Lucas Manker
5/6/20
COSC 4570
Homework 5

1

All code can be found in file partOne.py

2.3.1

a. Map: map all integers with unique keys corresponding to the number value. Reduce: Filter the largest key pair.

b. This is similar to the previous question, but it's important to count all instances the number appears. For this reason, I simply used the value of the numbers as the key, and the actual value as the count. Then I iterated over each key, and multiplied the key value by the number of times the number was encountered to get a sum.

c. This can be done in the exact same way I implemented question a. Create key pairs with the key and value being the same.

d. This is also very similar to a and c. All that's necessary to change is to count the number of keys present in problem c.

3.2.1

First I stripped all punctuation, and made everything lowercase. Then I pulled the first 3 words from the list of words, and iterated over the string. This was my result:
[['the', 'most', 'effective'], ['most', 'effective', 'way'], ['effective', 'way', 'to'], ['way', 'to', 'represent'], ['to', 'represent', 'documents'], ['represent', 'documents', 'as'], ['documents', 'as', 'sets'], ['as', 'sets', 'for'], ['sets', 'for', 'the'], ['for', 'the', 'purpose']]

3.3.3

a.

	S1	S2	S3	S4
h(1)	5	1	1	1
h(2)	2	2	2	2
h(3)	0	1	4	0

b. h(3) is the only true permutation.

c.

	1&2	1&3	1&4	2&3	2&4	3&4
Columns	0	0	0.25	0	0.25	0.25
Signatures	0.33	0.33	0.67	0.67	0.67	0.67

The estimated Jaccard similarities are no where near the true Jaccard similarities.

3.3.6

Original:

	S_1	S_2
1	0	0
2	0	1
3	1	1

Permutation 1:

	S_1	S_2
2	0	1
3	1	1
1	0	0

Permutation 2:

	S_1	S_2
3	1	1
2	0	1
1	0	0

h_1, h_2, h_3 are hash functions:

	S_1	S_2
h_1	3	2
h_2	2	1
h_3	1	1

3.4.4

a. First input the signatures and map key-value pairs. Then to reduce, create buckets for each band of output. So instead of a key-value pair consisting of a key-element, we reduce to a key-list(of elements).

b. Next, map a list of combinations corresponding to the same bucket. So the key-list(elements) turns into a key-aggregation(list(elements)). The reduction step is the comparison between pairs within the list. For column i, there will be a list of columns j¿i for which to compare i.

4.3.2

Using k hash functions has a probability of $(1 - e^{-km/n})^k$

I visualized this in the same way as explained in class (with dartboards). Instead of one huge dartboard, there's now k dartboards of size n/k. So instead of a single probability, there must be a product of k probabilities for the likelihood of hitting the dartboard k times.

The original equation without taking hashing into account is: $1 - e^{-y/x}$
Originally $y = km$ where k was the number of hashing functions, but since we're only using a single hashing function with multiple array $y = m$. x originally was equal to n but since every "dartboard" needs to be broken down $x = n/k$.

So the equation is $1 - e^{\frac{-m}{n/k}}$, but this is only for a single dartboard.

The combined probability of a false positive is then the product of all the false positives for all "dartboards".

$\prod^k 1 - e^{\frac{-m}{n/k}}$

It appears like this might perform better than the original method with multiple hashes, but the memory requirement is massive which is prohibitive.

4.3.3

Finding the derivative of the function will allow us to minimize the false positive rate. The false positive rate is defined as:

$f = (1 - p)^k$

First minimize the log of f:

$g = ln(f)$

$g = k(ln(1 - p))$

$g = k(ln(1 - e^{-kn/m}))$

Then take the derivative:

$\frac{dg}{dk} = ln(1 - e^{-kn/m}) + \frac{kn}{m}(\frac{e^{-kn/m}}{1-e^{-kn/m}})$

The choice of k becomes optimal when the derivative is zero.

$k = (ln(2))\frac{m}{n}$

4.4.1

The number of distinct elements is $\frac{2^R}{\phi}$ where $\phi = 0.77351$ according to the Flajolet-Martin algorithm.

a.

Value	Hash Value	Binary	R	2^R
3	7	00000111	0	1
1	3	00000011	0	1
4	9	00001001	0	1
1	3	00000011	0	1
5	11	00001011	0	1
9	19	00010011	0	1
2	5	00000101	0	1
6	13	00001101	0	1
5	11	00001011	0	1

The estimated number of distinct elements is then $\frac{2^0}{0.77351} = 1.29$

b.

Value	Hash Value	Binary	R	2^R
3	16	00010000	4	16
1	10	00001010	1	2
4	19	00010011	0	1
1	10	00001010	1	2
5	22	00010110	1	2
9	2	00000010	1	2
2	13	00001101	0	1
6	25	00011001	0	1
5	22	00010110	1	2

The max tail length is 4 so the estimated number of distinct elements is $\frac{2^4}{0.77351} = 20.68$

c.

Value	Hash Value	Binary	R	2^R
3	12	00001100	2	4
1	4	00000100	2	4
4	16	00010000	4	16
1	4	00000100	2	4
5	20	00010100	2	4
9	4	00000100	2	4
2	8	00001000	3	8
6	24	00011000	3	8
5	20	00010100	2	4

The max tail length is again 4 so the estimated number of distinct elements is $\frac{2^4}{0.77351} = 20.68$

4.5.3

Using the Alon-Matias-Szegedy Algorithm we can create a table to match pairs of x_i elements and x_i values.

Starting Position(i)	x_i element	x_i value
1	3	2
2	1	3
3	4	2
4	1	2
5	3	1
6	4	1
7	2	2
8	1	1
9	2	1

Then, calculate the 2nd moment:

$$F_2 = \frac{\sum 9(2(x_{value} - 1)}{9}$$

$$= \frac{27+45+27+27+9+9+27+9+9}{9} = 21$$

2

I tested several different values for memory size, but I opted to go large because of the size of the data set. I picked a bit array of size 5,000,000. With the size of the 365 file, even 2% false positives can push the total number of matches to over the size of the initial data set. This means I should have 14 hash functions according to the optimization formula $k = ln(2)(m/n)$ as there are 255,478 email addresses.

By using the formula for false positives: $P = (1 - [1 - (1/m)^{kn})^k$ I should have 8.27e-05% false positives.

I ended up with 37,554 matches in the filter. This possibly seems incorrect after a cursory comparison between the files (all names from 30 seem to be in 365), but I wasn't able to fully verify this because of the size of the files.

3

So for this problem I had to use multithreading to keep the runtime as low as possible since there is so much data. I used a thread to scan each document, and then hashed each line that started with a Q into a 32 bit integer. The hash functions I used were from the Python hashlib library (not the built in hash, it has some randomized salt which will not yield the same result). The hashes I used were sha1 sha256 and sha512. After hashing I binarized the integers, then counted the trailing zeroes. As per the book's suggestion in 4.4.3 I kept a count of all trailing zeroes, calculated the 2^R value, and the total number of lines with Q. After all threads were finished the 2^R were summed, and divided by the total count of Q lines to provide an average. I then took the median value of the 3 hashes. This provided me with a median value of 12.5.

I was skeptical of this value so I decided to use 64 bit hashes which was suggested in the book. I used the FNV, FNV1a, and murmur hashes. The median of these 3 hashes was 14.2, but there was an interesting outlier for the fnv1a hash. The 2^R value for this hash was 33.52 which is more than double the median. I would like to do more testing because I suspect that the true median 2^R value is probably higher the more bits you hash, but to encode this with a higher bit hash value I would need to delve deeper into multithreading because the latest code took all night to run.