Assignment 3

CS-UH-2218: Algorithmic Foundations of Data Science

Assignments are to be submitted in groups of at most three*. Upload the solutions on NYU classes as one PDF file for theoretical assignments and separate source code files for each programming assignment. Submit only one copy per group. Clearly mention the participant names on each file you submit.*

Problem 1 (10 points)**.**
Suppose that given a stream of length m, we want to output all elements with frequency more than m/k but we do not want to output any element with frequency less than $(1-\epsilon)\,m/k$ for some given $\epsilon \in (0,1]$. How would you use the Misra-Gries algorithm to do this in one pass? How many counters (in terms of k and ϵ) do you need ?

Problem 2 (10 points)**.**
Suppose that we want to use Count-Min Sketch to select heavy hitters in a stream. All items that are $\frac{1}{k}$-heavy hitters should be selected. In addition, any item that is not a $\frac{1}{2k}$-heavy hitter should have only 0.1% chance of being selected.

- Explain how this can be done if the length of the stream m is known in advance.
- How would you modify your algorithm if m is not known in advance?

Remark: Note that Count-Min Sketch itself does not store any elements. It only maintains the counts. The goal here is to identify and store all the heavy hitters.

Problem 3 (10 points)**.**
The goal of this exercise is to test the efficacy of the Flajolet-Martin (FM) algorithm and its variants in estimating the number of distinct elements. We will first count distinct words and shingles in the text file `http://norvig.com/big.txt` and then test it on a larger stream of numbers generated according to the power law distribution.[1]

The following function converts a file into a stream of words.

```
import string
def wordStream(fileName):
  with open(fileName, "r") as infile:
    for line in infile:
      for w in line.strip().lower().split():
        yield w
```

We can compute the number of distinct integers in the stream exactly as follows:

[1]A wide variety of natural phenomena follow this distribution. See `https://en.wikipedia.org/wiki/Power_law`.

```
def countDistinct(stream):
    M = {}
    for x in stream: M[x]=1
            return len(M.keys())
```

The above code of course uses an amount of memory proportional to the size of the set of distinct words and would not work if this was too large. We will only use it to measure the quality of the estimates we get.

The FM algorithm, as discussed in class, computes a hash value for each integer in the stream and uses the maximum number of trailing zeros seen in hash values to estimate the number of distinct integers. In the algorithms discussed in class, the hash functions are 2-universal. However, for ease and speed, we will use `mmh3.hash128`. For any element x, we can use `mmh3.hash128(str(x))` but we will often require multiple hash functions. For this purpose, we use another value `salt` and compute `mmh3.hash128(str(x) + str(salt))`. For each distinct value of `salt`, this gives us a new hash function.

TASK 1. Write a function `FM` which takes as input the stream and a number r denoting the required number of estimates and returns an array of r independent values $[z_0, z_1, \cdots, z_{r-1}]$ where z_i is the maximum number of trailing zeros seen in the hash values of the stream elements using the i^{th} hash function. Your function should look something like:

```
  def FM(stream, r):  # r is the number of estimates needed
    salt = np.random.randint(1<<30, size=r)
    z = [0]*r # z[i] counts the max no. trailing zeros for ith hash fn.

    for x in stream:
      for i in range(r):
          y = mmh3.hash128( str(x) + str(salt[i]), signed=False )
                # YOUR CODE HERE
                # update z[i] if y has more than z[i] trailing 0's
    return z
```

Feel free to modify the code as necessary. For efficiency reasons, avoid function calls in the loop.

Combining Estimates. Let z be the array obtained by

```
  z = FM(wordStream("big.txt"), r)
```

for some value of `r`. There are several ways of combining the estimates to obtain a final estimate. Note that each $z[i]$ represents an estimate of $2^{z[i]}$ by the FM algorithm. Try the following ways of combining the estimates for different values of r and state which seems to be the best:

1. mean of the estimates: $(2^{z[0]} + \cdots + 2^{z[r-1]})/r$.
2. median of the estimates: $\text{median}\{2^{z[0]}, \cdots, 2^{z[r-1]}\}$.
3. harmonic mean of the estimates: $r/(2^{-z[0]} + \cdots + 2^{-z[r-1]})$.

Stochastic Averaging. The above approach has the drawback that we need to compute r hash functions on each stream element. One way around this is the following. Compute just one hash value for each element and use its last few bits (say k bits) to partition the elements into 2^k groups and use the remaining bits for the algorithm within the group. Each distinct element is thus assigned a unique group. In each group, we use the remaining bits of the hash value to estimate the number of distinct elements using the FM Algorithm. Thus, we get $r = 2^k$ estimates, each estimate a power of 2. Since each distinct element is mapped to exactly one group, the sum of the r group estimates would be an estimate of the overall number of distinct elements. Instead of taking the sum of the group estimates, the HyperLogLog(HLL) algorithm does something different. For each group i, let $z[i]$ be the maximum number of trailing zeros in the hash values (after excluding the last k bits) of the elements in the group. HLL returns the estimate $\alpha_r\, r^2/(2^{-z[0]} + \cdots + 2^{-z[r-1]})$ where α_r is a correction factor.

TASK 2. Write a function `HLL` which takes a stream as input and returns an estimate of the number of distinct elements computed as described above using $k = 6$ and $r = 2^k = 64$. For any hash value h, you can obtain, the last 6 bits using `h & 63` (bitwise AND) and the remaining bits using $h >> 6$ (bit shifting). For $r = 64$, $\alpha_r = 1.418$ [2]. Use your `HLL` function to estimate the number of distinct

1. words in the file `big.txt`
2. 9-shingles[3] in the file `big.txt`
3. elements in the stream produced by

```python
def numStream():
    t = 1<<30
    for i in range(t):
        yield int(t*np.random.power(3.5))
```

In each case, report how accurate the estimates are by indicating the relative errors. If the actual number of distinct elements is d and our estimate is $\hat{d}$, the relative error is $|d - \hat{d}|/d$.

Problem 4 (10 points)**.**
Each row of the file `data.txt` contains data point $\mathbb{R}^5$. Use the Elbow method along with Lloyd's algorithm for k-means clustering to estimate the number of clusters in the data set.

Also try the Elbow method with Lloyd's algorithm on the `digits` dataset obtained in `sklearn` as follows.

```python
from sklearn.datasets import load_digits
data = load_digits().data
```

If we did not already know that the dataset has 10 clusters (corresponding to the ten digits), what would you have guessed from the Elbow method?

[2] If you compare with the Wikipedia article `https://en.wikipedia.org/wiki/HyperLogLog`, you will notice that our α values are twice the values stated there. This is because instead of counting the number of trailing zeros, they count the number of leading zeros (which is equivalent) but also include the count of the first 1 after the leading zeros.

[3] Recall that a k-shingle is any substring of the text with k consecutive characters (including spaces).

Problem 5 (10 points)**.**
In this exercise, we will use k-means clustering to compress an image. When an image is stored in the PNG format, for each pixel, the red, blue and green components are stored as 8 bit numbers each. Thus, we need 24 bits per pixel [4].

The idea for compressing is the following. We think of each pixel with components (r, g, b) as a point in three dimensions. Thus there are as many points as there are pixels in the image. Then, using k-means clustering, we compute k cluster centers. Each cluster center is a point in three dimensions and thus represents a color. If a point p belongs to a cluster with center c, we will change the color of the pixel corresponding to p to the color represented by c. We expect the error caused by this change to be small.

Let us take $k = 64$ so that each cluster number is encoded by 6 bits. Thus, we will only use 64 distinct colors corresponding to the 64 cluster centers. Each pixel can now store 6 bits of information indicating which of the 64 colors it uses. Thus, apart from the overhead of storing the 64 colors using 64×3 bytes, we only use 6 bits per pixel, leading to a reduction in space by a factor of about 4.

1. Use the above idea to convert the image `Neuschwanstein_small.png` into an image that uses only 64 distinct colors.

2. If you try to use the same idea for a larger image like `Neuschwanstein_large.png`, you will find that k-Means takes a long time since the image has over a million pixels. Use `MiniBatchKMeans` instead of `KMeans` from Scikit-learn. Skim this short paper to learn how MiniBatchKMeans works.

You can start with the code below which shows how to convert an image into a numpy array and how to display such an array. You should also look at the sample code for k-Means available on NYU classes.

```
import matplotlib.image as mpimg

img=mpimg.imread('Neuschwanstein_small.png')
img  = img[:,:,:3] # remove the fourth coordinate alpha
plt.imshow(img)
print(img.shape)
```

To use `MiniBatchKMeans` instead of `KMeans`, you just need to import it and provide an additional parameter called `batch_size`. For instance, you can use code like this:

```
from sklearn.cluster import MiniBatchKMeans

mbkmeans = MiniBatchKMeans(n_clusters = 64, batch_size=10000)
mbclusters = mbkmeans.fit_predict(data) # some data
mbcenters = mbkmeans.cluster_centers_
```

[4] We are assuming RGB format here. In the RGBA format, there is a fourth component called "alpha" which denotes transparency. The RGBA format requires 32 bits per pixel.

Problem 6 (10 points)**.**
In the standard version of the k-means clustering problem, the centers are not required to be data points themselves. Consider a variant where we need the cluster centers to be picked from among the given data points. We will show that the cost of the optimal solution of this variant is at most twice the cost of the optimal solution of the standard version. It suffices to show that in each cluster of the optimal solution to the standard version, we can replace the centroid by one of the data points without increasing the cost of the cluster by more than a factor 2. This can be shown as follows.

Let $X = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_m\}$ be the data points in one cluster and let $\mathbf{c}$ be their centroid. Note that the cost of this cluster is $\sum_{\mathbf{x}\in X} \|\mathbf{x} - \mathbf{c}\|^2$.

1. Prove that for any $\mathbf{x}_i$ and $\mathbf{x}_j$, $\|\mathbf{x}_i - \mathbf{x}_j\|^2 = \|\mathbf{x}_i - \mathbf{c}\|^2 + \|\mathbf{x}_j - \mathbf{c}\|^2 - 2\,(\mathbf{x}_i - \mathbf{c}) \cdot (\mathbf{x}_j - \mathbf{c})$.
2. Use the above to show that if we replace the centroid by a data point $\mathbf{x}_i \in X$ then the new cost of the cluster i.e., $\sum_{\mathbf{x}\in X} \|\mathbf{x} - \mathbf{x}_i\|^2$, is equal to $\sum_{\mathbf{x}\in X} \|\mathbf{x} - \mathbf{c}\|^2 + m \cdot \|\mathbf{x}_i - \mathbf{c}\|^2$.
3. Show by averaging that for some $\mathbf{x}_i \in X$, the above cost is at most $2 \cdot \sum_{\mathbf{x}\in X} \|\mathbf{x} - \mathbf{c}\|^2$.

To show that the above bound is tight, give a simple example of a single cluster where replacing the cluster center by any of the input data points increases the cost by a factor of 2.