Homework 2

Dejan Porjazovski

October 20, 2018

1 Problem 1

1.1 Argue that the extension of the Jaccard coefficient to bags, as defined above, is meaningful and well motivated

If we have a bag like this:

$$A = \{(a, 3), (b, 1)\}$$

we can treat each bag of words as a finite, unordered set.
Jaccard similarity is used for comparing two sets, therefore the extension of the proposed Jaccard coefficient to bags can be applied to a set represented as bag of words, so, the extension is meaningful and well motivated.

1.2 Provide a locality-sensitive hashing (LSH) scheme for the Jaccard coefficient to bags. In other words, desifn a family of hash functions F such that:

$$Pr[f(A) = f(B)] = J(A, B)$$

when f is drawn uniformly at random from F. Prove that the previous equality holds

We can use the MinHash algorithm.
We have 2 bags of words: A and B, we take the union of the two sets $A \cup B$ and randomly shuffle them and take the minimum element in the permutation

$$h(x) = min_i\{r(x_i)\}$$

We repeat this n number of times.
Proof

Let:

$$A = \{(a, 3), (b, 1), (d, 1)\}$$

$$B = \{(a, 1), (b, 2), (c, 1)\}$$

We can unpack the sets and get:

$$A = \{(a, 1), (a, 1), (a, 1), (b, 1), (d, 1)\}$$

$$B = \{(a, 1), (b, 1), (b, 1), (c, 1)\}$$

The union $A \cup B = \{(a, 3), (b, 2), (c, 1), (d, 1)\}$

The intersection $A \cap B = \{(a, 1), (b, 1)\}$
$||A \cup B|| = 7$
$||A \cap B|| = 2$

The Jaccard similarity is:

$J = \frac{||A \cap B||}{||A \cup B||} = 0.28$

If we randomly shuffle $A \cup B$, the probability that one of the items in the intersection $A \cap B$ ends up first in the list is $\frac{||A \cap B||}{||A \cup B||}$, which is the Jaccard coefficient.
We can say that the probability that the MinHash value will come from one of the items in the intersection $A \cap B$ is equal to the Jaccard similarity.

If we repeat the MinHash for 10 times (10 components), the number of values expected to be from the intersection is $num\_of\_components * prob\_of\_match = 10 * \frac{||A \cap B||}{||A \cup B||} = 2.8$. The expected value of the MinHash is:
$\frac{components\_intersection}{num\_components} = \frac{2.8}{10} = 0.28$, which is same as the Jaccard coefficient.

1.3 Discuss how exactly you will implement the locality-sensitive hashing scheme you designed. Provide pseudocode for finding similar documents in a document collection given a query document, where documents are represented as bags of words. Your pseudocode should describe both the preprocessing and the querying part of the similarity-search algorithm.

Notation:

U - union of all the documents
s_number - number of documents
S - all the documents
M[U][s] - characteristic matrix that has row number = the union of all the words in all the documents and column number = number of documents.
hash family = (ax + b) mod p where x is the row index and p is prime. number ¿ size(U)
$M\_sig[size(hash\_family)][s]$ - signature matrix

```
# preprocessing

# generate prime number greater than size(U)
p = generate_prime()
num_hash_functions = 100
union_size = size(U)

# we initialize the signature matrix with inf values (maybe a really large numbers or just -1
# and then handle that in the code)
M_sig = inf

for i in num_hash_functions
   h.append(gen_hash_function)
```

```
for row in union_size
   row_idx = indexOf(row)
   for column in s
      # if element in set
      if M[row][column] == 1
         for hash in h
            hash_idx = indexOf(hash)
            hash_val = hash(row_idx)
            if M_sig[hash_idx][s] > hash_val
               # update signature matrix
               M_sig[hash_idx][s] = hash_val
            else
               # dont update
               pass

#querying part
q = new_document
q_size = size(q)
# we initialize the signature matrix for the query document with inf
#values (maybe a really large numbers or just -1 and then handle that in the code)
M_sig_q = inf

for row in q_size
   row_idx = indexOf(row)
   for hash in h
      hash_idx = indexOf(hash)
      hash_val = hash(row_idx)
      if M_sig_q[hash_idx][s] > hash_val
         # update signature matrix
         M_sig_q[hash_idx][s] = hash_val
      else
         # dont update
         pass
```

---

2 Problem 2

Consider we need to find *max* elemt in an array.
The best case scenarion would be is the first element in the array is the largest. In that case the time complexity would be $O(1)$.
On the other hand, consider that the array is ordered in ascending order. In that case the *max* will be the last element, therefore the complexity would be $O(n)$, whaere n in the number of elements in the array.
Generally we are not interrested in those scenarion since the probability that they will occur is small.
What we are interested in is the complexity when the *max* element in the array is somewhere in the middle.
If we assume that $X[i]$ is drawn independently and uniform at random from the interval $(0, 1)$, then the expected value will be:

$$E[x] = \sum_{i=1}^{m} P(X_i)$$

where $P(X_i)$ is the probability that the *i-th* element is the *max* element in $X[1, 2, 3, 4, ..., m]$. Following that, we can see that:

$$E[x] = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{m} = \sum_{i=1}^{m} \frac{1}{i} = \log m$$

.
With this, we have say that the update step $max = X[i]$ will be executed $O(\log m)$ times on expectation.

We can use the same logic to prove that if we use *priority sampling for sliding window*, the space requirement would be $O(\log w * \log n)$, where w is the length of the sliding window and n is the size of the universe where numbers are drawn from.

Each element that comes in has uniform random number assigned.
In each time step we need to scan the window and choose which values to store.
We do this by starting from the right side of the window and select each element that has smaller probability then all the previously encountered elements.
This step is similar to the update that we have been doing in the previous algorithm when we encounter a value that is bigger than the current max. The only difference is that now we are looking for a min value instead of max.
With all that said, the algorithm will keep on average $O(\log w)$ elements.
Storing each of those elements requires $O(\log m)$ bits. Therefore the space requirement for the *priority sampling sliding window* is $O(\log w * \log n)$.

3 Problem 3

3.1 Write in English what does it mean that a k-sample in a data stream is uniform

If we store a sample of size K, each element in the stream has equal chance to be selected at any given time step to the sample.

3.2 What should the value of the probability p be for the k- RESERVOIR algorithm to give uniform samples?

As the number of element in the stream (t) increases, the probability p should decrease.
Therefore, it makes sense the value of p to be: $p = \frac{k}{t}$.

3.3 Prove that for the value of p that you specified in 4.2 the k- RESERVOIR algorithm gives indeed uniform samples.

If we have $t = k + 1$, which means that the reservoir S has just been filled.
Each new element has $\frac{k}{t}$ probability that it will be selected.

$$\left(1-\frac{k}{t}\right)+\frac{k}{t}\left(1-\frac{1}{k}\right)=1-\frac{k}{t}+\frac{k}{t}-\frac{k}{kt}=1-\frac{1}{t}=\frac{t-1}{t}$$

For the time step $t=k+1$ we get:

$$\frac{k}{t}$$

For a continuing stream we have the following:

$$\left(1-\frac{k}{t+1}\right)\frac{k}{t}+\left(\frac{k}{t+1}(1-\frac{1}{k})\right)\frac{k}{t}$$

$$=\left(\frac{k}{t}-\frac{k^2}{t^2+t}\right)+\left(\frac{k}{t+1}-\frac{k}{tk+k}\right)\frac{k}{t}$$

$$=\frac{k}{t}-\frac{k^2}{t^2+t}+\frac{k^2}{t^2+t}-\frac{k^2}{t^2k+kt}$$

$$=\frac{k}{t}-\frac{k^2}{t^2k+tk}$$

$$=\frac{k}{t}-\frac{k}{t(t+1)}$$

$$=\frac{k}{t}\left(1-\frac{1}{t+1}\right)$$

$$=\frac{k}{t}\frac{t}{t+1}$$

$$=\frac{k}{t+1}$$

We have showh that if the probability is $\frac{k}{t}$, every element in the stream has equal chance to be in the sample.

4 Problem 4

4.1 Describe a real-world application that you may consider applying the above algorithm.

Consider an online store that has database with all the customers that have purchased something from the store.
We are interested in the customers that have spent more than 1000 euro *"good customers"*.
Since the database is huge, we can't check all the existing customers and calculate how many of them have spent more than 1000 euro.
What we ca do is use a subset of the database and find out how many of them have spent more than 1000 euro. We can cosinder those as the *"good"* items.
Based on that, we can calculate p and use it to estimate the number of the *"good"* customers in the whole database.

4.2 Explain the intuition of MONTE CARLO algorithm.

The intuition behind the *Monte Carlo* algorithm is that if we take large enough samples from the population that we are interested in, that sample will have a good representation of the whole population.
By looking at the drawn samples we can make assumptions for the whole population which can be really beneficial if we have large database.

4.3 Explain Equation (1) in English.

In the equation, Z is the estimation of the *"good"* values. $|G|$ is the value that Z is trying to estimate.
$(1-\epsilon)$ means how further away can Z be from the true value of $|G|$. In other words, how much "freedom" do we give Z to vary from the true value.
$1-\delta$ is the lower-bound for the probability that the expected value $E[z]$ is in range ϵ of the true value $|G|$.

4.4 Use the Chernoff bound to show that if $N \geq \frac{4}{\epsilon^2 p} \quad in \quad \frac{2}{\delta}$, then the MONTECARLO algorithm returns an estimate Z that is an (ϵ, δ) - approximation of $|G|$

We know that the following is true:

$$Pr[(1-\epsilon)|G| \leq Z \leq (1+\epsilon)|G|] \geq 1-\delta$$

We also know that:

$$Z = \frac{|Y||U|}{N} \quad and \quad p = \frac{|G|}{|U|}$$

From there we get:

$$Pr\left[(1-\epsilon)|G| \leq \frac{|U||Y|}{N} \leq (1+\epsilon)|G|\right] \geq 1-\delta$$

$$Pr\left[(1-\epsilon)\frac{|G|N}{|U|} \leq |Y| \leq (1+\epsilon)\frac{|G|N}{|U|}\right] \geq 1-\delta$$

$$Pr\left[(1-\epsilon)p*N \leq |Y| \leq (1+\epsilon)p*N\right] \geq 1-\delta$$

The Chernoff bound is defined as:

$$Pr(X \geq (1+\delta)\mu) \leq e^{-\frac{\delta^2\mu}{2}}$$

$$Pr(X \leq (1-\delta)\mu) \leq e^{-\frac{\delta^2\mu}{3}}$$

If we replace δ with ϵ and X with Y, we get:

$$Pr(Y \geq (1+\epsilon)\mu) \leq e^{-\frac{\epsilon^2\mu Y}{2}}$$

$$Pr(Y \leq (1-\epsilon)\mu) \leq e^{-\frac{\epsilon^2\mu Y}{3}}$$

We can calculate the upper bound the following way:

$$Pr((1-\epsilon)\mu Y \leq Y \leq (1+\epsilon)\mu Y) \geq 1 - Pr(Y \leq (1-\epsilon)\mu Y) - Pr(Y \geq (1+\epsilon)\mu Y)$$

$$Pr((1-\epsilon)\mu Y \leq Y \leq (1+\epsilon)\mu Y) \geq 1 - e^{-\frac{\epsilon^2 \mu Y}{3}} - e^{-\frac{\epsilon^2 \mu}{2}}$$

In our case $pN = \mu Y$
We also know that: $e^{-\frac{\epsilon^2 pN}{4}} > e^{-\frac{\epsilon^2 pN}{2}}$ and $e^{-\frac{\epsilon^2 pN}{4}} > e^{-\frac{\epsilon^2 pN}{3}}$

$$Pr((1-\epsilon)\mu Y \leq Y \leq (1+\epsilon)\mu Y) \geq 1 - e^{-\frac{\epsilon^2 \mu Y}{3}} - e^{-\frac{\epsilon^2 \mu}{2}} \geq 1 - 2\epsilon^{-\frac{\epsilon^2 pN}{4}}$$

To get the bounds for Z, we do the following:

$$Pr((1-\epsilon)|G| \leq Z \leq (1+\epsilon)|G|) \geq 1 - 2\epsilon^{-\frac{\epsilon^2 pN}{4}} \geq 1 - \delta$$

Now we need to check for which N, the inequality holds:

$$1 - 2\epsilon^{-\frac{\epsilon^2 pN}{4}} \geq 1 - \delta$$

If we solve this, we get:

$$N \geq \frac{4}{\epsilon^2 p} \quad in \quad \frac{2}{\delta}$$

With this, we have shown that is we select N based on the last equation, we can get a good approximation of Z.

4.5 Discuss the bound (2) in terms of the dependence of N with respect to the approximation parameters ϵ and δ and the problem parameters $|U|$ and p. What are the implications of the bound (2)?

The desired accuracy ϵ is in the denominator, so, if we want to have a smaller error, we need to take a bigger sample.
Lowering the error requirement increases quadratically because ϵ is squeared.

p is the proportion of "good" items. Since p is also in the denominator, the lower the proportion of good items is (the lower the value of p is), the bigger the amount of the sample size that we need to take is.

δ is the probability that Z is not in the desired accuracy.
Since δ is in the denominator, if we increase δ, we will get a bigger bound which means we will need to take less number of samples.
If we decrease δ, we will have a smaller bound, which means we will need to take bigger number of samples.

$|U|$ is the whole set of items. If $|U|$ does not have enough "good" items, it will be hard to take a sample and try to find the estimate of the "good" items.