，可以解出：

k 最优值.

$$k = (\ln 2)(m/n) \tag{3.11}$$

这很容易将其带入公式 3.10 去验证。

这里还有另一种解法，令 $p = e^{-kn/m}$，可以将公式 3.9 化简成为：

k 最优值.

$$g(p) = -\frac{m}{n}\ln(p)\ln(1-p) \tag{3.12}$$

求该方程最小值更容易求解，在 $p = 0.5$ 时 $g(p)$ 取得最小值:

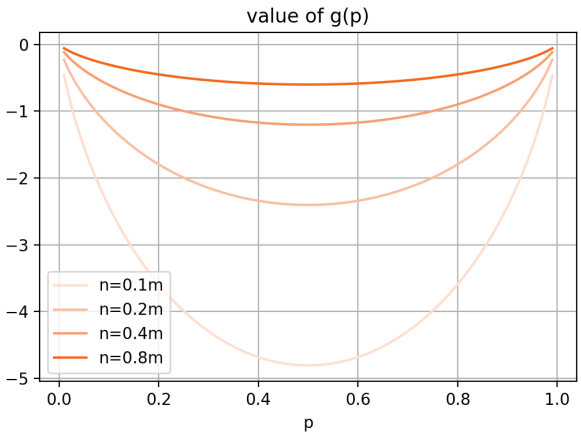

Figure 3.5: g(p) 函数变化曲线

于是我们的到 $p = e^{-kn/m} = 0.5$ 时，g(p) 取最小值，也就是当 $k = (\ln 2)(m/n)$ 的时候，f(k) 取最小值，这时候 false positive rate 最低。将 $e^{-kn/m} = 0.5$ 带入公式 3.6，可以得到：

false positive 最小值.

$$f(k)_{min} = (\frac{1}{2})^k = (\frac{1}{2})^{(\ln 2)(m/n)} \approx 0.6185^{m/n} \tag{3.13}$$

因为 k 必须是整数，所以我们总是选取小于最优解的第一个整数，以减少 hash 函数计算的开销。

m/n	k	k=1	k=2	k=3	k=4	k=5	k=6	k=7
2	1.39	0.393	0.400					
3	2.08	0.283	0.237	0.253				
4	2.77	0.221	0.155	0.147	0.160			
5	3.46	0.181	0.109	0.0918	0.0919	0.101		
6	4.16	0.154	0.080	0.061	0.056	0.058	0.064	
7	4.85	0.133	0.0618	0.0423	0.0359	0.0347	0.0364	
8	5.55	0.118	0.0489	0.0306	0.024	0.0217	0.0216	0.0229
9	6.24	0.105	0.0397	0.0228	0.0166	0.0141	0.0133	0.0135

Table 3.1: false positive rate 表

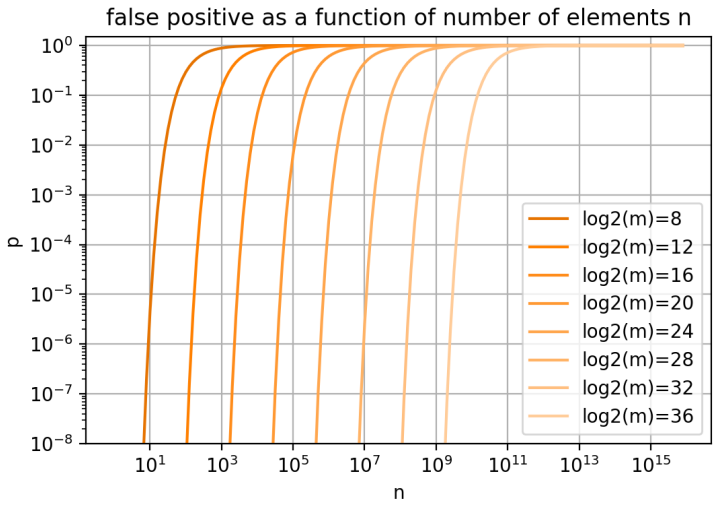

Figure 3.6: false positive 概率变化图

figure 3.6 是以 BloomFilter 的大小 m 以及插入的数据多少 n 为变量变化的，这里假设每个 BloomFilter 都有理想情况下的 $k = (m/n)ln2$ 个 hash 函数。 ■

3.3 其他性质

3.3.1 数据量估计

BloomFilter 中插入的数据总量可以被估计出来 [5]

估计值.

$$n^* = -\frac{m}{k}\ln[1-\frac{X}{m}] \tag{3.14}$$

n^* 是 BloomFilter 中数据量的估计值，m 是 Bloomfilter 的长度，k 是使用的 hash 函数个数，X 则是所有被置 1 的 cell 的个数。

3.3.2 交并计算

BloomFilter 可以看做一个集合的压缩表示形式，可以被用来估计两个集合的交集和并集的元素个数。Swamidass 和 Baldi 在论文 [5] 中提出了计算方法：

A,B 集合数据量.

$$n(A^*) = -\frac{m}{k}\ln[1-\frac{n(A)}{m}] \tag{3.15}$$

$$n(B^*) = -\frac{m}{k}\ln[1-\frac{n(B)}{m}] \tag{3.16}$$

A,B 集合运算.

$$n(A^* \cup B^*) = -\frac{m}{k}\ln\left[1-\frac{n(A\cup B)}{m}\right] \tag{3.17}$$

$$n(A^* \cap B^*) = n(A^*)+n(B^*)-n(A^*\cup B^*) \tag{3.18}$$

除了计算集合运算后的数量，还可以通过对两个同样大小的 BloomFilter 进行 OR 或者 AND 运算来实现并集和交集运算。求并集运算对 BloomFilter 来说是无损的，可以预见对于一个新的集合插入两个 set 中的数据和求 OR 运算得到的结果是相同的。但是，求交集运算则不同：交集得到的结果，其 false positive rate 最差情况下等于两个 set 中的 false positive rate 最大的一个，同时可能会比单独插入两个 set 的交集数据的 false positive rate 要高。

3.4 扩展和应用

3.4.1 Cache filtering

在 CDN 网络中，为了加速用户端访问速度，会把一些数据分布存储在不同的服务器中。因为这些 cache 服务器较原服务器离客户端更近，所以客户端的访问特定内容速度获得提升。那么将哪些数据存储在哪些 cache 服务器中，就需要我们仔细进行考虑。如果将原服务器的所有数据都复制到每一个 cache 服务器中，是能够提供最大程度的加速，但是却会浪费很多存储空间。根据 Akamai Tech, Inc 公司的调查，cache 服务器中有近 3/4 的 url 请求是"one-hit-wonders" 的（即这些 url 只被请求一次）。那么如果把这些 url 的请求数据存储在 cache 服务器中就会很浪费空间，因为不会再有其他访问请求出现。所以为了避免存储"one-hit-wonders" 的数据，这里 BloomFilter 就用来跟踪被用户访问的 url，只有在 BloomFilter 中存在的数据才会被保存到 cache 服务器中用以加速访问。[1]

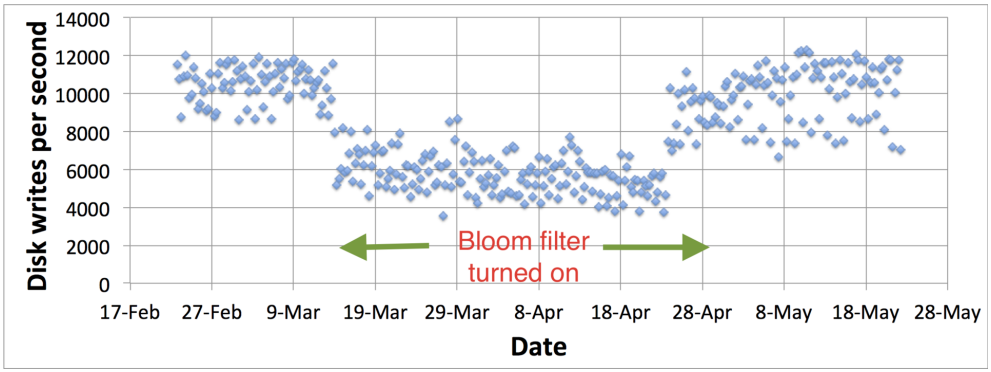

Figure 3.7: BloomFilter 在 cache 服务器上的作用

通过启用 cache 服务器上的 BloomFilter，明显减少了对磁盘的请求（只被访问一次的数据不被 cache）■

3.4.2 Counting filters

Counting filter 提供删除操作。在 BloomFilter 中，一个 cell 占 1 bit。Counting filter 则将 cell 空间变为 x bit。插入操作变为 hash address 对应的 cell 计数加一；查找则是检测对应 hash address 的 cell 是否为 0；删除则对所有对应的 cell 计数减一。

3.4.3 Bloomier filters

TODO

3.4.4 Stable Bloom filters

TODO

3.4.5 Scalable Bloom filters

TODO

绘图源码

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  ################## false positive rate ###################
5  m = 100000
6  n = [m*0.1, m*0.2, m*0.4, m*0.8]
7  k = np.arange(0, 100, 0.1);
8
9  l_color = ['#fcdfcf','#fabe9e', '#f79e6e','#f36618' ]
10 labels = ["n=0.1m", "n=0.2m", "n=0.4m", "n=0.8m"]
11 x = k
12 y = [[]] * 4
13 y2 = y
```

```
for i in range(4):
    y[i] = np.power(1 - np.power( 1 - 1/m, k*n[i]), k)
    y2[i]= np.power(1 - np.exp(-k*n[i]/m), k)
    plt.figure(1)
    plt.plot(x, y[i], l_color[i], label=labels[i])
    plt.figure(2)
    plt.plot(x, y2[i], l_color[i],label=labels[i])

plt.figure(1)
plt.grid()
plt.legend()
plt.title("false positive rate")
plt.xlabel("k")
plt.savefig("false positive rate", dpi=200)

plt.figure(2)
plt.grid()
plt.legend()
plt.xlabel("k")
plt.title("approximate false positive rate")
plt.savefig("approximate false positive rate", dpi=200)

plt.show()

###################### value of g(p) ######################
p = np.arange(0.01,1,0.01)
gp = [[]] * 4
for i in range(4):
    gp[i] = -m/n[i] * np.log(p) *np.log(1-p)
    plt.figure(3)
    plt.plot(p, gp[i], l_color[i], label=labels[i])

plt.grid()
plt.legend()
plt.title("value of g(p)")
plt.xlabel("p")
plt.savefig("value-of-gp", dpi=200)

###################### value of f(n) ######################
l_color = ['#e67300','#ff8000', '#ff8c1a','#ff9933',
'#ffa64d', '#ffb366', '#ffbf80', '#ffcc99' ]
labels = ["log2(m)=8", "log2(m)=12", "log2(m)=16", "log2(m)=20",
"log2(m)=24", "log2(m)=28", "log2(m)=32", "log2(m)=36"]
m = np.power(2,[8,12,16,20,24,28,32,36])
n = np.arange(0,16,0.1)
n = np.power(10,n)

x = []
y = [[]] * 8

for i in range(8):
    ax = plt.subplot(111)
    ax.set_xscale("log", nonposx='clip')
    ax.set_yscale("log", nonposy='clip')
    x = []
    for _ in n:
        k = (m[i]/_)*np.log(2)
        y_ = np.power(1 - np.power( 1 - 1/m[i], k*_), k)
        x = np.append(x,_)
        y[i] = np.append(y[i],y_)
    plt.plot(x, y[i], l_color[i], label = labels[i])
plt.grid()
plt.legend()
plt.ylim((1e-8, 1.5))
plt.title("false positive as a function of number of elements n")
plt.xlabel("n")
plt.ylabel("p")
plt.savefig("value-of-f_n", dpi=200)
```

引用

[1] *Bloom filter*. Dec. 2017. URL: https://en.wikipedia.org/wiki/Bloom_filter (cited on page 26).

[2] Burton H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13.7 (July 1970), pages 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692. URL: http://doi.acm.org/10.1145/362686.362692 (cited on page 19).

[3] Andrei Broder and Michael Mitzenmacher. "Network applications of bloom filters: A survey". In: *Internet mathematics* 1.4 (2004), pages 485–509 (cited on page 23).

[4] Pei Cao. *Bloom Filters - the math*. July 1998. URL: http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html (cited on page 20).

[5] S Joshua Swamidass and Pierre Baldi. "Mathematical correction for fingerprint similarity measures to improve chemical retrieval". In: *Journal of chemical information and modeling* 47.3 (2007), pages 952–964 (cited on pages 25, 26).

[6] 吴军. 数学之美. Volume 5. 人民邮电出版社, 2012 (cited on page 19).