[IS] Domača naloga 1

Niki Bizjak, Martin Preradović

Task 1: Maximization of a non-convex function

A variation of the Rosenbrock function we are interested in is defined as $f(x, y) = (1x)^2 + e(yx^2)^2$ (e is the base of the natural logarithm). It is non-convex. As a warm-up, what does it mean for a function to be non-convex?

A function $f : X \to \mathbb{R}$ is **non-convex** if there exist two points $a, b \in X$, so that there exists a $\lambda \in [0, 1]$, so that $f(\lambda a + (1 - \lambda)b) > \lambda f(a) + (1 - \lambda)f(b)$.

Prerequisites

To solve this task, we will be using the `GA` or **Genetic Algorithm** library, which we can include like so:

```
library(GA)
```

Coding the Rosenbrock function

Code a method f(x, y) that computes a value z, given an input tuple (x, y)

```
f <- function(x, y) {
  (1 - x)^2 + exp(1) * (y - x^2)^2
}
```

Visualizing the function

Code a method that visualizes the Rosenbrock function in 3D, where $x \in [1, 1]$ and $y \in [1, 1]$.

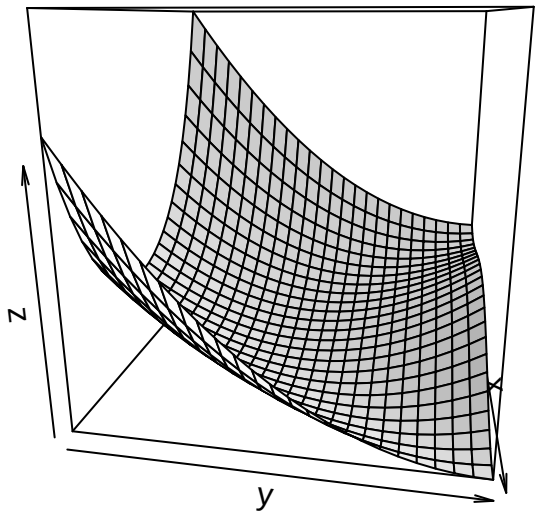
The following function accepts function *f*, *lower* and *upper* bounds for drawing and list of points *xy* to mark on our graph and plots it. It also marks the points in 2d matrix `xy`.

```
plotGraph <- function(f, lower, upper, xy = NULL) {
  x <- seq(from = lower[1], to = upper[1], length = 25)
  y <- seq(from = lower[2], to = upper[2], length = 25)
  z <- outer(x, y, f)
  persp(x, y, z, theta=100, shade=0.2)->transformationMatrix

  if (!is.null(xy)) {
    z = apply(xy, MARGIN = 2, FUN=function(tuple) { f(tuple[1], tuple[2]) })
    points(trans3d(xy[1,], xy[2,], z, pmat = transformationMatrix), col="red", pch=19)
  }
}
```

To actually plot the function f without marking any points, we can do that like so:

```
plotGraph(f, c(-1, -1), c(1, 1))
```



Finding the global maximum

Code a genetic algorithm, that attempts to find the global maximum of this function. Plot the maximum value it finds.

To calculate results, we call the `ga` function and pass the lower and upper bounds of our calculation, the probabilities of crossover and mutations, the maximum number of iterations and population size.

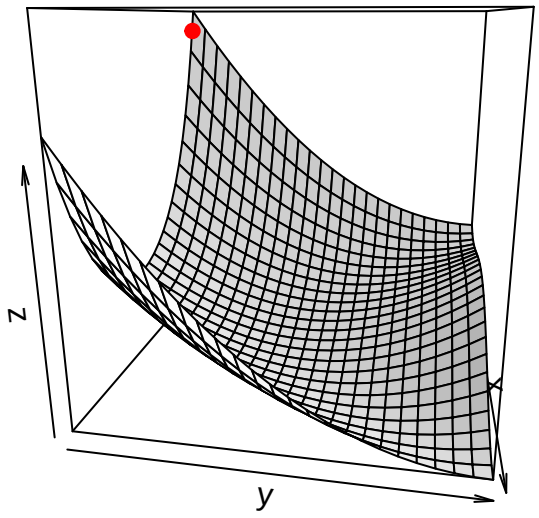
For the fitness function we can simply use the previously defined Rosenbrock function.

```
GA <- ga(
  type = "real-valued",
  fitness =  function(v) { f(v[1], v[2]) }, # Our fitness function
  lower = c(-1, -1), upper = c(1, 1), # The lower and upper search bounds
  popSize = 50, # Population size
  pcrossover = 0.8, pmutation = 0.1, # Probability of crossover and mutation
  maxiter = 100, # Maximum iterations
  monitor = function(obj) {}
)
```

We can plot our result using the next block of code.

```
# Plot the current population
# plotGraph(GA@fitness, GA@lower, GA@upper, t(GA@population))

# Plot the best solution that our genetic algorithm found
plotGraph(GA@fitness, GA@lower, GA@upper, t(GA@solution))
```



If we want to plot the trace of the evolution we can pass a monitor function, that is called every iteration and use our function `plotGraph`, that plots it.

```
GA <- ga(
  type = "real-valued",
  fitness =  function(v) { f(v[1], v[2]) }, # Our fitness function
  lower = c(-1, -1), upper = c(1, 1), # The lower and upper search bounds
  popSize = 50, # Population size
  pcrossover = 0.8, pmutation = 0.1, # Probability of crossover and mutation
  maxiter = 100, # Maximum iterations
  # The monitor function that will plot our graph every iteration
  monitor = function(obj) {
     plotGraph(obj@fitness, obj@lower, obj@upper, t(obj@population))
     Sys.sleep(0.1)
  }
)
```

Discussion

Performance - iterations

How does performance vary when you are increasing the number of iterations?

If the parameter `run = n` is supplied to our ga function, then our genetic algorithm ends when the value of the best candidate does not change for `n` iterations. Otherwise the parameter `maxiter` is used as the `run` parameter and our genetic algorithm does all `maxiter` iterations.

That means, that if we increase the number of iterations, our run time also increases, as it must calculate

all `maxiter` iterations.

Performance - population size

How does performance vary when you are increasing the population size?

The run time of our genetic algorithm increases, as it has to calculate more function values in more points and combine more elements to get new generation.

Local and global maxima

Explain the difference between local and global maxima.

Point x_0 is a global maximum of function $f : X \to \mathbb{R}$ if $(\forall x \in X) f(x_0) \geq f(x)$.

Point x_0 is a local maximum of function $f : X \to \mathbb{R}$ if there exists some $\epsilon > 0$, so that the $f(x_0) \geq f(x)$ for all x within ϵ-distance of x ($|x_0 - x| < \epsilon$).

Task 2: Genetic feature selection

High-dimensional data sets are quite common nowdays. The task of feature selection adresses the issue of identifying such features (= columns), that are the most relevant for a given e.g., classification problem. Individual solution, however you choose to represent it, must be evaluated as follows.

Evaluation of solutions

First, select your preferable classifier. The classifier shall be evaluated using three-fold cross validation. The output of this part is the average performance (justify the measure of your choice). Represent the final solution so that you take into account both the classifier's performance, as well as the number of features.

The dataset you are to use is `DLBCL.csv`, and represents a set of genes associated with B-cell Lymphoma. The target variable is called `class`. The hard constraint on the minimum number of features is 2. The hard constraint on the maximum number of features is 1000

Prerequisites

For this task, a `caret` will be used. It can be included using the following statement:

```
library(caret)
```

Parsing data

The first step is reading and parsing the `DLBCL` csv file. After reading the file, we have to convert the `class` variable of each row to factor and remove the `X` variable, which is the id of each row.

```
dataset = read.table("DLBCL.csv", header = TRUE, sep = ",")
dataset$class <- as.factor(dataset$class)
dataset$X <- NULL
```

Three-fold cross validation

To evaluate our classifier, we will need a three-fold cross validation control. We can create it using the `trainControl` function.

```
control <- trainControl(
  method = "cv", # cross validation
  number = 3, # three fold
)
```

Classifier

For this task, we need a classifier that works in high-dimensional data set. We have decided to use the **k-nearest-neighbors** classifier algorithm. To create model with specified classifier, we can use the following code. It creates a new classifier that is classifying our data using *all* possible features.

```
model = train(
  class ~ .,
  data = dataset,
  method = "knn", # Classifier type (k-nearest-neighbors)
  trControl = control # Control (three-fold cross validation)
)
```

Genetic algorithm

Now it is time to write a genetic algorithm that tries to maximize our model accuracy and minimize number of features needed. Because the feature can either be used in the model training or not, we will be using `binary` genetic algorithm.

Fitness function

So let's first write a fitness function. It should accept a vector $v \in \{0, 1\}^n$ where n is a number of all possible features and $v_i = 1$ if the feature should be included training of our classifier. Then it should train our classifier and calculate its accuracy. It should also take into account both the classifier accuracy and number of features.

For this task, we have decided to use the `k-nearest neighbors` classifier type, as it was the fastest to train. We found that out by trial and error.

In the first part of our function, we convert our numerical vector of ones and zeros to logical vector with values of `true` and `false` and use the R vector indexing to only get the needed columns. After that, our classifier is trained on reduced dataset and its accuracy is used to calculate the result of our fitness function.

```
fitness_function <- function(X) {
  number_of_features = sum(X)

  if (number_of_features < 2 || number_of_features > 1000)
    return(0)

  X = as.logical(X)
  X[length(dataset)] <- T

  current_dataset <- dataset[,X]
  model = train(
    class ~ .,
    data = current_dataset,
    method = "knn", # Classifier type (k-nearest-neighbors)
    trControl = control # Control (three-fold cross validation)
  )

  # Calculate current model accuracy
  max(model$results["Accuracy"]) / number_of_features
}
```

Initial population

The default population of our genetic algorithm consists of vectors, that have aproximately 50% of ones and

50% of zeroes in them. This means, that our genetic algorithm will decrease the number of features rather slowly. Because we want as little features as possible, we should start with vectors that have less than 50% of ones. We have found out that genetic algorithm works best if we generate our starting vectors randomly, with 1% of vector elements being one.

```
initialPopulation <- function(object) {
  matrix(data = as.numeric((runif(object@popSize * object@nBits, 0, 1) < 0.1)),
         nrow = object@popSize,
         ncol = object@nBits)
}
```

Genetic algorithm

```
GA <- ga(
  type = "binary", # binary genetic algorithm

  # Custom fitness and initial population functions
  fitness = fitness_function,
  population = initialPopulation,

  # Number of bits the binary ga should work with
  nBits = length(dataset) - 1,

  popSize = 50, # population size

  # Maximum iterations and iterations without change to end
  maxiter = 150,
  run = 20,

  # The crossover and mutation probablities
  pcrossover = 0.9, pmutation = 0.1,

  parallel = T, seed = 1
)
```

When the genetic algorithm is finished, we get a solution as a vector. To get a classifier we can use the following code.

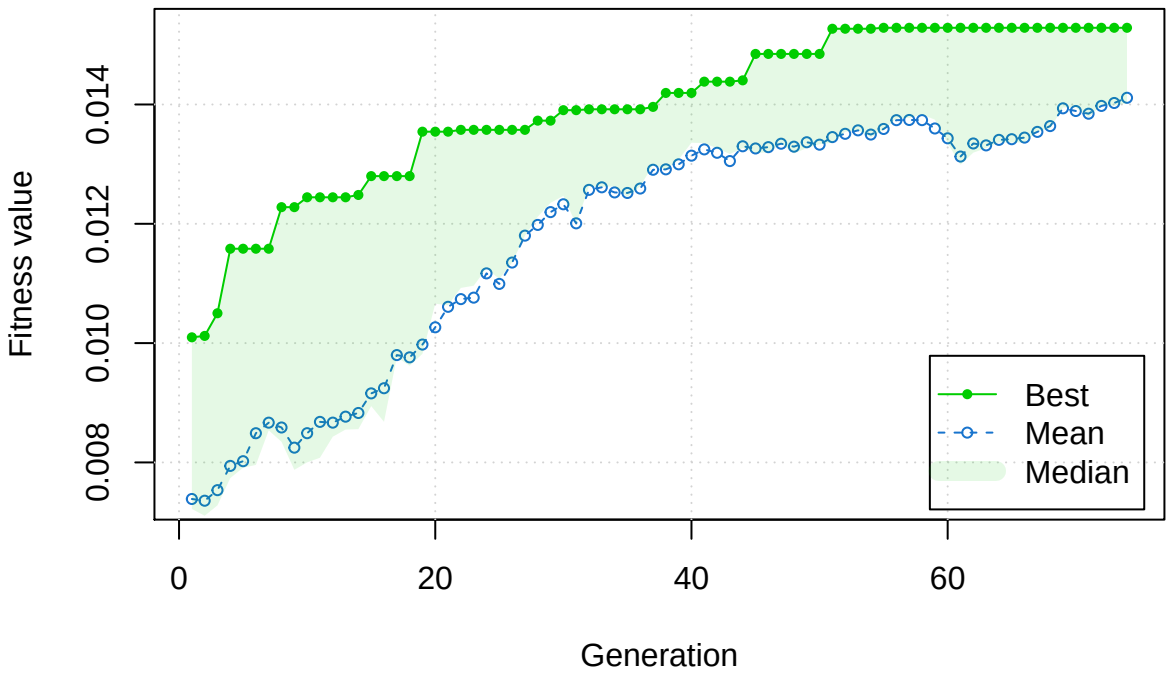
```
result <- as.logical(GA@solution)
result[length(dataset)] <- T
current_dataset <- dataset[,result]
model = train(
  class ~ .,
  data = current_dataset,
  method = "knn", # Classifier type (k-nearest-neighbors)
  trControl = control # Control (three-fold cross validation)
)
```

Summary

The feature selection was successful, the best solution we got had 95% model accuracy and used 51 features.

In the following image, we can see that the value of fitness function improves, that is because of the decrease of the number of features and increase in model accuracy.

```
plot(GA)
```



Discussion

Selection process

Suggest how to improve the selection process.

The selection process can be improved by excluding similar features. This also helps us simplify the model and avoid data overfitting.

Data overfitting

The procedure might overfit the data set. How would you prevent it?

To avoid potential data overfitting we can simplify the model by selecting fewer features.

Properties of the fitness function

What are some of the key properties of the fitness function?

The fitness function takes into account model accuracy and number of features. It divides accuracy of the model with number of features.