PiRaP

# Lab 1 - Python

## Task 1.

Write a function findPositions that takes two or more parameters. The function should return a dictionary, which keys are characters and the values are lists of positions of characters occurrences in the string passed as the first function parameter. The characters to be found are passed as parameters 2, 3, . . . . For example, when the function is called as pos = findPositions("test"\*2 + "xD", "a", "e", "s", "x"), then print(pos) should show {’e’: [1, 5], ’s’: [2, 6], ’x’: [8]}.

|  |
| --- |
| def **find\_positions**(**string**, \*characters):  dict = {}  **for** **char** **in** characters:  dict[**char**] = [pos **for** pos, current **in** **enumerate**(**string**) **if** current == **char**]  **if** not dict[**char**]:  del dict[**char**]  **return** dict |

## Task 2.

Write functions positionsToTupleList and positionsToTupleList comprehension. Both of them do the same, which is converting input dictionary of letters as keys and lists of integers as values to list of tuples. Each tuple contains two elements. The second element of a tuple is a dictionary key, and the first one is an element of list related to this key. The returned list should be sorted. For example for a call L = positionsToTupleList(pos), printing L, show show [(1, ’e’), (2, ’s’), (5, ’e’), (6, ’s’), (8, ’x’)]. The first variant should use loops, and the second one should use list comprehensions.

|  |
| --- |
| **def** **positions\_to\_tuple\_list**(position):  list = []  **for** char **in** position:  **for** item **in** position[char]:  list.append((item,char))  **return** sorted(list, key=**lambda** x: x[0])  **def** **positions\_to\_tuple\_list\_comprehension**(position):  array = [(item, char) **for** char **in** position **for** item **in** position[char]]  **return** sorted(array, key=**lambda** x: x[0]) |

## Task 3.

Write a generator dictSortedGenerator that allows to iterate over dict in ascending order of keys. Usage example: for a, b in dictSortedGenerator({5: 1, 1: 5}): print(f"k={a}, v={b}").

|  |
| --- |
| **def** **dict\_sorted\_generator**(dict):  **for** key **in** sorted(dict):  **yield** key, dict[key] |

## Task 4.

Write a function drawHisto that draws a histogram of a dictionary using start symbol (\*). It takes one input parameter: the dictionary being a result of task 1. In the first step new dictionary should be created from the parameter. Its keys should be the same as in the input, but values should be lengths of appropriate lists. This dictionary should be used to display a histogram. Usage example: drawHisto(pos). It should then print: e \*\* s \*\* x \* In the implementation use dictSortedGenerator from task 3.

|  |
| --- |
| **def** **draw\_histo**(dict):  new\_dict={key: len(dict[key]) **for** key **in** dict}  **for** key, value **in** dict\_sorted\_generator(new\_dict):  print(key, "\*" \* new\_dict[key]) |

## Task 5.

Write function convertToUnique that takes a list as a parameter, and return new list contaiting unique elements from input list. For example for a following call print(convertToUnique([1, 2, 1, 2, 6, 7, 6, 9, 9, 9, 10])) it should print [1, 2, 6, 7, 9, 10].

|  |
| --- |
| **def** **convert\_to\_unique**(dict):  **return** list(set(dict)) |

## Task 6.

Jaccard index is one of the measures of sets similarities. For sets S1 and S2 it is defined as |S1 ∩ S2|/|S1 ∪ S2|. Write a function that computes the Jaccard index of two sets passed as arguments.

|  |
| --- |
| **def** **jaccard\_index**(s1, s2):  **return** len(s1.intersection(s2))/len(s1.union(s2)) |

## Task 7.

Write a function optimizeJaccardInxed that takes three parameters: set1, set2 and limit (which default value is 0.9). Function should remove an element from bigger set that do not exists in the smaller set as long as the size of both sets is greater than zero and the Jaccard index for those sets is below limit.

|  |
| --- |
| **def** **optimize\_jaccard\_index**(s1, s2, limit=0.9):  index=jaccard\_index(s1, s2)  **while** len(s1)>0 **and** len(s2)>0 **and** index < limit:  **if** len(s1)<= len(s2):  b, s = s2, s1  **else**:  b, s = s1, s2  unique = b.difference(s)  b.remove(unique.pop())  s1 = b  s2 = s  index = jaccard\_index(s1, s2)   position = find\_positions("test"\*2 + "xD", "a", "e", "s", "x") **print** (position) tuple\_list1 = positions\_to\_tuple\_list(position) **print** (tuple\_list1) tuple\_list2 = positions\_to\_tuple\_list\_comprehension(position) **print** (tuple\_list2) **for** a, b **in** dict\_sorted\_generator({5: 1, 1:5}):  print(f"k={a}, v={b}") draw\_histo(position) print(convert\_to\_unique([1,2,1,2,6,7,6,9,9,9,10])) s1 = set([0,5,4]) s2 = set([5,2,4]) print(s1, s2, jaccard\_index(s1, s2)) optimize\_jaccard\_index(s1, s2, 0.55) print(s1, s2, jaccard\_index(s1, s2)) |

# Lab 2 - Python

## Task 1.

Define a function filter that takes as parameters a two dimensional array 𝐴 and a threshold value 𝑡. It returns a list of two element tuples with indices of 𝐴 elements not smaller than 𝑡 (use a comprehension). Show how the function works on an example array.

|  |
| --- |
| **def** **filter**(stuff: list, t: int):  result = [(x, y) **for** x, y **in** numpy.ndindex(stuff.shape) **if** stuff[x, y] >= t]  **return** result |

## Task 2.

Define a function getQueenMoves(x : int, y : int) that takes as an input 0-based indices of queen on a chessboard and returns an 8x8 integer array with 1s on fields where the queen can move (exclude the queen position itself). Steps: determine a list of moves using comprehension, then use a list for array indexing. Loops are not allowed.

|  |
| --- |
| **def** **get\_queen\_moves**(x: int, y: int):  **return** numpy.array([[-1000 **if** x == i **and** y == j **else** 1 **if** (x == i **or** j == y **or** (x-i == y-j) **or** (x+y == i+j)) **and** **not** (x == i **and** j == y) **else** 0 **for** i **in** range(8)] **for** j **in** range(8)]) |

## Task 3.

Define a function checkQueenPuzzle which takes an 8x8 integer array with 1s indicating queens (there can be any number of them) and returns logical value indicating whether no two queens threaten each other. Hints: determine queens positions with filter function, apply getQueenMoves for all queens, and aggregate the results. Loops are not allowed.

|  |
| --- |
| **def** **check\_queen\_puzzle**(stuff: numpy.array):  *#Uses the fact that in previous task such a low value (-1000) was assigned to fields where the queen could move to. If the queen is on a field where another one can move to the sum is bigger then -1k, but still less then 0*  neptunes = filter(stuff, 1)  dogoos = [get\_queen\_moves(neptune[0], neptune[1]) **for** neptune **in** neptunes]  dogoos = [sum(x) **for** x **in** zip(\*dogoos)]  dogoos = numpy.array(dogoos)  test = numpy.argwhere(numpy.logical\_and(dogoos < 0, dogoos > -1000))  **return** len(test) == 0 |

## Task 4.

Define a function generateSystem which generates a system of linear equations 𝐴𝑋 = 𝐵: The function takes as a parameter a column vector 𝑋 and returns a tuple containing matrix 𝐴 of coefficients and column vector 𝐵 of free terms. Elements of 𝐴 are drawn from uniform distribution [-1,1] with an exception of diagonal elements which can be any values fulfilling the diagonal dominance requirement: |𝑎𝑖𝑖| ≥ ∑ |𝑎𝑖𝑗| 𝑗≠𝑖 . Loops are not allowed.

|  |
| --- |
| **def** **generateSystem**(X: numpy.array):  n = X.shape[0]  A = numpy.random.uniform(-1, 1, size=(n, n))  numpy.fill\_diagonal(A, 0)  S = numpy.sum(numpy.sum(abs(A), 0), 0)  numpy.fill\_diagonal(A, S + numpy.random.uniform(S, S \* 2))  B = numpy.dot(A, X)  result = (A, B)  **return** result |

## Task 5.

Define a function solveSystem which solves 𝐴𝑋 = 𝐵 system using Jacobi algorithm. By dividing each equation by the corresponding diagonal coefficient we get: [ 1 𝑎12/𝑎11 𝑎21/𝑎22 1 ⋯ 𝑎1𝑛/𝑎11 𝑎2𝑛/𝑎22 ⋮ ⋱ ⋮ 𝑎𝑛1/𝑎𝑛𝑛 𝑎𝑛2/𝑎𝑛𝑛 ⋯ 1 ] [ 𝑥1 𝑥2 ⋮ 𝑥𝑛 ] = [ 𝑏1/𝑎11 𝑏2/𝑎22 ⋮ 𝑏𝑛/𝑎𝑛𝑛 ] which can be transformed to 𝑋 = 𝛼𝑋 + 𝛽 form: [ 𝑥1 𝑥2 ⋮ 𝑥𝑛 ] = [ 0 −𝑎12/𝑎11 −𝑎21/𝑎22 0 ⋯ −𝑎1𝑛/𝑎11 −𝑎2𝑛/𝑎22 ⋮ ⋱ ⋮ −𝑎𝑛1/𝑎𝑛𝑛 −𝑎𝑛2/𝑎𝑛𝑛 ⋯ 0 ] [ 𝑥1 𝑥2 ⋮ 𝑥𝑛 ] + [ 𝑏1/𝑎11 𝑏2/𝑎22 ⋮ 𝑏𝑛/𝑎𝑛𝑛 ] The Jacobi algorithm is an iterative procedure which works according to the equation X 𝑘+1 = αX 𝑘 + β, where: 𝑘 – iteration number, 𝑋 0 – initial solution (e.g., zero vector). The iteration stops when differences between corresponding X 𝑘 and X 𝑘+1 elements drop below assumed threshold. The function takes as parameters matrix 𝐴 and vector 𝐵. It returns a column vector 𝑋 of calculated unknowns. Loops are allowed only for performing consecutive iterations.

|  |
| --- |
| # TO DO |

## Task 6.

Verify the correctness of the Jacobi method by generating example system (let 𝑋 contain consecutive natural numbers) and comparing the solution with the input 𝑋. Measure the execution times of the Jacobi algorithm and the method with matrix inversion for systems of 100, 200, 500, 1000, 2000, and 5000 equations.

|  |
| --- |
| # TO DO |

# Lab 3 - R

Load the Carseats dataset from the ISLR package. Use the help section to find out about the variables in this dataset.

|  |
| --- |
| library("ISLR")  library(ggcorrplot)  data(Carseats)  attach(Carseats) |

## Task 1.

Investigate the structure of the dataset by summarizing all of the variables. Create two new data frames: one containing the qualitative variables and one containing the quantitative variables.

|  |
| --- |
| summary(Carseats)  t1\_quant<- data.frame(Carseats$Sales,  Carseats$CompPrice,  Carseats$Income,  Carseats$Advertising,  Carseats$Population,  Carseats$Price,  Carseats$Age,  Carseats$Education)  t1\_qual <- data.frame(Carseats$ShelveLoc,  Carseats$Urban,  Carseats$US) |

## Task 2.

Calculate Pearson’s linear correlation coefficients between the quantitative variables. Present the results on a correlogram using the ggplot2 package. Include correlation coefficient values on the diagram. The correlation coefficient should be presented on a color scale from blue (negative values) through white (zero) to red (positive values). Identify the variable that has the strongest linear relationship with sales. Using ggplot2 present this relation on a scatterplot with a fitted linear model represented by an orange line. The plot background should be white. Include a plot title, subtitle, caption and axes labels.

|  |
| --- |
| t2\_correl<-cor(t1\_quant)  t2\_g1 <- ggcorrplot(t2\_correl, type="lower", lab=TRUE, lab\_size=3, colors= c("blue","white","red"))  t2\_g2 <- ggplot(Carseats, aes(x = Price, y = Sales)) +  geom\_point() +  geom\_smooth(method="lm", se=FALSE, col="orange") +  labs(title = "Carseats" ,  subtitle = "Sales vs Prices",  x = "Price",  y = "Sales",  caption = "ISLR Dataset")  plot(t2\_g2) |

## Task 3.

Using an appropriate grouping function calculate the mean and median Price for each shelving location group. Compare the distribution of these prices using ggplot2 boxplots. The boxplots should be plotted with different contrasting colors.

|  |
| --- |
| t3\_mean <- aggregate(Carseats[,6], list(Carseats$ShelveLoc), mean)  t3\_median <- aggregate(Carseats[,6], list(Carseats$ShelveLoc), median)  t3\_g1 <- ggplot(Carseats,aes(x = as.factor(ShelveLoc), y = Price, fill = Carseats$ShelveLoc)) +  geom\_boxplot() +  xlab("ShelveLoc") +  ylab("Price")  plot(t3\_g1) |

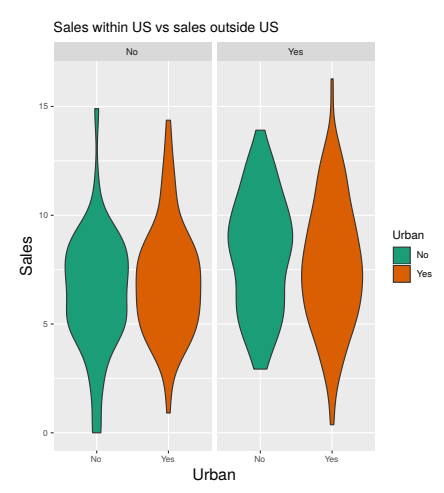
## Task 4.

Calculate the mean Sales of every Education group using a grouping function. Add a variable to the resulting data frame containing deviations from the mean. Add another variable that will have the value ”Yes” if at least 2/3 of stores in the Education groups are in the US, and ”No” otherwise. Sort the data frame according to deviations from the mean. Use the obtained data frame to plot a deviation bar plot with ggplot. The bars should be colored by the US location groups. Hint: Remember to transform the Education variable to a factor, so that the sorting by deviations takes effect.

|  |
| --- |
| # TO DO |

## Task 5.

Reproduce the following violin plot (colors from RColorBrewer Dark2 palette) :



|  |
| --- |
| g5 <- ggplot(Carseats,aes(Urban, Sales)) +  geom\_violin(aes(fill=Urban), trim = T) +  labs(title="Sales within US vs sales outside US") +  facet\_wrap(Carseats$US, ncol = 2) +  scale\_fill\_brewer(palette = "Dark2")  plot(g5) |

# 

# Lab 4 - R

Download the Heart dataset from the Educational Platform and load the data (you may use the read.csv function suitable for comma-separated files).

|  |
| --- |
| library(ISLR)  library(boot)  library(ade4)  library(ggplot2)  library(viridis)  library(doParallel)  library(dplyr)  library(boot)  library(doParallel)  heart<-read.csv("Heart.csv",header=TRUE,sep=",") |

## Task 1.

Using t-tests, check whether there are statistically significant differences in:

* Cholesterol levels (Chol) in different Fasting Blood Sugar groups (Fbs: 0 - normal, 1 elevated)
* Maximum Heart Rates (MaxHR) in people with and without Heart Disease (AHD)
* Resting Blood Pressure (RestBP) depending on Sex (0 - female, 1 - male).

|  |
| --- |
| ta <- t.test(Chol~Fbs, data=heart) #p=0.8634, no statistical differences ta$p.value tb <- t.test(MaxHR~AHD, data=heart) #p=9.106e-14, statistically sign. differences tb$p.value tc <- t.test(RestBP~Sex, data=heart) #p=0.2896814, no statistical differences tc$p.value |

## Task 2.

Build three linear regression models with Maximum Heart Rate as the dependent variable and:

* Age
* Resting Blood Pressure
* Cholesterol

as predictor variables. Divide the data into a training set containing 2/3 of the measurements and a test set with the remaining 1/3. For each predictor variable calculate the mean squared error. Which variable out of the three serves as the best predictor in a linear model? Using that variable perform Leave One Out cross-validation and 20-fold cross-validation with the boot package. Compare the error rates.

|  |
| --- |
| smp\_size <- floor(0.66 \* nrow(heart)) set.seed(42) train\_ind <- sample(seq\_len(nrow(heart)),size=smp\_size) train <- heart[train\_ind, ] test <- heart[-train\_ind, ] model1 <- lm(MaxHR~Age, data=train) model2 <- lm(MaxHR~RestBP, data=train) model3 <- lm(MaxHR~Chol, data=train) rsq <- c() *# model fit metric - the higher, the better* trainerr <- c() *# training set error - the lower, the better* testerr <- c() *# test set error - the lower, the better* rsq1 <- c(rsq,summary(model1)$r.squared) beta1 <- model1$coef[-1] intercept1 <- model1$coef[1] rsq2 <- c(rsq,summary(model2)$r.squared) beta2 <- model2$coef[-1] intercept2 <- model2$coef[1] rsq3 <- c(rsq,summary(model3)$r.squared) beta3 <- model3$coef[-1] intercept3 <- model3$coef[1] trainerr <- c() *# training set error - the lower, the better* testerr <- c() *# test set error - the lower, the better* testerr <- c(testerr,mean((heart$Age\*beta1+intercept1 - heart$MaxHR)^2)) testerr <- c(testerr,mean((heart$RestBP\*beta2+intercept2 - heart$MaxHR)^2)) testerr <- c(testerr,mean((heart$Chol\*beta3+intercept3 - heart$MaxHR)^2)) *# Leave-one-out crossval* dat <- data.frame(x=heart$Age,y=heart$MaxHR) mod <- glm(y~.,data=dat) cverr <- c(cv.glm(dat,mod)$delta[1]) *# perform LOO crossvalidation* cverr2 <- c(cv.glm(dat,mod,K=20)$delta[1]) *# perform LOO crossvalidation* |

## Task 3.

Take the division into training set (2/3) and test set (1/3) from the previous task and based on all 14 variables build a:

* logistic regression classifier
* Support Vector Machine classifier with a radial kernel

for Heart Disease (AHD) occurrence. Compare the error rates. Which classifier gives better performance?

|  |
| --- |
| testerr3log <- c() testerr3svm <- c() **for** (i **in** 1:14) {  mod <-glm(AHD~heart[,i],data=heart,family="binomial")  dat = data.frame(x=heart[,i], y=as.factor(heart$AHD))  pred.prob = predict(mod, type="response")  pred.prob = ifelse(pred.prob > 0.5, 1, 0)  er1=table(pred.prob)  er2=table(heart$AHD)  testerr3log=c(testerr3log, ((abs(er1[1]-er2[1])+abs(er1[2]-er2[2]))/392))  svm <- svm(y~.,data=dat,kernel="radial",cost=1,type="C-classification")  pred.prob = predict(svm, type="response")  pred.prob = ifelse(pred.prob > 0.5, 1, 0)  er1=table(pred.prob)  er2=table(heart$AHD)  testerr3svm=c(testerr3svm, ((abs(er1[1]-er2[1])+abs(er1[2]-er2[2]))/392)) } |

## Task 4.

Download the wine dataset from the Educational Platform and import the data into R. The wine type labels are stored in the first column. The dataset description is available in the wine.names file. Perform hierarchical clustering with average linkage and visualize the results on a dendrogram. Which wine type separates best from the others? Use K-means clustering on the Alcohol level and Color intensity variables. Present the results on a scatterplot. Do you think that these two variables are sufficient to get a comparably good grouping with regard to hierarchical clustering on all the variables?

|  |
| --- |
| # TO DO |

## Task 5.

Download the Dmel.RData file from the Educational Platform and load the data into R (use the load function). The data consists of a string vector containing the genome sequences of Drosophila melanogaster (the fruit fly). Using parallel computing on two cores in R first complement the sequences, and then, count the nucleotides in each sequence. Output a matrix containing the letter counts (N-by-4, where N is the number of sequences). Compare the run times for parallel and sequential operations. Hint: In the DNA guanine (G) is complemented by cytosine (C), adenine (A) by thymine (T).

|  |
| --- |
| # TO DO |

# 

# Lab 5 - Python

## Object oriented programming:

## Task 1.

Create a class StatePool representing the pool of states of a simple state machine. Inside class define method GetInst(self, cls), the second parameter, cls, is a class representing a single state. Class StatePool should collect states in a private (as far as python allows) dictionary with state classes being keys and their instances—values. GetInst should check if specified state is in the dictionary, if not it should create a new object and add appropriate entry to the dictionary, after that existing entry should be returned.

|  |
| --- |
| **class** **StatePool**:   **def** **\_\_init\_\_**(self):  self.\_\_states = dict()   **def** **get\_inst**(self, cls):  **if** cls **not** **in** self.\_\_states:  self.\_\_states[cls] = cls()  **return** self.\_\_states[cls] |

## Task 2.

Create a class HeroState representing a single state of a hero in some imaginary game. It should contain a private static field of type StatePool and public class method GetInst that gets instance of a state from the pool and returns it.

|  |
| --- |
| **class** **HeroState**:   \_\_state\_pool = StatePool()   **@classmethod**  **def** **get\_inst**(cls):  **return** HeroState.\_\_state\_pool.get\_inst(cls) |

## Task 3.

Define three subclasses of a HeroState class. They should represent following hero states: Standing, Moving, Shooting. Each of them should contain a str method that is automatically called when instance of an object is converted to string with str function. The str method should return a string describing a hero state, for example for Standing state it should return ”Hero is standing”. Furthermore, each class should define on event(self, event) method, where event is a string describing an action. Basing on an event parameter a new state should be returned. If event is not allowed for a current state, the current state should be returned. In a Standing state two actions are allowed: go (the Moving state should be returned) and shoot (the Shooting state should be returned). In Moving state also two events are allowed: stop (the Standing state should be returned) and shoot (the Shooting state should be returned). The Shooting state is a little tricky, because independently of the event it should always return a previous state (hint: store a previous state as a data field in an instance of class representing Shooting class).

|  |
| --- |
| **class** **Standing**(HeroState):   **def** **on\_event**(self, event:str):  **if** event == 'go':  **return** Moving.get\_inst()  **elif** event == 'shoot':  Shooting.prev\_state = self  **return** Shooting.get\_inst()  **else**:  **return** 'Error'   **def** **\_\_str\_\_**(self):  **return** 'HeRo iS StAnDiNg'  **class** **Moving**(HeroState):   **def** **on\_event**(self, event:str):  **if** event == 'stop':  **return** Standing.get\_inst()  **elif** event == 'shoot':  Shooting.prev\_state = self  **return** Shooting.get\_inst()  **else**:  **return** 'Error'   **def** **\_\_str\_\_**(self):  **return** 'HeRo iS MoViNg'  **class** **Shooting**(HeroState):   prev\_state = Standing()    **def** **on\_event**(self, event:str):  **if** event == 'go' **or** event == 'stop' **or** event == 'shoot':  **return** self.prev\_state  **else**:  **return** 'Error'   **def** **\_\_str\_\_**(self):  **return** 'HeRo iS ShOoTiNg' |

## Task 4.

Create a class Hero, it should contain, as a private field, current hero state (initially Standing). Define ProcessEvents method that will in an infinite loop print a current state (use str to convert state to a string). After that, an action should be read from the keyboard. If the action is a string exit the loop should be interrupted. In the opposite case the current state should be replaced by the new one (being a result of calling on event method on an object representing current state). Create an instance of Hero class and call ProcessEvents method.

|  |
| --- |
| **class** **Hero**:   **def** **\_\_init\_\_**(self):  self.\_\_current\_hero\_state = Standing()   **def** **process\_events**(self):   loop = **True**   **while** loop:  print(str(self.\_\_current\_hero\_state))  user\_input = input('Select action: ')  **if** user\_input == 'exit':  loop = **False**  **else**:  self.\_\_current\_hero\_state = self.\_\_current\_hero\_state.on\_event(user\_input)   hero = Hero() hero.process\_events() |

## Parallel computing:

In order to avoid Python pickling problems, define necessary functions at the top level and put the remaining code in the main scope (if name == ’ main ’:). Execute the entire script at once.

## Task 1.

Draw a real number x from a uniform distribution (1,100). Allocate empty numpy array of size N = 106 and fill it with consecutive roots of x (2nd, 3rd, etc.) Measure the execution time.

|  |
| --- |
| **def** **consecutiveRoots**(number, order):  **return** number\*\*(1./order)   random.seed(123)  x = random.uniform(1,100)  numWorkers = mp.cpu\_count()  array= np.empty(1000000)  t = time.time()  **for** i **in** range(1000000):  array[i] = consecutiveRoots(x, i + 1)  print(time.time()-t) |

## Task 2.

Parallelize the loop from the previous task using process pool from multiprocessing library. Use all logical CPU cores. Make sure that the result is converted to an array. What is the performance compared to the serial solution? Note: due to overhead related to the pool initialization, execute any operation on it before benchmarking.

|  |
| --- |
| array = np.empty(1000000)   **with** mp.Pool(processes=numWorkers) **as** pool:  pool.\_\_sizeof\_\_()  t = time.time()  A= pool.starmap(consecutiveRoots, ((x, i+1) **for** i **in** range(1000000)))   array = np.array(A)   print(time.time()-t) |

## Task 3.

Suggest an optimized parallelization scheme which reduces overheads and provides better scalability than the simple method from the previous task.

|  |
| --- |
| **def** **consecutiveRootsSlice**(x , start, size):  buffer= np.zeros((start+size)-start)  **for** i **in** range(size):  start=start+1  buffer[i] = consecutiveRoots(x, start+1)   array = np.zeros(1000000)  bufA = np.array( array)  sliceSize = 1000000 // numWorkers  **with** mp.Pool(processes=numWorkers) **as** pool:  pool.\_\_sizeof\_\_()  t = time.time()  A= pool.starmap(consecutiveRootsSlice,((x , sid \* sliceSize, sliceSize) **for** sid **in** range(numWorkers)))   print((time.time() - t)) |

## Task 4.

Define a function task representing some job to be done in parallel with other jobs. The function takes two arguments: an integer identifier to be printed on the console and a multiprocessing.Lock object which secures the access to the standard output. Create explicitly 10 processes (each assigned with a single task), run them, and wait until they finish.

|  |
| --- |
| **def** **task**(id, lock):  **with** lock:  sys.stdout.write(str(id)+ '\n')   lock = Lock()  t= time.time()  workers = [mp.Process(target= task, args= (sid ,lock)) **for** sid **in** range(10)]   **for** w **in** workers: w.start()  **for** w **in** workers: w.join()   print(time.time() -t) |

# Exam

## R - Task 1

data = read.csv("battles.csv")

head(data)

# zadanie 1

joffry\_atc = data[data["attacker\_king"] == "Joffrey/Tommen Baratheon",]

joffry\_def = data[(data["defender\_king"] == "Joffrey/Tommen Baratheon"),]

balon\_atc = data[data["attacker\_king"] == "Balon/Euron Greyjoy",]

balon\_def = data[(data["defender\_king"] == "Balon/Euron Greyjoy"),]

won\_joff = table(joffry\_atc["attacker\_outcome"])[3] + table(joffry\_def["attacker\_outcome"])[2]

def\_joff = table(joffry\_atc["attacker\_outcome"])[2] + table(joffry\_def["attacker\_outcome"])[3]

won\_balon = table(balon\_atc["attacker\_outcome"])[3] + table(balon\_def["attacker\_outcome"])[2]

def\_balon = table(balon\_atc["attacker\_outcome"])[2] + table(balon\_def["attacker\_outcome"])[3]

win\_loss = data.frame(row.names = c("Joffrey","Balon"))

win\_loss["win"] = c(won\_joff, won\_balon)

win\_loss["loss"] = c(def\_joff, def\_balon)

tf = fisher.test(as.matrix(win\_loss))

## Task 2

#task2

rm(list = ls())

winterBattles <- data[data$summer == 0,]

summerBattles <- data[data$summer == 1,]

summerSample <- sample(26, 21)

winterSample <- sample(12, 10)

trainSet <- rbind(winterBattles[winterSample,], summerBattles[summerSample,])

testSet <- rbind(winterBattles[-winterSample,], summerBattles[-summerSample,])

ytest <- testSet$region

svmfit = svm(region~battle\_type+summer+attacker\_king+attacker\_size+defender\_size,data=trainSet,scale=F)

predicted <- predict(svmfit, testSet)

xtab <- table(predicted, ytest)

install.packages("caret")

library(caret)

confusionMatrix(xtab)

load("nextAttack.RData")

attach(nextAttack)

predictedNew <- predict(svmfit, nextAttack)

predictedNew

## Task 3

library(ggplot2)

g <- ggplot(data[c(-30,-23, -38),], aes(attacker\_size, defender\_size, color=attacker\_king, pch=attacker\_outcome)) +

geom\_point(size=2) +

scale\_color\_manual(values=c("black", "red", "grey","yellow"))+

facet\_wrap(~year, ncol = 3) +

labs(title = "War of the five kings", subtitle = "Carseats data")+

theme(axis.text.x = element\_text(angle=90, hjust=1) )

plot(g)

## Task 4

## Python - Task 5

Define a function filter

import numpy

import math

def linear\_combination(A, M, i, j):

# if len(M) / 2 > int(len(M) / 2):

# filterx = len(M) / 2

sum = 0

filterx = int(len((M[0])) / 2) - (len(M[0]) % 2 == 0)

filtery = int((len(M)) / 2) - (len(M) % 2 == 0)

#print("cent", i, j, len(M))

for y in range(0, len(M)):

tmp = M[y]

for x in range(len(M[0])):

#print(x, y, i, j)

#print(y + i, x + j, tmp[x] \* A[y + i - filtery][x + j - filterx])

sum = sum + tmp[x] \* A[y + i - filtery][x + j - filterx]

return sum

def filter(A, M):

B = numpy.zeros(shape=(len(A), len(A[0])))

filterx = int(len((M[0])) / 2) - (len(M[0]) % 2 == 0)

filtery = int((len(M)) / 2) - (len(M) % 2 == 0)

#print("cent", filterx, filtery)

for i in range(len(A)):

for j in range(len(A[0])):

if i >= filtery and len(A) - i >= len(M) - filtery and j >= filterx and len(A[0]) - j >= len(M[0]) - filterx:

B[i][j] = linear\_combination(A, M, i, j)

else:

B[i][j] = A[i][j]

return B

A = numpy.array([

[2, 3, 1],

[4, 5, 6],

])

M = numpy.array([

[1, 2]

])

A2 = numpy.zeros((10, 10))

A2[2:8,3:7] = 1

Sx = numpy.array([

[-1, 0, 1],

[-2, 0, 2],

[-1, 0, 1]

])

Sy = numpy.array([

[-1, -2, -1],

[0, 0, 0],

[1, 2, 1]

])

print(filter(A, M))

print(A2, filter(A2, Sx))

print(filter(A2, Sy))

## Task 6

class TypedSet:

def \_\_init\_\_(self):

self.mainSet = dict()

def \_\_iter\_\_(self):

for group in self.mainSet.values():

for el in group:

yield el

def add(self, element):

if type(element).\_\_name\_\_ not in self.mainSet:

self.mainSet[type(element).\_\_name\_\_] = set()

self.mainSet[type(element).\_\_name\_\_].add(element)

def add\_many(self, \*args):

for arg in args:

self.add(arg)

ts = TypedSet()

ts.add(False)

ts.add\_many(5, "R", 10, (5, "some tuple"), ("🙂", "smile"))

ts.add("Python")

ts.add(True)

for item in ts:

print(item)

## Task 7

INPUT\_FILE\_PATH = "/home/miku/Idea/Projects/POLSL/PiRaP/Test/graph.txt"

class Graph:

# edges - dictionary, where:

# key - node

# value - list of (node, cost)

def \_\_init\_\_(self):

self.edges = dict()

def \_\_add\_edge(self, starting\_node, ending\_node, cost: float):

if starting\_node not in self.edges:

self.edges[starting\_node] = []

self.edges[starting\_node].append((ending\_node, cost))

if ending\_node not in self.edges:

self.edges[ending\_node] = []

self.edges[ending\_node].append((starting\_node, cost))

@staticmethod

def from\_file(file\_path: str):

file = open(file\_path, "r")

lines = file.readlines()

gr = Graph()

for line in lines:

elements = line.split()

gr.\_\_add\_edge(elements[0], elements[1], float(elements[2]))

return gr

def check\_connection(self):

if len(self.edges) < 2:

return True

unvisited\_nodes = list(self.edges.keys())

visited\_nodes = [unvisited\_nodes[0]]

unvisited\_nodes.remove(unvisited\_nodes[0])

while len(unvisited\_nodes) > 0:

found = False

for edges in self.edges[unvisited\_nodes[0]]:

if edges[0] in visited\_nodes:

found = True

visited\_nodes.append(unvisited\_nodes[0])

unvisited\_nodes.remove(unvisited\_nodes[0])

break

if not found:

return False

return True

# Validation:

gr = Graph.from\_file(INPUT\_FILE\_PATH)

print(gr.check\_connection())