Stat4DS / Homework 01

Pierpaolo Brutti

Due Sunday, December 05 (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**. Alternatively, a `zip`-file with all the material inside will be fine too.

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.

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*** *(no more) 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 and your group*** *only.*

Exercise 1: Randomize this...

1. Background: Random Projections for Data Sketching (more info)

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 summary statistics about that data.

Let us formalize this model using a **frequency vector** $\boldsymbol{x}$.
The raw stream is a growing sequence of *indices* $\mathcal{D}_n = (i_1, i_2, \ldots, i_k, \ldots, i_n)$, where each $i_k \in \{\text{data-alphabet}\} = \{1, \ldots, d\}$, 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 initialize, store and update *sequentially*[1] the frequency vector $\boldsymbol{x} = [x_1, \ldots, x_d]^\mathsf{T}$ with

$$x_j = \{\text{number of indices } i_k \text{ in } \mathcal{D}_n \text{ equal to } j\} = \text{cardinality}\big(\{k : i_k = j\}\big), \quad j \in \{1, \ldots, d\},$$

and then to output some properties of $\boldsymbol{x}$ at the end of the stream, such as its length, or the number of non-zero entries, etc.
If the algorithm were to maintain $\boldsymbol{x}$ **explicitly**, it could initialize $\boldsymbol{x}^{(0)} \leftarrow [0, 0, \ldots, 0]^\mathsf{T}$, then, at each step k, it receives the index i_k and just increments $x_{i_k}^{(k-1)}$ by 1 to get the updated frequency vector $\boldsymbol{x}^{(k)}$. Notice that, in terms of notation, $\boldsymbol{x} \equiv \boldsymbol{x}^{(n)}$. Given this *sequential*, **explicit** representation of $\boldsymbol{x}$ at the end of the stream, one can easily compute the desired properties.

[1] That is, *on-the-fly*, just by "looking" at each i_k as they appear without storing the raw data in $\mathcal{D}_n$ at any step.

So far the problem is trivial. The algorithm *can explicitly* store the frequency vector $\boldsymbol{x}$, or even the entire sequence $(i_1, \dots, i_n)$, 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 $\mathcal{D}_n = (i_1, \dots, i_n)$.
2. The algorithm should **never explicitly** build and maintain the frequency vector $\boldsymbol{x}$.
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 to process the entire stream $\mathcal{D}_n$.

This list rules out trivial solutions. Remarkably, numerous interesting summary statistics can still be computed under this restrictive observational model **if** we allow for randomized algorithms that output approximate answers.

2. The Algorithm: "length" matters...

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 this exercise, its *length* $\|\boldsymbol{x}\|$. 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 but with a different "preservation target".

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

Here for you the basic steps:

1. Let's start by selecting a *tuning parameter* p, the "size" of the projection, with $p << n << d$.
2. Define L to be a $(p \times d)$ random (projection) matrix whose entries are drawn *independently* as $\mathrm{N}_1(0, 1/p)$.
3. The algorithm will maintain **only** the p-dimensional vector $\boldsymbol{y}$ (after its obvious initialization $\boldsymbol{y}^{(0)}$), defined at step k as

$$\boldsymbol{y}^{(k)} = \mathsf{L} \cdot \boldsymbol{x}^{(k)}.$$

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

The *Johnson-Lindenstrauss lemma* (`JL`) then says that, for every tolerance $\epsilon > 0$ we pick,

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

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

3. The Exercise: Comment every line of code and result you get!

$\rightsquigarrow$ Your job $\leftsquigarrow$

1. Show the validity of the update step (i.e., the one you need to code): increasing by 1 the $j^{\mathtt{th}}$ coordinate of $\boldsymbol{x}^{(k-1)}$ corresponds to add the $j^{\mathtt{th}}$ column of L to $\boldsymbol{y}^{(k-1)}$.

2. Use `R` to generate the matrix L many times (say `M = 1000`) and setup a suitable simulation study to double-check the `JL`-lemma. You must play around with different values of d, n, ϵ and p (just a few, well chosen values, will be enough). The raw stream $\mathcal{D}_n = (i_1, \dots, i_n)$ can be fixed or randomized too, but notice that the probability appearing in Equation (1) simply accounts for the uncertainty implied by the randomness of L – that's why I stressed that you must generate "many times" L...

3. The main and **only** 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.

Exercise 2: Faraway, So Close!

1. Background: Statistical Distances and Losses (more info)

Suppose that $\boldsymbol{X}$ is a random vector in $\mathbb{R}^m$ with distribution $\mathcal{P}_X$ and $\boldsymbol{Y}$ is another random vector in $\mathbb{R}^m$ with distribution $\mathcal{P}_Y$ then, for $p \geqslant 1$, the p–Wasserstein distance based on the ground distance d is defined as

$$W_{d,p}(\mathcal{P}_X, \mathcal{P}_Y) \equiv W_{d,p}(\boldsymbol{X}, \boldsymbol{Y}) = \left(\inf_{\mathcal{J}} \int_{\mathbb{R}^m \times \mathbb{R}^m} d(\mathbf{x}, \mathbf{y})^p \, d\mathcal{J}(\mathbf{x}, \mathbf{y}) \right)^{1/p},$$

where the infimum is over all joint distributions $\mathcal{J}$ having $\mathcal{P}_X$ and $\mathcal{P}_Y$ as given marginals... sounds complicated, does it? Anyway, these optimal transport techniques are extremely useful and widespread nowdays with tons of "*... truly marvelous Machine Learning applications, which this homework is too narrow to contain...*".

Nevertheless, when $m = 1$, that is for univariate distributions, things simplify quite a bit and it can be shown that, by picking the $\mathtt{L}_1$ metric (i.e. the *absolute distance*) as ground distance, we obtain

$$\mathrm{W}_{\mathtt{L}_1,p}(\mathcal{P}_X, \mathcal{P}_Y) = \left(\int_0^1 \left| \mathrm{F}_X^{-1}(z) - \mathrm{F}_Y^{-1}(z) \right|^p dz \right)^{1/p},$$

where $\mathrm{F}_X(\cdot)$ and $\mathrm{F}_Y(\cdot)$ are the cumulative distribution functions of $\mathcal{P}_X$ and $\mathcal{P}_Y$, respectively and, consequently, $\mathrm{F}_X^{-1}(\cdot)$ and $\mathrm{F}_Y^{-1}(\cdot)$ are essentially their associated **quantile functions**.

2. The Learning Framework

1. Think about $\mathrm{F}_X(\cdot)$ as the **true** population model and assume that $X \sim \mathrm{Beta}(\alpha, \beta)$ for some value of (α, β) that you will choose (read about this flexible parametric family with bounded support at page 54-56 of our Gallery).
   Let $f(\cdot)$ denote the associated **true** density and $F^{-1}(\cdot)$ its quantile function.
2. Think about $\mathrm{F}_Y(\cdot) = \mathrm{F}_Y(\cdot\,; \boldsymbol{\theta})$ as an approximation to $\mathrm{F}_X(\cdot)$ that you need to *tune/optimize* with respect to the parameter vector $\boldsymbol{\theta}$ in order to achieve some specific goal.
   For this exercise we will consider a *stepwise function* (just like an *histogram*), whose density can be described as follow:
   - Divide the inteval $[0, 1]$, the support of $f(\cdot)$, into bins of length h: there are $N \approx 1/h$ bins.
     Denote the bins by $\{B_1, \ldots, B_N\}$.
   - The approximating density is then
     $$\widehat{f}(x; \boldsymbol{\theta}) = \sum_{j=1}^{N} \frac{\pi_j}{h} \mathbb{1}(x \in B_j),$$
     where $\boldsymbol{\theta} = (h, \pi_1, \ldots, \pi_N)$ and $\mathbb{1}(x \in B_j)$ is the indicator function of the $j^{\mathtt{th}}$ bin.
3. Assume that we are working under **perfect information**, meaning that we can access/query the **true** model.

3. The Exercise: Comment every line of code and result you get!

⇝ Your job ⇜

1. For a fixed h, what kind of constrains we need to impose over the parameter vector $(\pi_1, \ldots, \pi_N)$ for $\widehat{f}(x; \boldsymbol{\theta})$ to be a legit density?
2. Implement $\widehat{f}(x; \boldsymbol{\theta})$ and its corresponding quantile function, say $\widehat{F}^{-1}(x; \boldsymbol{\theta})$, in `R`
3. Pick a specific (α, β) pair and, for any $h > 0$, fix $\pi_j = \int_{B_j} f(x)\, \mathrm{d}x$. Notice that now $\widehat{f}(\cdot\,; \boldsymbol{\theta})$ depends on a single tuning parameter, the binwidth h. For this reason, let's simplify the notation and use $\widehat{f}_h(\cdot)$ and $\widehat{F}_h^{-1}(\cdot)$ to denote the approximant density and its quantile function respectively.
   Let $\epsilon > 0$ be an approximation level you are targeting in terms of $p = 1$ Wasserstein distance, meaning that you're looking for the **largest** binwidth h (i.e. the **coarsest** approximation) such that
   $$\mathrm{W}_{\mathtt{L}_1,1}(f, \widehat{f}_h) = \left(\int_0^1 \left| \mathrm{F}^{-1}(z) - \widehat{\mathrm{F}}_h^{-1}(z) \right| dz \right) \leqslant \epsilon$$
   Use any (**reasonable**) technique you like, to study how h varies with ϵ (... and properly comment the results).
4. (**Bonus**). Repeat the previous point letting $(\pi_1, \ldots, \pi_N)$ free to vary (quite harder!). Compare the results.