Applied Algorithms Assignment 6

Fei Fan (Peter) Chen

February 17, 2021

Exercise 1 (10 pts)

Consider at time n, you have your median value m at position $n/2$. In order to find the median at time $n+1$, you need to know the numbers at position $n/2-1$ and $n/2+1$. Hence, by induction, you need to know all n numbers at time n to be able to calculate the median at any time $t > n$.

Exercise 2 (10 pts)

a) Using Hoeffding's Inequality $Prob[S_n \geq (p+\varepsilon)n] \leq e^{-2\varepsilon^2 n}$ with $S_n = HT[h(x)] - f_x$, $p = \frac{1}{b}$ and $\varepsilon = \frac{1}{b}$:

$$Prob[HT[h(x)] - f_x \geq pn + \varepsilon n] = Prob[HT[h(x)] - f_x \geq \frac{2n}{b}] \leq e^{-\frac{2n}{b^2}}$$

b) Given that we only have 4 distinct values occuring uniformly, our probability distribution is also discrete. In fact, for $HT[h(x)] > f_x + \frac{2n}{b}$, none of the other distinct values can hash to the bucket that x hashes to (for all $n > 8$). Given that probability that another distinct value hashing to x is $\frac{b-1}{b}$, then the probability of not exceeding the actual count by $\frac{2n}{b}$ is:

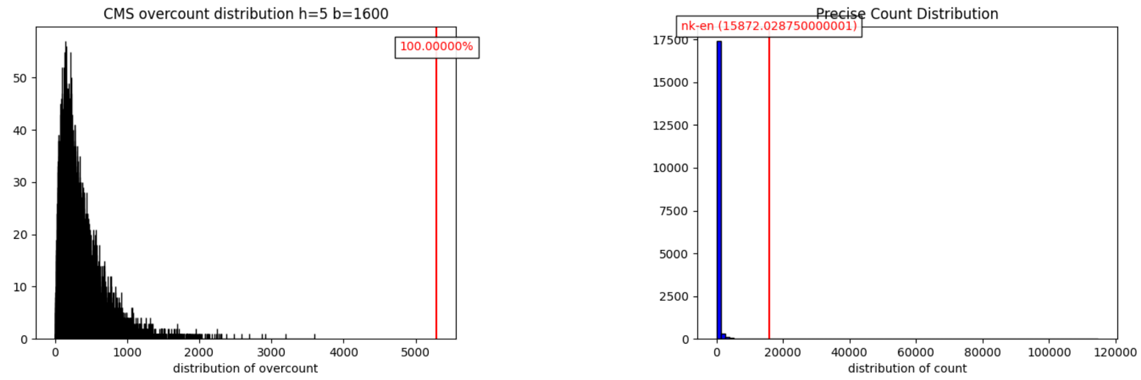
$$Prob[HT[h(x)] - f_x \geq \frac{2n}{b}] = 1 - (\frac{b-1}{b})^3$$

Exercise 3 (15 pts)

See code section for code. To see distribution of over counts and precise counts, see 1a and 1b.

The top cities with $counts > n/k$ where $k = 200$ are (precise, estimate, over count with $b = 1600$ and $h = 10$):

- Houston_TX: 114815 115616 801
- Los Angeles_CA 92701 92986 285
- Charlotte_NC: 88719 88846 127
- Dallas_TX: 76997 77065 68

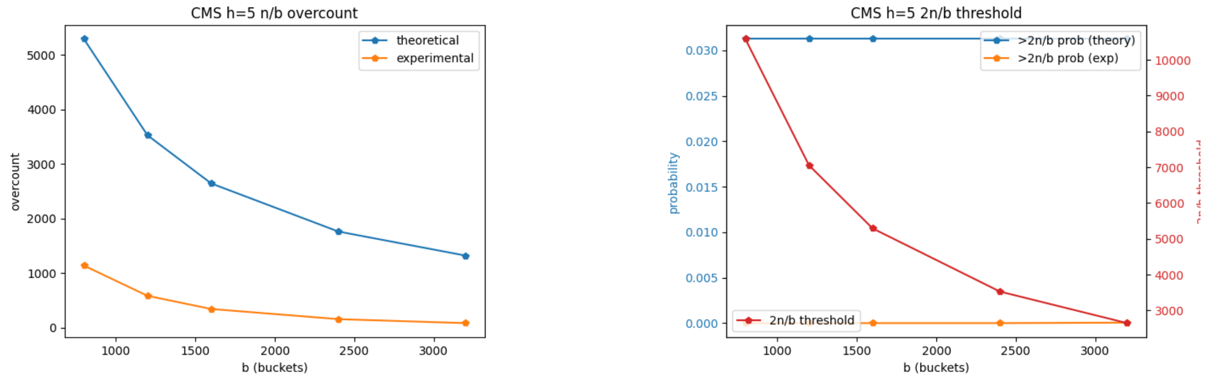


(a) Over Count

(b) Precise-count

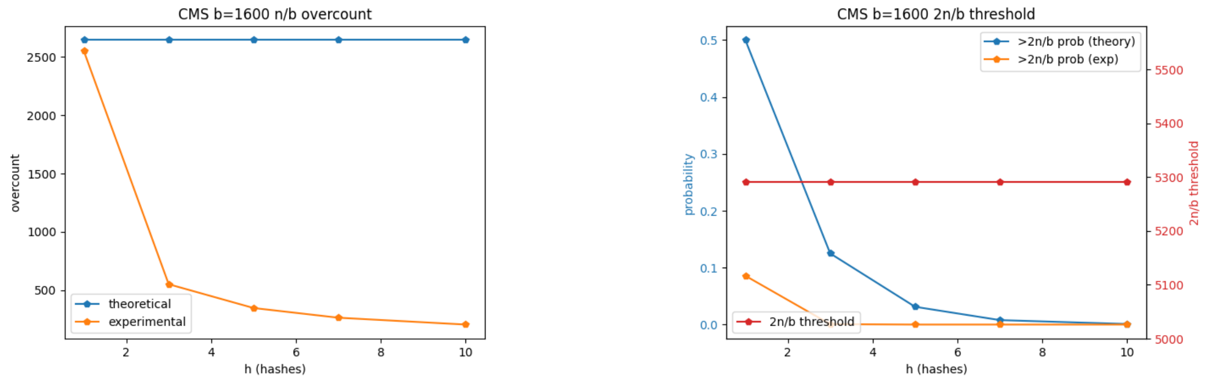
Figure 1: Distribution

- Austin_TX: 70250 70713 463
- Miami_FL: 63085 63136 51
- Raleigh_NC: 52871 53250 379
- Atlanta_GA: 46309 46458 149
- Baton Rouge_LA: 42814 43028 214
- Nashville_TN: 41767 42491 724
- Orlando_FL: 39552 40455 903
- Oklahoma City_OK: 39484 39574 90
- Sacramento_CA: 38061 38254 193
- Phoenix_AZ: 32597 32670 73
- Minneapolis_MN: 31781 31928 147
- San Diego_CA: 29416 29642 226
- Seattle_WA: 28004 28964 960
- San Antonio_TX: 27154 27364 210
- Saint Paul_MN: 23722 24256 534
- Jacksonville_FL: 23658 23833 175
- Richmond_VA: 23460 23883 423
- Portland_OR: 23349 23697 348
- San Jose_CA: 22953 23139 186
- Indianapolis_IN: 22479 23101 622
- Greenville_SC: 21664 21909 245



(a) Over-count (b) 2n/b prob/threshold

Figure 2: Sweep on bucket size.



(a) Over-count (b) 2n/b prob/threshold

Figure 3: Sweep on hash size.

Exercise 4 (15 pts)

There are 4232541 data records in the data set, so for $\varepsilon = \frac{1}{800}$ and $k = 200$, we want do not want to see any items that occur less than $n/k - \varepsilon n = \frac{4232541}{200} - \frac{4232541}{800} \approx 15872$ times. It seems the behavior is similar to theory which is an upper bound.

Code

```
#!/usr/bin/python
import math
import time
import random
import statistics
import matplotlib.pyplot as plt

# Carter-Wegman universal hash functions
#
class Hasher():
  """
  Carter-Wegman universal hash functions. For any x, it hashes x to ax+b
  where p is a prime, and a and b are integers in range [1...p-1] randoml
  when hash function is initiated.
  """
  def __init__(self, buckets: int):
    self._p = 2147483629  # prime less than 2^31
    self._p1 = 4294967291 # prime less than 2^32
    self._p2 = 65521      # prime less than 2^16

    self._a = int(random.uniform(1, self._p))
    self._b = int(random.uniform(1, self._p))
    self._buckets = buckets

  def __call__(self, s: str):
    """
    strings are converted to integers by treating the characters as coeff
    a polynomial, which is then evaluated at a fixed value. This arithmet
    again done mod a different prime p'.
    """
    val = 0
    x = 1
    for i in range(len(s)):
      c = ord(s[i])
      val = (val + c * x) % self._p1
      x = (x * self._p2) % self._p1

    result = ( val * self._a + self._b ) % self._p
    return result % self._buckets

class FakeHasher():
  """
  Simulated hash that randomly spreads results for each object and rememl
  it last stored them.
  """
```

```python
  def __init__(self, buckets: int):
    self._index = {} # remember where something was assigned
    self._buckets = buckets

  def __call__(self, s: str):
    if s not in self._index:
      self._index[s] = int(random.uniform(0, self._buckets))
    return self._index[s]


class CountMinSketch():
  def __init__(self, k: int, buckets: int , hashes: int):
    self.b = buckets
    self.epsilon = 1 / buckets
    self.h = [ Hasher(buckets) for _ in range(hashes) ]
    self.k = k # identify objects occuring n/k times
    self.count_cms = [[0] * buckets] * hashes # Count-Min-Sketch table
    self.count_precise = {} # Precise count table
    self.n = 0 # total number of datapoints seen in the stream

  def process(self, data: str):
    """
    process data to add to Count-Min-Sketch and keep an actual count
    for comparison
    """
    # add to precise count
    if data not in self.count_precise:
      self.count_precise[data] = 0
    self.count_precise[data] += 1

    # add to min-sketch count
    for i in range(len(self.h)):
      pos = self.h[i](data)
      self.count_cms[i][pos] += 1

    self.n += 1

  def cms_count(self, data: str):
    """
    return the precise and cms count of the data item
    """
    minCount = self.n
    for i in range(len(self.h)):
      pos = self.h[i](data)
      if self.count_cms[i][pos] < minCount:
        minCount = self.count_cms[i][pos]

    return minCount
```

```python
def analyze(self):
  """
  Analyze error rate of all items that appear > n/k times, we look
  at the overcount for these items (bucket them into histograms and
  average overcount).
  NOTE: this overcount average is not weighted by occurence of each
  item.

  Given theoretical analysis, we expect the count to exceed 2*n/b less
  than 50% of time, with expected overcount of n/b.

  Precision: true positives / ( true positives + false positives ) wher
  positive is a number cms found to have occured n/k times and actually
  did occur n/k times

  Recall: true positives / ( true positives + false negatives )
  """
  overcount = []
  true_positives = 0
  false_positives = 0
  false_negatives = 0
  overcnt_threshold = 0

  threshold = self.n / self.k
  pct_50_threshold = 2 * self.n / self.b
  for data, precise_count in self.count_precise.items():
    cms_count = self.cms_count(data)
    overcount.append( cms_count - precise_count )
    #if cms_count - precise_count > pct_50_threshold and precise_count
    if cms_count - precise_count > pct_50_threshold:
      overcnt_threshold += 1

    if precise_count >= threshold and cms_count >= threshold:
      true_positives += 1
    elif precise_count >= threshold and cms_count < threshold:
      false_negatives += 1
    elif precise_count < threshold - pct_50_threshold and cms_count > t
      # we define false positives to at n/k - epsilon*n
      false_positives += 1

  precision = true_positives / (true_positives + false_positives)
  recall = true_positives / (true_positives + false_negatives)

  pct_50_prob_theoretical = 0.5 ** len(self.h) #upper bound
  pct_50_prob_experimental = overcnt_threshold / len(self.count_precise

  overcount_expected = self.n / self.b
```

```
    overcount_experimental = statistics.mean(overcount)

    return {
      "overcnt_dist": overcount,
      "precision": precision,
      "recall": recall,
      "pct_50_threshold": pct_50_threshold,
      "pct_50_prob_theoretical": pct_50_prob_theoretical,
      "pct_50_prob_experimental": pct_50_prob_experimental,
      "overcnt_theoretical": overcount_expected,
      "overcnt_experimental": overcount_experimental,
    }

class Graph():
  def __init__(self, x, xlabel, title):
    self.precision = []
    self.recall = []
    self.pct_50_threshold = []
    self.pct_50_prob_theoretical = []
    self.pct_50_prob_experimental = []
    self.overcnt_theoretical = []
    self.overcnt_experimental = []
    self.x = x
    self.xlabel = xlabel
    self.title = title

  def append_result(self, result):
    self.precision.append(result["precision"])
    self.recall.append(result["recall"])
    self.pct_50_threshold.append(result["pct_50_threshold"])
    self.pct_50_prob_theoretical.append(result["pct_50_prob_theoretical"]
    self.pct_50_prob_experimental.append(result["pct_50_prob_experimental
    self.overcnt_theoretical.append(result["overcnt_theoretical"])
    self.overcnt_experimental.append(result["overcnt_experimental"])

  def make_graph(self):
    plt.figure()
    plt.plot(self.x, self.precision, "p-", label="precision")
    plt.plot(self.x, self.recall, "p-", label="recall")
    plt.xlabel(self.xlabel)
    plt.title( f"CMS {self.title} precision/recall" )
    plt.legend()
    plt.figure()
    plt.title( f"CMS {self.title} 2n/b threshold" )
    ax = plt.gca()
    ax.set_xlabel(self.xlabel)
    ax.set_ylabel("probability", color='tab:blue')
    ax.plot(self.x, self.pct_50_prob_theoretical, "p-", label=">2n/b prob
```

```
    ax.plot(self.x, self.pct_50_prob_experimental, "p-", label=">2n/b pro
    ax.tick_params(axis='y', labelcolor="tab:blue")
    ax.legend(loc=1)
    ax2 = ax.twinx()
    ax2.set_ylabel("2n/b threshold", color='tab:red')
    ax2.plot(self.x, self.pct_50_threshold, "p-", color="tab:red", label=
    ax2.tick_params(axis='y', labelcolor="tab:red")
    ax2.legend(loc=3)
    plt.figure()
    plt.plot(self.x, self.overcnt_theoretical, "p-", label="theoretical")
    plt.plot(self.x, self.overcnt_experimental, "p-", label="experimental
    plt.xlabel(self.xlabel)
    plt.ylabel("overcount")
    plt.title( f"CMS {self.title} n/b overcount" )
    plt.legend()

if __name__ == "__main__":
  random.seed()
  k = 200 # Top 200 cities with reported accidents
  buckets = [400, 800, 1200, 1600, 3200, 6400] # bucket size
  #buckets = [400, 1600]
  buckets_constant = 1600
  hashes = [1,3,5,7,10,13]   # number of hashes
  #hashes = [1,5]
  hashes_constant = 5

  # run over buckets
  bucket_graph = Graph(buckets, "b (buckets)", f"h={hashes_constant}")
  for b in buckets:
    cms = CountMinSketch(k=k, buckets=b, hashes=hashes_constant)
    st = time.time()
    # Data is a list of city/state where incidient occured
    for line in open('testdata.txt'):
      cms.process(line)

    result = cms.analyze()
    bucket_graph.append_result(result)
    print(f"buckets: {b}, hashes: {hashes_constant}, k: {k}")
    print("runtime", time.time() - st)

  bucket_graph.make_graph()

  # run over hashes
  hashes_graph = Graph(hashes, "h (hashes)", f"b={buckets_constant}")
  for h in hashes:
    cms = CountMinSketch(k=k, buckets=buckets_constant, hashes=h)
    st = time.time()
    # Data is a list of city/state where incidient occured
```

```
  for line in open('testdata.txt'):
    cms.process(line)

  result = cms.analyze()
  hashes_graph.append_result(result)
  print(f"buckets: {buckets_constant}, hashes: {h}, k: {k}")
  print("runtime", time.time() - st)

hashes_graph.make_graph()

# generate overcnt distribution at b=800 and h=5
print("Generate overcnt distribution")
st = time.time()
cms = CountMinSketch(k=k, buckets=buckets_constant, hashes=hashes_const
for line in open('testdata.txt'):
  cms.process(line)
result = cms.analyze()
print(result)
print(time.time() - st)
plt.figure()
ax = plt.gca()
plt.hist(result["overcnt_dist"], color='blue', edgecolor='black',
  bins=len(result["overcnt_dist"]))
plt.xlabel( "overcount" )
plt.ylabel( "distribution" )
plt.title( f"CMS overcount distribution h={hashes_constant} b={buckets_
plt.axvline(result["pct_50_threshold"], color="red")
threshold_pct= 1 - result["pct_50_prob_experimental"]
plt.text(result["pct_50_threshold"], ax.get_ylim()[1]-4, f"{threshold_p
  horizontalalignment='center', verticalalignment='center', color="red"
  bbox=dict(facecolor='white', alpha=0.9))
plt.show()
```