ICD_4

June 2, 2023

[ ]:
```
!apt-get install texlive texlive-xetex texlive-latex-extra pandoc
!pip install pypandoc
```

1 Feature selection

In this lesson, we will reduce the number of inputs for a machine learning technique. This can be modelled as an optimization problem: Feature selection problem. In this problem, we need to find the optimal feature subset that contributes most to our predicted variable.

In order to solve this problem, we will use the algorithms presented in previous lessons: GA and VNS.

[2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

1.1 The dataset

The dataset used is Breast Cancer Wisconsin (Diagnostic) which consists of 30 real-valued features. Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image. It also includes the diagnosis (M = malignant, B = benign).

In the next code, we provide a function to load and prepare the dataset for our experiments. It requires that the dataset file is in "Colab Notebook" directory of your drive. In the *Campus Virtual* you can find the dataset file. You have to download this file and upload it in the appropriate directory in your google drive.

[3]:
```
# libraries
import numpy as np
import pandas as pd
from google.colab import drive
import math
import random
import statistics
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import cross_val_score
```

```python
# Single seed for consistent results among different tests
#random.seed(11)

# For using our drive
drive.mount('/content/drive')

def load_data():
  # Load CSV file from your own google drive
  data = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/ICD_4_data.csv')

  # Removed unnecesary columns
  data.drop(['id','Unnamed: 32'],axis=1,inplace=True)

  # Separate values from class
  y = data['diagnosis']
  x = data.drop('diagnosis',axis =1)

  # Final preparation
  y.replace(to_replace='M',value= 1,inplace=True)
  y.replace(to_replace='B',value = 0,inplace=True)
  return x, y
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

1.2 Fitness function

In feature selection problem, we usually can consider two kinds of objectives: * The number of features selected (`noFeatures`). It should be minimized. * The model performance (`accuracy`). It should be maximized.

For this problem, we are going to use a Multi-layer Perceptron classifier (`MLPClassifier()`). This is an Artificial Neural Network for binary classification.

We use the cross-validation F1 score from trained on the individual's solution as the accuracy value. F1 score is used as the accuracy metric because:

- The dataset is quite imbalanced, and using `accuracy_score` could indicate a slight bias towards the majority class.
- A high f1 score indicates a better classification, i.e. a better model performance.

Therefore, as the final fitness function, we will aggregate these two objectives as follows:

$f = \alpha \cdot ((maxFeatures - noFeatures)/maxFeatures) + (1 - \alpha) \cdot accuracy$

We use a lower α value since accuracy is our main objective.

[4]:
```python
# Evaluate a solution
def evaluate_solution(s, cfg):
  # Remove the unselected features
  df = cfg["data"]
```

```
  i=0
  noFeatures = cfg["dimension"]
  for column in data:
    if s[i]==0:
      df = df.drop(column,axis=1)
      noFeatures -= 1
    i=i+1

  model = MLPClassifier()
  scores = cross_val_score(
      model,df,cfg["target"],scoring='f1_macro',n_jobs=-1,cv=2
    ) #using f1_score as it is an imbalanced dataset
  accuracy = scores.mean()
  fitness = cfg["alpha"]*((cfg["dimension"]-noFeatures)/cfg["dimension"]) +
 ↪(1-cfg["alpha"])*accuracy
  return fitness, accuracy, noFeatures
```

To test this function, in the next code, two solutions are provided: * A solution using all the features * A biased random solution

Also, the code includes a utility function to print a solution.

[5]:
```
# return a solution using all the features and its fitness
def complete_solution(cfg):
  s = [1]*cfg["dimension"]
  return s

def random_solution(cfg):
    s = [0]*cfg["dimension"]
    for p in range(len(s)):
      if random.uniform(0,1) < cfg["bias"]:
        s[p] = 1
    return s

# Get the names of the features selected
def print_solution(s, cfg):
  print("\tSolution:",s)
  list_of_features= []
  for i in range(len(s)):
    if s[i]:
      list_of_features.append(cfg["data"].columns[i])
  f, a, nof = evaluate_solution(s, cfg)
  print("\tNumber of selected features:", nof)
  print("\tAccuracy:", a)
  print("\tFitnes:", f)
  print("\tFeatures:", list_of_features)

# Main program
```

```python
data, target = load_data()
configuration = {
    "dimension": data.shape[1], # Number of features
    "data": data, # Values of the features
    "target": target, # classification for each sample in data
    "alpha": 0.2, # Weight for the aggregative function
    "bias": 0.3 # A bias for reducing the number of features in the random
 ↪solution
}

s0 = complete_solution(configuration)
s1 = random_solution(configuration)

print("Complete:")
print_solution(s0, configuration)

print("Random:")
print_solution(s1, configuration)
```

```
Complete:
        Solution: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
        Number of selected features: 30
        Accuracy: 0.9293546955236415
        Fitnes: 0.7434837564189132
        Features: ['radius_mean', 'texture_mean', 'perimeter_mean', 'area_mean',
'smoothness_mean', 'compactness_mean', 'concavity_mean', 'concave points_mean',
'symmetry_mean', 'fractal_dimension_mean', 'radius_se', 'texture_se',
'perimeter_se', 'area_se', 'smoothness_se', 'compactness_se', 'concavity_se',
'concave points_se', 'symmetry_se', 'fractal_dimension_se', 'radius_worst',
'texture_worst', 'perimeter_worst', 'area_worst', 'smoothness_worst',
'compactness_worst', 'concavity_worst', 'concave points_worst',
'symmetry_worst', 'fractal_dimension_worst']
Random:
        Solution: [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0,
1, 1, 0, 1, 1, 1, 1, 0, 0, 0]
        Number of selected features: 10
        Accuracy: 0.8514675728441827
        Fitnes: 0.8145073916086796
        Features: ['concavity_mean', 'radius_se', 'perimeter_se',
'compactness_se', 'radius_worst', 'texture_worst', 'area_worst',
'smoothness_worst', 'compactness_worst', 'concavity_worst']
```

1.3 Algorithms to solve this problem

We will use three different techniques to solve this problem: GA (lesson 1), HC and VNS (lesson 2).

Some considerations about this problem. This problem is a quite time-consuming process. To reduce the execution time in our experiments: * We only use a 2-cross validation process. * The number of steps of the technique is very low (100). * Each technique is only executed one time.

To get more accurate/robust results, we should use 10-cv, run the algorithm 30 or more times and the number of steps should be increased, but we will need a dedicated machine or even use a parallel platform.

In the next cell, we provide the HC code as an example, while you should implement the GA and VNS code (adapting the code from previous lessons).

1.3.1 Hill-climbing algorithm

[6]:
```
# ------------------------- FUNCTIONS ----------------------------------

# Generate a neighbour
# sigma is the maximum number of bits affected by this operator
def mutate(sol, sigma):
    changed_bits = random.randint(1, sigma)
    for i in range(changed_bits):
        pos = random.randint(0, len(sol)-1)
        sol[pos] = 1 - sol[pos] # bitflip
    return sol

# A simple HC
def HC(cfg):
    # Generate an initial solution
    best_sol = random_solution(cfg)
    best_fitness,_,_ = evaluate_solution(best_sol, cfg)
    best_sol_iter = 0
    for i in range(1, cfg["steps"]+1):
        if (i % 10):
            print(".", end="")
        else:
            print("|", end="")
        # Generate a neighbour
        sol = mutate(best_sol, cfg["sigma"])
        fitness,_,_ = evaluate_solution(sol, cfg)
        # Update the best solution if better
        if fitness > best_fitness:
            best_fitness = fitness
            best_sol = sol
            best_sol_iter = i
    return best_sol, best_fitness, best_sol_iter

# --------------------------- MAIN PROGRAM ----------------------------

# Configuration
```

```
# Additional parameters for HC
configuration["steps"] = 100
configuration["sigma"] = 1
print("Running HC ", end="")
s, _, _ = HC(configuration)
print()
print_solution(s, configuration)
```

```
Running HC …|…|…|…|…|…|…
|…|…|…|
        Solution: [0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0,
0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
        Number of selected features: 10
        Accuracy: 0.8450421013424191
        Fitnes: 0.8093670144072687
        Features: ['perimeter_mean', 'smoothness_mean', 'concavity_mean',
'concave points_mean', 'symmetry_mean', 'fractal_dimension_mean',
'perimeter_se', 'area_se', 'compactness_se', 'texture_worst']
```

1.3.2 VNS

Adapt the VNS code of second lesson to solve this problem. For variable neighbourhoods, you can change `sigma` value of `mutate` function from 1 to 10. Use the functions provided in the previous cells (`mutate`, `evaluation_solution`, `random_solution` and `print_solution`).

Note that in the second lesson, we **minimize** the fitness value, and in this problem, we want to **maximize** it.

[7]:
```
# VNS code

def VNS(cfg):
    # Generate an initial solution
    best_sol = random_solution(cfg)
    best_fitness,_,_ = evaluate_solution(best_sol, cfg)
    best_sol_iter = 0

    neighbourhood = 1 #
    MAX_NEIGHBOURHOOD = 10 #

    for i in range(1, cfg["steps"]+1):
        if (i % 10):
            print(".", end="")
        else:
            print("|", end="")
        # Generate a neighbour
        sol = mutate(best_sol, neighbourhood) # cfg["sigma"]
        fitness,_,_ = evaluate_solution(sol, cfg)
        # Update the best solution if better
```

```
        if fitness > best_fitness:
            best_fitness = fitness
            best_sol = sol
            best_sol_iter = i
            neighbourhood = 1 #
        else:
            neighbourhood += 1
            if neighbourhood > MAX_NEIGHBOURHOOD:
                neighbourhood = 1

    return best_sol, best_fitness, best_sol_iter

# --------------------------- MAIN PROGRAM ----------------------------

# Configuration
# Additional parameters for VNS
configuration["steps"] = 100
print("Running VNS ", end="")
s, _, _ = VNS(configuration)
print()
print_solution(s, configuration)
```

```
Running VNS …|…|…|…|…|…|…
.|…|…|…|
        Solution: [1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0,
1, 1, 0, 1, 0, 0, 1, 0, 0, 1]
        Number of selected features: 14
        Accuracy: 0.9115549032133919
        Fitnes: 0.8359105892373803
        Features: ['radius_mean', 'perimeter_mean', 'area_mean',
'smoothness_mean', 'concavity_mean', 'symmetry_mean', 'radius_se',
'concavity_se', 'symmetry_se', 'radius_worst', 'texture_worst', 'area_worst',
'concavity_worst', 'fractal_dimension_worst']
```

1.3.3 GA

Adapt the GA code of the first lesson for this problem. Use the functions provided in the previous cells (`mutate`, `evaluation_solution`, `random_solution` and `print_solution`).

[8]:
```
# GA code

# -------------------------- FUNCTIONS -------------------

def initialize(cfg):
    pop = []
    for i in range(cfg["pop_size"]):
        sol ={
            "string": random_solution(cfg),
```

```python
            "fitness": None
            }
        pop.append(sol)
    return pop

def evaluate_solution_GA(s, cfg):
    fitness, _, _ = evaluate_solution(s, cfg)
    return fitness

def evaluate(pop, cfg):
    for sol in pop:
        sol["fitness"] = evaluate_solution_GA(sol["string"], cfg)

def select(pop):
    return random.choice(pop), random.choice(pop)

# crossover operator (SPX)
def recombination(p1, p2, cfg):
    point = random.randint(1,len(p1)-2)
    child1 = p1[:point]+p2[point:]
    child2 = p2[:point]+p1[point:]
    if evaluate_solution_GA(child1, cfg) > evaluate_solution_GA(child2, cfg):
        return child1
    else:
        return child2

def replacement(pop, s):
    pop.append(s)
    new_pop = sorted(pop, key = lambda i: i["fitness"], reverse=True)
    return new_pop[:-1] # remove the last one

def best_solution(pop):
    new_pop = sorted(pop, key = lambda i: i["fitness"], reverse=True)
    return new_pop[0]

# main algorithm
def GA(cfg):
    # initial seed for replicability
    random.seed(cfg["seed"])

    # Initial steps
    population = initialize(cfg) # build the initial population
    evaluate(population, cfg) # evaluate all solutions in the population
    best_sol = best_solution(population)
    iter_best_sol = 0

    # Evolutive steps
```

```python
    for i in range(1,cfg["steps"]+1):
        if (i % 10):
            print(".", end="")
        else:
            print("|", end="")

        p1, p2 = select(population) # select two parents for crossover
        new_sol = recombination(p1["string"], p2["string"], cfg) # apply␣
↪crossover operator
        new_sol = mutate(new_sol, cfg["sigma"]) # apply mutation operator
        sol = {"string": new_sol, "fitness": evaluate_solution_GA(new_sol,␣
↪cfg)} # evaluate the new solution
        population = replacement(population, sol) # generate the next␣
↪population including the new solution

        # Get the best solution
        best_sol_current = best_solution(population)
        if best_sol_current["fitness"] > best_sol["fitness"]:
            iter_best_sol = i
            best_sol = best_sol_current
    return best_sol["string"], best_sol["fitness"], iter_best_sol

# -------------------------- MAIN PROGRAM -------------------

# Configuration
# Additional parameters for GA
configuration["seed"] = 11
configuration["pop_size"] = 20
configuration["steps"] = 100
configuration["sigma"] = 1

print("Running GA ", end="")
s, _, _ = GA(configuration)
print()
print_solution(s, configuration)
```

```
Running GA …|…|…|…|…|…|…
|…|…|…|
        Solution: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
        Number of selected features: 3
        Accuracy: 0.9054230215744563
        Fitnes: 0.9043384172595651
        Features: ['radius_mean', 'symmetry_se', 'radius_worst']
```

1.4 Exercises

1. Complete the next table with the obtained results

Answer 1:

	Complete solution	HC	VNS	GA
Number of features	30	10	14	3
Model accuracy	93%	85%	91%	91%
Fitness	74%	81%	84%	90%

2. Analyzing these results, which algorithm provides better results? Justify your answer.

Answer 2:

Due to the small number of iterations, tests results are unstable, which makes comparison between HC and VNS hard, as depending on initial random solution, which is better changes.

However, GA prooves to be better than any of them. Due to recombination, it searches widers solution space much faster, and can converge in smaller number of iterations. VNS, for higher sigmas, can also search wider space, however it is random, while GA combines previous solutions from continously upgraded population.

VNS should converge faster than HC (which, especially due to small changes, can get stuck in a local optimum), however in this case, probably due to small number of iterations, when it starts from a very bad solution it may have problem upgrding in restricted time.

Complete

Solution: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Features: [‘radius_mean’, ‘texture_mean’, ‘perimeter_mean’, ‘area_mean’, ‘smoothness_mean’, ‘compactness_mean’, ‘concavity_mean’, ‘concave points_mean’, ‘symmetry_mean’, ‘fractal_dimension_mean’, ‘radius_se’, ‘texture_se’, ‘perimeter_se’, ‘area_se’, ‘smoothness_se’, ‘compactness_se’, ‘concavity_se’, ‘concave points_se’, ‘symmetry_se’, ‘fractal_dimension_se’, ‘radius_worst’, ‘texture_worst’, ‘perimeter_worst’, ‘area_worst’, ‘smoothness_worst’, ‘compactness_worst’, ‘concavity_worst’, ‘concave points_worst’, ‘symmetry_worst’, ‘fractal_dimension_worst’]

HC

Solution: [0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]

Features: [‘perimeter_mean’, ‘smoothness_mean’, ‘concavity_mean’, ‘concave points_mean’, ‘symmetry_mean’, ‘fractal_dimension_mean’, ‘perimeter_se’, ‘area_se’, ‘compactness_se’, ‘texture_worst’]

VNS

Solution: [1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1]

Features: [‘radius_mean’, ‘perimeter_mean’, ‘area_mean’, ‘smoothness_mean’, ‘concavity_mean’, ‘symmetry_mean’, ‘radius_se’, ‘concavity_se’, ‘symmetry_se’, ‘radius_worst’, ‘texture_worst’, ‘area_worst’, ‘concavity_worst’, ‘fractal_dimension_worst’]

GA

Solution: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Features: ['radius_mean', 'symmetry_se', 'radius_worst']

3. (Optional) In the previous experiment, we use an Artificial Neural Network as the machine learning model. Change it for a different technique that you are used in part I or II from this course (K-means, SVM, regression…) and repeat the experiments.

[9]:
```
from sklearn.svm import SVC

# Evaluate a solution - SVM version
def evaluate_solution(s, cfg):
  # Remove the unselected features
  df = cfg["data"]
  i=0
  noFeatures = cfg["dimension"]
  for column in data:
    if s[i]==0:
      df = df.drop(column,axis=1)
      noFeatures -= 1
    i=i+1

  # model = MLPClassifier() - ORIGINAL CLASSIFIER
  model = SVC(C=.1, kernel='linear', gamma=1)
  #model = SVC(kernel='rbf')
  #model = SVC(kernel='poly')

  scores = cross_val_score(
      model,df,cfg["target"],scoring='f1_macro',n_jobs=-1,cv=2
    ) #using f1_score as it is an imbalanced dataset
  accuracy = scores.mean()
  fitness = cfg["alpha"]*((cfg["dimension"]-noFeatures)/cfg["dimension"]) +
 ↪(1-cfg["alpha"])*accuracy
  return fitness, accuracy, noFeatures

# Get the names of the features selected
def print_solution(s, cfg):
  print("\tSolution:",s)
  list_of_features= []
  for i in range(len(s)):
    if s[i]:
      list_of_features.append(cfg["data"].columns[i])
  f, a, nof = evaluate_solution(s, cfg)
  print("\tNumber of selected features:", nof)
  print("\tAccuracy:", a)
  print("\tFitnes:", f)
  print("\tFeatures:", list_of_features)
```

[10]:
```
# Full solution
data, target = load_data()
configuration = {
    "dimension": data.shape[1], # Number of features
    "data": data, # Values of the features
    "target": target, # classification for each sample in data
    "alpha": 0.2, # Weight for the aggregative function
    "bias": 0.3 # A bias for reducing the number of features in the random␣
 ↪solution
}

s0 = complete_solution(configuration)

print("Complete:")
print_solution(s0, configuration)
```

```
Complete:
        Solution: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
        Number of selected features: 30
        Accuracy: 0.9335982044269258
        Fitnes: 0.7468785635415407
        Features: ['radius_mean', 'texture_mean', 'perimeter_mean', 'area_mean',
'smoothness_mean', 'compactness_mean', 'concavity_mean', 'concave points_mean',
'symmetry_mean', 'fractal_dimension_mean', 'radius_se', 'texture_se',
'perimeter_se', 'area_se', 'smoothness_se', 'compactness_se', 'concavity_se',
'concave points_se', 'symmetry_se', 'fractal_dimension_se', 'radius_worst',
'texture_worst', 'perimeter_worst', 'area_worst', 'smoothness_worst',
'compactness_worst', 'concavity_worst', 'concave points_worst',
'symmetry_worst', 'fractal_dimension_worst']
```

[11]:
```
# HC

# A simple HC
def HC(cfg):
    # Generate an initial solution
    best_sol = random_solution(cfg)
    best_fitness,_,_ = evaluate_solution(best_sol, cfg)
    best_sol_iter = 0
    for i in range(1, cfg["steps"]+1):
        if (i % 10):
            print(".", end="")
        else:
            print("|", end="")
        # Generate a neighbour
        sol = mutate(best_sol, cfg["sigma"])
        fitness,_,_ = evaluate_solution(sol, cfg)
```

```python
        # Update the best solution if better
        if fitness > best_fitness:
            best_fitness = fitness
            best_sol = sol
            best_sol_iter = i
    return best_sol, best_fitness, best_sol_iter

# --------------------------- MAIN PROGRAM ----------------------------

# Configuration
# Additional parameters for HC
configuration["steps"] = 100
configuration["sigma"] = 1
print("Running HC ", end="")
s, _, _ = HC(configuration)
print()
print_solution(s, configuration)
```

```
Running HC …|…|…|…|…|…|…
|…|…|…|
	Solution: [1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1,
1, 0, 0, 1, 1, 1, 0, 1, 0, 1]
	Number of selected features: 16
	Accuracy: 0.910557580174927
	Fitnes: 0.8217793974732751
	Features: ['radius_mean', 'texture_mean', 'perimeter_mean',
'smoothness_mean', 'concavity_mean', 'radius_se', 'texture_se', 'smoothness_se',
'symmetry_se', 'fractal_dimension_se', 'radius_worst', 'area_worst',
'smoothness_worst', 'compactness_worst', 'concave points_worst',
'fractal_dimension_worst']
```

[12]:

```python
# VNS

def VNS(cfg):
    # Generate an initial solution
    best_sol = random_solution(cfg)
    best_fitness,_,_ = evaluate_solution(best_sol, cfg)
    best_sol_iter = 0

    neighbourhood = 1 #
    MAX_NEIGHBOURHOOD = 10 #

    for i in range(1, cfg["steps"]+1):
        if (i % 10):
            print(".", end="")
        else:
            print("|", end="")
```

```
        # Generate a neighbour
        sol = mutate(best_sol, neighbourhood) # cfg["sigma"]
        fitness,_,_ = evaluate_solution(sol, cfg)
        # Update the best solution if better
        if fitness > best_fitness:
            best_fitness = fitness
            best_sol = sol
            best_sol_iter = i
            neighbourhood = 1 #
        else:
            neighbourhood += 1
            if neighbourhood > MAX_NEIGHBOURHOOD:
                neighbourhood = 1

    return best_sol, best_fitness, best_sol_iter

# --------------------------- MAIN PROGRAM ----------------------------

# Configuration
# Additional parameters for VNS
configuration["steps"] = 100
print("Running VNS ", end="")
s, _, _ = VNS(configuration)
print()
print_solution(s, configuration)
```

```
Running VNS …|…|…|…|…|…|…
.|…|…|…|
        Solution: [0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1,
0, 0, 0, 1, 0, 0, 1, 0, 1, 0]
        Number of selected features: 14
        Accuracy: 0.9200484353870688
        Fitnes: 0.8427054149763218
        Features: ['texture_mean', 'smoothness_mean', 'compactness_mean',
'concavity_mean', 'concave points_mean', 'fractal_dimension_mean', 'texture_se',
'perimeter_se', 'area_se', 'concave points_se', 'fractal_dimension_se',
'area_worst', 'concavity_worst', 'symmetry_worst']
```

[13]:
```
# GA

def evaluate_solution_GA(s, cfg):
    fitness, _, _ = evaluate_solution(s, cfg)
    return fitness

def evaluate(pop, cfg):
    for sol in pop:
        sol["fitness"] = evaluate_solution_GA(sol["string"], cfg)
```

```
def select(pop):
    return random.choice(pop), random.choice(pop)

# crossover operator (SPX)
def recombination(p1, p2, cfg):
    point = random.randint(1,len(p1)-2)
    child1 = p1[:point]+p2[point:]
    child2 = p2[:point]+p1[point:]
    if evaluate_solution_GA(child1, cfg) > evaluate_solution_GA(child2, cfg):
        return child1
    else:
        return child2

# main algorithm
def GA(cfg):
    # initial seed for replicability
    random.seed(cfg["seed"])

    # Initial steps
    population = initialize(cfg) # build the initial population
    evaluate(population, cfg) # evaluate all solutions in the population
    best_sol = best_solution(population)
    iter_best_sol = 0

    # Evolutive steps
    for i in range(1,cfg["steps"]+1):
        if (i % 10):
            print(".", end="")
        else:
            print("|", end="")

        p1, p2 = select(population) # select two parents for crossover
        new_sol = recombination(p1["string"], p2["string"], cfg) # apply␣
↪crossover operator
        new_sol = mutate(new_sol, cfg["sigma"]) # apply mutation operator
        sol = {"string": new_sol, "fitness": evaluate_solution_GA(new_sol,␣
↪cfg)} # evaluate the new solution
        population = replacement(population, sol) # generate the next␣
↪population including the new solution

        # Get the best solution
        best_sol_current = best_solution(population)
        if best_sol_current["fitness"] > best_sol["fitness"]:
            iter_best_sol = i
            best_sol = best_sol_current
    return best_sol["string"], best_sol["fitness"], iter_best_sol
```

```python
# -------------------------- MAIN PROGRAM -------------------

# Configuration
# Additional parameters for GA
configuration["seed"] = 11
configuration["pop_size"] = 20
configuration["steps"] = 100
configuration["sigma"] = 1

print("Running GA ", end="")
s, _, _ = GA(configuration)
print()
print_solution(s, configuration)
```

```
Running GA …|…|…|…|…|…|…
|…|…|…|
        Solution: [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
1, 0, 1, 0, 0, 0, 0, 0, 0, 0]
        Number of selected features: 5
        Accuracy: 0.9367322131377256
        Fitnes: 0.9160524371768473
        Features: ['radius_mean', 'texture_mean', 'smoothness_se',
'radius_worst', 'perimeter_worst']
```

Answer 3:

Used model:

As an alternative model, Suport Vector Machine was choosen for the following reasons:

- it is mostly used for classification
- it is effective in high-dimensional cases (Which in given example means up to 30)

As can be seen in the table, starting with basic complete dataset model, recived accuracy is little better, which means that the model suits the case better overall.

It is further prooven for the rest of the algorithyms, where in each case accuracy is highly improved and overall fitness as well.

Results:

	Complete solution	HC	VNS	GA
Number of features	30	16	14	5
Model accuracy	93%	91%	92%	94%
Fitness	75%	82%	84%	92%

Details:

Complete

Solution: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Features: ['radius_mean', 'texture_mean', 'perimeter_mean', 'area_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean', 'concave points_mean', 'symmetry_mean', 'fractal_dimension_mean', 'radius_se', 'texture_se', 'perimeter_se', 'area_se', 'smoothness_se', 'compactness_se', 'concavity_se', 'concave points_se', 'symmetry_se', 'fractal_dimension_se', 'radius_worst', 'texture_worst', 'perimeter_worst', 'area_worst', 'smoothness_worst', 'compactness_worst', 'concavity_worst', 'concave points_worst', 'symmetry_worst', 'fractal_dimension_worst']

HC

Solution: [1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1]

Features: ['radius_mean', 'texture_mean', 'perimeter_mean', 'smoothness_mean', 'concavity_mean', 'radius_se', 'texture_se', 'smoothness_se', 'symmetry_se', 'fractal_dimension_se', 'radius_worst', 'area_worst', 'smoothness_worst', 'compactness_worst', 'concave points_worst', 'fractal_dimension_worst']

VNS

Solution: [0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0]

Features: ['texture_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean', 'concave points_mean', 'fractal_dimension_mean', 'texture_se', 'perimeter_se', 'area_se', 'concave points_se', 'fractal_dimension_se', 'area_worst', 'concavity_worst', 'symmetry_worst']

GA

Solution: [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0]

Features: ['radius_mean', 'texture_mean', 'smoothness_se', 'radius_worst', 'perimeter_worst']

[14]:
```
#!jupyter nbconvert --to PDF "/content/drive/MyDrive/Colab Notebooks/ICD_4.
↪ipynb"
```