Cardinality estimate in short

基数计数（cardinality counting）是实际应用中一种常见的计算场景，在数据分析、网络监控及数据库优化等领域都有相关需求。精确的基数计数算法由于种种原因，在面对大数据场景时往往力不从心，因此如何在误差可控的情况下对基数进行估计就显得十分重要。目前常见的基数估计算法有Linear Counting、LogLog Counting、HyperLogLog Counting及Adaptive Counting等。这几种算法都是基于概率统计理论所设计的概率算法，它们克服了精确基数计数算法的诸多弊端（如内存需求过大或难以合并等），同时可以通过一定手段将误差控制在所要求的范围内。

作为“解读Cardinality Estimation算法”系列文章的第一部分，本文将首先介绍基数的概念，然后通过一个电商数据分析的例子说明基数如何在具体业务场景中发挥作用以及为什么在大数据面前基数的计算是困难的，在这一部分也同时会详述传统基数计数的解决方案及遇到的难题。

后面在第二部分-第四部分会分别详细介绍Linear Counting、LogLog Counting、HyperLogLog Counting及Adaptive Counting四个算法，会涉及算法的基本思路、概率分析及论文关键部分的解读。

第一部分：基本概念

基数的定义

简单来说，基数（cardinality，也译作势），是指一个集合（这里的集合允许存在重复元素multi-set，与集合论对集合严格的定义略有不同，如不做特殊说明，本文中提到的集合均允许存在重复元素）中不同元素的个数。例如看下面的集合：

{1,2,3,4,5,2,3,9,7}{1,2,3,4,5,2,3,9,7}

这个集合有9个元素，但是2和3各出现了两次，因此不重复的元素为1,2,3,4,5,9,7，所以这个集合的基数是7。

如果两个集合具有相同的基数，我们说这两个集合等势。基数和等势的概念在有限集范畴内比较直观，但是如果扩展到无限集则会比较复杂，一个无限集可能会与其真子集等势（例如整数集和偶数集是等势的）。不过在这个系列文章中，我们仅讨论有限集的情况，关于无限集合基数的讨论，有兴趣的同学可以参考实变分析相关内容。

容易证明，如果一个集合是有限集，则其基数是一个自然数。

基数的应用场景

假设一个淘宝网店在其店铺首页放置了10个宝贝链接，分别从Item01到Item10为这十个链接编号。店主希望可以在一天中随时查看从今天零点开始到目前这十个宝贝链接分别被多少个独立访客点击过。所谓独立访客（Unique Visitor，简称UV）是指有多少个自然人，例如，即使我今天点了五次Item01，我对Item01的UV贡献也是1，而不是5。

用术语说这实际是一个**实时数据流统计**分析问题。

要实现这个统计需求。需要做到如下三点：

1、对独立访客做标识

2、在访客点击链接时记录下链接编号及访客标记

3、对每一个要统计的链接维护一个数据结构和一个当前UV值，当某个链接发生一次点击时，能迅速定位此用户在今天是否已经点过此链接，如果没有则此链接的UV增加1

下面分别介绍三个步骤的实现方案：

- 对独立访客做标识

  > 客观来说，目前还没有能在互联网上准确对一个自然人进行标识的方法，通常采用的是近似方案。例如通过登录用户+cookie跟踪的方式：当某个用户已经登录，则采用会员ID标识；对于未登录用户，则采用跟踪cookie的方式进行标识。为了简单起见，我们假设完全采用跟踪cookie的方式对独立访客进行标识。

- 记录链接编号和访客标记
- 实时UV计算

  > 可以看到，如果将每个链接被点击的日志中访客标识字段看成一个集合，那么此链接当前的UV也就是这个集合的基数，因此UV计算本质上就是一个**基数计数**问题。
  >
  > 在实时计算流中，我们可以认为任何一次链接点击均触发如下逻辑（伪代码描述）：
  >
  > ```
  > cand_counting(item_no, user_id) {
  >     if (user_id is not in the item_no visitor set) {
  >         add user_id to item_no visitor set;
  >         cand[item_no]++;
  >     }
  > }
  > ```
  >
  > 逻辑非常简单，每当有一个点击事件发生，就去相应的链接被访集合中寻找此访客是否已经在里面，如果没有则将此用户标识加入集合，并将此链接的UV加1。

传统的基数基数实现

1. 基于B树的基数计数
2. 基于bitmap的基数计数

   > 一种替代方案是使用bitmap表示集合。也就是使用一个很长的bit数组表示集合，将bit位顺序编号，bit为1表示此编号在集合中，为0表示不在集合中。例如“00100110”表示集合 {2，5，6}。bitmap中1的数量就是这个集合的基数。
   >
   > 与B树不同bitmap可以高效的进行合并，只需进行**按位或（or）**运算就可以，而位运算在计算机中的运算效率是很高的。但是bitmap方式也有自己的问题，就是内存使用问题。
   >
   > 很容易发现，bitmap的长度与集合中元素个数无关，而是与基数的上限有关。例如在上面的例子中，假如要计算上限为1亿的基数，则需要12.5M字节的bitmap，十个链接就需要125M。关键在于，这个内存使用与集合元素数量无关，即使一个链接仅仅有一个1UV，也要为其分配12.5M字节。
   >
   > 由此可见，虽然bitmap方式易于合并，却由于内存使用问题而无法广泛用于大数据场景。

第二部分：Linear Counting

简介

Linear Counting（以下简称LC）在1990年的一篇论文“A linear-time probabilistic counting algorithm for database applications”中被提出。作为一个早期的基数估计算法，LC在空间复杂度方面并不算优秀，实际上LC的空间复杂度与上文中简单bitmap方法是一样的（但是有个常数项级别的降低），都是$O(N_{max})$，因此目前很少单独使用LC

基本算法

LC的基本思路是：设有一哈希函数H，其哈希结果空间有m个值（最小值0，最大值m-1），并且哈希结果服从均匀分布。使用一个长度为m的bitmap，每个bit为一个桶，均初始化为0，设一个集合的基数为n，此集合所有元素通过H哈希到bitmap中，如果某一个元素被哈希到第k个比特并且第k个比特为0，则将其置为1。当集合所有元素哈希完成后，设bitmap中还有u个bit为0。则：

$\hat{n} = -mlog\frac{u}{m}$，为n的一个估计，且为最大似然估计

1. 推导证明

> 已知n个元素的哈希值服从独立均匀分布。设A_j为事件“经过n个不同元素的哈希操作后，第j个桶的值为0”，则：
>
> $P(A_j) = (1 - \frac{1}{m})^n$
>
> 又每个桶式独立的，则u的期望为：
>
> $E(u) = \sum_{j=1}^{m} P(A_j) = m(1 - \frac{1}{m})^n = m((1 + \frac{1}{-m})^{-m})^{-\frac{n}{m}}$
>
> 当n和m趋于无穷大，其值约为$me^{-\frac{n}{m}}$，令$E(u) = me^{-\frac{n}{m}}$
>
> 则：$n = -mlog\frac{E(u)}{m}$
>
> 显然每个桶的值服从参数相同0-1分布，因此u服从二项分布。由概率论知识可知，当n很大时，可以用正态分布逼近二项分布，因此可以认为当n和m趋于无穷大时u渐进服从**正态分布**。正态分布的期望的最大似然估计是样本均值。
>
> 有如下定理：
>
> 设$f(x)$是可逆函数，$\hat{x}$是x的最大似然估计，则$f(\hat{x})$是$f(x)$的最大似然估计
>
> 且$-mlog\frac{x}{m}$是可逆函数，则$\hat{n} = -mlog\frac{u}{m}$是$-mlog\frac{E(u)}{m} = n$的最大似然估计

2. 误差分析与控制

> 暂不讨论

小节

> LC算法虽然由于空间复杂度$O(N_{max})$不够理想已经很少被单独使用，但是由于其在元素数量较少时表现非常优秀，因此常被用于弥补LogLog Counting在元素较少时误差较大的缺陷，实际上LC及其思想是组成HyperLogLog Counting和Adaptive Counting的一部分。

第三部分：LogLog Counting[1]

上一节介绍的Linear Counting算法相较于直接映射bitmap的方法能大大节省内存（大约只需后者1/10的内存），但毕竟只是一个常系数级的降低，空间复杂度仍然为$O(N_{max})$。而本文要介绍的LogLog Counting却只有$O(log_2(log_2(N_{max})))$。例如，假设基数的上限为1亿，原始bitmap方法需要12.5M内存，而LogLog Counting只需不到1K内存（640字节）就可以在标准误差不超过4%的精度下对基数进行估计，效果可谓十分惊人。

简介

LogLog Counting（以下简称LLC）出自论文“Loglog Counting of Large Cardinalities”。LLC的空间复杂度仅有$O(log_2(log_2(N_{max})))$，使得通过KB级内存估计数亿级别的基数成为可能，因此目前在处理大数据的基数计算问题时，所采用算法基本为LLC或其几个变种。下面来具体看一下这个算法。

基本算法

1. 均匀随机化

> 与LC一样，在使用LLC之前需要选取一个哈希函数H应用于所有元素，然后对哈希值进行基数估计。H必须**满足如下条件**（定性的）：
>
> 1. H的结果具有很好的均匀性，也就是说无论原始集合元素的值分布如何，其哈希结果的值几乎服从均匀分布（完全服从均匀分布是不可能的，D. Knuth已经证明不可能通过一个哈希函数将一组不服从均匀分布的数据映射为绝对均匀分布，但是很多哈希函数可以生成几乎服从均匀分布的结果，这里我们忽略这种理论上的差异，认为哈希结果就是服从均匀分布）。
> 2. H的碰撞几乎可以忽略不计。也就是说我们认为对于不同的原始值，其哈希结果相同的概率非常小以至于可以忽略不计。
> 3. H的哈希结果是固定长度的。
>
> 即哈希后的结果是服从均匀分布的数据

2. 思想来源

> 设a为待估集合（哈希后）中的一个元素，由上面对H的定义可知，a可以看做一个长度固定的比特串（也就是a的二进制表示），设H哈希后的结果长度为L比特，我们将这L个比特位从左到右分别编号为1、2、…、L：
>
> 
>
>
> 其中a是服从均匀分布的样本空间中随机抽取的一个样本，a的每个比特位服从如下独立分布：
>
> $$P(x=k)=\begin{cases}0.5 & (k=0)\\ 0.5 & (k=1)\end{cases}$$
>
> 设$\rho(a)$为a的比特串中第一个“1”出现的位置，显然$1\le\rho(a)\le L$，遍历集合中所有元素的取值，令ρ_{max}为所有$\rho(a)$的最大值。
>
> 则，我们可以使用$2^{\rho_{max}}$作为基数的一个粗糙估计，即$\hat{n}=2^{\rho_{max}}$

3. 推导[2]

> 一次伯努利过程：由于比特串每个比特都独立且服从0-1分布，因此从左到右扫描上述某个比特串寻找第一个“1”的过程从统计学角度看是一个伯努利过程，例如，可以等价看作不断投掷一个硬币（每次投掷正反面概率皆为0.5），直到得到一个正面的过程。在一次这样的过程中，投掷一次就得到正面的概率为$\frac{1}{2}$，投掷两次得到正面的概率是$\frac{1}{2^2}$，…，投掷k次才得到第一个正面的概率为$\frac{1}{2^k}$
>
> 注意：这里的k代表最大的ρ_{max}
>
> Question1：抛n次硬币，所有抛掷次数不大于k的概率P1是多少？

> 所有抛掷次数不大于k的概率=一次抛掷不大于k的概率^n $=(1-\frac{1}{2^k})^n$
>
> 一次抛掷次数不大于k的概率=1-一次抛掷次数大于k的概率=$1-\frac{1}{2^k}$;即前k次均为反面

Question2：抛n次硬币，至少有一次抛掷次数等于k的概率P2是多少？

> 至少一次抛掷次数等于l的概率=1-N次抛掷次数都不等于k的概率=$1-(1-\frac{1}{2^{k-1}})^n$
>
> 一次抛掷次数不等于k的概率=$(1-\frac{1}{2^{k-1}})$

分析：

$$P\begin{cases} n \ll 2^k & P2 \to 0 \\ n \gg 2^k & P1 \to 0 \end{cases}$$

自然语言解释：

> 当试验次数远远小于2^k 时，至少有一次抛掷次数等于k的事件概率为0
>
> 当试验次数远远大于2^k 时，所有抛掷次数不大于k的事件概率为0
>
> 设一个集合的基数为n，ρ_{max}为所有元素中首个“1”的位置最大的那个元素的“1”的位置，如果n远远小于$2^{\rho_{max}}$，则我们得到ρ_{max}为当前值的概率几乎为0（它应该更小），同样的，如果n远远大于$2^{\rho_{max}}$，则我们得到ρ_{max}为当前值的概率也几乎为0（它应该更大），因此$2^{\rho_{max}}$可以作为基数n的一个粗糙估计。

4. 分桶平均

> 上述分析给出了LLC的基本思想，不过如果直接使用上面的单一估计量进行基数估计会由于偶然性而存在较大误差。因此，LLC采用了**分桶平均**的思想来消减误差。具体来说，就是将哈希空间平均分成m份，每份称之为一个桶（bucket）。对于每一个元素，其哈希值的**前k**比特作为桶编号，其中$m=2^k$，而后**L-k个**比特作为真正用于基数估计的比特串。桶编号相同的元素被分配到同一个桶，在进行基数估计时，首先计算每个桶内元素最大的第一个“1”的位置，设为M[i]，然后对这m个值取平均后再进行估计，即：

$$\hat{n}=2^{\frac{1}{m}\sum M[i]}$$

> 这相当于物理试验中经常使用的多次试验取平均的做法，可以有效消减因偶然性带来的误差。(HLLC使用**调和平均数**代替平均数更好的进行粗糙估计)
>
> 一个小例子：
>
> > 假设H的哈希长度为16bit，分桶数m定为32。设一个元素哈希值的比特串为“0001001010001010”，由于m为32，因此前5个bit为桶编号，所以这个元素应该归入“00010”即2号桶（桶编号从0开始，最大编号为m-1），而剩下部分是“01010001010”且显然ρ(01010001010)=2，所以桶编号为“00010”的元素最大的ρ 即为M[2]的值。

5. 误差分析与修正

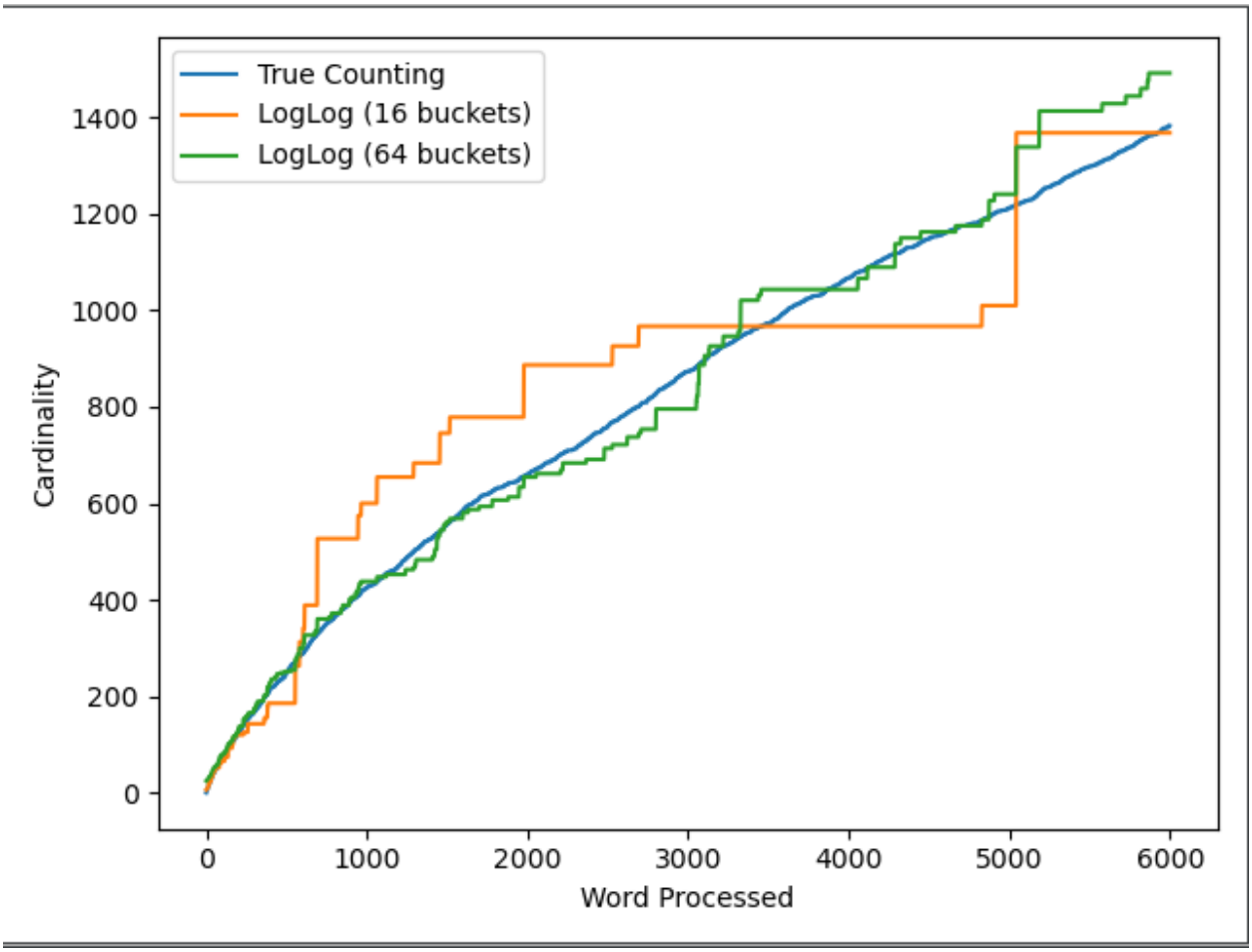
> 暂未讨论
>
> 空间复杂度为啥是$loglog(N_{max})$
>
> $StdError(\hat{n}/n)\approx\frac{1.30}{\sqrt{m}}$

代码示例[3]

```
import hashlib
import zlib
import nltk     # Natural Language Toolkit用于引入word数据
from collections import Counter     # find different elements
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
"""
Hashing Function and Dataset
The next algorithms/data structures make use of hashing functions,
so before diving into them lets define the hashing functions that will be used.
I note that some algorithms/data structures make assumptions about these hash functions,
but none of them were tested.
Futhermore, I constrain every hashing to have 24 bits.
"""

def hash_CRC32(s):
    return zlib.crc32(s) & 0xffffff

def hash_Adler32(s):
    return zlib.adler32(s) & 0xffffff

def hash_MD5(s):
    return int(hashlib.md5(s).hexdigest(), 16) & 0xffffffff

def hash_SHA(s):
    return int(hashlib.sha1(s).hexdigest(), 16) & 0xffffffff


# note that this implementation is returning 2 to power of the first 1-bit
def least1(x, L):
    if x == 0:
        return 2**L
    return x & -x

def index_least1(x):
    if x == 0:
        return 0
    index = 1
    while (x & 1) == 0:
        x >>= 1
        index += 1
    return index
    # index = 1
    # while x % 2 == 0:
    #     x >>= 1
    #     index += 1
    # return index

def cardinality_LogLog(buckets):
    buckets = [index_least1(x) for x in buckets]
    return 0.39701 * len(buckets) * 2 ** (np.mean(buckets))

if __name__ == '__main__':

    words = nltk.corpus.gutenberg.words('austen-persuasion.txt')
    words = [x.lower().encode('utf-8') for x in words]  # len:98171 type:bytes
```

```
    subset = words[0:6000]
    s = set([])
    df = pd.DataFrame(data=np.nan, index=range(0, 3 * len(subset)), columns=['f', 'x', 'count'])
    buckets16 = np.array([0] * 16)
    buckets64 = np.array([0] * 64)

    for idx, w in enumerate(subset):
        s.add(w)        # set自动去重，len获得UV
        hashed = hash_SHA(w)
        print(idx)

        buckets16[hashed % 16] = max(buckets16[hashed % 16], least1(hashed >> 4, 24))
        buckets64[hashed % 64] = max(buckets64[hashed % 64], least1(hashed >> 6, 24))

        df.loc[idx * 3] = ['True Counting', idx, len(s)]
        df.loc[idx * 3 + 1] = ['LogLog (16 buckets)', idx, cardinality_LogLog(buckets16)]
        df.loc[idx * 3 + 2] = ['LogLog (64 buckets)', idx, cardinality_LogLog(buckets64)]


    # 分组为了画图
    groups = df.groupby("f")
    df_count = groups.get_group("True Counting")
    df_16 = groups.get_group("LogLog (16 buckets)")
    df_64 = groups.get_group("LogLog (64 buckets)")

    df_plot = [df_count, df_16, df_64]
    for i in range(3):
        plt.plot(df_plot[i]["x"], df_plot[i]["count"], label=df_plot[i]["f"].values[0])

    plt.xlabel("Word Processed")
    plt.ylabel("Cardinality")
    plt.legend(loc='best')
    plt.show()
```

1. 实验结果：



小节

内存使用与m的大小及哈希值得长度（或说基数上限）有关。假设H的值为32bit，则$\rho_{max} \leq 32$，因此每个桶需要5bit空间存储这个桶的ρ_{max}，m个桶就是$\frac{5 \cdot m}{8}$字节。例如基数上限为一亿（约2^{27}），当分桶数m为1024时，每个桶的基数上限约为$2^{27}/2^{10} = 2^{17}$，而$loglog(2^{17}) = 4.09$，因此每个桶需要5bit，需要字节数就是5×1024/8=640字节，误差为$1.30/\sqrt{1024} = 0.040625$，也就是约为4%

在合并操作，可以使用桶编号相同位置较大的元素代替该桶的ρ_{max}

本文主要介绍了LogLog Counting算法，相比LC其最大的优势就是**内存使用极少**。不过LLC也有自己的问题，就是当n不是特别大时，其估计误差过大，因此目前实际使用的基数估计算法都是基于LLC改进的算法，这些改进算法通过一定手段抑制原始LLC在n较小时偏差过大的问题。后面要介绍的HyperLogLog Counting和Adaptive Counting就是这类改进算法。

第四部分：HyperLogLog Counting

基本算法

HLLC的算法和LLC类似，它使用了调和平均数代替几何平均数

LLC对各个桶取算数平均数，算数平均数最终被应用到2的指数上，总体看LLC取得是几何平均数。但是几何平均数对离群值（如0）特别敏感，因此当存在离群值时，LLC的偏差就会很大。当n较小时，可能有较多的空桶，这些离群值强烈干扰了几何平均数的稳定性，LLC在n较小时，表现不好。

调和平均数的定义：

$$H = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n}} = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$$

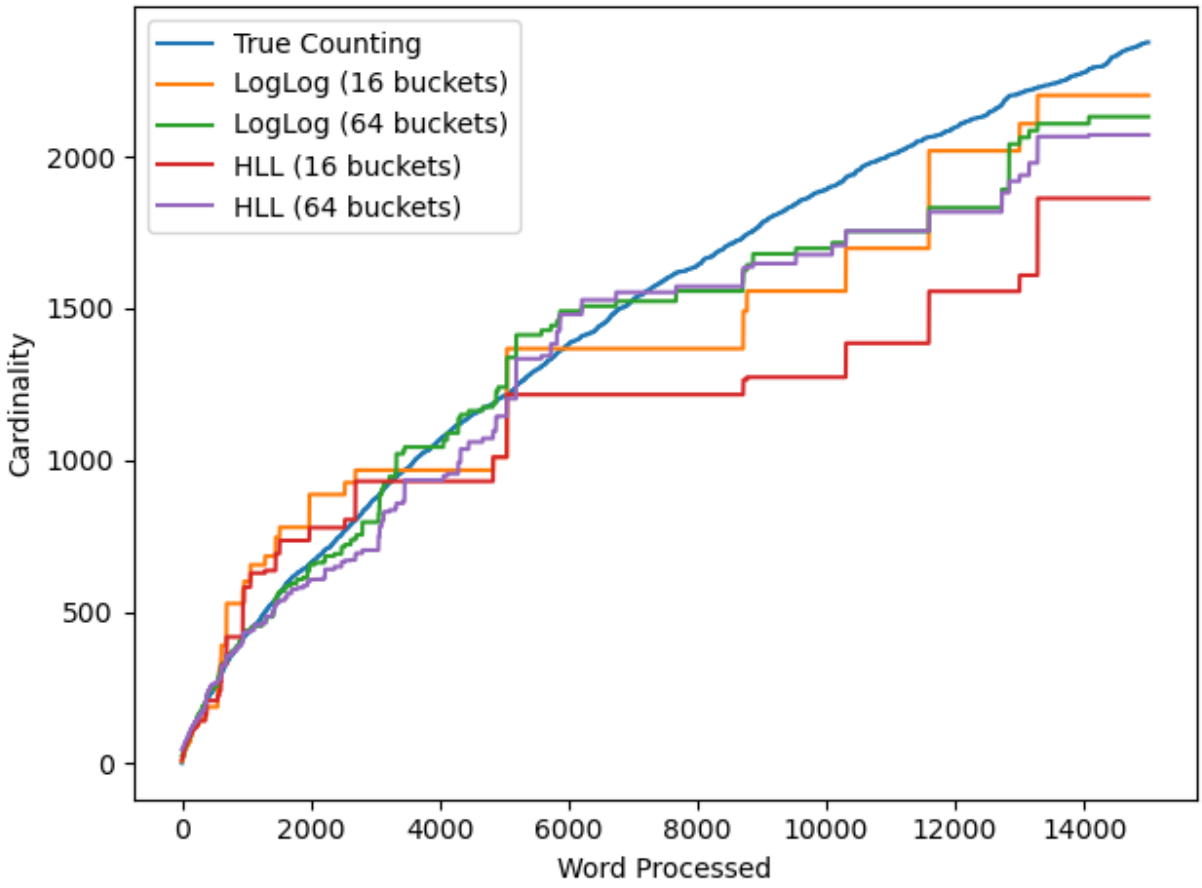
HLLC的基数估计：

$$\hat{n} = \frac{\alpha_m \cdot m^2}{\sum 2^{-M}}$$

1. 偏差分析

$$SE_{HLLC}(\frac{\hat{n}}{n}) = 1.04/\sqrt{m}$$

2. 实验结果



参考链接

1. CodingLabs - 解读Cardinality Estimation算法（第一部分：基本概念）

2. CodingLabs - 解读Cardinality Estimation算法（第二部分：Linear Counting）

3. CodingLabs - 解读Cardinality Estimation算法（第三部分：LogLog Counting）

4. CodingLabs - 解读Cardinality Estimation算法（第四部分：HyperLogLog Counting及Adaptive Counting）

5. CodingLabs - 五种常用基数估计算法效果实验及实践建议

6. CodingLabs - 基数估计算法概览

1. LNCS 2832 - Loglog Counting of Large Cardinalities (inria.fr) ↩

2. 基数估计算法辅助推导 - 知乎 (zhihu.com) ↩

3. code_近似计数、Flajolet-Martin、LogLog、HyperLogLog、Bloom Filters (github.com) ↩