Unsupervised Learning and Evolutionary Computation Using R

Winter Term 2024/2025

Exercise Sheet 1 (21st October, 2024)

Installing the Jupyter notebook R kernel: please following this tutorial[1]. Essentially, once Jupyter and R are installed executing the following commands in an interactive R session should be enough to get it running:

```
install.packages('IRkernel')
IRkernel::installspec()  # to register the kernel in the current R installation
```

Exercise 1 (Coin tossing simulation)
The function `sample(...)` can be used to sample random numbers uniformly at random from a given set. Read the documentation of the function (`?sample`). Write a simulation study on tossing a single fair coin n times for different values of $n \to \infty$. Report the absolute/relative number of head/tail in some way (functions `barplot` or `table` could be useful).

Exercise 2 (Trimmed mean)
The arithmetic mean is likely the most popular and frequently used measure of mass. A related measure is the so-called *α-trimmed arithmetic mean* which for a series of data points $x_1, \ldots, x_n$ and $\alpha \in [0, 1)$ is defined as

$$\bar{x}_t(x_1, \ldots, x_n) = \frac{1}{n - 2\lfloor \alpha n \rfloor} \sum_{i=\lfloor \alpha n \rfloor}^{n-\lfloor \alpha n \rfloor} x_{(i)}.$$

Here, $\lfloor x \rfloor$ rounds its argument x to the nearest integer value lower than x and $x_{(i)}$ for $i = 1, \ldots, n$ is the ith order statistic (the ith order statistic is the value that comes at the ith position if the data is sorted in increasing order). Hence, the trimmed mean cuts of a fraction of $\alpha\%$ lowest and highest observations and calculates the mean of the remaining ones. Implement a function `trimmedMean(x, alpha)` that implements the trimmed mean following the formula given above. Why should it be useful to use the trimmed mean instead of the standard arithmetic mean?

Exercise 3 (Re-scaling)
We will see later in the semester that for many multi-variate methods, i.e., methods that deal with data of at least two (numeric) variables $X_1, \ldots, X_p, p \geq 2$, different variable scales are problematic. Here, it is often useful to transform these variables to a common scale; oftentimes $[0, 1]$. For realisations $x_1, \ldots, x_n$ of a numeric variable X this *re-scaling* can be realised by calculating

$$\tilde{x}_i = \frac{x_i - \min_i x_i}{\max_i x_i - \min_i x_i}, i = 1, \ldots, n.$$

[1] URL: https://github.com/IRkernel/IRkernel

1. Implement a function `rescale(x)` that expects a numeric vector `x` as input and returns the re-scaled version.
2. Test your function on different input vectors.
3. Imagine the input vector contains at least one so-called NA-value (NA for not available, a special value type in R). Check your input on a vector with at least one NA value, e.g., `c(1, 5, NA, 10, 3)`. Modify your function by adding a logical parameter `na.rm` which defaults to `FALSE`. If set to `TRUE` NA values should be ignored during re-scaling.
4. Imagine now the input vector potentially contains either `Inf` or `Inf`. Again, check how your implementation behaves and come up with a possible solution.
   Hint: `is.infinite()` might be useful.

Exercise 4 (R loop performance)
In the presentation slides you learned that loops should be avoided in R unless there is a dependency. For a numeric vector `x` the build-in function `cumsum(x)` builds the cumulative sum vector of the same length, i.e., the ith entry of `cumsum(x)` is $\sum_{j=1}^{i} x_j$.

1. Write a function `cumsum_loop(x)` that calculates the cumulative sum in R using basic R loops. Why can't we use one of the `*apply` functions to solve the task?
2. Generate random numeric vectors of different length $n \in \{1\,000, 100\,000, 1\,000\,000, ...\}$. Run both the build-in version and your implementation several times and store the elapsed running time. Visualise the running time difference, e.g., with box-plots.