Assignment 3

TA email: ji_zhe@sjtu.edu.cn

Concept questions

1. In the Bloom filter, given the size m of the key set and the length n of the bit string as parameters, derive the optimal value of k, the number of independent hash functions. (20 points)
2. Compute the surprise number (second moment) for the stream 3, 1, 4, 1, 3, 4, 2, 1, 2. What is the third moment of this stream? (10 points)
3. Suppose we have a dataset of customer transactions, each containing the customer's ID, the item purchased, the date of purchase, and the purchase price.

   We want to answer certain queries approximately from a 1/20th sample of the data. For each of the queries below, indicate how you would construct the sample. That is, tell what the key attributes should be.

   (a) For each item, estimate the average purchase price.

   (b) Estimate the fraction of customers who made a purchase of $50 or more.

   (c) Estimate the fraction of items that were purchased by at least 10 customers.

Coding problems

Your task is to complete the notebook file `streaming_algorithm.ipynb` . Find more details of tasks in `streaming_algorithm.ipynb` .

You can add markdown cells to write things you want to report. You can also add additional code cells to complete the tasks.

You can choose to install jupyter server on your own PC, or to create a notebook on some online platforms such as Jupyter and Kaggle and upload the data file `stream_data_dgim.txt` to that platform.

A preview of the notebook file is listed at the end of this pdf.

Submission

You should submit a zip file via Canvas.

The zip file should contain only one folder named as [StudentID]_[StudentName]

There should be

1. a pdf file as your answer to concept questions
2. a completed ipynb file named `streaming_algorithm.ipynb`
3. the data file `stream_data_dgim.txt` . (You don't need to change it, but you need to include it to your submission.)

Scoring criteria

Concept questions: 20 + 10 + 10 points, 40 points in total

Coding problems: 20 points per task, 60 points in total. We mainly consider the correctness of your codes. Other aspects include code efficiency and your analysis.

streaming_algorithm

May 9, 2023

1 EE359-Coursework 4 Streaming Algorithm

Existing codes in this file are just hints. You can modify these codes as you want

1.1 Task1 DGIM

DGIM is an efficient algorithm in processing large streams. When it's infeasible to store the flowing binary stream, DGIM can estimate the number of 1-bits in the window. In this coding, you're given the stream_data_dgim.txt (binary stream), and you need to implement the DGIM algorithm to count the number of 1-bits. Write code below.

1.1.1 1. Set the window size to 1000, and count the number of 1-bits in the current window.

[ ]:
```
from collections import defaultdict

class DGIM:
    def __init__(self, filepath, windowsize, maxtime = None):
        self.fileHandler = open(filepath, 'r')
        self.windowSize = windowsize
        self.buckets = defaultdict(list)
        self.timeMod = maxtime if maxtime else windowsize << 2
        self.timestamp = 0

    def update(self):
        pass

    def run(self):
        f = self.fileHandler
        x = f.read(2).strip()
        while x:
            if x == '1':
                self.buckets[1].append(self.timestamp)
                self.update()
            self.timestamp = (self.timestamp + 1) % self.timeMod
            x = f.read(2).strip()

    def count(self, start):
```

```
        pass
```

1.1.2 2. With the window size 1000, count the number of 1-bits in the last 500 and 200 bits of the bitstream.

[ ]:
```
# code here
```

1.1.3 3. Write a function that accurately counts the number of 1-bits in the current window. Caculate the accuracy of your own DGIM algorithm and compare the running time difference.

[ ]:
```
# Your code here, you can add cells if necessary
import numpy as np
import time
def accurateCountTask1():
    pass
accurateCountTask1()
```

1.2 Task2: Bloom Filter

A Bloom filter is a space-efficient probabilistic data structure. Here the task is to implement a bloom filter by yourself.

1.2.1 Data loading:

From the NLTK (Natural Language ToolKit) library, we import a large list of English dictionary words, commonly used by the very first spell-checking programs in Unix-like operating systems.

[ ]:
```
import nltk
from nltk.corpus import words
word_list = words.words()
```

Then we load another dataset from the NLTK Corpora collection: movie_reviews.

The movie reviews are categorized between positive and negative, so we construct a list of words (usually called bag of words) for each category.

[ ]:
```
from nltk.corpus import movie_reviews

neg_reviews = []
pos_reviews = []

for fileid in movie_reviews.fileids('neg'):
  neg_reviews.extend(movie_reviews.words(fileid))
for fileid in movie_reviews.fileids('pos'):
  pos_reviews.extend(movie_reviews.words(fileid))
```

Here we get a data stream (word_list) and 2 query lists (neg_reviews and pos_reviews).

1.2.2 1. Write a function that accurately determines whether each word in neg_reviews and pos_reviews belongs to word_list.

[ ]:
```
import numpy as np
import time

def linear_search(word_list, target_word):
    pass
```

[ ]:
```
# Binary search
neg_flag_b = np.zeros(len(neg_reviews), dtype=bool)
pos_flag_b = np.zeros(len(pos_reviews), dtype=bool)
# to store the accurate result

t = time.time()
sorted_word_list = sorted(word_list)
for i, neg_word in enumerate(neg_reviews):
    neg_flag_b[i] = linear_search(sorted_word_list, neg_word)
for i, pos_word in enumerate(pos_reviews):
    pos_flag_b[i] = linear_search(sorted_word_list, pos_word)
print("Search Time:", time.time() - t)
```

2. Implement the bloom filter by yourself and add all words in word_list in your bloom filter. Compare the running time difference between exact search (Task2 Question1) and multiple hash computations in a Bloom filter.

[ ]:
```
# Your code here, you can add cells if necessary
import hashlib

class BloomFilter:
    def __init__(self, m, k):
        self.m = int(m)
        self.k = int(k)
        self.table = np.zeros((self.m), dtype = bool)
    def add(self, item):
        hash1 = hashlib.sha256()
        hash1.update(item.encode())
        hash1 = int(hash1.hexdigest(), 16) % self.m
        hash2 = hashlib.md5()
        hash2.update(item.encode())
        hash2 = int(hash2.hexdigest(), 16) % self.m
        h = hash1
        for i in range(self.k):
            pass
    def check(self, item):
        pass
```

```
[ ]: bf = BloomFilter(1e7, 4)
neg_bf = np.zeros(len(neg_reviews), dtype=bool)
pos_bf = np.zeros(len(pos_reviews), dtype=bool)
n = neg_bf.shape[0] + pos_bf.shape[0]
t = time.time()
for word in word_list:
    bf.add(word)
for i, neg_word in enumerate(neg_reviews):
    neg_bf[i] = bf.check(neg_word)
for i, pos_word in enumerate(pos_reviews):
    pos_bf[i] = bf.check(pos_word)
print("Bloom Filter Time:", time.time()-t)
print("Accuracy:")
print(f"neg: {(neg_bf == neg_flag_b).sum()} / {neg_bf.shape[0]}")
print(f"pos: {(pos_bf == pos_flag_b).sum()} / {pos_bf.shape[0]}")
```

1.2.3 3. Use different bit array length 'm' and number of hash functions 'k' to implement the bloom filter algorithm. Then compare the impact of different m and k on the false positive rate.

```
[ ]: # Your code here, you can add cells if necessary
```

1.3 Task3: Count-Min sketch

In computing, the count–min sketch (CM sketch) is a probabilistic data structure that serves as a frequency table of events in a stream of data.

Here we use the query stream (neg_reviews or pos_reviews) from task 2.

1.3.1 1. Write a function that accurately counts the occurrence times of each word in neg_reviews or pos_reviews.

```
[ ]: # Your code here, you can add cells if necessary

def accCount(words):
    pass
neg_accCount = accCount(neg_reviews)
```

1.3.2 2. Implement the Count-Min sketch by yourself. Set different width w and depth d of the internal data structure of CM-Sketch. Compare the influence of different w and d on the error.

```
[ ]: # Your code here, you can add cells if necessary
class CountMin:
    def __init__(self, w, d):
        self.w = int(w)
        self.d = int(d)
```

```
        self.table=np.zeros((self.d, self.w), dtype=int)
    def add(self, item):
        hash1 = hashlib.sha256()
        hash1.update(item.encode())
        hash1 = int(hash1.hexdigest(), 16) % self.w
        hash2 = hashlib.md5()
        hash2.update(item.encode())
        hash2 = int(hash2.hexdigest(), 16) % self.w
        h = hash1
        for i in range(self.d):
            pass
    def count(self, item):
        pass
```