Stat4DS / Homework 01

Pierpaolo Brutti

Due Wednesday, October 30, 2019, 23:59 PM on Moodle

General Instructions

I expect you to upload your solutions on Moodle as a **single running** `R Markdown` file (`.rmd`) + its `html` output, named with your surnames.

You will give the commands to answer each question in its own code block, which will also produce plots that will be automatically embedded in the output file. Your responses must be supported by both textual explanations and the code you generate to produce your results. *Just examining your various objects in the "Environment" section of RStudio is insufficient – you must use scripted commands and functions.*

R Markdown Test

To be sure that everything is working fine, start `RStudio` and create an empty project called `HW1`. Now open a new `R Markdown` file (`File > New File > R Markdown...`); set the output to `HTML mode`, press `OK` and then click on `Knit HTML`. This should produce a web page with the knitting procedure executing the default code blocks. You can now start editing this file to produce your homework submission.

Please Notice

- For more info on `R Markdown`, check the support webpage that explains the main steps and ingredients: R Markdown from RStudio.
- For more info on how to write math formulas in LaTex: Wikibooks.
- Remember our **policy on collaboration**: *Collaboration on homework assignments with fellow students is **encouraged**. However, such collaboration should be clearly acknowledged, by listing the names of the students with whom you have had discussions concerning your solution. You may **not**, however, share written work or code after discussing a problem with others. The solutions should be written by **you**.*

Exercise 1: So unfair to go first...

1. The game

At first glance some games of chance seem completely fair and not biased in any way, but in fact, if you think a bit harder, it may be the case that you can always select a strategy able to turn the odds in your favor.

Consider the following two player game based on a pack of 52 ordinary cards. We are interested only in their colour, that of coursse can be `Red` (R) or `Black` (B).

At the start of a game each player announces, one player after the other, a three colour sequence he/she will be watching for the whole game. For example, `Player-1` go first and picks RBR, and then `Player-2` selects RRB.

At this point we start drawing cards from the deck, one by one, and every time `Player-1` or `Player-2` sequence of cards appears, all those cards are removed from the game as a "winning trick". Keep going with the example, if the following five cards are dealt

$$\mathsf{R}\,\mathsf{B}\,\mathsf{B}\,\mathsf{R}\,\mathsf{B},$$

no one has won yet. A sixth card is put down:

$$\mathsf{R}\,\mathsf{B}\,\mathsf{B}\,\underbrace{\mathsf{R}\,\mathsf{B}\,\mathsf{R}}_{\texttt{Player-1 seq.}},$$

and `Player-1` won, which give him/her one point. `Player-1` gathers up the six cards and put them by his/her side, and the dealing continues with the remaining 46 cards.

Once they run out of cards, the player with the most tricks is declared the winner.

This sounds like a perfectly even game, but in fact `Player-2` has a strategy that will given him/her a significant advantage...

2. You job

1. Start by assuming the two players pick their sequences completely at random. Use the `R` function `sample()` – and all the loops and data structures you like – to simulate `N = 1000` times this game to show his undisputable fairness (under these conditions at least).
2. Next consider the following weird strategy: when `Player-1` has chosen his/her sequence, say RBR as above, `Player-2` changes the middle color (in this case from B to R), adds it to the start of the sequence, discards the last color, and announces the resulting sequence (in this case RRB). In general, this strategy gives a decided advantage to `Player-2` no matter which sequence his opponent has chosen. Here's some numbers:

`Player-1`	`Player-2`	Pr(`P2` wins)	Pr(Draw)	Pr(`P1` wins)
RRR	BRR	99.5%	0.5%	0.1%
RRB	BRR	93.5%	4%	2.5%
BRR	BBR	88.5%	6.5%	5%

   Your goal is of course to double-check these values adjusting the simulation scheme adopted above. You do **not** have to cover all three cases, just pick one.
3. Grab your phone + a stand, and make a short video (max 2 minutes) of you and your homework partner (if alone, ask a friend!) playing this game say 10 times over the next two weeks. You'll upload the video (so make it low res!) and report here on the result of your games. Alternatively you can place the video on any platform you like (YouTube?) and provide the link. In the end, does this strategy really pay in practice?
4. Quite impressive, uh? Try to explain this phenomenon at least qualitatively.

Exercise 2: Randomize this...

1. Background

Imagine a network switch which must process dozens of gigabytes per second, and may only have a few kilobytes or megabytes of fast memory available. Imagine also that, from the huge amount of traffic passing by the switch, we wish to compute basic stats like the number of distinct traffic flows (source/destination pairs) traversing the switch, the variability of the packet sizes, etc. Lots of data to process, not a lot of space: the perfect recipe for an epic infrastructural failure.

Streaming model of computation to the rescue! The model aims to capture scenarios in which a computing device with a very limited amount of storage must process a huge amount of data, and must compute some aggregate statistics about that data.

Let us formalize this model using frequency vectors. The data is a sequence of indices $(i_1, i_2, \dots, i_n)$, where each $i_k \in \{1, \dots, d\}$, where d, the size of the "data-alphabet", may be very large. For our developments, think d way larger than n, possibly infinite! At an abstract level, the goal is to maintain the d-dimensional frequency vector $\boldsymbol{x}$ with components

$$x_j = \{\text{number of indices } i_k \text{ equal to } j\} = \text{card}\big(\{k : i_k = j\}\big), \quad j \in \{1, \dots, d\},$$

and then to output some properties of $\boldsymbol{x}$, such as its lenght, or the number of non-zero entries, etc. If the algorithm were to maintain $\boldsymbol{x}$ explicitly, it could initialiaze $\boldsymbol{x} \leftarrow [0, 0, \dots, 0]^{\mathsf{T}}$, then at each time step k, it receives the index i_k and increments x_{i_k} by 1. Given this explicit representation of $\boldsymbol{x}$, one can easily compute the desired properties.

So far the problem is trivial. The algorithm can explicitly store the frequency vector $\boldsymbol{x}$, or even the entire sequence $(i_1, i_2, \dots)$, and compute any desired function of those objects. What makes the model interesting are the following desiderata:

1. The algorithm should never explicitly store the whole data stream $(i_1, i_2, \dots, i_n)$.
2. The algorithm should never explicitly build and maintain the frequency vector.
3. The algorithm only see one index i_k per round...BTW, that's the very idea of a *streaming* algorithm!
4. The algorithm should use $\mathcal{O}(\log(n))$ words of space.

This rules out the trivial solutions. Remarkably, numerous interesting statistics can still be computed in this model, if we allow randomized algorithms that output approximate answers.

2. The Algorithm

Here we will focus on a simple randomized algorithm to evaluate the **length** (a.k.a. ℓ_2 norm) of $\boldsymbol{x}$, namely

$$\|\boldsymbol{x}\| = \sqrt{\sum_{i=1}^{d} x_i^2}.$$

The idea of the algorithm is very simple: instead of storing $\boldsymbol{x}$ explicitly, we will store a **dimensionality reduced** form of $\boldsymbol{x}$. *Dimensionality reduction* is the process of mapping a high dimensional dataset to a lower dimensional space, while preserving much of the important structure. In statistics and machine learning, this often refers to the process of finding a few directions in which a high dimensional random vector has maximimum variance. Principal component analysis is a standard technique for that purpose.

Now, how do we get this compressed version of $\boldsymbol{x}$? Simple: **random projection**! Why? You might ask. Because there's a wonderful result known as Johnson-Lindenstrauss lemma which assures that, with high probability, a well designed random projection will (almost) preserve pairwise distances (and **lengths**!) between data points. Of course random projections are central in many different applications, and **compressed sensing** was one of the big, fascinating thing not too long ago...

Okay, let's start by picking a tuning parameter $p << n$. Now define L to be a $(p \times d)$ matrix whose entries are drawn independently as $\mathrm{N}(0, 1/p)$.

The algorithm will explicitly maintain the p dimensional vector $\boldsymbol{y}$, defined as

$$\boldsymbol{y} = \mathsf{L} \cdot \boldsymbol{x}.$$

At time step k, the algorithm receives the index $i_k = j$ with $j \in \{1, \ldots, d\}$, so implicitly the $j^{\mathtt{th}}$ coordinate of $\boldsymbol{x}$ increases by 1. The corresponding explicit change in $\boldsymbol{y}$ is to add the $j^{\mathtt{th}}$ column of L to $\boldsymbol{y}$.

Johnson-Lindenstrauss lemma then says that, for every tolerance $\epsilon > 0$ we pick,

$$\Pr\Big((1-\epsilon) \cdot \|\boldsymbol{x}\| \leqslant \|\boldsymbol{y}\| \leqslant (1+\epsilon) \cdot \|\boldsymbol{x}\|\Big) \geqslant 1 - \mathrm{e}^{-\epsilon^2 \cdot p}. \tag{1}$$

So if we set the tuning parameter p to a value of the order $1/\epsilon^2$, then $\|\boldsymbol{y}\|$ gives a $(1+\epsilon)$ approximation of $\|\boldsymbol{x}\|$ with constant probability. Or, if we want $\boldsymbol{y}$ to give an accurate estimate at each of the n time steps, we can take $p = \Theta(\log(n)/\epsilon^2)$. Notice the remarkable fact that p, the suggested dimension of the embedding, ignores completely the (presumably huge or even infinite) alphabet size d.

3. Your Job

1. Using the `R` function `rnorm()` to generate the matrix L many times (say `N = 1000`), setup a suitable simulation study to double-check the previous result. You must play around with different values of d, n, ϵ and p (just a few, well chosen values, will be enough). The sequence of indices $(i_1, \ldots, i_n)$ can be fixed or randomized too, but notice that the probability appearing in Equation (1) simply accounts for the uncertainty implied by the stochasticity of L – that's why I stressed that you must generate "many times" L...
2. The main object being updated by the algorithm is the vector $\boldsymbol{y}$. This vector consumes $p \sim \log(n)/\epsilon^2$ words of space, so... have we achieved our goal? Explain.