**Big Data Analytics: HOMEWORK 2**
**Professor: Mohamed Ndaoud**
**DUE ON: Monday, NOVEMBER 27.**

---

The homework can be done in groups of 5 students. Please type your solutions (LaTeX and code) on a Jupyter Notebook that you will upload on Moodle. Make sure to write the names of the 5 students of your group in the top of your notebook. It is enough that one students uploads the work of the whole group.
In case you struggle writing Latex on a Jupyter notebook, you can return a (single) pdf file that includes both the executed code and your solutions to the theoretical questions.

## Problem 1, 30 points: (On the Flajolet-Martin Algorithm)

The goal of this algorithm is to compute an approximation of the number of distinct elements in a stream. Consider a stream $S$ with $N$ items. Let $F$ be the number of distinct items in $S$, our goal is to estimate $F$. In order to do so, we have access to a perfect hash function $h$ such that for all $k$ in $S$, $h(k) \sim \text{Unif}[0, 2^w - 1]$ where $w = \lfloor \log(N) \rfloor$. In particular for $k_1 \neq k_2$ we have that $h(k_1)$ and $h(k_2)$ are independent.

We follow the following scheme:

- Let $z_k$ be the number of 0 at the tail of the binary representation of $h(k)$.
- $Z = \max_{k \in S} z_k$.
- $\tilde{F} = 2^Z$.

For example, if $w = 5$ and $h(k) = 4 = (00100)_2$, $z_k = 2$. We want to prove that

$$\forall c \geq 3, \quad \mathbb{P}\left(1/c \leq \frac{\tilde{F}}{F} \leq c\right) \geq 1 - 3/c.$$

1. Show that for all $r \in [0, w]$, $\mathbb{P}(z_k \geq r) = 2^{-r}$.

   For $r \in [0, w]$ and $k \in S$, we define the random variable $X_k(r) = \mathbf{1}(z_k \geq r)$ and $X(r) = \sum_{\text{distinct } k \in S} X_k(r)$.

2. Compute the expectation and variance of $X_k(r)$ and $X(r)$.

   Let $r_1$ be the smallest integer $r$ such that $2^r > cF$ and $r_2$ the smallest integer such that $2^r \geq F/c$.

3. Prove that the scheme is correct if $X(r_1) = 0$ and $X(r_2) \neq 0$.

4. Show that $\mathbb{P}(X(r_1) \geq 1) \leq 1/c$.

5. Show that $\mathbb{P}(X(r_2) = 0) \leq 2/c$.

6. Conclude.

## Problem 2, 40 points: (Bloom Filters)

For this problem you need to install bloom_filter, a Python library which offers an implementation of Bloom filters. Run the following code *!pip install bloom_filter*.

From the NLTK (Natural Language ToolKit) library, you need to import a large list of English dictionary words, commonly used by the very first spell-checking programs in Unix-like operating systems.

```
import nltk
nltk.download('words')
from nltk.corpus import words
word_list = words.words()
print(f'Dictionary length: len(word_list)')
print(word_list[:15])
```

Then you will load another dataset from the NLTK Corpora collection: movie_reviews. The movie reviews are categorized between positive and negative, so we construct a list of words (usually called bag of words) for each category.

```
from nltk.corpus import movie_reviews
nltk.download('movie_reviews')

neg_reviews = []
pos_reviews = []

for fileid in movie_reviews.fileids('neg'):
    neg_reviews.extend(movie_reviews.words(fileid))
for fileid in movie_reviews.fileids('pos'):
    pos_reviews.extend(movie_reviews.words(fileid))
```

The goal is to develop a very simplistic spell-checker. By no means you should think of using it for a real-world use case, but it is an interesting exercise to highlight the strenghts and weaknesses of Bloom Filters!

1. Complete the following code to create word_filter a BloomFilter of the words in word_list, also define word_set as the corresponding Python set of word_list.

   ```
   from bloom_filter import BloomFilter
   word_filter = BloomFilter(max_elements=236736)
   word_filter = ...
   word_set = ...
   ```

   You now have 3 different variables in your scope:

   - word_list, a Python list containing the English dictionary (in case insensitive order)
   - word_filter, a Bloom filter where we have already added all the words in the English dictionary
   - word_set, a Python set built from the same list of words in the English dictionary.

2. Using "*from sys import getsizeof*" inspect the size of each of the above data structures and comment.

   Now let's find out how fast is the main operation for which we construct Bloom filters: membership testing. To do so, we will use the timeit IPython magic command, which times the repeated execution of a single Python statement. For example using the code:
   *%timeit -r 3 "California" in word_filter.*

3. Compare the execution time of the previous data structures and comment.

   We now have all the building blocks required to build our spell-checker, and we understand the performance tradeoffs of each data structure we chose.

4. Write a function that takes as arguments (1) a list of words, and (2) any of the 3 dictionary data structures we constructed. The function must return the number of words which do not appear in the dictionary. Check the movie reviews using your code.

## Problem 3, 30 points: (Dead ends in PageRank computations)

Let the matrix of the Web $M$ be an $n$ by $n$ matrix, where $n$ is the number of Web pages. The entry $m_{ij}$ in row $i$ and column $j$ is 0, unless there is an arc from node (page) $j$ to node $i$. In that case, the value of $m_{ij}$ is $1/k$, where $k$ is the number of arcs (links) out of node $j$. Notice that if node $j$ has $k > 0$ arcs out, then column $j$ has $k$ values of $1/k$ and the rest 0's. If node $j$ is a dead end (i.e., it has zero arcs out), then column $j$ is all 0's.

Let $r = [r_1, r_2, ..., r_n]^T$ be (an estimate of) the PageRank vector; that is, $r_i$ is the estimate of the PageRank of node $i$. Define $w(r)$ to be the sum of the components of $r$; that is $w(r) = \sum_i r_i$.

In one iteration of the PageRank algorithm, we compute the next estimate $r'$ of the PageRank as: $r' = Mr$. Specifically, for each $i$ we compute $r'_i = \sum_j m_{ij} r_j$ .

1. Suppose the Web has no dead ends. Prove that $w(r') = w(r)$.

2. Suppose there are still no dead ends, but we use a teleportation probability of $1 - \beta$, where $0 < \beta < 1$. The expression for the next estimate of $r_i$ becomes $r'_i = \beta \sum_j m_{ij} r_j + (1 - \beta)/n$.

   Under what circumstances will $w(r') = w(r)$? Prove your conclusion.

3. Now, let us assume a teleportation probability of $1 - \beta$ in addition to the fact that there are one or more dead ends. Call a node "dead" if it is a dead end and "live" if not. Assume $w(r) = 1$. At each iteration, each live node $j$ distributes $(1 - \beta) r_j / n$ PageRank to each of the other nodes, and each dead node $j$ distributes $r_j / n$ PageRank to each of the other nodes.

   Write the equation for $r'_i$ in terms of $\beta, M, r, n$ and $D$ (where $D$ is the set of dead nodes). Then, prove that $w(r')$ is also 1.

## Problem 4, 60 points: (Implementing PageRank and HITS)

In this problem, you will learn how to implement the PageRank and HITS algorithms in Spark. You will be experimenting with a small randomly generated graph (assume graph has no dead-ends) provided on Moodle "graph.txt".

It has $n = 1000$ nodes (numbered 1, 2, …., 1000), and m = 8192 edges, 1000 of which form a directed cycle (through all the nodes) which ensures that the graph is connected. It is easy to see that the existence of such a cycle ensures that there are no dead ends in the graph. There may be multiple directed edges between a pair of nodes, and your solution should treat them as the same edge. The first column in graph.txt refers to the source node, and the second column refers to the destination node.

## PageRank Implementation

Assume the directed graph $G = (V, E)$ has $n$ nodes (numbered 1, 2, …, n) and m edges, all nodes have positive out-degree, and M is a an n × n matrix as defined in class such that for any i, j $M_{ji} = 1/deg(i)$ if $(i \to j) \in E$.

Here, $deg(i)$ is the number of outgoing edges of node $i$ in $G$. If there are multiple edges in the same direction between two nodes, treat them as a single edge. By the definition of PageRank, assuming $1 - \beta$ to be the teleport probability, and denoting the PageRank vector by the column vector $r$, we have the following equation:

$$r = \frac{(1-\beta)}{n}\mathbf{1}_n + \beta M r,$$

where $\mathbf{1}_n$ is the vector with all entries equal to 1. Based on this equation, the iterative procedure to compute PageRank works as follows:

1. Initialize: $r^0 = \frac{1}{n}\mathbf{1}_n$.
2. For $i$ from 1 to $k$, iterate: $r^i = \frac{(1-\beta)}{n}\mathbf{1}_n + \beta M r^{i-1}$.

Run the aforementioned iterative process in Spark for 40 iterations (assuming $\beta = 0.8$) and obtain the PageRank vector $r$. The matrix M can be large and should be processed as an RDD in your solution. Compute the following:

- List the top 5 node ids with the highest PageRank scores.
- List the bottom 5 node ids with the lowest PageRank scores.

## HITS Implementation

Assume the directed graph $G = (V, E)$ has n nodes (numbered 1, 2, …, n) and m edges, all nodes have non-negative out-degree, and L is a an n × n matrix referred to as the link matrix such that for any i, j $L_{ij} = 1$ if $(i \to j) \in E$.

Given the link matrix $L$ and some scaling factors $\lambda$, $\mu$, the hubbiness vector $h$ and the authority vector $a$ can be expressed using the equations:

$$h = \lambda L a, \qquad a = \mu L^T h.$$

Based on this equation, the iterative method to compute $h$ and $h$ is as follows:

1. Initialize $h$ with a column vector (of size $n$) of all 1's.
2. Compute $a = \mu L^T h$ and scale so that the largest value in the vector $a$ has value 1.
3. Compute $h = \lambda L a$ and scale so that the largest value in the vector $h$ has value 1.

4. Go to step 2.

Repeat the iterative process for 40 iterations, assume that $\lambda = 1, \mu = 1$ and then obtain the hubbiness and authority scores of all the nodes (pages). The link matrix $L$ can be large and should be processed as an RDD. Compute the following:

- List the 5 node ids with the highest hubbiness score.
- List the 5 node ids with the lowest hubbiness score.
- List the 5 node ids with the highest authority score.
- List the 5 node ids with the lowest authority score.

## Problem 5, 40 points: (PageRank for Sports Analytics)

Data for this exercise comes from the NCAA (National Collegiate Athletic Association) 2019 men's basketball regular (and post) seasons(s), limited to the ACC (Atlantic Coast Conference). The goal is to rank the teams applying the Pagerank algorithm.

1. Start by reading the data from "NCAA.csv" file available on Moodle. Take a look at the data.
2. We need to pre-process the data in such a way that PageRank can receive it as an argument. We will encode patterns of wins and losses into a matrix. There are different approaches to it. For this exercise we will follow the one proposed in this blog:`https://towardsdatascience.com/python-pagerank-meets-sports-analytics-28e4d395af57`. In this approach, we iterate over every regular season game, and find the point differential.

   For example, Team A beat Team B by 20 points; ergo Team B gives 20 points credibility to Team A. If two given teams played each other twice, and the same team won both times, the edge connecting them would be the average of their winnings:

   - For each GameID, find difference between scores and send it to a list.
   - Using this difference, build edges between teams. We need a directed graph. If team A beat Team B by 20 points, then Team B gives 20 points to Team A. Create a list edges with (loser,winner,points).
   - Install the igraph package. Using this package and edges, build a weighted and directed graph: game_graph.
3. Run the following code that computes and displays the ranking of the teams:

   *import operator*
   *vectors = game_graph.pagerank() #creates the vector of rankings*
   *e = {name:cen for cen, name in zip([v for v in vectors],game_graph.vs['name'])} #we create a dict. with the names and scores*
   *sorted_eigen = sorted(e.items(), key=operator.itemgetter(1),reverse=True) #we sort the teams accordingly the rankings*
   *sorted_eigen*