METAHEURÍSTICAS

PRÁCTICA 2: Técnicas de Búsqueda basadas en Poblaciones

UNIVERSIDAD
DE GRANADA

Problema: Agrupamiento con Restricciones

Alba Casillas Rodríguez
76738108B
3ºA – Grupo de Prácticas 1 (Miércoles 17:30-19:30)
albacaro@correo.ugr.es

Índice

1 - Descripción del problema

PROBLEMA DEL AGRUPAMIENTO CON RESTRICCIONES

El agrupamiento, también llamado ***clustering***, es una tarea de aprendizaje no supervisado que consiste en agrupar un conjunto de objetos de tal manera que los objetos que estén en el mismo ***grupo (cluster)*** sean más ***similares*** entre sí que con los de otros grupos (clusters).

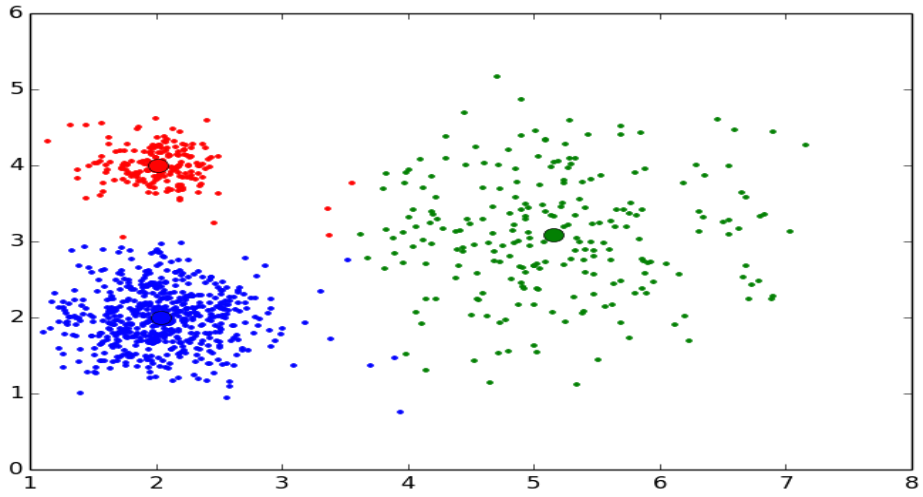
Esta similitud entre los objetos está definido mediante una ***distancia***, normalmente la Euclídea; por lo que haremos uso del término ***centroide***, siendo este el centro geométrico del cluster.

En nuestro problema, al clustering clásico le añadiremos restricciones; convirtiendo este problema en uno de aprendizaje semi-supervisado.

La partición de nuestro grupo de datos, deberá tener en cuenta ***restricciones fuertes***, que obligan a que todos los clusters deben contener al menos una instancia, cada instancia debe pertenecer a un único cluster, y la unión de los clusters debe ser el conjunto de datos.

Además, también deberán cumplirse unas ***restricciones débiles***, donde la partición del conjunto de datos debe minimizar el número de restricciones incumplidas (pudiendo incumplir algunas). Estas restricciones consisten en que dada una pareja de instancias, algunas deberán pertenecer al mismo cluster (***must-link***); y otras, por el contrario, no podrán estar en un mismo cluster (***cannot-link***).

El clustering es una tarea muy importante en exploración de data mining, y una técnica común en el análisis estadístico de datos, aprendizaje automático, reconocimiento de imágenes, etc.



2 - Consideraciones comunes de los algoritmos

- ***2.1- Representación de la solución :*** Representamos nuestra solución como un vector de tamaño "n", cuyos valores van desde 1 hasta "k", siendo este el número de clusters definido.

$$\text{Solución} = \{S_1 , S_2 , \dots , S_n\} \setminus S_i \in K$$

Inicialmente, esta solución es creada mediante un generador de números aleatorios.

Deberemos asegurarnos de que nuestra solución sea **factible**, es decir, no dejaremos ningún cluster vacío.

- ***2.2 – Función objetivo :*** Será nuestra función a minimizar. Está se calculará mediante la fórmula:

$$f = C + (\text{infeasibility} * \lambda)$$

- C será la desviación general.
- Infeasibility será el número de restricciones incumplidas.
- λ(lambda) será el parámetro de escalado para dar relevancia al infeasibility.

- ***2.3 – Cálculo del Infeasibility*** *[Modificada con respecto a la práctica1]:* Llamaremos infeasibility al número de restricciones incumplidas dada una pareja de instancias del conjunto de datos.

$$\text{Infeasibility} = \Sigma^{n}_{i=0} \Sigma^{n}_{j=i+1} V(x_i, x_j)$$

Aunque sea una de las funciones utilizadas en la práctica 1, para esta práctica he modificado la estructura de datos usada para guardar las restricciones, con motivo de agilizar las ejecuciones de conjuntos de datos.

He ustituido la matriz de restricciones por una lista que almacena datos de la manera:
[dato1, dato2, valor] , donde valor toma los valores: 1 (ML) o -1 (CL)

De esta manera, el valor de infeasibility pasa a ser calculado de la siguiente manera:

```
def calcular_infeasibility(datos, v_solucion, restricciones):
    para i ∈ {0...len(restricciones)}:
        si  restricciones[i][2] == -1 && v_solucion[i][0] == v_solucion[i][1]:
            # INCUMPLE CANNOT-LINK
            infeasibility++
        si  restricciones[i][2] == 1 && v_solucion[i][0] != v_solucion[i][1]:
            # INCUMPLE CANNOT-LINK
            infeasibility++
```

- *2.4 – Generación de la población :* La población estará formada por "long_población" (en los AG = 50 y en los AM = 10) cromosomas.

Para ello, se ha creado una clase Cromosoma compuesta por un constructor que inicializa a cero sus tres atributos : el vector solución, el valor de la función objetivo y los valores de los centroides. Esta clase solamente contendrá el constructor y una función usada como criterio de ordenación de los cromosomas (según qué tan bajo sea su valor de la función objetivo); no implementamos más métodos ya que simplemente queremos usar esta clase como una estructura equivalente a un "struct" que almacene los valores.

```
def generar_poblacion_aleatoria(long_poblacion, k, n_genes, d):
    poblacion = []

    for i ∈ {0...long_poblacion}:
        crom = Cromosoma(n_genes,d) #Creamos nuestro objeto de la clase Cromosoma
        poblacion ← cromosoma
        for j ∈ {0...n_genes}:
            poblacion[i].v_solucion[j] = random{1..k}

        for j ∈{0...k}:    #Las solcuiones tienen que ser factibles
            si cluster_vacio(poblacion[i].v_solucion, j+1):
                reparar( poblacion[i].v_solucion, j+1)

    return poblacion
```

- *2.5 – Cruce :* Para el cruce, se han implementado dos métodos:

- Cruce uniforme: el hijo estará formado por la mitad de los genes de un padre y la otra mitad del segundo padre. Para saber qué genes se copiarán de cada padre, calcularemos las posiciones de manera aleatoria

-Segmento fijo: el hijo estará formado por un segmento de tamaño "v_segmento" de genes fijo de un padre (en mi caso, he elegido el segmento del mejor padre, para así favorecer la explotación). Tanto el tamaño del segmento como el inicio de este, han sido calculados de forma aleatoria.

El segmento restante, será calculado mediante cruce uniforme.

```
def operador_uniforme(padre1, padre2, n_genes, d):
        aleatorios ← zeros[n_genes/2]
        hijo = Cromosoma(n_genes,d)

        for i ∈{0...n_genes/2}:
                aleatorios[i] = random(0...n_genes-1}

        #Hacemos una copia del segundo padre para que de esta forma solo tengamos que modificar
        #las posiciones generadas aleatoriamente
        hijo ← copy(padre2)

        for i ∈aleatorios:
                hijo.v_solucion[i] ← copy(padre1.v_solucion[i])

        return hijo

def segmento_fijo(padre1, padre2, n_genes, d):
        hijo = Cromosoma(n_genes/2)
        tam_segmento = random(0..n_genes-1)
        inicio_segmento = random(0..n_genes-1)
        num_elementos ← 0

        elegimos al mejor padre comparando los valores de su función objetivo. Será mejor el que
        menor valor de la función objetivo tenga.

        Posicion = inicio_segmento

        mientras que num_elementos < tam_segmento:
                hijo.v_solucion[posicion] ← copy(mejor_padre.v_solucion[posicion]
                posicion = (posicion + 1)%n_genes
                num_elementos++

        restates = n_genes – tam_segmento

        Los genes restantes se calcularán como hemos explicado en el coste uniforme. Generando
        números aleatorios teniendo en cuenta que hay que hacer un "%n_genes" para que los
        valores no se salgan del vector.

        return hijo
```

Ahora que tenemos los dos operadores de cruce implementado, procedo a explicar el cruce en sí. Para el algoritmo genético generacional, realizo un "num_esperado_cruces"

num_esperado_cruces = probabilidad_cruce * (long_poblacion/2))

de iteraciones, donde en cada una de ellas genero dos hijos aplicando el operador de cruce deseado.

Una vez generados los dos hijos, compruebo para cada uno de ellos si sus soluciones son factibles, recorriendo los clusteres de manera que si alguno de ellos no tiene ningún elemento, repararlo.

La reparación consiste en generar una posición aleatoria entre 0 y n_genes valores y asignarle el cluster vacio.

En el algoritmo genético estacionario, como se ha seleccionado una población de dos padres unicamente, siempre se cruzan, de la misma manera que en el algoritmo generacional: generando dos hijos cuyas soluciones sean factibles.

```
def cruce():
    contador ← 0
    individuo ← 0

    #copiamos la población por si al hacer el cruce, no se consiguen tantos hijos como
    #cromosomas hay en la población
    pob_intermedia ← copy(poblacion)

    #En el AGE , num_esperado_cruce = 1
    while contador < num_esperado_cruce:
        hijo ← operador(poblacion[individuo],poblacion[individuo+1],n_genes,d)
        otro_hijo ← operadores(poblacion[individuo],poblacion[individuo+1],n_genes,d)

        #Comprobamos para los dos hijos si generan soluciones factibles
        si hijo.solucion no factible:
            reparar

        pob_intermedia[individuo] ← hijo
        pob_intermedia[individuo+1]← otro_hijo

        individuo = individuo +2
        contador++

    return pob_intermedia
```

- *2.6– Mutación:* La mutación se realizará de dos maneras diferentes.

En el caso del algoritmo generacional, realizaremos la mutación un "num_esperado_mutacion"

num_esperado_mutacion = probabilidad_poblacion * (long_poblacion*n_genes))

donde en cada iteración se calculará el cromosoma y gen a mutar basado en el código de ejemplo proporcionado en prácticas. Tendremos en cuenta que si el nuevo valor tras la mutación era el mismo al original, lo recalcularemos.

En el algoritmo genético estacionario, al solo tener dos cromosomas, no tendría sentido realizar la mutación de la misma manera que en generacional. Por eso, generamos dos valores aleatorios, uno para cada cromosoma; si este valor es menor a una **tasa = probabilidad_mutacion * n_genes**, decidiremos

aleatoriamente el gen a mutar del cromosoma y su valor; asegurándonos se asignarle un valor distinto al original.

```
Def mutacion():
    si elitista == True:
        mu_next ← ceil(log(random / log(1.0-prob_mutacion)
        contador ← 0

        mientras que contador < num_esperado_mutacion:
            i ← (int(mu_next/n_genes)%log_poblacion)
            j ← (int(mu_next%n_genes))

            nuevo_valor ←  random(1..k)

            mientras que nuevo_valor == cluster:
                nuevo_valor ← random(1..k)

            poblacion[i].v_solucion[j] ← nuevo_valor

            mu_next ← ceil(log(random / log(1.0-prob_mutacion)
            contador++
        mu_next = mu_next – (n_genes*long_poblacion)
    else:
        aleatorio1 = random()
        aleatorio2 = random()

        if aleatorio < tasa: #se comprueba con ambos aleatorios
            gen ←  random(0..n_genes-1)
            nuevo_valor ← random(1..k)

            mientras que nuevo_valor == cluster:
                nuevo_valor ← random(1..k)

            poblacion[crom].v_solucion[gen] ← nuevo_valor
    return poblacion
```

3 – Descripción de los algoritmos

ALGORITMO GENÉTICO

Es un algoritmo ***basado en poblaciones*** el cual hace evolucionar una población de individuos (es decir, nuestros cromosomas) sometiéndola a acciones aleatorias que imitan el comportamiento de la ***evolución biológica*** (mutaciones y recombinaciones genéticas), así como a una selección que decide cuáles serán los individuos más adaptados para sobrevivir.

El criterio de parada de este algoritmo son *MAX_EVAL = 100000* evaluaciones (aproximadamente) de la función objetivo.

La estructura de nuestro algoritmo genético será:

```
def AG(datos, restricciones, n, d, long_poblacion, prob_cruce, prob_muta):
    num_eval = 0
    num_esperado_cruce, num_esperado_mutacion #Ya explicado su cálculo
    lambda ← calcular_lambda() #Función explicada en la práctica1

    poblacion ← generar_poblacion_aleatoria
    # Evaluar población: para cada uno de los cromosomas de la poblacion,
    # recalcularemos sus centroides y el valor de su función objetivo
    num_eval ← evaluar_poblacion(poblacion)

    # Calcula mejor peor: compara los valores de la función objetivo para cada
    # cromosoma de la población quedándonos con el más bajo (mejor) y el más alto (peor), y
    # sus posiciones
    mejor_cromosoma, peor_cromosoma, pos_mejor, pos_peor = calcula_mejor_peor()

    mientras que num_eval < MAX_EVAL:
        #Aplicamos el método de selección: SELECCIÓN POR TORNEO se eligen aleatoriamente
        # dos individuos de la población y nos  quedaremos con el mejor de ellos. En el AGG se
        # aplican tantos torneos como individuos existan en la población y en el AGE solamente
        # dos, para elegir a los dos padres.
        Si elitista:
            tope = long_poblacion
        si no:
            tope = 2

        num_torneos ←0   nueva_poblacion ←  []

        mientras que num_torneos < tope:
            aleatorio1 ←  random(0..long_poblacion-1)
            aleatorio2 ←  random(0..long_poblacion-1)

            elegimos cual es el mejor cromosoma (el demenor valor de la función objetivo)

            nueva_poblacion ← mejor_cromosoma

            num_torneos++

        nueva_poblacion ← cruce
```

```
nueva_poblacion_mutada ← mutar

# Tras el cruce y la mutación, pasamos a REEMPLAZAR nuestra población por la nueva.
Si elitista: # La nueva población sustituye a la actual, salvando su mejor cromosoma
        poblacion ← nueva_poblacion_mutada

        si mejor cromosoma no esta en poblacion:
                num_eval ← evaluar_poblacion(poblacion)
                mejor_cromosoma, peor_cromosoma, pos_mejor, pos_peor =
                                                        calcula_mejor_peor()
                poblacion[pos_peor] ← copy(mejor_cromosoma)
si no: #Los dos descendientes sustituyen a los dos peores de la poblacion actual
        # Segundo peor: recorremos la poblacion quedandonos con el cromosoma de
        # menor valor de la función objetivo que NO sea el peor cromosoma
        peor2, pos_peor2 = segundo_peor(peor_cromosoma)
        num_eval ← evaluar_poblacion(nueva_poblacion_mutada)
        mejor_hijo, mejor_hijo2, pos_mejor, pos_mejor2 =  calcula_mejor_peor()

        si nueva_poblacion_mutada[pos_mejor].f_objetivo < peor_cromosoma.f_objetivo:
                poblacion[pos_peor] ← copy(nueva_poblacion_mutada[pos_mejor]

                si nueva_poblacion_mutada[pos_mejor2].f_objetivo < peor_2.f_objetivo:
                        poblacion[pos_peor2] ← copy(nueva_poblacionmutada[pos_mejor2]

# Tras el reemplazo, solo evaluamos la poblacion de nuevo si nos enconramos en el AGG
# ya que en el AGE se evaluaron durante el reemplazamiento
si elitista:
        num_eval ← evaluar_poblacion(poblacion)

mejor_cromosoma, peor_cromosoma, pos_mejor, pos_peor = calcula_mejor_peor()
```

ALGORITMO MEMÉTICO

Los ***algoritmos meméticos*** son técnicas de optimización que combinan metaheurísticas, tales como algoritmos evolutivos, y la mejora local.

En nuestro problema, combinaremos el ***algoritmo genético generacional elitista de segmento fijo*** con una ***búsqueda local suave***, la cual aplicaremos cada 10 iteraciones a cada cromosoma de nuestra población. Podemos considerarlo una optimización ya que, aunque el algoritmo genético encamina bien hacia la solución óptima, puede estancarse durante varias evaluaciones en un óptimo local; para ello, realizamos la búsqueda local para que converja más rápidamente; además, el hecho de no realizarla en todas las iteraciones favorecerá a que haya un ***equilibrio entre la exploración y la explotación***.

Primero, procedo a indicar la estructura de la búsqueda local suave, la cual recorre en un orden aleatorio los genes del cromosoma al que se le aplica la búsqueda, de manera que le asignaremos los valores de los clusteres (distinto al suyo actual) en busca de aquel que nos proporcione una mayor minimización del valor de la función objetivo. Si el cromosoma es localmente optimizable, en el mejor de los casos se recorrerá una única vez, de lo contrario, un contador de fallos cuyo valor máximo es *MAX_ERROR = 0.1*n_genes*, crecerá rápidamente y la búsqueda local se detendrá.

```
def BusquedaLocal(cromosoma, n_genes):
        rsi ← shuffle(n_genes) #Desordenamos los indices para recorrerlos
        fallos ← 0   mejora ← True  i ← 0
        v_obj_antiguo ← cromosoma.f_objetivo

        crom_aux ← Cromosoma(n_genes,d)
        mejor_cluster ← crom_aux.v_solucion[rsi[0]]

        mientras que (mejora or fallos < MAX_ERROR) y i < n_genes:
                mejora ← False
                cluster_actual ← cromosoma.v_solucion[rsi[i]]
                for j € {0...k}:
                        crom_aux.v_sol[rsi[i]] ← j+1

                        si !cluster_vacio(crom_aux.v_solucion, j+1):
                                recalcular_centroides(crom_aux.centroides)
                                calcular_funcion_objetivo
                                num_eval++

                                # Nos quedaremos con el que más mejore
                                si crom_aux.f_objetivo < f_obj_antiguo:
                                        mejora ← True
                                        mejor_cluster ← j+1
                #Actualizamos los valores de crom_aux
                crom_aux.v_solucion[rsi[i]] ← mejor_cluster

                si NO mejora:
                        fallos++
                i++

        return cromosoma, num_eval
```

Por tanto, nuestro algoritmo memético resultará de la combinación del AGG-SF y la búsqueda local descrita:

La condición de parada será *MAX_EVAL = 100000* evaluaciones de la función objetivo; y se añadirán dos nuevos parámetros: tipo (para indicar si vamos a ejecutar AM-1.0-1.0MEJ o no) y tasa (1.0 o 0.1).

```
def AM(datos, restricciones, n, d, long_poblacion, prob_cruce, prob_muta, tipo, tasa):
        num_eval ← 0,  num_generaciones ← 0
        num_esperado_cruce, num_esperado_mutacion #Ya explicado su cálculo
        lambda ← calcular_lambda() #Función explicada en la práctica1

        poblacion ← generar_poblacion_aleatoria
        num_eval ← evaluar_poblacion(poblacion)
        mejor_cromosoma, peor_cromosoma, pos_mejor, pos_peor = calcula_mejor_peor()

        mientras que num_eval < MAX_EVAL:

                nueva_poblacion ← seleccion
                nueva_poblacion ← cruce
                nueva_poblacion_mutada ← mutar


                si num_genraciones%10 == 0: # La BL se aplicara cada 10 generaciones
                        si tipo = 'MEJ'
                                num_eval ← evaluar_poblacion(nueva_poblacion_mutada)
                                # Ordenaremos la población para elegir a los mejores
                                lista ← sorted(nueva_poblacion_mutada)

                                for i €{0..0.01*len(lista)}:
                                        nueva_poblacion_mutada[i], num_eval = BUSQUEDA_LOCAL
                        si no:
                                for i €{0..len(nueva_poblacion_mutada)}:
                                        valor ← random()
                                        si valor < tasa:
                                                nueva_poblacion_mutada[i], num_eval = BUSQUEDA_LOCAL

                poblacion ← reemplazar(nueva_poblacion_mutada)
                num_eval ← evaluar_poblacion(poblacion)
                mejor_cromosoma, peor_cromosoma, pos_mejor, pos_peor = calcula_mejor_peor()
                num_generaciones ++
```

4 – Algoritmos comparativos

ALGORITMO GREEDY

La solución greedy para este problema estará basado en el algoritmo ***k-medias***, donde además se tendrá en cuenta el cumplimiento de las ***restricciones*** débiles.

K-medias es un método de agrupamiento en el cual se divide un conjunto de n objetos en k grupos, donde cada objeto pertenece al gurpo cuyo valor medio es más cercano.

Por ello, inicialmente barajaremos los índices para recorrer el conjunto de datos de forma aleatoria sin repetición y asignaremos cada instancia al clúster más cercano de entre los que produzcan un menor aumento en el valor de infeasibility.

Guardaremos la ***infeasibility*** de asignar a cada instancia los clústeres, para que de haber varios con infeasibility minima, podamos elegir aquel que tiene menor distancia dato-centroide.

Una vez hemos obtenido la nueva partición, se deberá ***re-calcular los centroides.***

Iteraremos el algoritmo mientras no haya cambio en la partición.

```
def GREEDY(datos, restricciones, centroides, n, d):
    random.shuffle(rsi) # rsi será un vector cuyos valores van de 0 a n

    mientras que ( hay_cambio ) && ( num_iteraciones < MAX_ITERACIONES=10000):
        num_iteraciones++
        anterior_solucion ← v_solucion

        for i ∈ {0...rsi}:
            for j i ∈ {0...k}:
                v_solucion[i] ← j #Asignamos a cada instancia un cluster "j"
                infeasibility ← caculamos infeasibility(v_solucion)

                #Guardo en un vector los valores de infeasibility de asignar a una
                #instancia Xi al cluster j
                v_infeasibility[j] ← infeasibility

            #De los valores de v_infeasibility, guardaremos en un vector auxiliar
            #(v_minimos) las posiciones de los valores mínimos.
            for c ∈ {0...k}:
                si v_infeasibility[c] < minimo:
                    minimo ←  valor_infeasibility
                    v_minimos ←  posicion
```

```
                    #Tras obtener las posiciones cuyo valor del infeasibility es mínimo.
                    # Nos quedaremos con el centroide que tenga la menor distancia
                    for c ∈ {0...len(v_minimos)}:
                            distancia ← distancia(datos,centroides)
                            v_distancias[c] ← distancia

                    for c ∈ {0...len(v_distancias)}:
                            si v_distancias[c] ≤ minimo:
                                    minimo ←  v_distancias[c]
                                    centroide_cercano ←  v_minimos[c]

                    #Una vez obtenemos el centroide más cercano con menor valor de
                    #infeasibility, lo asignamos al vector solución

                    v_solucion[i]  ← centroide_cercano

            #Recalculamos centroides
            recalcular_centrodies(datos, v_solucion, centroides, d)

            #Comprobamos si ha habido un cambio en la partición
            cambio ← cambio(v_solucion, anterior_solucion)

    return v_solucion
```

5 – Manual de usuario

Para ejecutar la práctica solo debe indicarse qué conjunto de datos se desea ejecutar.

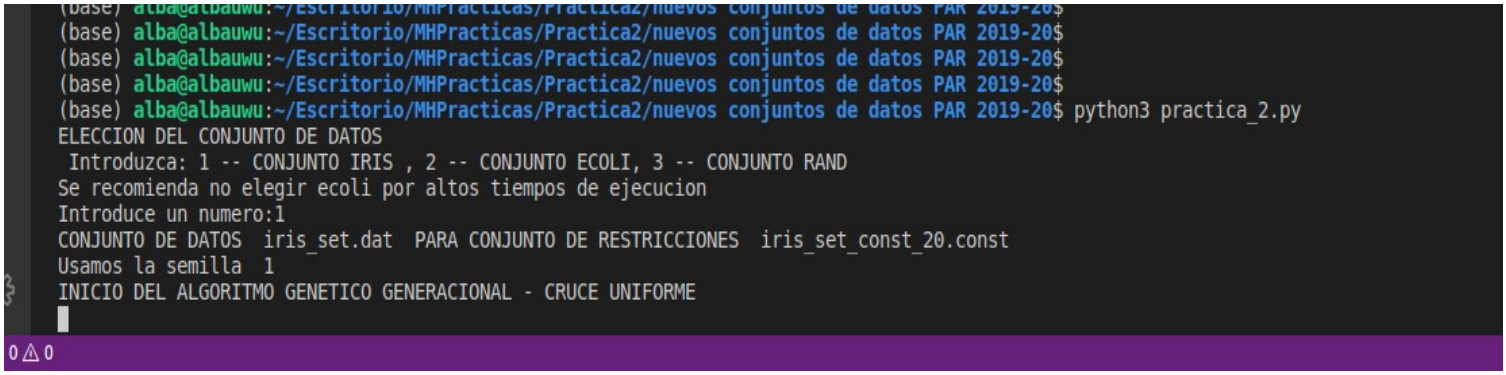
Tras indicar el conjunto de datos, se ejecutarán ambos conjuntos de restricciones 5 veces.

NO HACE FALTA INDICAR LAS SEMILLAS.

Las semillas utilizadas: (1,3,2,7, 5) han sido guardadas en un vector, de manera que por cada ejecución se seleccionará una semilla diferente.

Esta práctica se ha implementado en python3.

Ejemplo de ejecución:



** Cabe destacar que en el programa ejecutable se ha comentado el código con el que se realizan de manera automática las tablas y gráficas proporcionadas para el análisis de resultados.

6 – Tablas de resultados

6.1 – Algoritmo Genético Generacional

Resultados obtenidos en el AGG-UN con 10% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,11	0,00	0,11	248,82	7,20	161,00	11,49	823,83	0,12	0,00	0,12	246,18	2,67	0,00	2,67	339,31
Ejecución 2	0,11	0,00	0,11	244,47	8,04	146,00	11,93	809,41	0,12	0,00	0,12	230,04	2,15	77,00	4,96	391,46
Ejecución 3	0,11	0,00	0,11	236,75	8,04	106,00	10,87	814,64	0,12	0,00	0,12	134,04	2,67	0,00	2,67	419,34
Ejecución 4	0,11	0,00	0,11	203,35	7,94	257,00	14,79	819,83	0,12	0,00	0,12	137,46	2,67	0,00	2,67	417,12
Ejecuión 5	0,11	0,00	0,11	182,31	8,16	192,00	13,28	815,13	0,12	0,00	0,12	139,08	2,67	0,00	2,67	306,00
Media	0,11	0,00	0,11	223,14	7,88	172,40	12,47	816,57	0,12	0,00	0,12	177,36	2,57	15,40	3,13	374,65

Resultados obtenidos en el AGG-UN con 20% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,11	0,00	0,11	256,79	6,58	476,00	12,93	1028,77	0,12	0,00	0,12	223,29	2,67	0,00	2,67	424,78
Ejecución 2	0,11	0,00	0,11	247,86	6,64	204,00	9,36	1005,65	0,12	0,00	0,12	271,07	2,67	0,00	2,67	433,72
Ejecución 3	0,11	0,00	0,11	377,60	7,46	90,00	8,66	1016,64	0,12	0,00	0,12	323,99	2,67	0,00	2,67	432,12
Ejecución 4	0,11	0,00	0,11	305,40	7,74	354,00	12,45	1018,91	0,12	0,00	0,12	303,83	2,67	0,00	2,67	421,70
Ejecuión 5	0,11	0,00	0,11	277,34	7,35	232,00	9,72	1010,37	0,12	0,00	0,12	215,27	2,67	0.0	2,67	414,36
Media	0,11	0,00	0,11	293,00	7,16	271,20	10,62	1016,07	0,12	0,00	0,12	267,49	2.67	0.0	2,67	425,34

Resultados obtenidos en el AGG-SF con 10% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,11	0,00	0,11	221,58	7,13	502,00	20,51	815,39	0,12	0,00	0,12	259,06	2,67	0,00	2,67	368,86
Ejecución 2	0,11	0,00	0,11	223,68	6,50	64,00	8,20	812,44	0,12	0,00	0,12	136,45	2,67	0,00	2,67	306,73
Ejecución 3	0,11	0,00	0,11	203,35	7,57	363,00	17,25	821,06	0,12	0,00	0,12	140,89	2,67	0,00	2,67	478,08
Ejecución 4	0,11	0,00	0,11	220,53	7,04	462,00	19,35	812,24	0,12	0,00	0,12	141,17	2,67	0,00	2,67	331,89
Ejecución 5	0,11	0,00	0,11	210,69	7,20	96,00	9,76	815,39	0,12	0,00	0,12	189,19	2,67	0,00	2,67	503,95
Media	0,11	0,00	0,11	215,97	7,09	297,40	15,01	815,30	0,12	0,00	0,12	173,35	2,67	0,00	2,67	397,90

Resultados obtenidos en el AGG-SF con 20% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,11	0,00	0,11	246,03	6,82	275,00	10,48	1084,33	0,12	0,00	0,12	256,58	2,67	0,00	2,67	424,78
Ejecución 2	0,11	0,00	0,11	311,34	6,68	204,00	9,40	993,62	0,12	0,00	0,12	300,90	2,67	0,00	2,67	433,78
Ejecución 3	0,11	0,00	0,11	286,19	7,60	233,00	10,70	1013,56	0,12	0,00	0,12	333,35	2,67	0,00	2,67	432,14
Ejecución 4	0,11	0,00	0,11	347,52	7,93	225,00	10,93	1004,56	0,12	0,00	0,12	272,32	2,67	0,00	2,67	418,68
Ejecución 5	0,11	0,00	0,11	338,35	6,89	256,00	10,39	1023,78	0,12	0,00	0,12	222,41	2,67	0,00	2,67	414,36
Media	0,11	0,00	0,11	305,88	7,18	238,60	10,38	1023,97	0,12	0,00	0,12	277,11	2,67	0,00	2,67	424,75

Resultados obtenidos en el AGE-UN con 10% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,11	0,00	0,11	268,54	6,36	106,00	9,19	854,75	0,12	0,00	0,12	258,63	2,67	0,00	2,67	391,17
Ejecución 2	0,11	0,00	0,11	210,11	6,04	26,00	6,74	873,09	0,12	0,00	0,12	252,89	2,67	0,00	2,67	424,20
Ejecución 3	0,11	0,00	0,11	293,36	5,27	19,00	5,77	871,43	0,12	0,00	0,12	228,95	3,29	58,00	5,41	492,27
Ejecución 4	0,11	0,00	0,11	313,20	5,60	48,00	6,88	864,93	0,12	0,00	0,12	237,66	2,67	0,00	2,67	440,35
Ejecución 5	0,11	0,00	0,11	214,21	4,70	62,00	6,35	858,85	0,12	0,00	0,12	156,31	2,67	0,00	2,67	436,31
Media	0,11	0,00	0,11	259,88	5,59	52,20	6,98	864,61	0,12	0,00	0,12	226,89	2,79	11,60	3,22	436,86

Resultados obtenidos en el AGE-UN con 20% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,11	0,00	0,11	249,12	5,21	41,00	5,76	1117,20	0,12	0,00	0,12	250,78	2,67	0,00	2,67	438,64
Ejecución 2	0,11	0,00	0,11	319,93	4,57	81,00	5,65	1009,13	0,12	0,00	0,12	325,64	2,67	0,00	2,67	472,53
Ejecución 3	0,11	0,00	0,11	286,82	3,71	50,00	4,37	1085,44	0,12	0,00	0,12	253,71	3,29	58,00	5,41	418,92
Ejecución 4	0,11	0,00	0,11	240,19	4,88	61,00	5,70	1041,94	0,12	0,00	0,12	230,04	2,67	0,00	2,67	440,30
Ejecución 5	0,11	0,00	0,11	355,41	4,92	62,00	5,87	153,78	0,12	0,00	0,12	293,26	2,67	0,00	2,67	466,31
Media	0,11	0,00	0,11	290,29	4,66	59,00	5,47	881,50	0,12	0,00	0,12	270,69	2,79	11,60	3,22	447,34

Resultados obtenidos en el AGE-SF con 10% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,11	0,00	0,11	207,68	6,02	86,00	8,32	871,35	0,12	0,00	0,12	231,45	2,67	0,00	2,67	420,39
Ejecución 2	0,11	0,00	0,11	251,22	4,82	24,00	5,46	881,11	0,12	0,00	0,12	171,78	2,67	0,00	2,67	370,85
Ejecución 3	0,11	0,00	0,11	298,04	5,77	60,00	7,37	872,42	0,12	0,00	0,12	154,78	2,67	0,00	2,67	543,46
Ejecución 4	0,11	0,00	0,11	259,74	5,21	21,00	5,77	878,38	0,12	0,00	0,12	153,89	2,67	0,00	2,67	339,19
Ejecución 5	0,11	0,00	0,11	271,74	4,35	40,00	5,42	899,89	0,12	0,00	0,12	251,00	2,67	0,00	2,67	430,97
Media	0,11	0,00	0,11	257,68	5,24	46,20	6,47	880,63	0,12	0,00	0,12	192,58	2,67	0,00	2,67	420,97

Resultados obtenidos en el AGE-SF con 20% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,11	0,00	0,11	282,77	3,67	58,00	4,44	1134,46	0,12	0,00	0,12	287,61	2,67	0,00	2,67	440,88
Ejecución 2	0,11	0,00	0,11	331,29	4,90	60,00	5,70	1053,10	0,12	0,00	0,12	324,05	2,67	0,00	2,67	456,63
Ejecución 3	0,11	0,00	0,11	317,49	4,61	76,00	5,62	1096,85	0,12	0,00	0,12	285,78	2,67	0,00	2,67	546,81
Ejecución 4	0,11	0,00	0,11	366,95	5,06	41,00	5,61	1097,48	0,12	0,00	0,12	337,80	2,67	0,00	2,67	444,04
Ejecución 5	0,11	0,00	0,11	352,16	4,97	69,00	5,74	1110,32	0,12	0,00	0,12	288,80	2,67	0,00	2,67	439,01
Media	0,11	0,00	0,11	330,13	4,64	60,80	5,42	1098,44	0,12	0,00	0,12	304,81	2,67	0,00	2,67	465,48

6.2 – Algoritmo Memético

Resultados obtenidos en el AM-10-1.0 con 10% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,12	19,00	0,24	294,48	7,61	1308,00	42,47	1134,91	0,12	10,00	0,19	258,63	2,93	46,00	4.61	319,71
Ejecución 2	0,12	6,00	0,27	275,66	6,98	1349,00	40,71	1007,98	0,12	12,00	0,18	252,89	2,77	39,00	3,75	333,45
Ejecución 3	0,12	0,00	0,12	293,36	8,90	1212,00	39,20	1225,34	0,12	4,00	0,22	228,95	3,01	58,00	4,46	356,02
Ejecución 4	0,12	24,00	0.72	301,03	8,42	903,00	31,00	1164,08	0,12	10,00	0,19	237,66	2,67	60,00	4,17	307,12
Ejecución 5	0,12	13,00	0,45	264,21	7,03	1113,00	34,86	1135,23	0,12	10,00	0,19	156,31	2,93	43,00	4,01	322,09
Media	0,12	12,40	0,27	285,75	7,79	1177,00	37,65	1133,51	0,12	9,20	0,19	226,89	2,86	49,20	4,10	327,68

Resultados obtenidos en el AM-10-1.0 con 20% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,13	60,00	0,31	383,71	8,25	2363,00	39,70	996,03	0,12	16,00	0,18	165,65	2,85	45,00	3,67	433,70
Ejecución 2	0,13	18,00	0,58	334,19	7,98	1355,00	41,77	1054,98	0,12	16,00	0,18	203,45	2,78	37,00	3,75	423,44
Ejecución 3	0,13	33,00	0,47	303,99	8,90	1212,00	38,88	1009,97	0,12	8,00	0,20	178,89	3,23	59,00	4,21	456,71
Ejecución 4	0,13	75,00	0.38	366,65	8,10	2178,00	39,95	1122,58	0,12	10,00	0,15	202,44	2,66	56,00	3,93	412,13
Ejecución 5	0,13	60,00	0,31	378,22	8.03	2226,00	37,23	1342,34	0,12	12,00	0,17	160,33	2,93	42,00	3,86	438,22
Media	0,13	49,20	0,42	353,35	8,31	1866,80	39,51	1105,18	0,12	12,40	0,18	182,15	2,89	47,80	3,88	432,84

Resultados obtenidos en el AM-10-0.1 con 10% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,10	0,00	0,10	286,41	7,86	554,00	22,62	968,23	0,12	0,00	0,12	276,89	2,67	0,00	2,67	302,62
Ejecución 2	0,10	0,00	0,10	315,12	7,50	234,00	17,20	1001,30	0,12	0,00	0,12	298,76	2,67	0,00	2,67	299,89
Ejecución 3	0,10	0,00	0,10	276,88	7,74	477,00	20,03	973,31	0,12	0,00	0,12	239,80	2,67	0,00	2,67	312,11
Ejecución 4	0,10	0,00	0,10	298,98	7,97	434,00	18,28	995,73	0,12	0,00	0,12	298,77	2,67	0,00	2,67	299,10
Ejecución 5	0,10	0,00	0,10	286,81	7,00	505,00	21,33	971,44	0,12	0,00	0,12	2,00	2,67	0,00	2,67	308,06
Media	0,10	0,00	0,10	292,84	7,61	440,80	19,89	982,00	0,12	0,00	0,12	223,24	2,67	0,00	2,67	304,36

Resultados obtenidos en el AM-10-0.1 con 20% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,10	0,00	0,10	236,00	7,62	933,00	20,06	1305,67	0,12	0,00	0,12	195,95	2,67	0,00	2,67	302,62
Ejecución 2	0,10	0,00	0,10	215,13	7,75	566,00	21,90	1201,99	0,12	0,00	0,12	185,76	2,67	0,00	2,67	299,89
Ejecución 3	0,10	0,00	0,10	216,88	7,86	712,00	31,86	1345,55	0,12	0,00	0,12	202,33	2,67	0,00	2,67	312,11
Ejecución 4	0,10	0,00	0,10	234,55	7,97	960,00	28,97	1287,72	0,12	0,00	0,12	198,81	2,67	0,00	2,67	299,10
Ejecución 5	0,10	0,00	0,10	229,76	7,96	894,00	28,31	1129,26	0,12	0,00	0,12	197,66	2,67	0,00	2,67	308,06
Media	0,10	0,00	0,10	226,46	7,83	813,00	26,22	1254,04	0,12	0,00	0,12	196,10	2,67	0,00	2,67	304,36

Resultados obtenidos en el AM-10-0.1MEJ con 10% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,10	0,00	0,10	181,41	7,43	645,00	24,61	938,91	0,12	0,00	0,12	147,35	2,67	0,00	2,67	309,97
Ejecución 2	0,10	0,00	0,10	178,92	7,64	220,00	18,36	925,55	0,12	0,00	0,12	175,72	2,67	0,00	2,67	279,56
Ejecución 3	0,10	0,00	0,10	186,67	7,38	323,00	19,93	973,31	0,12	0,00	0,12	150,11	2,67	0,00	2,67	303,01
Ejecución 4	0,10	0,00	0,10	200,34	7,74	235,00	21,28	995,73	0,12	0,00	0,12	187,96	2,67	0,00	2,67	307,77
Ejecución 5	0,10	0,00	0,10	187,74	7,56	601,00	23,34	971,44	0,12	0,00	0,12	153,44	2,67	0,00	2,67	295,27
Media	0,10	0,00	0,10	187,02	7,55	404,80	21,50	960,99	0,12	0,00	0,12	162,92	2,67	0,00	2,67	299,12

Resultados obtenidos en el AM-10-0.1MEJ con 20% de restricciones.

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,10	0,00	0,10	347,65	8,37	507,00	15,13	1165,40	0,12	0,00	0,12	261,98	2,67	0,00	2,67	402,63
Ejecución 2	0,10	0,00	0,10	289,13	8,03	601,00	16,97	1334,98	0,12	0,00	0,12	243,56	2,67	0,00	2,67	415,87
Ejecución 3	0,10	0,00	0,10	276,90	8,87	643,00	22,89	1445,09	0,12	0,00	0,12	288,81	2,67	0,00	2,67	382,07
Ejecución 4	0,10	0,00	0,10	305,67	8,23	767,00	25,44	1297,79	0,12	0,00	0,12	276,02	2,67	0,00	2,67	404,44
Ejecución 5	0,10	0,00	0,10	296,12	8,77	543,00	15,68	1566,03	0,12	0,00	0,12	257,96	2,67	0,00	2,67	405,67
Media	0,10	0,00	0,10	303,09	8,45	612,20	19,22	1361,86	0,12	0,00	0,12	265,67	2,67	0,00	2,67	402,14

6.3 – Tabla global comparativa

Resultados globales con 10% de restricciones

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
COPKM	0.14	55,40	7,55	4,39	8,13	448,40	645,61	1652,94	0.16	45,80	8,54	3,64	x	x	x	x
BL	0,11	0,00	0,11	36,13	7,69	709,20	1015,93	2388,08	0,13	0,00	0,13	34,72	x	x	x	x
AGG-UN	0,11	0.00	0.11	223,14	7,88	172,40	12,47	816,57	0,12	0,00	0,12	177,36	2,57	15,40	3,13	374,65
AGG-SF	0,11	0,00	0,11	215,97	7,09	297,40	15,01	815,3	0,12	0,00	0,12	173,35	2,67	0,00	2,67	397,9
AGE-UN	0,11	0,00	0,11	259,88	5,59	52,20	6,98	864,61	0,12	0,00	0,12	226,89	2,79	11,60	3,22	436,86
AGE-SF	0,11	0,00	0,11	257,68	5,24	46,20	6,47	880,63	0,12	0,00	0,12	192,58	2,67	0.0	2,67	420,97
AM10-1.0	0,12	12,40	0,27	285,75	7,79	1177,00	37,65	1133,51	0,12	9,20	0,19	226,89	2,86	49,20	4,10	327,68
AM10-0.1	0,1	0.00	0,10	292,84	7,61	440,00	19,89	982	0,12	0,00	0,12	223,24	2,67	0,00	2,67	304,36
AM10-0.1MEJ	0,1	0.00	0,10	187,02	7,55	404,80	21,50	960,99	0,12	0,00	0,12	162,92	2,67	0.0	2,67	299,12

Resultados globales con 20% de restricciones

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
COPKM	0.12	113,40	7,95	3,29	9,97	386,40	288,68	1652,89	0.17	106,00	10,17	3,39	x	x	x	x
BL	0,11	0,00	0,11	35,88	7,73	1383,60	1005,71	2250,57	0,13	0,00	0,13	29,42	x	x	x	x
AGG-UN	0,11	0.00	0.11	293	7,16	271,20	10,62	1016,07	0,12	0,00	0,12	267,49	2,67	0,00	2,67	425,34
AGG-SF	0,11	0,00	0,11	305,88	7,18	238,60	10,38	1023,97	0,12	0,00	0,12	277,11	2,67	0,00	2,67	424,75
AGE-UN	0,11	0,00	0,11	290,29	4,66	59,00	5,47	1157,49	0,12	0,00	0,12	270,69	2,79	11,60	3,22	447,34
AGE-SF	0,11	0,00	0,11	330,13	4,64	60,80	5,42	1098,44	0,12	0,00	0,12	304,81	2,67	0.0	2,67	465,48
AM10-1.0	0,13	49,20	0,42	353,35	8,31	1866,80	39,51	1105,18	0,12	12,40	0,18	182,15	2,89	47,80	3,88	432,84
AM10-0.1	0,1	0.00	0.1	226,46	7,83	813,00	26,22	1254,04	0,12	0,00	0,12	196,1	2,67	0.0	2,67	304,36
AM10-0.1MEJ	0,1	0.00	0,10	303,09	8,45	612,20	19,22	1361,86	0,12	0,00	0,12	265,67	2,67	0.0	2,67	402,14

7– Análisis global de resultados

Para analizar el rendimiento sobre los algoritmos, se han ejecutado sobre cuatro conjuntos de datos, uno más que en la práctica anterior:

- ***Iris***: Contiene información sobre características de tres tipos de flor de Iris. Tiene tres clases (k = 3).

- ***Ecoli***: Contiene medidas sobre ciertas características de diferentes tipos de células. Tiene ocho clases (k=8).

- ***Rand***: Conjunto de datos artificial. Contiene tres clases (k = 3).

- ***Newthyroid***: Contiene medidas cuantitativas tomadas sobre la glándula tiroides de 215 pacientes. Tiene tres clases (k = 3).

Compararemos ambos algoritmos basándonos en sus capacidades para obtener soluciones de calidad, el número de restricciones no satisfechas, y rapidez de estos, comenzando por comparar los algoritmos genéticos.

Comenzamos observando los resultados de los ***algoritmos genéticos generacionales elitistas***: donde claramente se puede ver que no han encontrado dificultades para encontrar una solución óptima en los conjuntos “iris”, “rand” y “newthyroid”, mostrándonos los mismos resultados en tiempos muy similares. Sin embargo, podemos detenernos en el conjunto de datos “ecoli” el cual, debido

a su complejidad y al mayor número de cluster, no ha alcanzado el óptimo. Deteniéndonos en sus resultados, podemos observar que, aunque estos no difieren mucho los unos de los otros en el valor de la desviación, el AGG-UN en general es capaz de encontrar una solución que incumple menos restricciones que el AGG-SF, disminuyendo el valor de "infeasibility" y el de la función objetivo.

Algo similar sucede entre las dos versiones de algoritmos ***genéticos estacionarios***, tanto el AGE-UN como el AGE-SF alcanzan óptimos en los tres conjuntos de datos menos el Ecoli, y de nuevo, el AGE-UN proporciona unos valores más pequeños (y por tanto, mejores) en el cálculo de la "infeasibility" y la función objetivo; aunque en este caso la diferencia entre valores sea menos notoria.

A continuación, comparamos los algoritmos generacionales elitistas con los algoritmos estacionarios. Observamos que en ambos tipos de algoritmos, al aumentar el porcentaje de restricciones (de 10% a 20%), a pesar de que la desviación general no sufre ningún cambio significativo, el valor de "infeasibility" aumenta un poco su valor (y a su vez el de la funcón objetivo); pero sobre todo, notamos cómo los algoritmos pasan a ser algo más lentos en sus ejecuciones.

Destaquemos de nuevo el "Ecoli" como conjunto de datos decisivo a la hora de elegir qué algoritmo genético es mejor. Sin ninguna duda, el **algoritmo estacionario** consigue resultados sorprendentemente mejores disminuyendo significativamente el valor de la función objetivo. Esto quizás podría deberse a que, a pesar de tener una convergencia más lenta, el favorecer a la exploración del entorno proporcione mejores valores.

A la hora de implementar los ***algoritmos meméticos*** , lo construimos sobre el algoritmo generacional elitista, al cual tras cada 10 generaciones le aplicamos a un porcentaje de su población una búsqueda local; combinando la estrategia de explorar de los genéticos con la de explotar de la búsqueda local; de esta forma, se esperaba que los resultados fuesen similares a los del genético o un poco mejor; sin embargo, observamos cómo los valores empeoran cuando le aplicamos la búsqueda local a todos los cromosomas de la población, los cuales se quedan "atrapados" en lo que parecen ser óptimos locales en vez de encontrarse con una mejor solución.

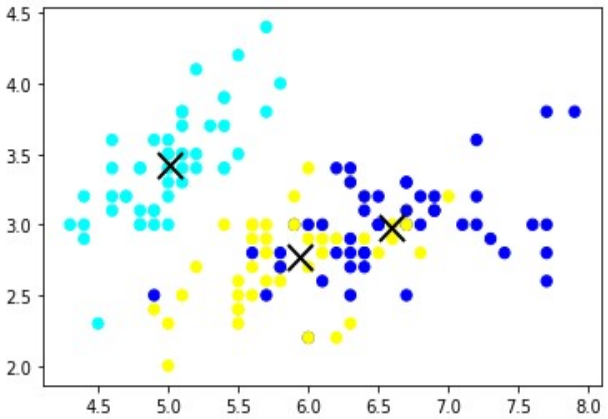
Aún así, esto no ocurre cuando aplicamos la búsqueda local a un porcentaje más reducido de cromosomas, donde sí se consigue llegar a una solución óptima, la cual, además, es muy similar a la del resto de algoritmos implementados. Aunque el algoritmo memético mejore, muy disimuladamente, el valor de la función objetivo en el conjunto de datos "Iris"; considero que el mejor algoritmo es el ***algoritmo genético estacionario***, puesto que consigue dar mejores soluciones en el conjunto de datos más problemático.

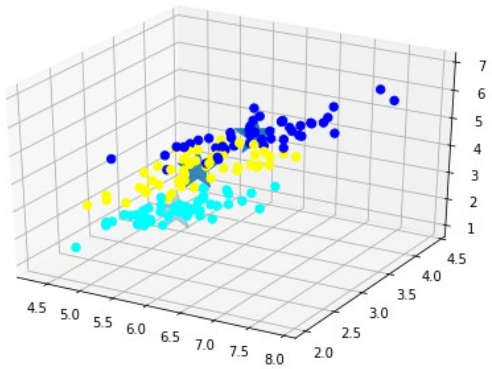
Como se puede ver, cualquiera de estos algoritmos han conseguido unos mejores resultados que nuestro algoritmo de comparación ***Greedy***, el cual no nos proporcionó valores muy buenos en la práctica anterior, sin conseguir alcanzar el óptimo en ninguna de sus soluciones. Aún así, es interesante compararlo con la ***Búsqueda Local*** de la práctica anterior, que a pesar de proporcionar resultados bastante mediocres en el "Ecoli", encuentra soluciones de calidad muy similares a las del resto de conjuntos de datos, teniendo además "el plus" del poco tiempo de ejecución que conlleva.

Aunque por falta de tiempo no he podido ejecutar el nuevo conjunto de datos "Newthyroid" en los algoritmos Greedy y Búsqueda Local, me resulta interesante destacar cómo podemos ver la complejidad del conjunto de datos a través de los resultados obtenidos; ya que, aunque se obtenga una desviación más alta que en los conjuntos "Iris" y "Rand", podemos asegurar que llega a un óptimo (local o global) cuando obtenemos un valor de "infeasibility" cero; sin embargo, el hecho de que no siempre sea capaz de encontrar este óptimo (como bien podemos observar en las tablas de los resultados), sugiere que sus datos son de una complejidad algo mayor al de los conjuntos anteriormente citados.

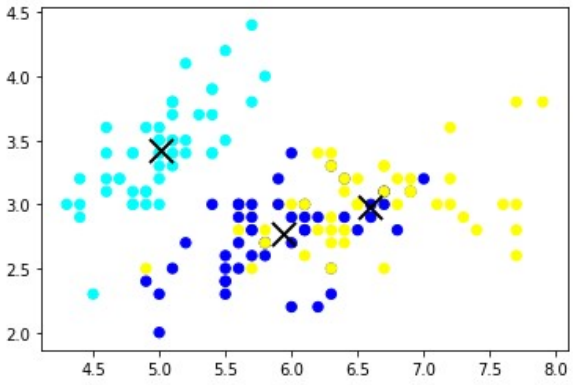
Adjunto gráficas de convergencia y representación gráfica de las soluciones para el conjunto "Iris" con un porcentaje de restricciones del 10%:

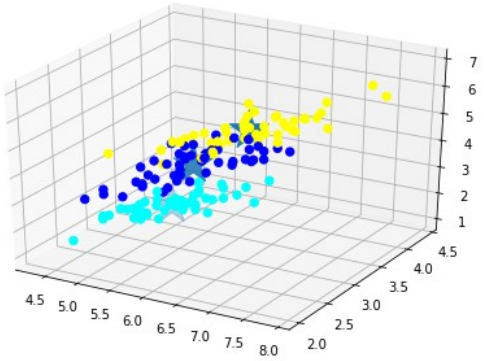
AGG-UN IRIS 10%



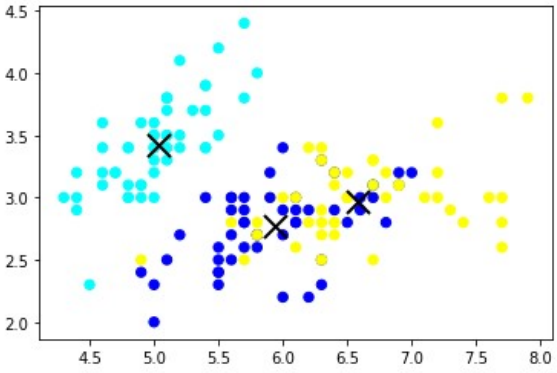


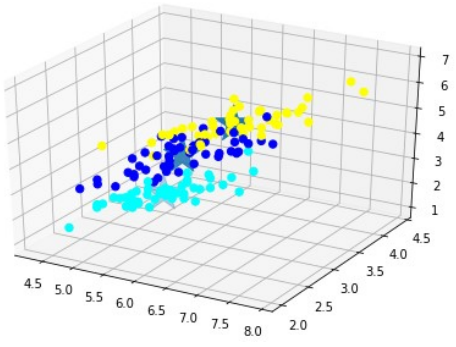
AGE-UN IRIS 10%



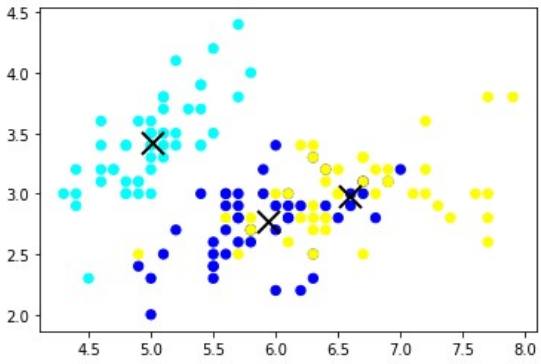


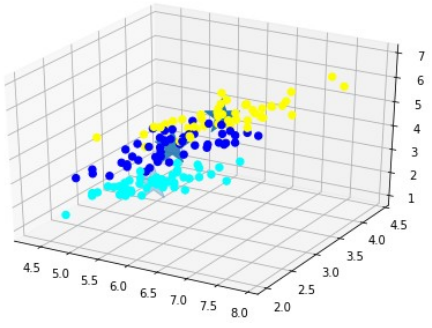
AM 10-1.0 IRIS 10%



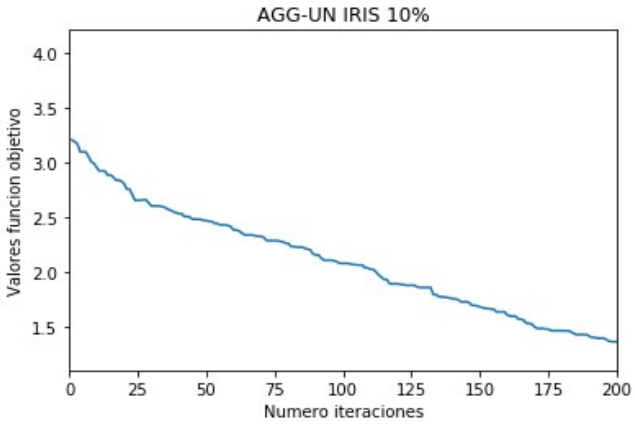


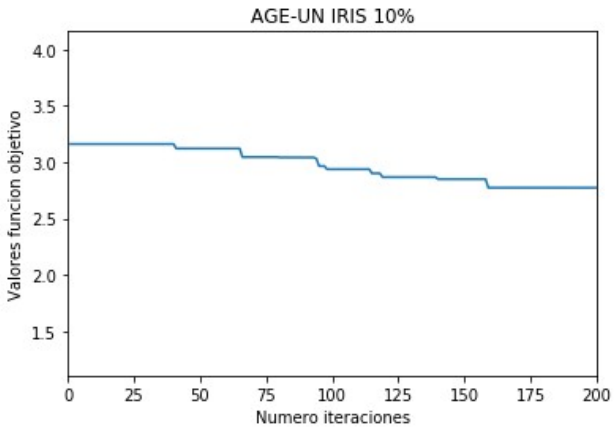
AM 10-0.1 IRIS 10%

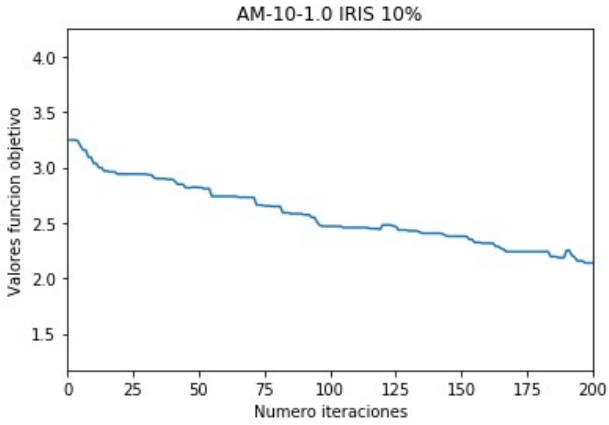


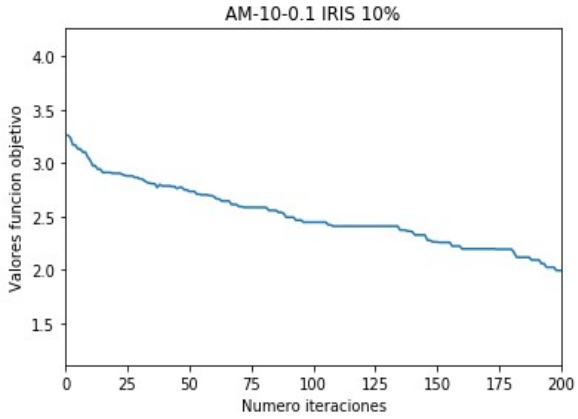


Al ejecutar el conjunto de datos con los diferentes algoritmos, observamos cómo se obtienen gráficas de soluciones prácticamente idénticas, destacando la solución del AM 10-1.0, cuya solución está ligeramente menos agrupada que las del resto (reflejándose esto en los valores medios de sus ejecuciones, donde “infeasibility” no consigue llegar a cero.









A través de las gráficas de convergencia, obtenemos una visión más clara de cómo funciona cada algoritmo. Como podemos observar, el algoritmo ***genético estacionario*** opta por una convergencia lenta, donde se favorece la exploración del entorno y los cambios en la función objetivo son menos frecuentes pero, sin embargo, más significativos (como se ve reflejado en los “picos” que pega la gráfica).

Por otro lado, el algoritmo *genético generacional* es elitista, y produce una presión selectiva muy alta al reemplazar sus peores cromosomas, lo que le proporciona una convergencia muy rápida.

En los *algoritmos meméticos*, aunque tengan una convergencia más o menos similar, encontramos claramente los efectos de la búsqueda local en los "picos" que se producen cada ciertas iteraciones; los cuales bajan bastante excepto a partir de la mitad-final, donde se estancan y van subiendo y bajando. Cabe a destacar que estos "picos" son más visibles en el AM-10-1.0, por lo que podemos asegurar que al aplicar la búsqueda local sobre toda la población hace que se explote más de lo que se explora, siendo esto quizás la causa de unos peores resultados en comparación a otros algoritmos.

7 – Bibliografía

- Entendimiento del problema

- Tema 3: Metaheurísticas basadas en poblaciones – Parte I
- Seminario 3: Problemas de optimización con técnicas basadas en poblaciones
-https://es.wikipedia.org/wiki/Algoritmo_gen%C3%A9tico
-https://es.wikipedia.org/wiki/Algoritmo_mem%C3%A9tico

-Programación del problema

- https://stackoverflow.com/
- https://docs.python.org/3/library/

-Análisis de resultados y muestra de resultados

- https://www.aprendemachinelearning.com/k-means-en-python-paso-a-paso/
- https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html
-https://datatofish.com/export-dataframe-to-excel/