Stat4DS / Homework 01

Leandro Bernardino Gentili (1527999)

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

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

$$\text{R B B R B},$$

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

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

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).

Note: comments within code contain useful explanations and information about the reasoning.

```r
rnb_game <- function(patterns=NULL, pattern_size=3) {
  deck_R_count <- 26 # No. of Red cards in the deck
  deck_B_count <- 26 # No. of Black cards in the deck
  deck_size <- deck_R_count + deck_B_count # it's a regular deck of 52 cards
  deck_primitives <- c("R", "B") # We care about cards' colour, nothing more...
  board <- c() # where dealt cards will be placed

  ## Assign random patterns (the fair way)
  assign_fair_patterns <- function () {
    patterns <- list(c("R", "R", "R"), c("R", "R", "R"))

    # to avoid having the same pattern for both players
    while(identical(patterns[1], patterns[2])) {
      patterns[[1]] <- sample(deck_primitives, pattern_size, replace=TRUE)
      patterns[[2]] <- sample(deck_primitives, pattern_size, replace=TRUE)
    }

    return(patterns)
  }
```

```r
# Assign *fair* patterns to players if not provided
if (is.null(patterns)) {
  patterns <- assign_fair_patterns()
}

# unpack patterns list
player_1 <- patterns[[1]]
player_2 <- patterns[[2]]

# Score of each player
players_trick <- c(0, 0)

## Pick a new card considering the ones dealt before.
## In this case, we are informing sample() about the probability of having
## a R/B card considering how many of each kind are still in the deck.
pick_card <- function (n_cards) {
  return(sample(deck_primitives, size=n_cards,
                prob=c(deck_R_count/deck_size, deck_B_count/deck_size),
                replace=TRUE
               )
        )
}

## Determine if a player has won or not
is_a_winner <- function (chosen_pattern) {
  board_len = length(board)

  # We need to check the last three cards from the board
  item_1 <- board[board_len - pattern_size + 1]
  item_2 <- board[board_len - pattern_size + 2]
  item_3 <- board[board_len - pattern_size + 3]
  cards_from_board <- c(item_1, item_2, item_3)

  # check whether a chosen pattern has been found
  if (all(chosen_pattern == cards_from_board)) {
    return(TRUE)
  } else {
    return(FALSE)
  }
}

# The game will continue until every card will be dealt
while (deck_R_count + deck_B_count > 0) {

  # pick up a card from the deck
  picked_card <- pick_card(1)
  board <- append(board, picked_card)

  # updating cards' count
  if (picked_card == "R") {
    deck_R_count <- deck_R_count - 1
  } else {
    deck_B_count <- deck_B_count - 1
  }

  # we need to reach the size of the pattern in order
  # to check whether we have a winner
  if (length(board) < pattern_size) { next }
```

```r
    # if Player_1 wins the game, +1 and board clean up
    if (is_a_winner(player_1)) {
      players_trick[1] = players_trick[1] + 1
      board <- c()

    # same applies for Player_2
    } else if (is_a_winner(player_2)) {
      players_trick[2] = players_trick[2] + 1
      board <- c()
    }
  }

  return(players_trick)
}

## Simulation
N = 1000
results_fair <- replicate(N, rnb_game())

## Results of the "fair" game
p1_win <- ncol(results_fair[,results_fair[1,] > results_fair[2,], drop=FALSE])
draws <- ncol(results_fair[,results_fair[1,] == results_fair[2,], drop=FALSE])
p2_win <- ncol(results_fair[,results_fair[1,] < results_fair[2,], drop=FALSE])
```

Results of the simulation of a fair game:

Player 1: 43.1% **Draw**: 11.6% **Player 2**: 45.3%

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.

```r
rigged_patterns = list(c("R", "R", "B"), c("B", "R", "R"))
results_rigged <- replicate(N, rnb_game(patterns = rigged_patterns))

## Results of the "rigged" game
p1_win <- ncol(results_rigged[,results_rigged[1,] > results_rigged[2,], drop=FALSE])
draws <- ncol(results_rigged[,results_rigged[1,] == results_rigged[2,], drop=FALSE])
p2_win <- ncol(results_rigged[,results_rigged[1,] < results_rigged[2,], drop=FALSE])
```

Results of the simulation of a rigged game:

Player 1 (RRB): 2.6% **Draw**: 4.1% **Player 2 (BRR)**: 93.3%

As can be seen above, the results match for the pattern RRB - BRR! Note the ridiculous difference between the rigged and fair game %.

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?

Note: Unfortunately, I didn't had enough time to dedicate to record a video due to work commitments. Of course, I had the chance to discover numerous interesting video that demonstrate the game and, clearly, I played the game myself too whilst studying the matter. Here's the highest quality video I found: Link to the video

4. Quite impressive, uh? Try to explain this phenomenon at least qualitatively.

This game follows almost the same rules applied to a similar game from where it comes, The Penney's Game, created by Walter Penney in 1969. In that game, both the players should choose a pattern composed of Heads and Tails, the main objective is the same (catching the right pattern before the other player does) but with any precise ending of the game... In fact, two Players could decide to go on flipping a coin for ever. Crucially, in this variation of The Penney's Game, rules are slightly different since both players should use a regular deck of 52 cards, composed of 26 Red and 26 Black, and the game shall end as soon as all cards have been dealt. Moreover, for each hand won by one of the two players, he/she will earn a "winning trick" (score).

From a probability perspective, as soon as the cards are shuffled the probability of picking one Red or Black is 1/2 (but this is not guaranteed in the long run) and it will *usually* tend to stay around that value since the game would last 7-8 tricks, at most. On one hand, one of the main differences between the game played using coins and cards is that the event of "flipping a coin" is independent from one another having an exact probability of 1/2. On the other hand, the event of "picking a card" from the deck is dependent since we have an finite set and that exact value cannot be guaranteed, as said before.

The fact that Player 2 will choose his pattern based on the first Player one puts P2 in advantage as he/she is one step ahead (note that the first two cards of the P1 are *always* the same last two of P2). There is a theory called "Collings's theory" cited in this paper from *Yutaka Nishiyama* which defines the average wait time $E(N)$ before sequence R or B will appear. Moreover, the deck of regular card provides a reasonable amount of trials to end the game and declare a winner. On that note, Colling provides a table which indicates that RBR vs RRB (our case study) winning probabilities are 1/2 against 2/3. Player 2 victory is not guaranteed at the first shot but the situation dramatically bends to P2 when considering 7 trials (winning 4 tricks at least):

$P(RRB,7) =^7 C_7(\frac{2}{3})^7 +^7 C_6(\frac{2}{3})^6(\frac{1}{3}) +^7 C_5(\frac{2}{3})^5(\frac{1}{3})^2 +^7 C_4(\frac{2}{3})^4(\frac{1}{3})^3 = 0.827$

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 $(i_1, i_2, \ldots, i_n)$, where each $i_k \in \{1, \ldots, d\}$, 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 maintain the d-dimensional frequency vector $\boldsymbol{x}$ with components

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

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 $\boldsymbol{x} \leftarrow [0, 0, \ldots, 0]^{\mathsf{T}}$, 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 $(i_1, i_2, \ldots)$, 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 $(i_1, i_2, \ldots, i_n)$.
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,

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

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 $p = \Theta(\log(n)/\epsilon^2)$. Note 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 note 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...

Note: comments within code contain useful explanations and information about the reasoning.

```r
# Generate a stream of random indices as explained below
generate_stream <- function (n_len=10, alphabet_len=100, allow_dupl=FALSE) {
  return(sample(1:alphabet_len, n_len, replace=allow_dupl))
}

random_proj_JL_bound <- function (stream, hyper_params=NULL) {
  if (is.null(hyper_params)) throw("Missing hyperparams! ex. c(n, d, e, p)")

  # Unpack hyperparameters
  n <- hyper_params[1]
  d <- hyper_params[2]
  e <- hyper_params[3]
  p <- if (!is.integer(hyper_params[4])) floor(hyper_params[4]) else hyper_params[4]

  # frequency vector x to work with stream
  x <- as.matrix(rep(0, times=d))

  # We need to store a dimensionality reduced version of x, called y
  # using JL-Lemma which is the foundation of random projection.
  # In order to proceed with RP, we need to compute a random matrix called L
  L <- matrix(data = rnorm(p * d, sd = sqrt(1 / p)), nrow = p, ncol = d)

  # Compressed version of vector x which is defined as follow:
  # y = L %*% x but instead of computing this matrix multiplication
  # we'll update its placeholder from pieces of L
  # We now need to create its placeholder...
  y <- as.matrix(rep(0, times=p))

  # Update y for each index belonging to the stream
  for (j in stream) {
    x[j] <- x[j] + 1 # freq vector update
    y <- y + L[,j]   # compressed vector update
  }

  # We need to verify Eq. 1
  x_norm <- norm(x, type="2")
  y_norm <- norm(y, type="2")
  is_verified_1 <- (1 - e) * x_norm <= y_norm
  is_verified_2 <- y_norm <= (1 + e) * x_norm
  is_verified <- is_verified_1 && is_verified_2

  return(is_verified)
}

##   Hyperparams
n <- 125

# the size of the data-alphabet, may be very large!
d <- n ** 2

# The epsilon is the error tolerant parameter and it is inversely proportional
# to the accuracy of the result.
e <- 0.25

# We need to choose a tuning parameter p << n (p much less than n)
# We can pick p value considering JL-Lemma: p = log(n)/eps^2
p <- log(n)/e**2

# The stream is a set of indices coming in an overwhelming quantity
# (i(1), ..., i(n) where i(n) is bound {1, ..., d}
stream <- generate_stream(n_len=n, alphabet_len=d, allow_dupl=TRUE)

# execute Random Projection based on JL-Lemma
outcome <- random_proj_JL_bound(stream, hyper_params=c(n, d, e, p))
```

Of course, we could play around with hyperparameters (n, d, e, p) in order to obtain the best solution which would depend on data dimensionality, first of all. In fact, Random Projection is well-known for its being fast at computing a good stochastic solution whereas other noteworhty methods, like PCA (Principal Component Analysis), provide "the perfect solution" but in a slow way because of more complex operations are needed. Crucially, as in this case dealing with a huge stream of data, being quick and accurate with a high probability counts more than the rest. For this reason, RP is currently used for dimensionality reduction in high dimensional unsupervised learning, for instance.

Is it important to note that changing ϵ (error tolerant parameter, inversely proportional to the accuracy of the result) to lower values it would increase the dimension, and viceversa.
For the sake of this experiment, we fixed the hyperparameters as follows, chosen in order to stay behind the bounds stated in

the exercise's statement and commented within the R code above.

- $n = 125$
- $d = n^2$
- $\epsilon = 0.25$
- $p = \log(n)/\epsilon^2$

We can now proceed with replicating the experiment $N = 1000$ times since the probability appearing in the Eq. 1 accounts for the uncertanty implied by the stochasticity of matrix L, as stated above, since that matrix is randomly generated using the rnorm(). During each simulation we will validate the probability distribution conditions and store their results as boolean whether they have been satisfied or not. The latter ones will be useful to test the Eq. 1.

```r
# Simulation N-times
N = 1000
outcomes <- replicate(N, random_proj_JL_bound(stream, hyper_params=c(n, d, e, p)))
```

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.

Yes, we managed to obtain a p-dimensional vector $\boldsymbol{y}$ which is a compressed representation of $\boldsymbol{x}$:

$$\begin{bmatrix} & y & \end{bmatrix}_{p\times n} = \begin{bmatrix} & L & \end{bmatrix}_{p\times d} \begin{bmatrix} & x & \end{bmatrix}_{d\times n}$$

Finally, we can verify if the Eq.1 is satisfied considering all the outcomes (TRUE if probability conditions have been verified; FALSE, otherwise) out of N times we ran the simulation. Then, we can verify the right part of the Eq. 1.

```r
# Computing the Probability value considering the results
# coming from the N simulation
JL_lemma_prob <- length(outcomes[outcomes == TRUE]) / N
equation_right_bound <- 1 - exp(-e**2*p)

# Output
cat(JL_lemma_prob, " >= ", equation_right_bound, JL_lemma_prob >= equation_right_bound, "\n")
```

```
## 0.997  >=  0.992 TRUE
```

$0.997 >= 0.992 -> \text{TRUE}$

The Eq. 1 is satisfied. That's a sign we managed to maintain the pair-wise distance between our data reducing its dimensionality.