Big Data Computing, Spring 2016/17
Homework 1

Marco Favorito
Master of Science in Engineering in Computer Science
Department of Computer, Control, and Management Engineering
University of Rome "La Sapienza"
favorito.1609890@studenti.uniroma1.it

April 28, 2017

Contents

1 Problem 1

Introduction

First of all, let's make some considerations:

- Since $p << n$, we can model the event "a user j buys an item i" with a Bernoulli r.v.:
$$X_j \sim \text{Bern}\,(p)$$
- We can go further from this, defining the following random variable:
$$X_{rs} \sim \text{Bern}(p^2)$$
Which represents the event "two users buy the item i";
- Finally, we define a Binomial random variable as the following:
$$Y_{rs} = X_{rs}^1 + X_{rs}^2 + \cdots + X_{rs}^n = \text{Bin}(p^2, n)$$
Which represents the number of two users' common purchased items. Note that the number of common items is the same of $(u_r \cdot u_s)$, by definitions of the problem.

Solution

Let us compute the probability of a pair of user to be retrieved by our recommending system. By Markov's Inequality:

$$\mathbf{Pr}(Y_{rs} \geq qn) \leq \frac{\mathbb{E}[Y_{rs}]}{qn} = \frac{p^2 n}{qn} = \frac{p^2}{q}$$

It follows that the expected number of candidate pairs is simply:

$$\binom{m}{2} \cdot \mathbf{Pr}(Y_{rs} \geq qn) \leq \binom{m}{2} \cdot \frac{p^2}{q}$$

So, in order to do not violate the constraint over the maximum number of false positives, we can just set q (depending on α) in the following way:

$$\binom{m}{2} \cdot \frac{p^2}{q} \leq \alpha \binom{m}{2} \iff \frac{p^2}{\alpha} \leq q$$

Which means that, if we want at most an α fraction of false positive over all the possible pairs, we can ensure us setting q greater than $\frac{p^2}{\alpha}$, implying that, even in the *worst case* that all candidate pairs are false positive, we are not violating the constraint.

2 Problem 2

2.1 Choosen similarity measure

In the case of social networks, we can choose the similarity function as the "neighbor similarity", i.e. consider, in some way, the common friends [8]. One good idea is to use Jaccard Similarity:

$$J(X,Y) = \frac{|\Gamma(X) \cap \Gamma(Y)|}{|\Gamma(X) \cup \Gamma(Y)|}$$

Where X and Y are two nodes of the graph G and $\Gamma : V \to 2^V$, i.e. given a node, returns its neighborhood.

Using this choice of similarity function, we can easily reduce this problem to the one we have seen in the class: using min-hashing to estimate Jaccard Similarity, thinking the neighborhood of a node as a document like a "shingles-occurence array". To be clear, let us see an example.

Consider, for instance, the following simple adjacency list of a graph:

```
0: 1, 2;
1: 0;
2: 0;
```

We can rewrite the same graph in the following table:

```
\ 0 1 2
0 0 1 1
1 1 0 0
2 1 0 0
```

With this format, we can easily apply the min-hashing algorithm to estimate Jaccard Similarity, as we have already seen. In the following section, I will provide all the details, in order to satisfy the requirements.

2.2 Data structure/Algorithmic tools

Assuming that the total amount of memory for represent the graph is kn RAM words[1], we can build a table $\widehat{G}_{k \times n}$, where in the column i_{th} we have exactly k integers, and in a generic h_{th} cell of column C_i we will store the min-hash of $\Gamma(i)$, i.e.:

$$C_i = (h^1_{\min}(\Gamma(i)), h^2_{\min}(\Gamma(i)), \ldots, h^k_{\min}(\Gamma(i))) \tag{1}$$

[1] We are supposing that the word dimension is 32 bit

Where $h_{\min(S)}$ stand for min-hash of a set S, defined as the following:

$$h_{\min}(S) := \min_{x \in S} h(x)$$

As we know, the probability that two set have the same min-hash is equal to the Jaccard similarity of the sets, in symbols:

$$\mathbf{Pr}(h_{\min}(X) = h_{\min}(Y)) = \frac{|X \cap Y|}{|X \cup Y|} = J(X, Y)$$

And this justify, intuitively, why we need this settings.

2.3 Algorithm

In the following we show and describe the algorithm (Algorithm 1).

Algorithm 1 Estimate $sim(\cdot, \cdot)$

function INITIALIZE(G)
Require: $G = (V, E)$, with $n = |V|$
    $\widehat{G} \leftarrow (\infty)_{k \times n}$
    $\mathbf{h} \leftarrow$ pick randomly k hash function from a Universal family
    for edge $(i, j) \in E$ **do**
        for $l : 1, \ldots, k$ **do**
            %% Update i's neighborhood with the hash of j
            $\widehat{G}[l, i] \leftarrow \min(\mathbf{h}[l](j), \widehat{G}[l, i])$
            %% Update j's neighborhood with the hash of i
            $\widehat{G}[l, j] \leftarrow \min(\mathbf{h}[l](i), \widehat{G}[l, j])$
        end for
    end for
    %% At the end, for all columns $\widehat{G}[\cdot, i]$, we'll have C_i as in (1)
end function

function SIMILARITY(u, v)
Require: $\widehat{G}$ initialized
    Y $\leftarrow \{l : \widehat{G}[l, u] = \widehat{G}[l, v]\}$
    return $\frac{|Y|}{k}$
end function

The function INITIALIZE initializes the data structure described in Section 2.2: it scans the whole edge list and, for each (i, j) encountered, updates both columns C_i and C_j which contain, respectively, as explained in (1),

the min-hash of the i's and j's neighborhood, computed for each l_{th} hash function, $l = 1, \ldots, k$.

The function SIMILARITY estimates the similarity of nodes u and v by simply counting the common min-hash values stored in the data structure $\widehat{G}$, Y, scaled by the total number of hash functions stored in $\mathbf{h}$, where $|\mathbf{h}| = k$. In symbols:

$$\hat{J}(u, v) = \frac{|\{l : h^l_{\min}(\Gamma(u)) = h^l_{\min}(\Gamma(v))\}|}{k}$$

Note: in the algorithm we use many hash functions. There exists a variant that uses a single hash function; in every column i, we can store the k smallest hash of $\Gamma(i)$, all of them computed with the unique h. Both approaches are equivalent, although the latter is slightly better in terms of performances.

2.4 Accuracy

By Chernoff Bounds [2], the expected error of the estimated Jaccard similarity is $o(1/\sqrt{k})$. More precisely, if we want an (ϵ, δ) approximation, we need to choose $k = O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$.

3 Problem 3

This problem is a main one in the field of streaming algorithm: *Count-distinct problem* [7]. A well-known simple algorithm exists for such kind of problems, and its name is "Flajolet-Martin Algorithm"[4]. In the following, we will design an algorithm in a distributed fashion (due to the scenario of application) that relies on the Flajolet-Martin Algorithm.

Algorithm 2 implements the module DistinctElementsCounter that every

---

Module:

Name: DistinctElementsCounter, **instance** c.

Events:

Request: $<c$, *Update* $|id>$: Update the estimate with the new *id*.

Properties:

C1: *Liveness*: Eventually, every update will be taken into account in the estimate.

Algorithm 2 Estimate number of distinct robot

```
1:  Implements: DistinctElementsCounter, instance c
2:
3:  upon event ⟨ c, Init ⟩ do:
4:      E := [0]^{K×L}, the matrix of the estimates;
5:      H := K × L different hash function;
6:
7:  upon event ⟨ c, Update| id ⟩ do:
8:      forall i = 1, ..., K do:
9:          forall j = 1, ..., L do:
10:             R = number of trailing zeros in H[i][j](id)
11:             E[i][j] = max(E[i][j], R)
```

robot need to have installed in its software. The event *Update* is triggered whenever the robot's sensors detect another robot in the nearest. The called procedure simply implements the Flajolet-Martin schema: at the end, all the estimates (i.e. the maximum number of trailing zeros of the hashes on the encountered ids) are stored in the matrix $\mathcal{E}$. In order to answer to the query "Number of distinct fellow robots", the procedure is the following:

1. Compute the means of all the K groups of L elements, stored in $\mathcal{E}$, considering a generic element in the following way:

$$e_{ij} = 2^{\mathcal{E}[i][j]}$$

2. Compute the medians among all the means computed in the step 1.

This trick is very important, since the presence of outliers (i.e. very high values of zeros of the hash values) is very likely. The median operation achieves variance reduction, and the mean averages over all the medians, in order to avoid that all the estimates are in the form 2^r, for some integer r.

3.1 Accuracy estimation

Considering [1], we know that in order to reach an (ϵ, δ) approximation, we need $O(\frac{1}{\epsilon^2} \cdot \log \frac{1}{\delta} \log m))$ space, where $O(\frac{1}{\epsilon^2} \log m)$ is the space required for the algorithm, and $\log \frac{1}{\delta}$ is the number of times we need to run the algorithm in order to achieve probability guarantee.

4 Problem 4

This problem is known in literature as *Heavy Distinct Hitter Problem*, and the (ϵ, δ)-*approximate* version, the *Approximate Heavy Distinct Hitter Problem*, as explained in [6]. We will report here the definition:

Definition 1. ***Approximate Heavy Distinct Hitter Problem****. Given a stream S, parameters $\epsilon, \delta \in (0,1)$, and a threshold value $T > 0$, output a set $\mathcal{L}$ of pairs $(e, \tilde{w}(e))$ for which it holds that:*

1. *if element e is in $\mathcal{L}$, then $w(e) \geq (1-\epsilon)T$*
2. *if element e is not in $\mathcal{L}$, then $w(e) < (1+\epsilon)T$*
3. *for all $(e, \tilde{w}(e)) \in \mathcal{L}$ we have that $|w(e) - \tilde{w}(e)| \leq \epsilon T$*

with probability at least $1 - \delta$.

The adopted solution will use algorithmic tools from [3] but, instead of considering mere counting of items, we will use a *distinct counter primitive*, (in particular [4]). This construction is called CM-FM, as described in [5].

4.1 Notation

Now we clarify the use of some notation that will be used in the following sections. Note: we omit the dependency from time t in order to simplify the notation.

- N is the number of users, M is the number of items;
- S is the multiset of pairs (i, j), where i is an user and j is an item that he purchased;
- $\bar{m}$ is the number of distinct items until now.
- $B_i = \{(i, j) | (i, j) \in S\}$ is the *set* of distinct item that an user i bought.
- $f_i = |B_i|$, i.e. the number of distinct item that an user purchased.
- $\mathcal{I} = \{j : (i, j) \in S\}$, the set of item purchased until now. Notice that $|\mathcal{I}| = \bar{m}$
- $\hat{m}$ is the estimate of the number of distinct items seen so far.

4.2 Solution: Brief description

The main issue here is about memory: we cannot store N counter for every user, in order to determine f_i for every user and then the ϕ-heavy hitters. This is the main reason why we need a *Count-Min* sketch data structure. Moreover, differently from [3], since we need to count the number of distinct item per user, we'll use a *distinct counting primitive*, such as the Flajolet-Martin sketch [4]: in particular, since we need strong precision guarantees, we'll use its extended version fully described in [1].

To summarize, the steps are:

1. Initialize a FM counter for counting the number of distinct item purchased, without considering users. In other words, $FM_M = \hat{m} \approx |\mathcal{I}| = \bar{m}$;

2. Initialize a Count-Min sketch matrix CM with a certain depth d and width w (depending on the desired accuracy, as we will se later), and for each cell use a FM_{rc} counter ($r = 1, \ldots, d$, $c = 1, \ldots, w$);

3. For every incoming transaction (i, j), update FM_M and all the $CM[r, h_r(i)]$ counters ($r = 1, \ldots, d$) with an *insert()* operation (in the same fashion of [6]);

4. The *point query* $\mathcal{Q}(i)$ (where i is an user) will be simply:

$$\mathcal{Q}(i) = \min_{r \in \{1,\ldots,d\}} CM[r, h_r(i)].getNumberDistinct()$$

5. The general update procedure (called for every incoming transaction) for maintain the output set $\mathcal{L}$ of heavy hitters is the following: update the Count-Min sketch inserting , after value is greater than $\phi\hat{m}$, keep /insert it in the output set $\mathcal{L}$; otherwise don't. Moreover, scan all the other current heavy-hitters and check that they can still be in the output set.

4.3 Solution Architecture

In this section we will describe more in details all algorithmic tools and data structures used in our solution. We will start from $FMsketch$ components, then the CM-FM data structures that allow point queries $\mathcal{Q}$ already defined and finally the update and query algorithm, which is actually the same of the algorithm we saw in class.

4.3.1 FM sketches

First of all, a brief focus on FM sketch [4]. From [1], we know that an extended version of the FM sketch can provide an $(\epsilon_{fm}, \delta_{fm})$-approximation for the distinct counting problem, using:

- **Space**: $t = \log(1/\epsilon_{fm}^2)$ elements to store, each of them requiring $O(\log m)$ space; $O(\log \frac{1}{\delta_{fm}})$ instances of the algorithm to run in parallel and taking the median, in order to boosting the probability (as usual);
- **Time**: using a balanced binary search tree for store the t smallest values, each step requires time $O(\log(1/\epsilon_{fm}) \cdot \log M)$

So, we have space $O(\frac{1}{\epsilon_{fm}^2} \log(\frac{1}{\delta_{fm}}) \log M)$ and time $O(\log(1/\epsilon_{fm}) \cdot \log M)$ for every FM sketch used.

4.3.2 CM-FM sketch

What we are going to prove now is that our CM-based construction (namely, Count-Min Flajolet-Martin sketch, CM-FM) will provide a good estimate of the distinct item seen so far for each user $i = 1, \dots N$ or, in other words, that a *point query* $\mathcal{Q}(i)$, where i denote the user, will return $\hat{f}_i$ (our estimate of f_i, the number of distinct purchased item by the user i) with the usual (ϵ, δ)-approximation requirements. The following theorem, similar to Theorem 2 in [5], will take into account the error bounds for the estimates and ensure us about the correctness of our approach.

Theorem 1. *For a CM-FM sketch with* $w = \lceil (1+\epsilon_{fm})e/\epsilon_{cm} \rceil$, $d = \lceil \ln(1/\delta_{cm}) \rceil$ *and user defined constants* $(\epsilon_{fm}, \epsilon_{cm}, \delta_{fm}, \delta_{cm})$, *let* f_α *be the exact number of distinct element of user* $\alpha \in [N]$, *and* $\hat{f}_\alpha$ *be the estimated number computed using the sketch. Then, with probability at least* $(1-\delta_{fm})^d : \hat{f}_\alpha \geq (1-\epsilon_{fm}) f_\alpha$ *and with probability at least* $(1-\delta_{fm})^d(1-\delta_{cm}) : \hat{f}_\alpha \leq (1+\epsilon_{fm}) f_\alpha + \epsilon_{cm}|\bar{m}|$.

Proof. The proof is divided into two parts: the one which proves the lower bound and the other one which proves the upper bound. It is strongly inspired by [3] and [5], with some difference on the errors introduced by the collisions of the Count-Min sketch and on the probability guarantees.

Upper bound Let define an indicator variable $I^i_{\alpha,\beta}$ as follows:

$$I^i_{\alpha,\beta} = \begin{cases} 1, & \alpha \neq \beta \wedge h_i(\alpha) = h_i(\beta) \\ 0, & otherwise \end{cases}$$

Now we introduce the following random variable:

$$X_\alpha^i = \sum_{\beta=1}^{N} I_{\alpha\beta}^i |\mathcal{I}_{\beta\setminus\alpha}|$$

Where $\mathcal{I}_{\beta\setminus\alpha} = \mathcal{I}_\beta \setminus \mathcal{I}_\alpha$, and $\mathcal{I}_u = \{j|(u,j) \in B_u\}$. In other words, this random variable quantifies the number of extra insertions due to collisions, which happen, for a generic row i, when both the following conditions hold:

1. $h_i(\alpha) = h_i(\beta)$, otherwise it means that β refers to another "counter", or "cell of the matrix".

2. $\mathcal{I}_\beta \setminus \mathcal{I}_\alpha \neq \emptyset$, i.e. there exists an item that user β purchased but α do not; otherwise the counter "recognizes" that it already "saw" that item and will "ignore" it[2]

This is a non-negative random variable.

Assuming that the accuracy of every FM sketch is $\epsilon_{fm}, \delta_{fm}$, we have that each estimate is bounded by:

$$(1-\epsilon_{fm})Y \leq \widehat{Y} \leq (1+\epsilon_{fm})Y$$

with probability $1-\delta_{fm}$. So, we have[3]:

$$CM[i,h_i(\alpha)].getNumberDistinct() \geq^{1-\delta_{fm}} (1-\epsilon_{fm})(f_\alpha + X_\alpha^i) \geq (1-\epsilon_{fm})f_\alpha$$

and using the following as estimation function:

$$\hat{f_\alpha} = \min_{i\in\{1,\ldots,d\}} CM[i,h_i(\alpha)].getNumberDistinct()$$

the lower bound follows. Notice that the lower bound succeeds with probability $(1-\delta_{fm})^k$, i.e., the probability that the estimates of all FM sketches will be correct.

[2] In other words, this is due to *duplicate insertions insensitiveness*. It is not a completely "sufficient" condition for an increment of our estimate, but for our use of that random variable, it is ok.

[3] In $(f_\alpha + X_\alpha^i$, the "+" has a slightly different meaning than the usual. It should be interpreted as an "eventual" addition to the true estimate, since it may happen that the current insertion of an item has no effect on the estimate, even if the FM sketch has never seen that item until now. As said before, since X_α^i is non-negative, this construction it's enough for our purposes.

Upper bound To prove upper bound, assuming perfect hash functions. It holds that:

$$E[I^i_{\alpha,\beta}] = Pr[h_i(\alpha) = h_i(\beta)] \leq \frac{1}{w}$$

and notice that:

$$E[X^i_\alpha] = \sum_{\beta=1}^{N} E[I^i_{\alpha\beta}] \cdot |\mathcal{I}_{\beta\setminus\alpha}| \leq \frac{1}{w}\bar{m}$$

since $\bar{m} = |\mathcal{I}| = |\mathcal{I}_\alpha| + \sum_{\beta=1}^{N} |\mathcal{I}_{\beta\setminus\alpha}|$. From the above considerations, it follows that:

$$Pr[\hat{f}_\alpha > (1+\epsilon_{fm})f_\alpha + \epsilon_{cm}\bar{m}] =$$

$$Pr[\forall i : CM[i, h_i(\alpha)].getNumberDistinct() > (1+\epsilon_{fm})f_\alpha + \epsilon_{cm}\bar{m}] \leq^{(1-\delta_{fm})^k}$$

$$Pr[\forall i : (1+\epsilon_{fm})(f_\alpha + X^i_\alpha) > (1+\epsilon_{fm})f_\alpha + \epsilon_{cm}\bar{m}] \leq$$

$$Pr[\forall i : (1+\epsilon_{fm})(f_\alpha + X^i_\alpha) > (1+\epsilon_{fm})f_\alpha + \epsilon_{cm}wE[X^i_\alpha]] =$$

$$Pr[\forall i : X^i_\alpha > \frac{\epsilon_{cm}}{1+\epsilon_{fm}}wE[X^i_\alpha]] \leq \left(\frac{1+\epsilon_{fm}}{\epsilon_{cm}w}\right)^d$$

Now, by setting $e = \frac{\epsilon_{cm}w}{1+\epsilon_{fm}}$ we get:

$$Pr[\hat{f}_\alpha \leq (1+\epsilon_{fm})f_\alpha + \epsilon_{cm}\bar{m}] \leq 1 - e^{-d}$$

Thus, $\delta_{cm} = e^{-d} \implies d = \ln(\frac{1}{\delta_{cm}})$. Notice that the above holds only if all the FM sketches do not fails. So, the estimation will succeeds with probability $(1-\delta_{fm})^d(1-\delta_{cm})$.

□

The above result ensure us that, tuning $(\epsilon_{fm}, \epsilon_{cm}, \delta_{fm}, \delta_{cm})$ we can reach any accuracy at the cost of:

- **Space**: $O(dw) = O(\frac{1+\epsilon_{fm}}{\epsilon_{cm}} \log \frac{1}{\delta_{cm}})$ number of cells, each of them with cost $O(\frac{1}{\epsilon^2_{fm}} \log(\frac{1}{\delta_{fm}}) \log M)$. Join them together we have

$$O(\frac{1}{\epsilon^2_{fm}\epsilon_{cm}} \log \frac{1}{\delta_{fm}} \log \frac{1}{\delta_{cm}} \log M)$$

- **Time**: $O(\log(1/\epsilon_{fm}) \cdot \log M)$ for update one FM sketch. Joining it with the number of insertions per update, which is $O(d) = O(\log(\frac{1}{\delta_{cm}}))$, we have:
$$O(\log \frac{1}{\epsilon_{fm}} \log \frac{1}{\delta_{cm}} \cdot \log M)$$

So, how to choose $(\epsilon_{fm}, \epsilon_{cm}, \delta_{fm}, \delta_{cm})$? If we want an $(\tilde{\epsilon}, \tilde{\delta})$-approximation of our CM-FM, we need:

- $\tilde{\epsilon} = \frac{\epsilon_{cm}}{1+\epsilon_{fm}}$

- $1 - \tilde{\delta} \simeq \frac{1}{\delta_{cm}} e^{\frac{1}{\delta_{fm}}}$, because for both lower and upper bounds we can impose[4]:
$$1 - \tilde{\delta} = (1 - \delta_{fm})^{\ln \frac{1}{\delta_{cm}}} \cdot (1 - \delta_{cm}).$$
Using $1 - x \leq e^{-x}$ (or even $1 - x \approx e^{-x}$ for $x \to 0$) we can rewrite that expression:
$$e^{\frac{1}{\delta_{fm}} \ln \frac{1}{\delta_{cm}}} (1 - \delta_{cm}) = \frac{1 - \delta_{cm}}{\delta_{cm}} e^{\frac{1}{\delta_{fm}}} \approx \frac{1}{\delta_{cm}} e^{\frac{1}{\delta_{fm}}}$$
Whose form is more amenable.

Finally, once we defined the point query $\mathcal{Q}(i)$ with the desired $(\tilde{\epsilon}, \tilde{\delta})$-approximation, we can estimate the desired value as the minimum among all the estimates, i.e.:

$$\hat{f}_i = \min_{j \in \{1,\ldots,d\}} CM[j, h_j(i)].getDistinctNumber()$$

4.3.3 Update and query algorithm for heavy hitters

Over this building block, we can simply apply our knowledge on CM-sketch for the heavy hitters problem, using the result in Theorem 6 [3]: by construction, the elements retrieved in this way will be the ones which fulfill the requirements defined in Definition 1. The space will be:

$$O(\frac{1}{\epsilon_{fm}^2 \epsilon_{cm}} \log \frac{1}{\delta_{fm}} \log \frac{N}{\delta_{cm}} \log M)$$

[4]Notice that, actually, for the lower bound is only $(1 - \delta_{fm})^{\ln \frac{1}{\delta_{cm}}}$. However, since it is a "success" probability, we can consider the same value used for the upper bound, which is a lower value (i.e. $(1 - \delta_{fm})^{\ln \frac{1}{\delta_{cm}}} \cdot (1 - \delta_{cm})$) and does not affect the validity of our arguments, but strongly simplify our analysis.

and the time per update is:

$$O(\log \frac{1}{\epsilon_{fm}} \log \frac{N}{\delta_{cm}} \cdot \log M)$$

The only important remark to do here is about the estimate of $\bar{m}$, $\hat{m}$, which is needed to retrieve all ϕ-heavy hitters[5]. In order to estimate $\hat{m}$, we need to initialize the FM_M and insert in it every incoming transaction.

Also for the FM_M counter we have to choose the pair (ϵ_M, δ_M). We can choose it as $(\epsilon_{fm}, \delta_{fm})$, or even a different one: for reasonable choices it does not affect the global space needs.

Hence, the global accuracy depends also from $(1 - \delta_M)$ and ϵ_M, since they influence the probability that the conditions in Definition 1 hold. For include them, simply merge with the other parameters of the solution.

The algorithm: The *Update and query* algorithm is the same of the one we saw during the class. The only main difference is the following: if an update does not affect $\hat{m}$, we do not need to check that the condition for the users in the output set $\mathcal{L}$ still holds, because it is surely the case.

[5]Recall that, in our scenario, an user is a ϕ-heavy hitter if $f_i \geq \phi\bar{m}$, i.e. $\hat{f}_i \geq \phi\hat{m}$

5 Problem 5

You will find the solution for this problem in the `.zip` provided in the email.

5.1 Algorithm description

5.1.1 Subquestions (a) and (b)

In the following all the map-reduce steps:

1. From the list of lines, transform each of them in lower case and without punctuations. Then split into a list of words.

   ```
   map(lambda line:  re.sub("[^ a-z]", "", line.lower()).split())
   ```

2. From a list of splitted lines (list of list of words), compute a list of pair (word1, [word11, word12,...]), where word1x appear in the same line of word1. Notice that we need to remove word1 in the current line to avoid to count it (this explains why we need `enumerate`: we use the index of the word in order to remove it when we form the combinations).

   ```
   flatMap(lambda word_list:
map(lambda x:  (x[1], word_list[:x[0]] + word_list[x[0] + 1:]),
enumerate(word_list)))
   ```

3. From a list of pair (word1, [word11, word12,...]) to a "flat" list of pair (word1, word11), (word1, word12).

   ```
   flatMapValues(lambda x:  x)
   ```

4. We're almost done. Add a "1" as value in every pair, making the word pairs as key.

   ```
   map(lambda x:  (x, 1))
   ```

5. Reduce by key.

6. Make word1 as key (remember: word1 is the word whose we want to know the counts). In detail: from a list of ((word1, word1x), occurrences) to a list of (word1, (word1x, occurrences)).

   ```
   map(lambda x:  (x[0][0], (x[0][1], x[1])))
   ```

7. Group by key and return the top-10, after sorted the list obtained.

5.1.2 Subquestions (c) and (d)

The only difference with the previous algorithm is only on the step number 2: hence, I will not report every step again.

- From a list of splitted lines (list of list of words), compute a list of pair (word1, [word11, word12,...]), where word1x appears after x positions, up to a maximum of k, after word1 (it is done thanks to `enumerate` and slice operations over list).

```
    flatMap(lambda wordlist:
map( lambda x:  (x[1], wordlist[x[0] + 1:x[0] + 1 + k]) if x[0]
+ 1 < len(wordlist) else (x[1], ""), enumerate(wordlist)))
```

5.2 Implementation

5.2.1 Project structure

- in `src/` you will find the solution for every subquestion, as a python module. The algorithms are in the functions called `execute_map`;
- in `data/` there is simply the input data provided by you;
- in `out/` there are all the outputs from every subquestion, in two format:
  - `.txt`: the list of words and, for each of them, a list of pairs (word, occurrences), one per line. i.e.:

    ```
    word_1\n
    \t word_11 \t occurrences_11
    \t word_12 \t occurrences_12
    ...
    word_2\n
    \t...
    ```

  - `.dict`: the result in a Python `dict` object, dumped through `pickle`. It may be useful for testing.

The structure of the solution is the following:

```
.
|-- README
|-- conf.py
|-- execute_all_subquestions.sh
|-- main.py
```

```
|-- data
|   `-- pg674.txt
|-- out
|   |-- a.dict
|   |-- a.txt
|   |-- b.dict
|   |-- b.txt
|   |-- c.dict
|   |-- c.txt
|   |-- d.dict
|   `-- d.txt
`-- src
    |-- utils.py
    |-- a.py
    |-- b.py
    |-- c.py
    `-- d.py
```

5.2.2 How it works

You should call the algorithms through the `main.py` script, in the following way:

```
$python main.py (a|b|c|d) <input_file>
```

It will pick the `OUTPUT_FOLDER` path in `conf.py` (default: `out`) and will call the script `src/<subquestion>.py`.

5.2.3 How to run it

If you want to run all the solution (i.e. every subquestions): Just call the script `execute_all_subquestions.sh`

If you want to run only one subquestion: e.g., for subquestion `a`:

```
$python main.py a data/pg674.txt
```

References

[1] Ziv Bar-Yossef et al. "Counting Distinct Elements in a Data Stream". In: *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques*. RANDOM '02. London, UK, UK: Springer-Verlag, 2002, pp. 1–10. ISBN: 3-540-44147-6. URL: http://dl.acm.org/citation.cfm?id=646978.711822.

[2] Herman Chernoff. "A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the sum of Observations". In: *Ann. Math. Statist.* 23.4 (Dec. 1952), pp. 493–507. DOI: 10.1214/aoms/1177729330. URL: http://dx.doi.org/10.1214/aoms/1177729330.

[3] Graham Cormode and S. Muthukrishnan. "An Improved Data Stream Summary: The Count-min Sketch and Its Applications". In: *J. Algorithms* 55.1 (Apr. 2005), pp. 58–75. ISSN: 0196-6774. DOI: 10.1016/j.jalgor.2003.12.001. URL: http://dx.doi.org/10.1016/j.jalgor.2003.12.001.

[4] Philippe Flajolet, G. N. Martin, and G. Nigel Martin. *Probabilistic Counting Algorithms for Data Base Applications*. 1985.

[5] Marios Hadjieleftheriou, John W. Byers, and George Kollios. *Robust sketching and aggregation of distributed data streams*. Tech. rep. 2005.

[6] Thomas Locher. "Finding Heavy Distinct Hitters in Data Streams". In: *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '11. San Jose, California, USA: ACM, 2011, pp. 299–308. ISBN: 978-1-4503-0743-7. DOI: 10.1145/1989493.1989541. URL: http://doi.acm.org/10.1145/1989493.1989541.

[7] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. New York, NY, USA: Cambridge University Press, 2011. ISBN: 1107015359, 9781107015357.

[8] Ahmad Rawashdeh and Anca L. Ralescu. "Similarity Measure for Social Networks - A Brief Survey". In: *MAICS*. 2015.