UNIVERSIDAD DE GRANADA

# Aprendizaje de Pesos en Características

12 de mayo 2022

***Metaheurísticas***

*Gallego Menor, Francisco Javier*
*javigallego@correo.ugr.es*
*74745747W*

# Índice

# 1 Introducción

El problema de clasificación consiste en, dado un conjunto $A = \{(a, b) \ : a \in R^n, b \ es \ una \ clase\}$ de datos ya clasificados, obtener un sistema que permita clasificar un objeto nuevo de forma automática.

Un ejemplo de clasificador, y el que utilizaremos en esta práctica,es el $k - NN$ , $k$ vecinos más cercanos. Este toma la clase que más se repita entre los $u_i \in A$ tales que su distancia al nuevo elemento $u$ sea mínima. En nuestro caso, en una versión sencilla del problema, consideraremos el clasificador $1-NN$.

Consideraremos como distancias la distancia trivial si las características son discretas (esto es, la distancia será 1 si las características son diferentes, y 0 si son iguales. La denotamos como $d_n$), y la distancia euclídea para características que sean continuas. Además, cada característica tendrá un peso asociado, por lo que dado un vector de pesos $w$, la distancia entre dos vectores $u$ y $v$ será de la forma:

$$d(u, v) = \sqrt{\sum_i w_i(u_i - v_i)^2 + \sum_j w_j d_n(u_j, v_j)}$$

El aprendizaje de pesos en características consiste en hallar un vector de pesos que maximice la siguiente función:

$$F(w) = \alpha T_{clas}(w) + (1 - \alpha)T_{red}(w)$$

Donde

- $T_{clas}$ es la función que indica cómo de bueno es nuestro clasificador, es decir, cuántos casos ha clasificado correctamente si entrenamos el clasificador usando el resto de datos ,la técnica *k-fold cross validation*, y dejando un elemento fuera (leave one out).
- $T_{red}$ que es la función que nos indica cuántas características de un dato tienen un peso menor que un valor establecido, en nuestro caso 0.2.

# 2 Descripción de la aplicación de los algoritmos

## 2.1 Esquemas de representación

Antes de comenzar, comentar brevemente que aquellas secciones que no añaden nada nuevo (aquellas cogidas de la P1) las marco en rojo. En nuestro problema, los datos de entrada poseen los elementos que siguen:

- **Clase del elemento (target)**: que es la categoría a la que corresponde el mismo, dependiendo de cada dataset
- **Ejemplo**: que es un par que tiene un vector de features y una clase (target).
- **Dataset**: que contendrá una lista de ejemplos

- **Vector de características**: vector de valores reales que trataremos de normalizar al intervalo $[0, 1]$ para trabajar con ellos.
- Para esta segunda práctica, con el fin de representar a un individuo de la población, he creado la clase **Chromosome**. Esta clase dispone únicamente de cuatro atributos. El primero de ellos, el vector de pesos, y el resto se corresponde con cada una de las tasas (clasificación, reducción y fitness).

A partir de los elementos previamente descritos, vamos a obtener nuestra solución al problema. Esta consistirá en un vector de pesos, cuyos valores reales se encontrarán también en el intervalo $[0, 1]$.

## 2.2 Operadores comunes

En esta sección, procederemos a describir aquellas funcionalidades que sean comunes para todos los algoritmos. En nuestro caso, solo es la que sigue:

### 2.2.1 Generación Soluciones Aleatorias - Clase Cromosoma

Cuando creamos un nuevo cromosoma, en caso de no pasar como parámetro un vector de pesos ya existente, se crea un vector de pesos de forma aleatoria.

Para la generación de los números aleatorios usamos **np.random.uniform**, el cual implica que cualquier valor dentro del intervalo dado tiene la misma probabilidad de ser extraído por uniforme. En nuestro caso el [0,1). El pseudocódigo es el siguiente:

1: **procedure** Crear Solución Aleatoria:
2: vector de pesos del cromosoma creado ← np.random.uniform(extremo inferior intervalo, extremo superior, longitud del vector)

### 2.2.2 Operador de Cruce BLX-$\alpha$

El operador de cruce BLX recibe como parámetros dos cromosomas y devuelve dos descendientes, que tienen unos vectores de pesos obtenidos de forma aleatoria en un intervalo concreto que depende de los pesos de los padres.

1: **procedure** BLX(c1, c2, data, classes):
2: $C_{max}$ ← Máximo entre: los máximos de C1 y C2
3: $C_{min}$ ← Mínimo entre: los mínimos de C1 y C2
4: L ← $C_{max} - C_{min}$
5: a ← $C_{min} - \alpha * L$ (Extremos inferior del intervalo)
6: b ← $C_{max} + \alpha * L$ (Extremo superior del intervalo)
7: H1, H2 ← generamos dos nuevos descendientes
8: **return** dos nuevos cromosomas con H1 y H2 como vectores de pesos respectivamente.

### 2.2.3 Operador de cruce Aritmético

Por su parte, este operador combina los pesos de los dos padres según la media aritmética para obtener el vector de pesos del hijo.

```
1: procedure ARITHMEITCCROSS(c1, c2, data, classes):
2:     α, β ← nº aleatorio entre 0.0 y 1.0
3:     H1 ← vector pesos C1 ·α + vector pesos C2 ·(1 – α)
4:     H2 ← vector pesos C1 ·β + vector pesos C2 ·(1 – β)
5:     return dos nuevos cromosomas con H1 y H2 como vectores de pesos
```

### 2.2.4 Operador de selección

En nuestro caso se trata del **torneo binario**. Cogemos dos individuos, de forma aleatoria, de la población y escogemos al que mejor valor de fitness posea entre los dos.

```
1: procedure SELECTION(pop):
2:     i1, i2 ← Indices aleatorios
3:     if pop[i1].fitness > pop[i2].fitness then
4:         return pop[i1]
5:     else
6:         return pop[i2]
```

### 2.2.5 Mutación

Este operador es común para todos los algoritmos de la Práctica 2. Al mutar, sumaremos a la componente *j–esima* un valor aleatorio y truncaremos al intervalo $[0, 1]$ si el nuevo valor se nos escapara del intervalo:

```
1: procedure MUTE(w,sigma,j)
2:     w(j) ← w(j) + random(0, 1)
3:     w(j) ← Normalize(w(j))
4:     return w
```

## 2.3 Función objetivo

En nuestro caso, se nos indica que tomemos como *alpha* el valor 0.5, así que en realidad lo que estamos haciendo es:

$$F = 0.5(T_{clas} + T_{red})$$

Hemos modificado la forma de calcular $T_{class}$ con respecto a su versión de la P1, pues era el cuello de botella en la ejecución. La calcularemos de la siguiente manera ahora:

```
1: procedure T-class(w, features, targets)
2:     dataw ← features * w
3:     classifier ← clasificador 1-NN de Sklearn
4:     Entrenamos el modelo
5:     ind_near ← indice vecino mas cercano
6:     tasa_class ← media (targets[ind_near] == targets)
7:     return tasa_class / 100
```

Y calcularemos $T_{red}$ así:

```
1: procedure T-red(weight)
2:     return (Nº Pesos < 0.2) / Nº Pesos Totales
```

# 3 Descripción de los algoritmos considerados

## 3.1 Algoritmo Greedy Relief

El algoritmo de Greedy Relief se encarga de recorrer todo el conjunto de datos, muestra por muestra. En cada uno de ellos, dependiendo del amigo y enemigo más cercanos, hace una modificación del vector de pesos. Entonces, lo que haremos será: por cada una de las muestras, obtener un conjunto de datos amigos (que pertenecen a su misma clase) y uno de enemigos (no pertenecen a su misma clase). Posteriormente, obtendremos el vecino más cercano de cada uno de ellos.

**Algorithm 1** Greedy Relief

```
1: procedure greedyRelief(features, targets)
2:     w ← [0, ...., 0]
3:     distances ← matriz cuadrada de distancias (euclídeas)
4: loop: para cada muestra del conjunto de entrenamiento
5:     en_indices ← índices de ejemplos con distinto target a la muestra
6:     fr_indices ← índices de ejemplos con igual target a la muestra
7:     friends ← features[fr_indices]
8:     enemies ← features[en_indices]
9:
10:    closestFriend ← Amigo mas cercano
11:    closestEnemy ← Enemigo mas cercano
12:
13:    w ← w + |features(i) - closestEnemy| - |features(i)-closestFriend|
14: endloop
15:    w ← Truncamos valores negativos a 0
16:    w ← Normalizamos w
17:    return w
```

## 3.2 Búsqueda local

Para este algoritmo, debemos definir el algoritmo que hemos usado para obtener una mutación de un ejemplo. Al mutar, sumaremos a la componente $j-esima$ un valor aleatorio y truncaremos al intervalo $[0,1]$ si el nuevo valor se nos escapara del intervalo:

1: **procedure** MUTE(w,sigma,j)
2: $w(j) \leftarrow w(j) + random(0,1)$
3: $w(j) \leftarrow Normalize(w(j))$
4: **return** w

En el procedimiento de cálculo de pesos mediante la búsqueda local intervendrá la función de evaluación, que será notada por $f(w)$.

Así, el procedimiento general para la generación de pesos para la búsqueda local sería:

**Algorithm 2** Local Search

1: **procedure** LOCALSEARCH(initialWeight,features,targets)
2: weight ← valor inicial para los pesos
3: bestF ← valor inicial para la función objetivo
4: index ← Ordenación de forma aleatoria de los índices
5: improve ← False
6: ***while*** (menos evaluaciones que 15000) and (Haya mejora antes de generar 20*n vecinos)
7: w ← vector de pesos mutando componente j-esima
8: newF ← Valor funcion objetivo para w
9: **if** newF > bestF **then**
bestF ← newF
weight ← w
notMuted ← 0
improve ← True
10: **else**
notMuted ← notMuted + 1
11: evaluaciones ← evaluaciones +1
12: **if** (Hay mejora) or (Todos los componentes han sido ya mutados) **then**
improve ← False
index ← Ordenación de forma aleatoria de los índices
13: *endloop*
14: **return** weight

## 3.3 Algoritmos Genéticos Generacionales

Este algoritmo consta de diferentes partes en su implementación, las cuales son:

- Selección: se lleva a cabo mediante el torneo binario. Este, selecciona aleatoriamente dos indivi-

duos de la población y se queda con el mejor de los dos. En este caso, formaremos una lista con tantos padres como individuos tenga la poblacion actual.

- Cruce: usaremos los operadores BLX-$\alpha$ y el de cruce aritmético.
- Mutación
- Reemplazamiento: este algoritmo es elitista. Por tanto, cuando se genera una nueva población, que sería la lista de hijos tras la mutación, reemplazaremos al peor cromosoma de los hijos por el mejor de la población actual.

**Algorithm 3** AGG

```
1:  procedure AGG(data,classes, trainIndex, testIndex, crossOperator)
2:      population ← generamos la población inicial
3:
4:  while (evaluaciones función objetivo < 15000)
5:      bpIndex ← indice del cromosoma con mejor fitness en la poblacion actual
6:      new population ← [selection(population) for i in range(TAMAÑO POBLACION = 30)]
7:
8:      for i in range(nº cromosomas a cruzar) do
            i1, i2 ← Generamos dos índices aleatorios
            h1, h2 ← Aplicamos el operador de cruce a new population[i1] y new population[i2]
            new population[i1], new population[i2] ← h1, h2
            evaluaciones ← evaluaciones+2
9:
10:     for i in range(nº mutaciones a realizar) do
            Generamos dos índices aleatorios, uno para el cromosoma y otro para el gen
            Mutamos el gen concreto
            evaluaciones ← evaluaciones+1
11:
12:     currentBestIndex ← indice del cromosoma con mejor fitness en new population
13:     if new population[currentBestIndex] < population[bpIndex] then
            Sustituimos el peor fitness de la nueva población, por el mejor de la antigua
14:     Actualizamos valores
15: endloop
16:
17:     Calculamos el cromosoma con mayor fitness, y obtenemos los resultados
18:     return tasa clasificación, tasa reducción
```

## 3.4 Algoritmos Genéticos Estacionarios

Este algoritmo consta de diferentes partes en su implementación, las cuales son:

- Selección: se lleva a cabo mediante el torneo binario. En este caso, formaremos una lista con tan solo 2 padres.
- Cruce: usaremos los operadores BLX-$\alpha$ y el de cruce aritmético.
- Mutación
- Reemplazamiento: los individuos obtenidos después de la mutación compiten por acceder a la población.

**Algorithm 4** AGE

```
1:  procedure AGE(data,classes, trainIndex, testIndex, crossOperator)
2:      population ← generamos la población inicial
3:
4:  while (evaluaciones función objetivo < 15000)
5:      new parents ← seleccionamos los 2 nuevos padres por torneo binario
6:      new parents ← aplicamos el operador de cruce
7:      nº evaluaciones ← +2
8:      new parents ← aplicamos la mutación
9:      Actualizamos nº evaluaciones actuales
10:     Incluimos a los 2 hijos generados en population
11:     Ordenamos population segun el valor del fitness
12:     Eliminamos los 2 peores cromosomas (últimas 2 posiciones)
13: endloop
14:
15:     Calculamos el cromosoma con mayor fitness, y obtenemos los resultados
16:     return tasa clasificación, tasa reducción
```

## 3.5 Algoritmos Meméticos

Los algoritmos meméticos se basan en el refinamiento de los cromosomas aplicando una búsqueda local sobre ellos. Los algoritmos meméticos son un híbrido entre los AGG y la búsqueda local. Ahora, incluimos el pseudocódigo para cada uno de los algoritmos meméticos:

**Algorithm 5** AM1

```
1: procedure AM1(population, data, classes, trainIndex)
2:     it ← 0
3:     new_population ← []
4:     for c in population do
5:         s, newC ← localsearch(data, classes, trainIndex, c)
6:         it ← it + s
7:         new_population ← añadir newC
8:     return it, new_population
```

**Algorithm 6** AM2

```
1: procedure AM2(population, data, classes, trainIndex)
2:     k ← indice de cromosoma elegido aleatorio
3:     it, population[k] ← localsearch(data, classes, trainIndex, population[k])
4:     return it, population
```

A continuación, mostramos el esquema general para los diferentes AM.

Por último, incluimos el esquema de la búsqueda local para los algoritmos genéticos:

**Algorithm 7** AM3

```
1: procedure AM3(population, data, classes, trainIndex)
2:     best ← indice de cromosoma con mejor fitness
3:     it, population[best] ← localsearch(data, classes, trainIndex, population[best])
4:     return it, population
```

**Algorithm 8** AM

```
1:  procedure AM(data,classes, trainIndex, testIndex, crossOperator, typeMemetic)
2:      population ← generamos la población inicial
3:      generation ← 1
4:  while (evaluaciones función objetivo < 15000)
5:      bpIndex ← indice del cromosoma con mejor fitness en la poblacion actual
6:      new population ← [selection(population) for i in range(TAMAÑO POBLACION = 30)]
7:
8:      for i in range(nº cromosomas a cruzar) do
          i1, i2 ← Generamos dos índices aleatorios
          h1, h2 ← Aplicamos el operador de cruce a new population[i1] y new population[i2]
          new population[i1], new population[i2] ← h1, h2
          evaluaciones ← evaluaciones+2
9:      for i in range(nº mutaciones a realizar) do
          Generamos dos índices aleatorios, uno para el cromosoma y otro para el gen
          Mutamos el gen concreto
          evaluaciones ← evaluaciones+1
10:     currentBestIndex ← indice del cromosoma con mejor fitness en new population
11:     if new population[currentBestIndex] < population[bpIndex] then
          Sustituimos el peor fitness de la nueva población, por el mejor de la antigua
12:     if generation % 10 then
          Aplicamos typeMemetic (AM1, AM2, AM3)
13:
14:     generation ← generation+1
15: endloop
16:     Calculamos el cromosoma con mayor fitness, y obtenemos los resultados
17:     return tasa clasificación, tasa reducción
```

**Algorithm 9** low_localSearch

```
1:  procedure LOW_LOCALSEARCH(data, classes, trainIndex, chromosome)
2:      bestF ← chromosome.fitness
3:      it ← 0
4:  while (it < 2 * longitud vector pesos)
5:      k ← gen aleatorio
6:      mutChrom ← chromosome
7:      mutChrom ← cromosoma con el gen k-ésimo mutado
8:      Calculamos fitness de mutChrom
9:      it ← it+1
10:     if mutChrom.fitness > bestF then
          Actualizamos chromosome y bestF
11: endloop
12:     return it, chromosome
```

# 4 Procedimiento considerado para desarrollar la práctica

Para desarrollar la práctica, he usado **Python** como lenguaje de programación, sin usar ningún framework de metaheurísticas.

Para poder ejecutar el código , hace falta tener instalado *Numpy, Scipy* y *Sklearn*. Este último es muy útil para la realización de prácticas de este estilo pues trae implementaciones de muchas funcionalidades básicas para *Machine learning*.

El fichero que hay que ejecutar dentro de la carpeta es el fichero ***main.py***. Para ello, basta con escribir en la terminal:

*python main.py*

Tras la ejecución, comenzará a ejecutar los algoritmos sobre los 3 ficheros de datos que tenemos, que se explicarán más adelante.

La lista de archivos que contiene la práctica son los siguientes:

- **main.py**: fichero principal a ejecutar para la ejecución de nuestro programa.
- **algorithms.py**: fichero en el que se encuentran los algoritmos y algunas funciones auxiliares programadas para la práctica.
- **genetics.py**: fichero con todo lo relacionado con los algoritmos genéticos
- **memetics.py**: fichero con todo lo relacionado con los algoritmos meméticos.
- **graficas.ipynb**: notebook mediante el cual he generado las gráficas.
- **Datasets**: carpeta en la que se encuentran los datasets almacenados.
- **Archivos_CSV**: carpeta con los archivos CSV necesarios para agilizar el proceso de creación de las gráficas.

# 5 Experimentos y análisis de resultados

## 5.1 Resultados de cada algoritmo

**Algoritmo 1-NN**

| | *Ionosphere* | | | | *Parkinson* | | | | *Spectf-Heart* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Nº** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** |
| 0 | 0.85 | 0.00 | 0.42 | 0.00283 | 0.72 | 0.00 | 0.36 | 0.00099 | 0.97 | 0.00 | 0.49 | 0.00309 |
| 1 | 0.77 | 0.00 | 0.39 | 0.00276 | 0.82 | 0.00 | 0.41 | 0.00100 | 0.90 | 0.00 | 0.45 | 0.00323 |
| 2 | 0.83 | 0.00 | 0.41 | 0.00279 | 0.95 | 0.00 | 0.47 | 0.00100 | 0.71 | 0.00 | 0.36 | 0.00312 |
| 3 | 0.91 | 0.00 | 0.46 | 0.00278 | 0.74 | 0.00 | 0.37 | 0.00100 | 0.77 | 0.00 | 0.39 | 0.00317 |
| 4 | 0.86 | 0.00 | 0.43 | 0.00285 | 0.67 | 0.00 | 0.33 | 0.00101 | 0.99 | 0.00 | 0.49 | 0.00307 |
| Media | 0.84 | 0.00 | 0.42 | 0.00280 | 0.78 | 0.00 | 0.39 | 0.00100 | 0.87 | 0.00 | 0.43 | 0.00314 |
| Std | 0.05 | 0.00 | 0.02 | 0.00003 | 0.10 | 0.00 | 0.05 | 0.00001 | 0.11 | 0.00 | 0.05 | 0.00006 |

**Algoritmo RELIEF**

| | *Ionosphere* | | | | *Parkinson* | | | | *Spectf-Heart* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Nº** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** |
| 0 | 0.86 | 0.03 | 0.44 | 0.01278 | 0.72 | 0.05 | 0.38 | 0.00627 | 0.97 | 0.11 | 0.54 | 0.01284 |
| 1 | 0.79 | 0.03 | 0.41 | 0.01250 | 0.82 | 0.00 | 0.41 | 0.00475 | 0.91 | 0.00 | 0.46 | 0.01303 |
| 2 | 0.80 | 0.09 | 0.44 | 0.01248 | 0.95 | 0.05 | 0.50 | 0.00474 | 0.74 | 0.00 | 0.37 | 0.01291 |
| 3 | 0.91 | 0.03 | 0.47 | 0.01277 | 0.69 | 0.05 | 0.37 | 0.00474 | 0.74 | 0.00 | 0.37 | 0.01297 |
| 4 | 0.86 | 0.03 | 0.44 | 0.01278 | 0.67 | 0.00 | 0.33 | 0.00476 | 0.99 | 0.09 | 0.54 | 0.01274 |
| Media | 0.84 | 0.04 | 0.44 | 0.01266 | 0.77 | 0.03 | 0.40 | 0.00505 | 0.87 | 0.04 | 0.46 | 0.01290 |
| Std | 0.05 | 0.02 | 0.02 | 0.00014 | 0.10 | 0.02 | 0.06 | 0.00061 | 0.11 | 0.05 | 0.08 | 0.00010 |

**Algoritmo de búsqueda local**

| | *Ionosphere* | | | | *Parkinson* | | | | *Spectf-Heart* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Nº** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** |
| 0 | 0.89 | 0.79 | 0.84 | 2.47617 | 0.77 | 0.68 | 0.73 | 0.54792 | 0.97 | 0.84 | 0.91 | 3.52364 |
| 1 | 0.87 | 0.85 | 0.86 | 2.03225 | 0.85 | 1.00 | 0.92 | 1.16207 | 0.89 | 0.86 | 0.87 | 4.97461 |
| 2 | 0.86 | 0.76 | 0.81 | 1.77548 | 0.87 | 0.82 | 0.84 | 0.81054 | 0.74 | 0.91 | 0.83 | 3.02352 |
| 3 | 0.97 | 0.97 | 0.97 | 2.93292 | 0.74 | 1.00 | 0.87 | 1.17324 | 0.80 | 0.82 | 0.81 | 2.94800 |
| 4 | 0.86 | 0.88 | 0.87 | 1.89319 | 0.62 | 0.91 | 0.76 | 0.53875 | 0.99 | 0.68 | 0.83 | 3.57642 |
| Media | 0.89 | 0.85 | 0.87 | 2.22200 | 0.77 | 0.88 | 0.83 | 0.84650 | 0.88 | 0.82 | 0.85 | 3.60924 |
| Std | 0.04 | 0.07 | 0.05 | 0.42744 | 0.09 | 0.12 | 0.07 | 0.27982 | 0.09 | 0.08 | 0.04 | 0.72841 |

**AGG-BLX**

| | *Ionosphere* | | | | *Parkinson* | | | | *Spectf-Heart* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Nº** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** |
| 0 | 0.90 | 0.59 | 0.75 | 17.78729 | 0.94 | 0.68 | 0.81 | 15.10886 | 0.80 | 0.61 | 0.71 | 20.81741 |
| 1 | 0.91 | 0.65 | 0.78 | 18.16027 | 0.91 | 0.68 | 0.80 | 12.36116 | 0.86 | 0.55 | 0.70 | 20.59248 |
| 2 | 0.87 | 0.65 | 0.76 | 17.92551 | 0.92 | 0.73 | 0.83 | 13.88676 | 0.90 | 0.59 | 0.75 | 21.65683 |
| 3 | 0.90 | 0.59 | 0.74 | 18.16812 | 0.94 | 0.73 | 0.83 | 12.77313 | 0.85 | 0.61 | 0.73 | 18.05538 |
| 4 | 0.87 | 0.62 | 0.74 | 19.33893 | 0.94 | 0.73 | 0.83 | 13.36502 | 0.85 | 0.55 | 0.70 | 22.23772 |
| Media | 0.89 | 0.62 | 0.75 | 18.27602 | 0.93 | 0.71 | 0.82 | 13.49899 | 0.85 | 0.58 | 0.72 | 20.67197 |
| Std | 0.02 | 0.03 | 0.01 | 0.55073 | 0.01 | 0.02 | 0.01 | 0.95724 | 0.03 | 0.03 | 0.02 | 1.43496 |

**AGG-CA**

| | *Ionosphere* | | | | *Parkinson* | | | | *Spectf-Heart* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Nº** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** |
| 0 | 0.92 | 0.68 | 0.80 | 17.82474 | 0.97 | 0.73 | 0.85 | 11.80979 | 0.80 | 0.50 | 0.65 | 21.29503 |
| 1 | 0.92 | 0.59 | 0.75 | 18.02119 | 0.96 | 0.77 | 0.86 | 11.81669 | 0.87 | 0.66 | 0.76 | 21.21374 |
| 2 | 0.93 | 0.71 | 0.82 | 17.84429 | 0.95 | 0.77 | 0.86 | 13.23995 | 0.94 | 0.68 | 0.81 | 23.44971 |
| 3 | 0.89 | 0.74 | 0.81 | 18.00936 | 0.98 | 0.86 | 0.92 | 13.59249 | 0.90 | 0.64 | 0.77 | 20.47733 |
| 4 | 0.90 | 0.62 | 0.76 | 17.63211 | 0.99 | 0.64 | 0.81 | 13.07948 | 0.84 | 0.73 | 0.78 | 21.82140 |
| Media | 0.91 | 0.66 | 0.79 | 17.86634 | 0.97 | 0.75 | 0.86 | 12.70768 | 0.87 | 0.64 | 0.76 | 21.65144 |
| Std | 0.01 | 0.05 | 0.03 | 0.14249 | 0.02 | 0.07 | 0.03 | 0.74893 | 0.05 | 0.08 | 0.06 | 0.99597 |

**AGE-BLX**

| | *Ionosphere* | | | | *Parkinson* | | | | *Spectf-Heart* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Nº** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** |
| 0 | 0.93 | 0.59 | 0.76 | 20.86034 | 0.96 | 0.64 | 0.80 | 17.22413 | 0.84 | 0.55 | 0.69 | 26.56079 |
| 1 | 0.90 | 0.65 | 0.77 | 20.88104 | 0.90 | 0.68 | 0.79 | 17.03955 | 0.86 | 0.61 | 0.74 | 24.52509 |
| 2 | 0.91 | 0.62 | 0.76 | 20.65761 | 0.87 | 0.73 | 0.80 | 17.19522 | 0.94 | 0.57 | 0.75 | 25.26890 |
| 3 | 0.88 | 0.62 | 0.75 | 20.91860 | 0.93 | 0.73 | 0.83 | 15.30046 | 0.89 | 0.55 | 0.72 | 22.21060 |
| 4 | 0.89 | 0.68 | 0.78 | 20.43069 | 0.95 | 0.73 | 0.84 | 16.04657 | 0.77 | 0.66 | 0.71 | 23.19795 |
| Media | 0.90 | 0.63 | 0.76 | 20.74966 | 0.92 | 0.70 | 0.81 | 16.56119 | 0.86 | 0.59 | 0.72 | 24.35267 |
| Std | 0.02 | 0.03 | 0.01 | 0.18345 | 0.03 | 0.04 | 0.02 | 0.76480 | 0.06 | 0.04 | 0.02 | 1.52753 |

**AGE-CA**

| | *Ionosphere* | | | | *Parkinson* | | | | *Spectf-Heart* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Nº** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** |
| 0 | 0.93 | 0.94 | 0.93 | 20.62278 | 0.95 | 0.91 | 0.93 | 17.61132 | 0.84 | 0.73 | 0.78 | 24.29962 |
| 1 | 0.94 | 0.91 | 0.92 | 20.76749 | 0.99 | 1.00 | 0.99 | 14.09746 | 0.91 | 0.80 | 0.85 | 22.94477 |
| 2 | 0.94 | 0.85 | 0.90 | 20.79059 | 0.98 | 0.95 | 0.97 | 15.76394 | 0.94 | 0.86 | 0.90 | 26.37363 |
| 3 | 0.91 | 0.85 | 0.88 | 20.64561 | 0.99 | 0.91 | 0.95 | 16.24410 | 0.90 | 0.80 | 0.85 | 21.78090 |
| 4 | 0.90 | 0.85 | 0.87 | 20.28353 | 0.97 | 0.91 | 0.94 | 15.62938 | 0.86 | 0.75 | 0.81 | 21.34640 |
| Media | 0.92 | 0.88 | 0.90 | 20.62200 | 0.98 | 0.94 | 0.96 | 15.86924 | 0.89 | 0.79 | 0.84 | 23.34907 |
| Std | 0.02 | 0.04 | 0.02 | 0.18150 | 0.01 | 0.04 | 0.02 | 1.12988 | 0.04 | 0.05 | 0.04 | 1.82673 |

**AM-(10, 1.0)**

| | *Ionosphere* | | | | *Parkinson* | | | | *Spectf-Heart* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Nº** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** |
| 0 | 0.92 | 0.82 | 0.87 | 19.41069 | 0.94 | 0.91 | 0.92 | 14.03663 | 0.87 | 0.80 | 0.83 | 21.42776 |
| 1 | 0.94 | 0.94 | 0.94 | 19.52858 | 0.95 | 0.95 | 0.95 | 13.05515 | 0.86 | 0.80 | 0.83 | 22.07166 |
| 2 | 0.94 | 0.82 | 0.88 | 19.45065 | 0.94 | 0.91 | 0.92 | 15.36115 | 0.94 | 0.93 | 0.94 | 19.10654 |
| 3 | 0.90 | 0.76 | 0.83 | 19.38353 | 0.95 | 0.77 | 0.86 | 14.82838 | 0.91 | 0.84 | 0.88 | 20.13135 |
| 4 | 0.91 | 0.85 | 0.88 | 19.21892 | 0.97 | 0.95 | 0.96 | 14.10570 | 0.84 | 0.64 | 0.74 | 22.59400 |
| Media | 0.92 | 0.84 | 0.88 | 19.39848 | 0.95 | 0.90 | 0.92 | 14.27740 | 0.88 | 0.80 | 0.84 | 21.06626 |
| Std | 0.02 | 0.06 | 0.03 | 0.10224 | 0.01 | 0.07 | 0.04 | 0.78223 | 0.04 | 0.10 | 0.07 | 1.27985 |

**AM-(10, 0.1)**

| | *Ionosphere* | | | | *Parkinson* | | | | *Spectf-Heart* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Nº** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** |
| 0 | 0.91 | 0.65 | 0.78 | 18.49086 | 0.91 | 0.73 | 0.82 | 12.31483 | 0.82 | 0.64 | 0.73 | 20.94851 |
| 1 | 0.90 | 0.85 | 0.88 | 18.48669 | 0.94 | 0.68 | 0.81 | 12.32023 | 0.86 | 0.64 | 0.75 | 23.23542 |
| 2 | 0.92 | 0.62 | 0.77 | 18.68449 | 0.94 | 0.73 | 0.83 | 13.31626 | 0.95 | 0.68 | 0.82 | 18.40112 |
| 3 | 0.85 | 0.74 | 0.79 | 22.17481 | 0.96 | 0.77 | 0.86 | 12.76525 | 0.88 | 0.75 | 0.82 | 19.42529 |
| 4 | 0.90 | 0.71 | 0.80 | 20.60894 | 0.95 | 0.68 | 0.82 | 12.90642 | 0.80 | 0.68 | 0.74 | 19.38266 |
| Media | 0.90 | 0.71 | 0.80 | 19.68916 | 0.94 | 0.72 | 0.83 | 12.72460 | 0.86 | 0.68 | 0.77 | 20.27860 |
| Std | 0.02 | 0.08 | 0.04 | 1.47754 | 0.02 | 0.03 | 0.02 | 0.37847 | 0.05 | 0.04 | 0.04 | 1.68804 |

**AM-(10, 0.1 mej)**

| | *Ionosphere* | | | | *Parkinson* | | | | *Spectf-Heart* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Nº** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** |
| 0 | 0.93 | 0.79 | 0.86 | 18.60021 | 0.98 | 0.73 | 0.85 | 11.54883 | 0.82 | 0.77 | 0.80 | 21.03617 |
| 1 | 0.92 | 0.68 | 0.80 | 18.40387 | 0.91 | 0.86 | 0.89 | 13.37438 | 0.87 | 0.75 | 0.81 | 22.23361 |
| 2 | 0.91 | 0.85 | 0.88 | 18.62032 | 0.92 | 0.91 | 0.92 | 14.54208 | 0.96 | 0.86 | 0.91 | 18.87223 |
| 3 | 0.91 | 0.79 | 0.85 | 23.08223 | 0.94 | 0.82 | 0.88 | 13.50264 | 0.91 | 0.52 | 0.71 | 20.93831 |
| 4 | 0.90 | 0.74 | 0.82 | 31.44513 | 0.98 | 0.82 | 0.90 | 13.82746 | 0.84 | 0.73 | 0.78 | 19.07014 |
| Media | 0.91 | 0.77 | 0.84 | 22.03035 | 0.95 | 0.83 | 0.89 | 13.35908 | 0.88 | 0.73 | 0.80 | 20.43009 |
| Std | 0.01 | 0.06 | 0.03 | 5.02573 | 0.03 | 0.06 | 0.02 | 0.99164 | 0.05 | 0.11 | 0.06 | 1.27708 |

## 5.2 Resumen Global - Tabla de Medias

| | *Ionosphere* | | | | *Parkinson* | | | | *Spectf-Heart* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Alg** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** | **Clas** | **Red** | **Agr** | **T** |
| 1-NN | 0.84 | 0.00 | 0.42 | 0.00280 | 0.78 | 0.00 | 0.39 | 0.00100 | 0.87 | 0.00 | 0.43 | 0.00314 |
| RELIEF | 0.84 | 0.04 | 0.44 | 0.02056 | 0.77 | 0.03 | 0.40 | 0.00487 | 0.87 | 0.04 | 0.46 | 0.01494 |
| BL | 0.89 | 0.85 | 0.87 | 2.22200 | 0.77 | 0.88 | 0.83 | 0.84650 | 0.88 | 0.82 | 0.85 | 3.60924 |
| AGG-BLX | 0.89 | 0.62 | 0.75 | 18.27602 | 0.93 | 0.71 | 0.82 | 13.49899 | 0.85 | 0.58 | 0.72 | 20.67197 |
| AGG-AC | 0.91 | 0.66 | 0.79 | 17.86634 | 0.97 | 0.75 | 0.86 | 12.70768 | 0.87 | 0.64 | 0.76 | 21.65144 |
| AGE-BLX | 0.90 | 0.63 | 0.76 | 20.74966 | 0.92 | 0.70 | 0.81 | 16.56119 | 0.86 | 0.59 | 0.72 | 24.35267 |
| AGE-AC | 0.92 | 0.88 | 0.90 | 20.62200 | 0.98 | 0.94 | 0.96 | 15.86924 | 0.89 | 0.79 | 0.84 | 23.34907 |
| AM1 | 0.92 | 0.84 | 0.88 | 19.39848 | 0.95 | 0.90 | 0.92 | 14.27740 | 0.88 | 0.80 | 0.84 | 21.06626 |
| AM2 | 0.90 | 0.71 | 0.80 | 19.68916 | 0.94 | 0.72 | 0.83 | 12.72460 | 0.86 | 0.68 | 0.77 | 20.27860 |
| AM3 | 0.91 | 0.77 | 0.84 | 22.03035 | 0.95 | 0.83 | 0.89 | 13.35908 | 0.88 | 0.73 | 0.80 | 20.43009 |

## 5.3 Análisis de los resultados

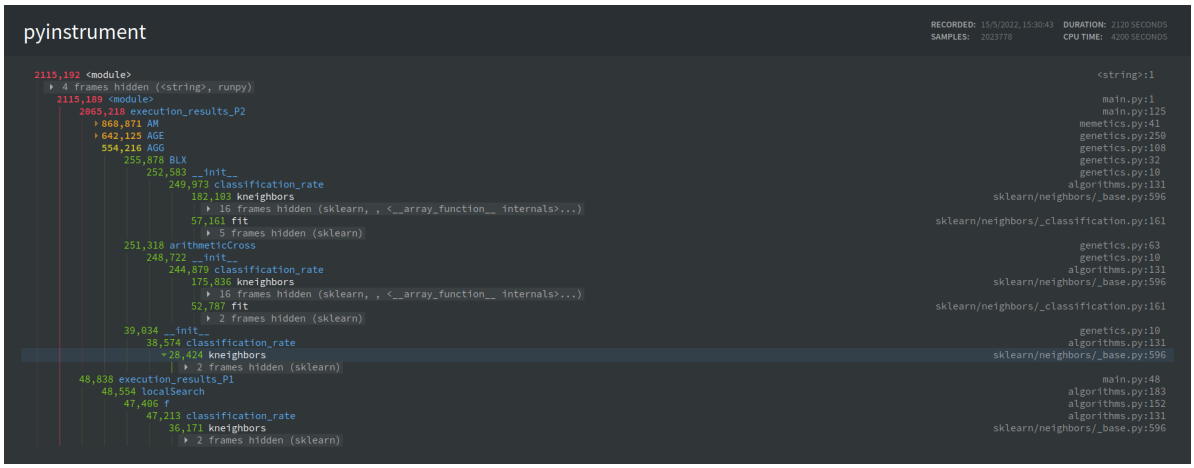
En esta sección realizaremos un extenso análisis a cerca de los resultados obtenidos. Para ello analizaremos cada una de las tasas. Empezamos en primer lugar por el tiempo de ejecución.

**Primero, mencionar que para un mejor análisis visual recomiendo acudir al archivo .ipynb. Puesto que Plotly es una librería que permite la realización de gráficos interactivos, se pueden estudiar mucho mejor las comparativas. A continuación incluyo mis conclusiones tras examinar las distintas gráficas.**

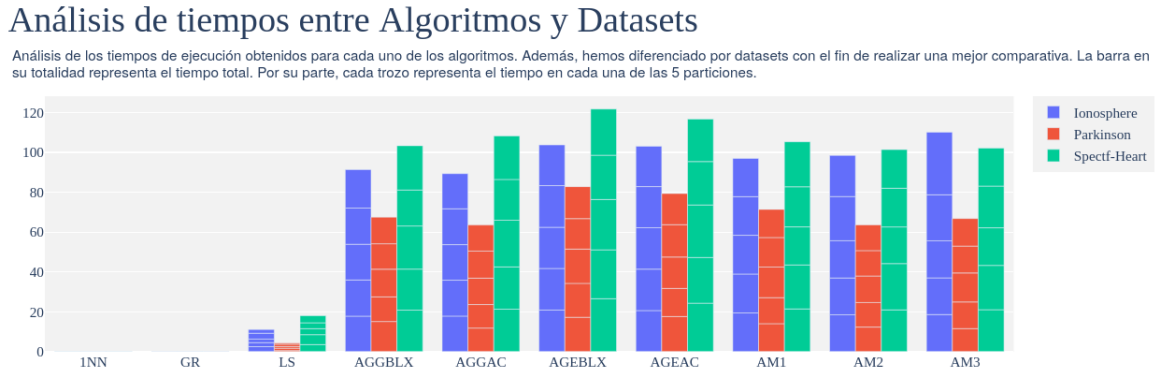
### 5.3.1 Tiempo de ejecución

Antes de comenzar, mencionar brevemente que el algoritmo de búsqueda local ha sido optimizado. Dicho resultado puede observarse comparando los tiempos de ejecución obtenidos con los de la memoria de

la práctica anterior. El cuello de botella que ralentizaba a este algoritmo también impedía una 'rápida' ejecución de los algoritmos genéticos y meméticos, así que fue clave la identificación y optimización de dicha sección de código. El cuello de botella fue fácilmente identificable tras hacer un profiling. En la siguiente imagen podemos observar como el **cálculo de la tasa de clasificación es el cuello de botella.**




Tal y como podemos apreciar en el siguiente gráfico de barras, los algoritmos implementados presentan unos tiempos de ejecución mayores que los de la práctica 1. **Dicha diferencia proviene principalmente del número de evaluaciones**. Para la búsqueda local, el criterio de parada hace que nunca lleguemos a evaluar los 15000 vecinos. Por el contrario, para los algoritmos evolutivos esto siempre pasa. En el profiling, se aprecia la gran diferencia de tiempos empleados en **classification_rate** entre la LS y los algoritmos generacionales.




Así mismo, podemos concluir que, al igual que ocurría en la primera práctica, el dataset de Parkinson se muestra como el más rápido a la hora de realizar la clasificación. Recordemos que era el dataset con **menor número de características**. En cuanto a los otros dos conjuntos de datos, vemos como los tiempos suelen ser algo más similares.

Centrémonos ahora en los algoritmos de esta práctica. Podemos observar como los **algoritmos genéticos estacionarios** son los que **mayor runtime** presentan. Como sabemos los AGE seleccionan únicamente a dos padres de la población, los cuales se cruzan y mutan. Esto implica que sea necesario un mayor número de generaciones (de la población) para llegar a las 15000 evaluaciones de la función objetivo.

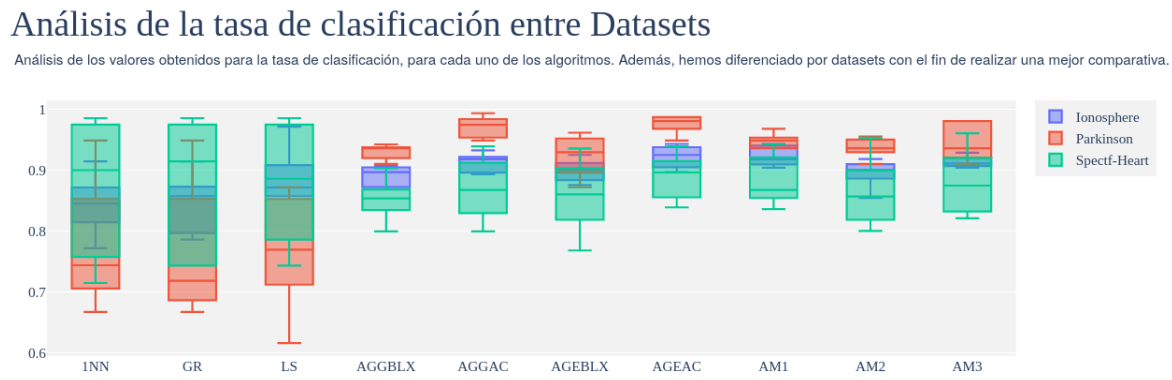
Hemos comprobado, que esto a su vez conlleva una mayor cantidad de números aleatorios generados (entorno a 2000 más). Ya sabemos que una de las cosas mas costosa en tiempo de ejecución de un

algoritmo genético es la generación de números aleatorios. Además, nos hemos tomado la libertad de medir el tiempo promedio que tardan, tanto los AGE como el AGG, en realizar el reemplazamiento. Dicho tiempo es un poco mayor para el reemplazamiento en los AGE. En definitiva, todo esto que he mencionado podrían ser las posibles causas que expliquen los tiempos más elevados de los AGE, en comparación con los AGG.

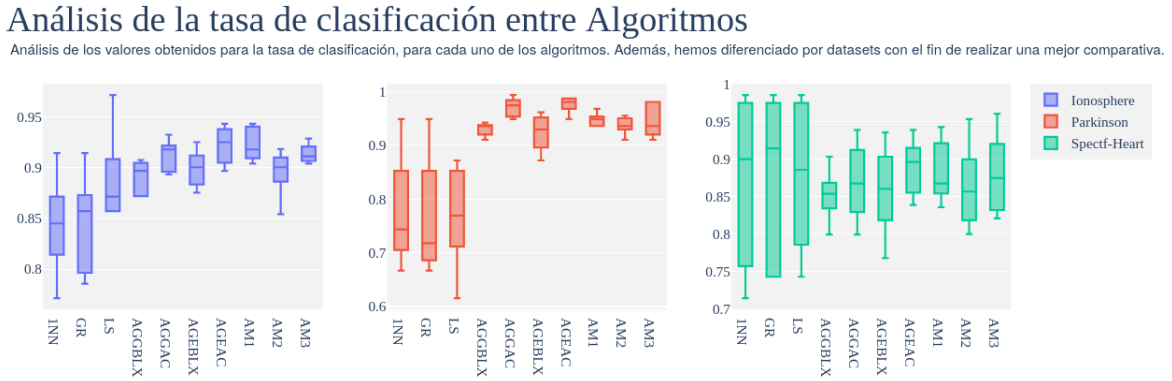
Por último, podemos observar como los tres algoritmos meméticos obtienen tiempos similares entre ellos. Dicho tiempo además, tal y como puede apreciarse es más o menos similar al de los algoritmos genéticos generacionales.

### 5.3.2 Tasa de Clasificación

Empezamos por realizar el análisis para los conjuntos de datos. Hay un cambio de tendencia entre los algoritmos de la P1 y la P2. Mientras que los de la 1era práctica obtenían mejores tasas para spectf-heart y las peores para parkinson, para los algoritmos de esta segunda práctica es al revés. El dataset **Parkinson** es el que mejores resultados obtiene, pues sus cajas sobresalen superiormente del resto. En cuanto a **Ionosphere**, este presenta la menor dispersión en cuanto a los valores obtenidos. Por último, cabe destacar la reducción en la dispersión de los valores obtenidos. Esto se ve a simple vista comparando parkinson y spectf-heart para los algoritmos de la P1 y P2.

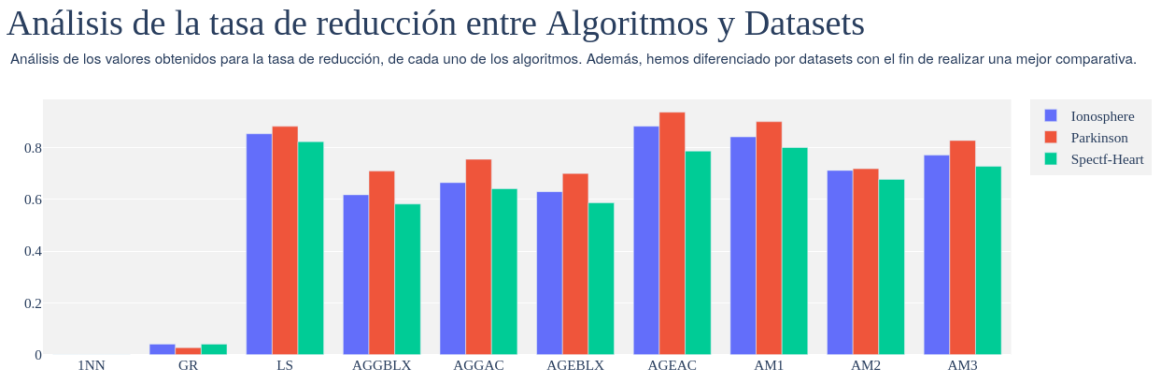



Comparemos ahora los algoritmos. Tal y como hemos comentado antes, se observan unos valores mucho menos dispersos para los nuevos algoritmos. Vemos como además, los algoritmos meméticos y genéticos suelen presentar mejores valores para cada uno de los cuartiles, que los algoritmos de la primera práctica. Esta diferencia es claramente visible sobre todo en el segundo dataset.



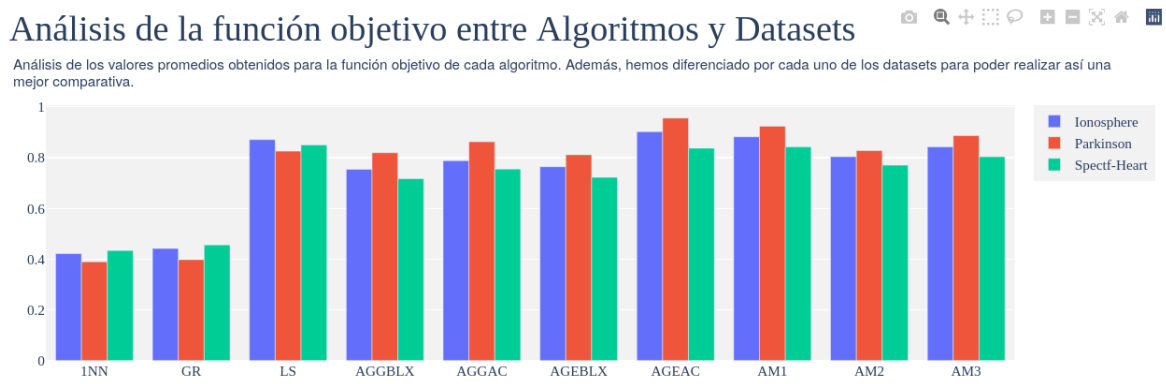
### 5.3.3 Tasa de Reducción

Observando la gráfica posterior, podemos observar como los mejores valores los obtiene el AGEAC, seguido muy de cerca por el de búsqueda local y el AM1. Podemos ver, que entre los algoritmos genéticos son los que usan el operador aritmético los que han obtenido unos mejores resultados. Entre los algoritmos meméticos, vemos como el que peores resultados obtiene es el AM2.




### 5.3.4 Función Objetivo

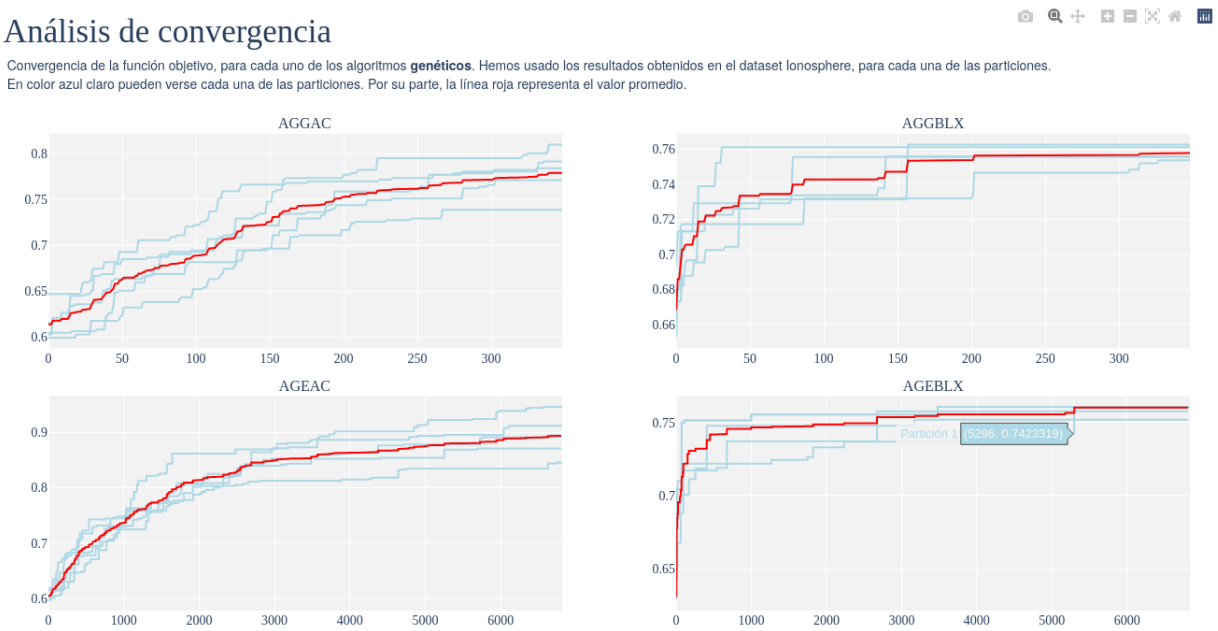
Pasemos a hablar de los valores obtenidos para la función objetivo, la cual queremos maximizar. Al igual que para el apartado del tiempo de ejecución, hemos condensado todo en una misma gráfica. Tanto diferencias entre algoritmo, como datasets.



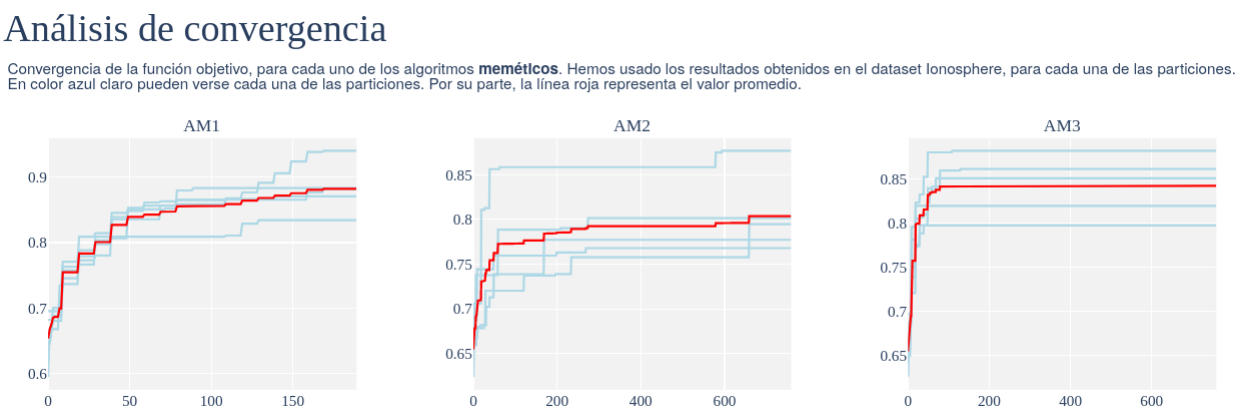

Podemos apreciar que para los **algoritmos de la P2** el **mejor resultado** se obtiene en el dataset **Parkinson**. Esto contrasta con lo asumido en la anterior práctica (dicho dataset es algo más complicado de clasificar comparado con los otros dos). En cuanto a la comparativa de algoritmos, se aprecia como es el AGEAC el algoritmo que obtiene unos mejores resultados promedio. Acordémonos, que era el que mejores tasas de reducción conseguía. Sus resultados son muy similares a los del AM1.

### 5.3.5 Convergencia hacia la solución

En esta sección analizaremos como ha sido la convergencia de cada uno de los algoritmos, en términos de la función objetivo. A continuación hacemos una comparativa entre los cuatro algoritmos genéticos.




Tal y como se puede apreciar, los algoritmos que emplean el cruce aritmético son los que han tenido una convergencia mas progresiva, haciendo uso para ello de todas las iteraciones (generaciones). Por su parte, vemos como por el contrario los algoritmos que han hecho uso del operador de cruce BLX-$\alpha$ presentan una convergencia muy rápida. De hecho, podemos observar como presentan largos períodos de estancamiento. Otra pequeña observación es, que como bien comentamos previamente, los algoritmos generacionales tienen un menor número de generaciones en total . Mientras que estos llegan aproximadamente a la nº 350, los estacionarios llegan a la 6500. Por último, procedemos a realizar dicha comparación entre los algoritmos meméticos.




Vemos que los AM1 y AM2 convergen de una forma más progresiva, mientras el algoritmo AM3 converge muy rápidamente hacia la solución. Este último presenta un período final de estancamiento muy largo.