Homework 2 example solution

Compiled by Han Xiao

November 5 2018

Question 1

Question 1.1

Credit: Joonas Lipping

The Jaccard similarity for sets of words reflects the notion that documents are similar when they contain approximately the same set of words. The Jaccard coefficient for bags of words extends this idea by the notion that similar documents should agree not only on whether a word occurs, but also in how many times it occurs.

It is reasonable to hypothesize that two different kinds of documents (which could mean different topics, authors, etc.) can agree that a word is likely to occur, while a meaningful distinction can be made in the relative frequency of that word. The motivation and meaning for the Jaccard coefficient for bags is then clear: it can capture this kind of distinction.

Question 1.2

We first describe how to transform bags to make it feasible to use. Then we explain the min-hash scheme.

Transforming bags: given a bag $X = \{(x, n)\}$, we transform it into $X^{'}$ as follows: each (x, n) is expanded into $\{x_1, \ldots, x_n)\}$. For example give $X = \{(a, 3), (b, 2), (c, 1)\}$, $X^{'} = \{a_1, a_2, a_3, b_1, b_2, c_1\}$.

Min-hash scheme: Assuming for each word $w \in U$, we know its maximum frequency, denoted as $w_{\max}$, then the min-hash function materializes a random permutation $r : U \to [1, \ldots, m]$, where $m = \sum_w w_{\max}$.

The min-hash function f simply returns the minimum element in the permutation:

$$f(X) = \min_{x \in X} r(x)$$

Why $\mathbf{Pr}[f(X) = f(Y)] = J(X, Y)$? Given two bags X and Y, we build a matrix M where columns correspond to $\{X, Y\}$ and rows corresponds to elements in $X^{'} \cup Y^{'}$, the entry is 1 if the corresponding column contains the element and 0 otherwise. An example is illustrated in 1.

Table 1: Illustration of a matrix M (with rows randomly permutated) built on $X = \{(a, 3), (b, 2), (c, 1)\}$ and $Y = \{(a, 4), (c, 2), (d, 2)\}$ with row types being annotated.

Element	$X^{'}$	$Y^{'}$	Type
a_1	1	1	I
a_2	1	1	I
a_3	1	1	I
a_4	0	1	J
b_1	1	0	J
b_2	1	0	J
c_1	1	1	I
c_2	0	1	J
d_1	0	1	J
d_2	0	1	J

We distinguish two types of rows in M, where

- *type-I*: a row that has *both* 1 in column X and Y
- *type-J*: a row that has 1 in *either* column X or Y

We denote the cardinality of the above rows as i and j respectively.

Consider random permutation of the rows (type-I and type-J only), the probability that type-I row comes on top of the rows is simply $i/(i+j)$. Such probability is equivalent to the probability that $f(X) = f(Y)$.

In addition, by definition of i and j, it's trivial to see $J(X,Y) = \frac{i}{i+j}$. Therefore, $\Pr[f(X) = f(Y)] = J(X,Y)$

Question 1.3

Credit:

- pseudo-code partially based on Lukas Bernwald's solution
- Python code by Christabella Irwanto

An algorithm that achieves this is illustrated in Algorithm 1. In addition, code in Python is provided (which however, does not implement gap amplification).

Algorithm 1: Finding similar documents in a document collection

Data: Document collection $D = \{d_1, \ldots d_n\}$, where $d = \{(w, i) \mid w$ is in document d with frequency $i\}$. Query document q has the has representation as d, and similarity threshold s. m and k are the stacking and repeating parameters respectively (for similarity gap amplification).

Result: Similar documents d_i with $J(q, d_i) > s$

```
preprocess(D, m, k);
Z ← ∅;
for i ← 1 to m do
    q_i ← hashDocumentKTimes(q, k);
    Z_i ← {documents found in the bucket q_i};
    Z ← Z ∪ Z_i;
return all z ∈ Z, such that J(q, z) ≥ s
Procedure preprocess(D, m, k)
    for i ← 1 to m do
        for d ∈ D do
            h ←hashDocumentKTimes(d, k);
            Store d in bucket given by h;
Procedure hashDocumentKTimes(X, k)
    X' ← transformDoc(X);
    h ← randomly sampled k elements from X';
    return h;
Procedure transformDoc(X)
    X' ← ∅;
    for (x, j) ∈ X do
        for i ← 1 to j do
            X' ← X' ∪ {(x, i)}
    return X';
```

```
import random


```

```
def find_similar_documents(Q, X):
    '''Given a query document (bag of words), find all similar bags.
    Args:
        Q (list): A single bag of words, e.g. [ ('a', 3), ('b', 1),... ].
        X (list): All bags in the considered universe of bags other than
                  Q, e.g. [[ ('a', 3), ('b', 1),... ], ...].
    Returns:
        list: A list of similar bags.
        E.g. [[ ('a', 3), ('b', 1),... ], [('b', 1), ...], ...].
    '''
    buckets = preprocess(X)
    similar_documents = query(Q, buckets)
    return similar_documents


def unpack_all_bags(X):
    '''Transforms a set of documents (bags of words) X, into a set of
    unpacked bags of uniquely indexed words, X'.
    Args:
        X (list): A list of bags.
                  E.g. [[ ('a', 3), ('b', 1),... ], ...].
    Returns:
        list: A list of unpacked bags.
              E.g. [['a1', 'a2', 'a3', 'b1', ...], ...].
    '''
    X_prime = []
    for i in len(X):  # Iterate over bags
        flattened_bag = []
        for j in len(X[i]):  # Iterate over a bag
            word, count = X[i][j]
            for k in range(count):  # Iterate over count
                # Concatenate word and int k, e.g. 'a', 3 -> 'a3'
                indexed_word = "%s%d" % word, count
                flattened_bag.append(indexed_word)
        X_prime[i] = flattened_bag
    return X_prime


def minhash(A_prime, U_prime):
    '''Generate a signature for bag A, given its unpacked equivalent.
    Args:
        A_prime (list): A non-empty unpacked bag, e.g. ['a1', 'b1', ...].
        U_prime (list): The universal set of unpacked bags.
                        E.g. ['a1', 'a2', 'a3', 'b1', 'b2', ...].
    Returns:
        string: The minhash of bag A, e.g. 'b1'..
    '''
    R = random.shuffle(U_prime)  # Would need to import random
    for indexed_word in R:
        # Return the first element of R found in A'
        if indexed_word in A_prime:
            return indexed_word


def preprocess(X):
    '''Transform and hash all bags in X, and bucket the bags by
```

```
    their signatures.
    Args:
        X (list): A list of bags.
                  E.g. [[ ('a', 3), ('b', 1),... ], ...].
    Returns:
        dict: A dictionary of lists of bags, indexed by their shared
              hash value.
    '''
    X_prime = unpack_all_bags(X)
    buckets = dict()
    for x, x_prime in zip(X, X_prime):
        signature = minhash(x_prime)
        if signature not in buckets:
            buckets[signature] = []  # Initialize empty array
        buckets[signature].append(X)
    return buckets


def query(Q, buckets):
    '''Given a query bag of words, return all bags with the same signature.

    Args:
        Q (list): A bag of words, e.g. [ ('a', 3), ('b', 1),... ].
        buckets (dict): A dictionary of lists of bags, indexed by their shared
                        hash value.

    Returns:
        list: A list of matching bags.
        E.g. [[ ('a', 3), ('b', 1),... ], [('b', 1), ...], ...].
    '''
    Q_prime = unpack_all_bags([Q])[0]
    Q_signature = minhash(Q_prime)
    matching_bags = buckets[Q_signature]
    return matching_bags
```

Problem 2

Question 2.1

Define a random variable Y_i to be 1 if $X[i]$ is the maximum among the previous $i-1$ elements and 0 otherwise. Then we have the following: [1]

$$\Pr[Y_i = 1] = \frac{1}{i}$$

And define another variable Z as

$$Z = \sum_{i=1}^{m} Y_i$$

The expected number of max updates is equivalent to

[1] Think about given a set of k random numbers, what's the probability of a specific number being the maximum in the set?

$$
\begin{aligned}
E[Z] &= E[\sum_{i=1^m} Y_i] \\
&= \sum_{i=1\ldots m} E[Y_i] \\
&= \sum_{i=1\ldots m} (0 \times \Pr[Y_i = 0] + 1 \times \Pr[Y_i = 1]) \quad \text{Linearity of expectation} \\
&= \sum_{i=1\ldots m} \frac{1}{i} \\
&< \ln m + 1
\end{aligned}
$$

Thus, the expected number of updates is $O(\log m)$.

Priority sampling for sliding window We scan the elements in sliding window from right to left and save the tth element if it is smaller than previous $t-1$ elements (on its right).

From this perspective, the analysis is the same as the first question. The expected number of elements to save is $\log w$

Finally, the total space complexity is $O(\log w \log n)$.

Problem 3

Credit: Jan Horesovsky

Question 3.1

First, we consider all the elements of the stream we have seen so far, let us call them D. Then we consider all possible subsets $A \subseteq D$ such that $|A| = k$. Now A' is a uniform k-sample of a data stream if it has the same probability to be chosen as all other subsets A do.

Questions 3.2 & 3.3

Claim. *The algorithm in question gives a uniform k-sample, if $p = \frac{k}{t}$ where t is the number of elements of the data stream it has seen so far.*

Proof. Following the notation in question 3.1, if $|D| = t$, then there are $\binom{t}{k}$ possible subsets A. Each $x \in D$ is in exactly $\binom{t-1}{k-1}$ of those subsets. If we have a uniform k-sample S, it follows that

$$\Pr[x \in S] = \frac{\binom{t-1}{k-1}}{\binom{t}{k}} = \frac{k}{t} \tag{1}$$

for all $x \in D$.

Going back to the algorithm, in time $t \geq i$, we have:

$$\frac{k}{t} = \Pr[x_i \in S] = \frac{k}{i} \cdot \prod_{j=i+1}^{t} \left(\left(1 - \frac{k}{j}\right) + \left(\frac{k}{j} \cdot \frac{k-1}{k}\right) \right) \tag{2}$$

where the term on the right-hand side corresponds to the probability of x_i being chosen in step i and then not being replaced by any subsequent items in the data stream. After simplification, we get:

$$\frac{k}{t} = \Pr[x_i \in S] = \frac{k}{i} \cdot \prod_{j=i+1}^{t} \frac{j-1}{j} = \frac{k}{t} \tag{3}$$

□

Problem 4

Credit:

- Joonas Lipping: 4.1 - 4.4
- Lukas Bernwald: 4.5

Question 4.1

Suppose that we want to measure the fraction of IP packets passing through a given router in a given timeframe whose address has some property (such as being in a specific country). It can be infeasible to look up the property for every single packet, especially if the lookup involves asking a remote service, so it is reasonable to instead choose a subset to check by Monte Carlo sampling.

Question 4.2

A random subset of U should, on expectation, be representative of U as far as concerns intensive quantities (those quantities that don't depend on the size of the system – things like densities and fractions, among them ρ). Therefore calculating an intensive quantity of a random subset gives a reasonable estimate of the same intensive quantity of U. The larger the subset, the better the estimate.

Question 4.3

If we say that our estimate is correct up to an error margin of $\pm 100\epsilon\%$, we are wrong with probability at most δ. The idea is that ϵ and δ are small, and the gist of the statement is, "if we say that our estimate is correct up to a *narrow margin*, we are *unlikely* to be wrong." The specific numbers ϵ, δ make the statement precise as to how narrow (ϵ) and how unlikely (δ).

Question 4.4

The Poisson trials in this case are the goodness tests of the elements of G. In terms of the statement of the Chernoff bound, $X \leftarrow |Y|$ and $\mu \leftarrow \rho N$. Now if the bound in (2) holds,

$$\begin{aligned} N &\geq \frac{4}{\epsilon^2 \rho} \log \frac{2}{\delta} \\ \frac{\epsilon^2 \rho N}{4} &\geq \log \frac{2}{\delta} \\ -\frac{\epsilon^2 \rho N}{4} &\leq \log \frac{\delta}{2} \\ e^{-\frac{\epsilon^2 \rho N}{4}} &\leq \frac{\delta}{2} \\ 2e^{-\frac{\epsilon^2 \rho N}{4}} &\leq \delta, \end{aligned}$$

and consequently (using the Chernoff bounds)

$$\begin{aligned} \delta &\geq 2e^{-\frac{\epsilon^2 \rho N}{4}} \\ &= e^{-\frac{\epsilon^2 \rho N}{4}} + e^{-\frac{\epsilon^2 \rho N}{4}} \\ &\geq e^{-\frac{\epsilon^2 \rho N}{2}} + e^{-\frac{\epsilon^2 \rho N}{3}} \\ &\geq \Pr[|Y| < (1-\epsilon)\rho N] + \Pr[|Y| > (1+\epsilon)\rho N] \\ &= \Pr[\tilde{\rho} < (1-\epsilon)\rho] + \Pr[\tilde{\rho} > (1+\epsilon)\rho] \\ &= 1 - \Pr[(1-\epsilon)\rho \leq \tilde{\rho} \leq (1+\epsilon)\rho], \end{aligned}$$

which can be rearranged as

$$\Pr[(1-\epsilon)\rho \leq \tilde{\rho} \leq (1+\epsilon)\rho] \geq 1 - \delta,$$

which is the statement of the (ϵ, δ)-approximation. Incidentally, it seems like the given threshold for N is generous; it seems that a numerator of 3 would have served instead of 4.

Question 4.5

We describe how N has to change, so that our algorithm still returns an estimate Z, that is an (ϵ, δ)-approximation of $|G|$. In other words we describe how our sample size has to change, so that the algorithm still gives a useful/desirable estimate of the number of "good" items in the whole dataset.

If ϵ shrinks linearly, N has to grow squarely. This makes sense, because ϵ shrinks down the ϵ-environment around $|G|$ from two sides. So our intuition tells us, that our accuracy also grows square.

If δ shrinks linearly, N has to grow exponentially. This makes sense, because we have a Poisson distribution.

If $|G|$ stays the same and $|U|$ grows linearly, N has to grow linearly. This makes sense, because if we have linearly more elements, we intuitively also have to take linearly bigger samples.

If ρ grows linearly, N can shrink linearly. This makes sense, because if we have linearly more "good" items in the whole set. The probability, that we get "good" items in the random subset also grows linearly.

The implications of the bound are that the number of items one has to sample depends on how big the fault tolerance of the estimate can be, how sharp the lower bound of the probability of a mistake should be and what the ratio of "good" items to all items in the whole set is.