UNIVERSIDAD DE GRANADA

Aprendizaje de Pesos en Características

12 de Junio 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

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.

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

Para la generación de las soluciones aleatorias 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 ← np.random.uniform(extremo inferior intervalo, extremo superior, longitud del vector)

2.2.2 Mutación

Este operador es común para todos los algoritmos de la práctica. 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: $w(j) \leftarrow w(j) + random(0, 1, scale = sigma)$
3: $w(j) \leftarrow Normalize(w(j))$
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 Big Bang - Big Crunch

El algoritmo Big Bang - Big Crunch, también conocido por la abreviatura BB-BC es fruto de la teoría de la evolución en física y astronomía. Este algoritmo está compuesto principalmente por dos fases:

1. **Big Bang:** esta fase engloba la creación de una población de soluciones iniciales aleatorios. Representa a su vez, un conjunto de procedimientos de disipación de energía en términos de aleatoriedad.
2. **Big Crunch:** consiste en la reducción de dicha población inicial a un conjunto de élite (también conocido como ***elite pool***, exhibido por un centro de masas. Representa a un proceso que distribuye partículas al azar y las reordena.

El coste de una solución dada representará una masa. Entonces, es intuitivo, que aquella solución que tenga el menor coste, es decir la mejor, será el centro de masa.

Como sabemos por la física, la gravedad depende de la masa de los objetos y de la distancia que los separa. Cuanta más masa tienen y más cerca están, mayor es la fuerza. Cuando se separan el doble por ejemplo, la fuerza se reduce a un cuarto.

En nuestro caso, el centro de masas atraerá soluciones potencialmente mejores. En otras palabras, aquellas más cercanas al centro del espacio de búsqueda (universo), o el punto donde el Big Crunch convergerá.

Tal y como podemos ver, presenta pues una gran similitud con los algoritmos genéticos, los cuales hemos estudiado a lo largo de la asignatura. Es por esto, que he elegido esta metaheurística, pues me permite realizar diversas comparaciones con los resultados ya obtenidos en las prácticas de la asignatura.

- **BIG BANG (generación de soluciones).**
  - ***Paso 1:*** generamos una población de tamaño N y evaluamos en la función objetivo. Si es la primera vez, generamos de forma aleatoria total. De lo contrario, partimos del elite pool.
- **BIG CRUNCH.**

  Repetir
  - ***Paso 2:*** Generamos Ns vecinos para cada solución y reemplazamos al padre por el mejor descendiente Ci.
  - ***Paso 3:*** buscar el centro de masa.
  - ***Paso 4:*** Aplicamos la búsqueda local al centro de masa.
  - ***Paso 5:*** Actualizamos el elite pool.
  - ***Paso 6:*** Nos deshacemos de las peores soluciones

  Hasta que solo quede una solución.
- ***Paso 7:*** Volver a (1) si no se cumple la condición de parada.
- ***Paso 8:*** Devolver la mejor solución

Mencionar que como en los criterios de evaluación del proyecto se evalúa por separado la metaheurística y una mejora de esta añadiéndole Búsqueda Local o cualquier otra hibridación, vamos a obviar el PASO 4 para la primera parte. Este paso lo añadiremos para la mejora.

3.1.1 Explicación de las partes del algoritmo

Paso 1: nuestro algoritmo comienza con la fase del Big Bang, donde generamos nuestras soluciones iniciales. En esta primera generación, dichas soluciones se generarán de manera totalmente aleatorias. Por el contrario, para el resto de generaciones de soluciones iniciales, partiremos del conjunto de élite. Luego cada vez se generarán poblaciones mejores.

- **Conjunto de élite**: este es uno de los términos que mayor importancia cobra en el algoritmo. Este conjunto almacena aquellas soluciones que sean mejores. Esto, nos permite realizar una mayor explotación de las soluciones.

Puesto que para la segunda práctica, para los algoritmos genéticos usamos poblaciones con tamaño 30, vamos a mantener el mismo tamaño de tal manera que podamos realizar una comparación coherente entre los algoritmos ya realizados.

Paso 2: una vez realizada la fase del Big Bang, pasamos a la fase del Big Crunch. Lo primero que haremos en esta sección es que para cada solución actual generaremos N_s vecinos, de tal manera que reemplazaremos al padre por el mejor descendiente C_i.

En el BB-BC original la generación de vecinos se rige por las siguientes dos fórmulas:

$$C_i^{new} = C_c + \sigma \tag{1}$$

$$\sigma = \frac{r\alpha(C_{max} - Cmin}{k}, 0 < \frac{\alpha}{k} < 1 \tag{2}$$

donde C_i^{new} es el vecino nuevo y σ la desviación estándar de una distribución normal.

Para el cálculo de σ, r es un nº aleatorio entre el 0 y 1. Así mismo, α es una tasa de reducción cuyo valor también estará comprendido en dicho intervalo. C_{max} y C_{min} son los límites superior e inferior del conjunto de élite en un momento dado. Por último, k es el nº máximo de evaluaciones.

En este paso es donde más adapto la metaheurística a nuestro problema. Como lo que queremos es modificar un vector de pesos, tenemos que darle otro sentido a la ecuación (1). Entonces, σ significará para nosotros el porcentaje de pesos que vamos a mutar. De tal manera que, como mediante (2) se obtienen valores muy pequeños he procedido de la siguiente forma:

$$\sigma = max\{C_{max} - C_{min}, 0.05\} \tag{3}$$

Sabemos que al principio el conjunto de élite tendrá soluciones con valores de masas más dispersos. Esto nos permite entonces, mutar una mayor cantidad de pesos al principio del algoritmo pues la diferencia entre C_{max} y C_{min} será bastante mayor que 0.05. Por el contrario, conforme avance la ejecución dicha diferencia irá reduciéndose. Cada vez se irán mutando menos pesos. Seguramente, llegue un punto en que sea muy pequeña, entonces nos aseguramos de que al menos se mutarán un 5 % de los pesos, de ahí que incluyamos el 0.05 en la ecuación (3).

Paso 3: Tras esto, buscaremos el centro de masas:

- **Centro de masas:** en física, el centro de masas de un sistema discreto o continuo es el punto geométrico que dinámicamente se comporta como si en él estuviera aplicada la resultante de las fuerzas externas al sistema. De manera análoga, se puede decir que el sistema formado por toda la masa concentrada en el centro de masas es un sistema equivalente al original.

En nuestro caso, el centro de masas hara referencia a aquella solución que mayor masa tenga, o lo que es lo mismo, aquella solución que mejor valor tenga para nuestra función objetivo.

Llegados a este punto, aquí es donde se aplicaría el paso 4 del algoritmo. Aplicaríamos una búsqueda local, con un determinado nº de evaluaciones máximas. Este paso solo se incluye en la hibridación con BL.

Paso 5: la siguiente tarea a realizar es la de actualizar el **elite pool**. Para ello, borramos el último elemento del array (nótese que dicho vector de Stellar Objects ha sido ordenado previamente de forma descendente, de mayor a menor, en función de los valores de las masas). Acto seguido, incluiríamos el nuevo centro de masa, y procederíamos a realizar una ordenación de nuevo. Así mismo, almacenaríamos los nuevos valores de c_{max} y c_{min}.

Paso 6: una vez realizadas todas las operaciones necesarias para actualizar la elite pool, procederíamos a reducir el tamaño de la población actual. En cada iteración reducimos en 6 el tamaño actual. En caso de haber únicamente 6 individuos, nos quedaríamos directamente con el mejor, y saldríamos del bucle interno.

Paso 7: una vez fuera, en caso de no cumplirse la condición de parada, volveríamos al PASO 1 del algoritmo, donde se generaría una nueva población inicial a partir del conjunto de élite actual resultante. Este proceso se repetirá hasta que se llegue al número de evaluaciones máximas permitidas, que es nuestro criterio de parada.

A continuación presentamos el pseudocódigo de una manera mucho más extensa y cercana al código. En primer lugar, incluyo el pseudocódigo de la función **neighbours**. Mediante este método generamos todos los vecinos para cada solución de la población actual. Posteriormente, retornamos aquel vecino cuyo valor de masa sea el más alto.

Algorithm 1 neighbours

```
1:  procedure NEIGHBOURS(st,data,classes, trainIndex, testIndex, cmax, cmin)
2:      σ ← max { 0.05, C_max − C_min }
3:      for i in range(NUM_NEIGHBOURS):
4:              for j in range(nº caracteristicas * σ)
5:                  index ← nº aleatorio entre [0, nº características -1]
6:                  mutar(st.w, index)
7:              new_st ← StellarObject(data[trainIndex], classes[trainIndex], st.w)
8:              neighbours.append(new_st)
9:
10:     return vecino con mayor valor de masa
```

Puesto que el pseudocódigo general es muy parecido para ambos algoritmos, explicaremos primero en qué consiste la hibridación y posteriormente incluiremos el pseudocódigo completo.

3.2 Hibridación. Big bang - Big crunch + Local Search

Tal y como comentamos antes, para la hibridación añadiremos el **Paso 4** del algoritmo propiamente dicho. Es decir, aplicaremos una búsqueda local al centro de masas.

Puesto que no se especifica una búsqueda local concreta a aplicar, he decidido usar la que ya había implementado para la sección de prácicas.

Algorithm 2 BBBC

```
1:  procedure BBBC(data,classes, trainIndex, testIndex)
2:      population ← población inicial aleatoria
3:      sort(population)
4:      centre_mass ← population[0]
5:      elite_pool ← population[ : ELITE_POOL_SIZE]
6:      c_max, c_min ← elite_pool[0], elite_pool[-1]
7:
8:      it ← 0
9:  while it < MAX_EVALUACIONES :
10:     seguir ← true
11:     new_tam ← 30
12:     while seguir == True
13:             index ← índice característica aleatoria
14:             Para cada elemento de la población
15:                 neighbour ← neighbours(population[i], data, classes, trainIndex, testIndex, cmax, cmin)
16:                 new_population.apppend(neighbour)
17:             endloop
18:
19:             it ← += new_tam * NUM_NEIGHBOURS
20:             sort(new_population)
21:             centre_mass ← new_population[0]
22:
23:
24:             Paso 4. Solo Híbrido: aplicamos la búsqueda local al centro de masas.
25:
26:
27:             Actualizamos elite_pool
28:             sort(elite_pool)
29:             c_max, c_min ← elite_pool[0].mass, elite_pool[-1].mass
30:             new_population ← reducimos la poblacion
31:     end while
32:
33:     if it + (POPULATION_SIZE - ELITE_POOL_SIZE) < MAX_EVALUACIONES then
34:         population ← generamos nueva población a partir del elite_pool
35:         it += POPULATION_SIZE - ELITE_POOL_SIZE
36: end while
37:     Entrenamos el modelo, predecimos y calculamos las tasas
38:     return tasa clasificación, tasa reducción
```

4 Procedimiento considerado para desarrollar el proyecto

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 de la P1, y algunas funciones auxiliares programadas para las prácticas.
- **genetics.py**: fichero con los algoritmos genéticos implementados en las prácticas.
- **memetics.py**: fichero con los algoritmos meméticos implementados en las prácticas.
- **bbbc.py**: fichero con la metaheurística Big Bang - Big Crunch, y su híbrido.
- **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 Big Bang - Big Crunch

	Ionosphere				*Parkinson*				*Spectf-Heart*			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
0	0.90	0.97	0.94	23.56071	0.69	1.00	0.85	14.11411	0.94	0.86	0.90	21.43855
1	0.83	0.97	0.90	21.53296	0.77	1.00	0.88	13.87712	0.84	0.91	0.88	21.63166
2	0.74	0.91	0.83	22.06799	0.85	1.00	0.92	14.12460	0.76	0.89	0.82	23.87600
3	0.96	0.91	0.93	22.12468	0.67	1.00	0.83	14.63036	0.83	0.80	0.81	22.49936
4	0.96	0.91	0.93	22.28493	0.72	1.00	0.86	13.73606	0.97	0.84	0.91	22.39183
Media	0.88	0.94	0.91	22.31425	0.74	1.00	0.87	14.09645	0.87	0.86	0.86	22.36760

Algoritmo Big Bang - Big Crunch + Búsqueda Local

	Ionosphere				*Parkinson*				*Spectf-Heart*			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
0	0.77	0.91	0.84	24.60504	0.74	1.00	0.87	14.72751	1.00	0.84	0.92	21.76347
1	0.83	1.00	0.91	22.65621	0.85	1.00	0.92	14.56661	0.90	0.89	0.89	22.40183
2	0.89	0.94	0.91	22.60146	1.00	1.00	1.00	15.42183	0.76	0.95	0.86	23.22098
3	0.99	1.00	0.99	22.02246	0.72	1.00	0.86	14.05384	0.89	0.86	0.87	23.19769
4	0.94	0.97	0.96	21.90966	0.67	1.00	0.83	14.01068	0.99	0.89	0.94	22.95220
Media	0.88	0.96	0.92	22.75897	0.79	1.00	0.90	14.55609	0.91	0.89	0.90	22.70724

5.2 Resumen Global - Tabla de Medias

	Ionosphere				*Parkinson*				*Spectf-Heart*			
Alg	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
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
BB-BC	0.88	0.94	0.91	22.31425	0.74	1.00	0.87	14.09645	0.87	0.86	0.86	22.36760
BB-BC + LS	0.88	0.96	0.92	22.75897	0.79	1.00	0.90	14.55609	0.91	0.89	0.90	22.70724

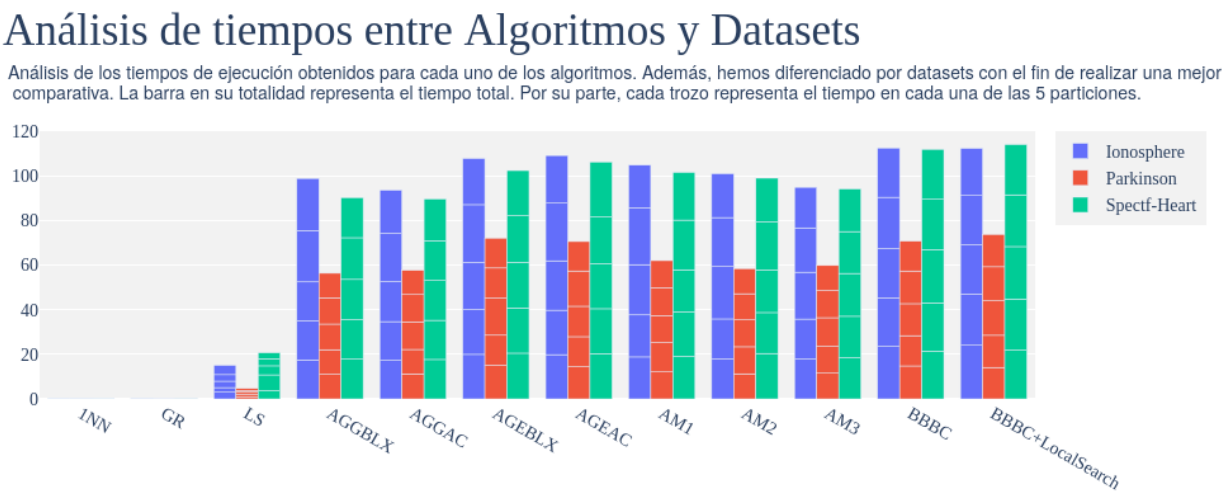
5.3 Análisis de los resultados

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.

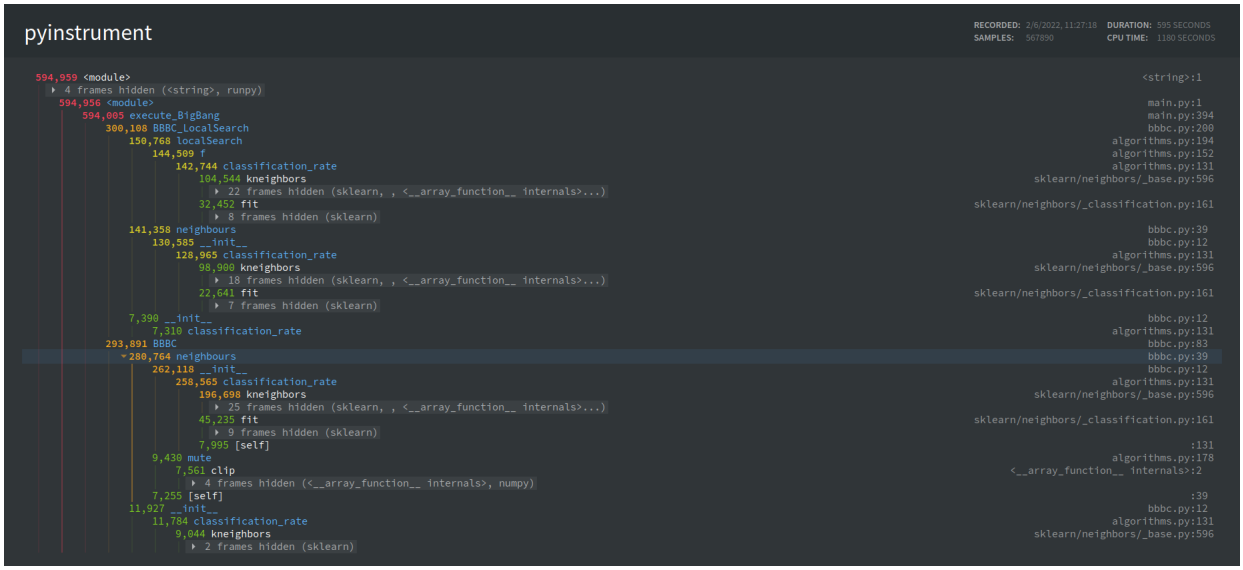
5.3.1 Tiempo de ejecución

Empezamos la sección de análisis con el tiempo de ejecución de los algoritmos. En primer lugar, vamos a realizar una comparativa entre los distintos conjuntos de datos que tenemos. Como podemos observar, los resultados obtenidos para los algoritmos de BB-BC se asemejan con los de los demás algoritmos. **Parkinson** es el dataset que menor runtime presenta. Cabe destacar que es 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.

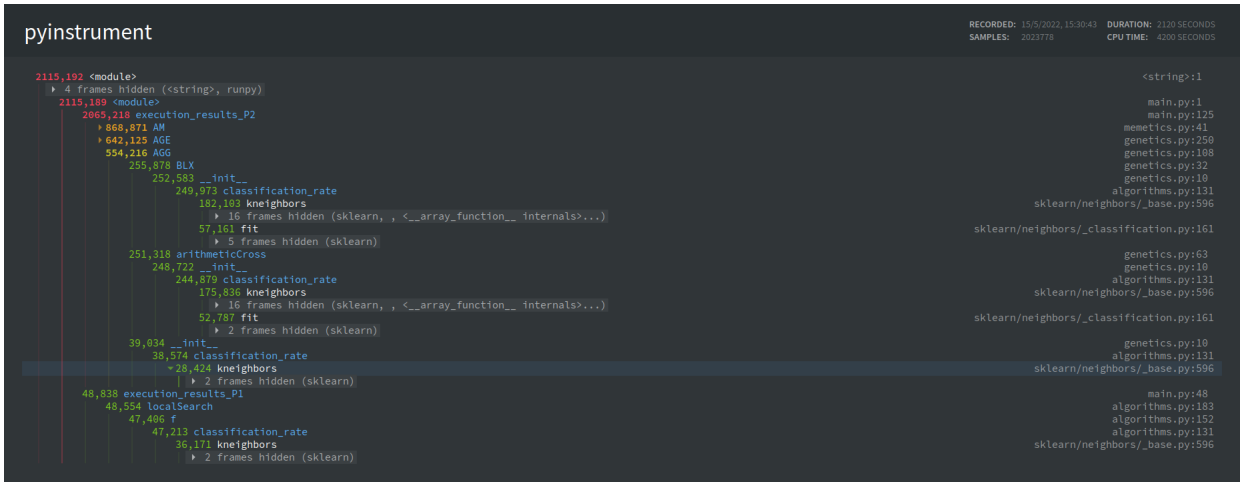


En cuanto a la comparativa entre algoritmos, vemos como los nuevos algoritmos (Big bang - Big crunch, y su hibridación) son los que más tiempo consumen en general. Dicha diferencia se puede notar especialmente para los conjuntos de datos de **Ionosphere** y **Spectf - Heart**. En cuanto a Parkinson, ambos están parejos con los algoritmos genéticos.

Pasemos entonces a analizar el tiempo que necesitan para llevar a cabo la clasificación nuestro algoritmos. Hemos realizado un profiling del tiempo de ejecución total de los algoritmos nuevos, mediante la herramienta **pyinstrument**.Esta, nos enseña cuales son las partes del código que mayor runtime presentan.

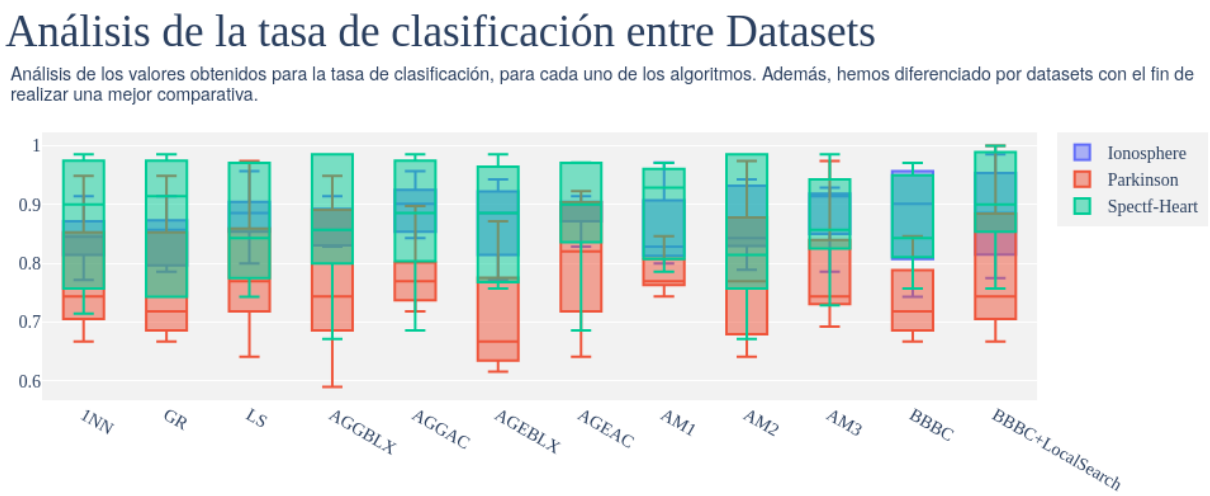


Como podemos apreciar en la imagen incluida, la mayor parte del tiempo se nos va en el método **classification_rate**, con el cuál obtenemos el valor de la tasa de clasificación obtenida para una solución durante el entrenamiento. Dicho cuello de botella es el mismo para el resto de los algoritmos al igual que se muestra en la siguiente imagen, la cual contiene el mismo tipo de profiling pero esta vez para los algoritmos genéticos.

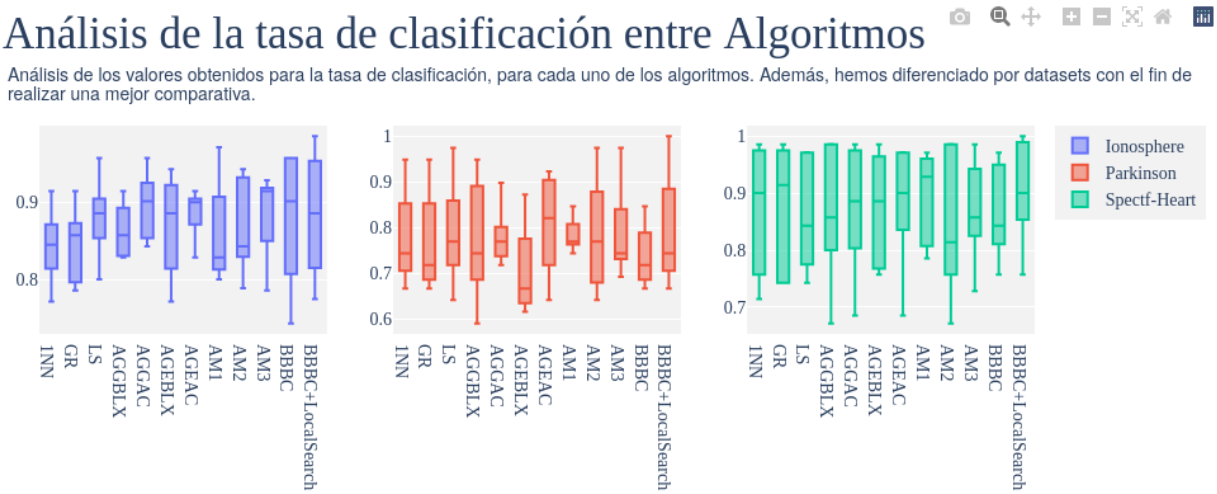


5.3.2 Tasa de Clasificación

A continuación pasamos a analizar los valores obtenidos para la clasificación. Esta sección la dividiremos en dos partes. En la primera, analizaremos los valores obtenidos haciendo una comparativa entre los conjuntos de datos. En la segunda, realizaremos la comparativa entre los diferentes algoritmos.



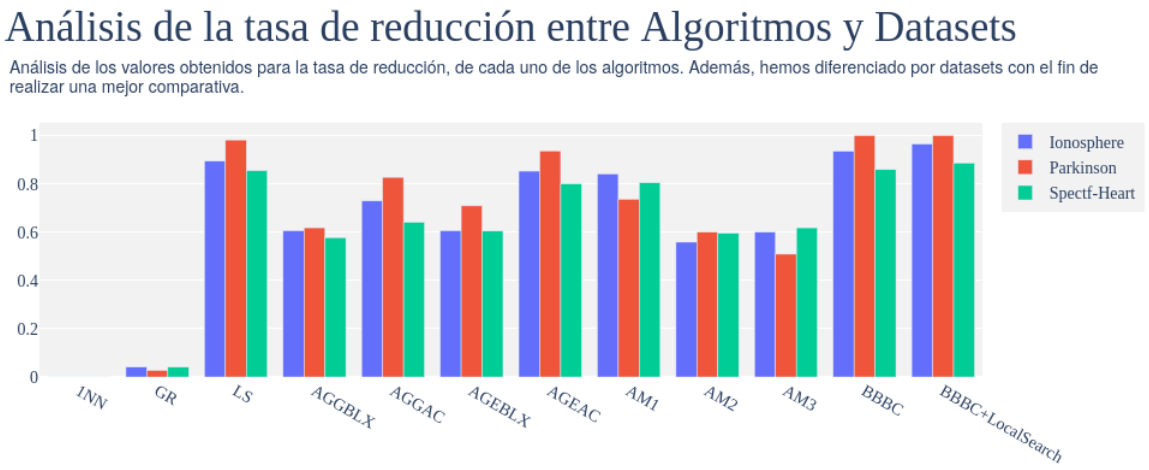
Vemos como para los nuevos algoritmos implementados la tendencia sigue siendo la misma. El dataset de **Parkinson** resulta ser el más complicado de clasificar. Puede observarse que sus cajas correspondientes, las rojas, siempre sobresalen por debajo. Esto puede deberse a que pese a poseer un menor nº de características, sus datos pueden estar más dispersos, dificultando la tarea de clasificación. Tanto para BBBC, como su híbrido, en Ionosphere y Spectf-Heart se obtienen valores similares. Así mismo, presentan un rango parecido en cuanto a la dispersión de los resultados obtenidos (las cajas son prácticamente del mismo tamaño).



En cuanto a la comparativa de algoritmos, podemos apreciar como la búsqueda local y la correspondiente explotación de las soluciones han tenido efecto. El híbrido es el que obtiene el máximo valor para una partición, de todos los algoritmos. Puede observarse a su vez, que suele ser uno de los algoritmos en los que valores obtenidos presentan una mayor desviación típica.

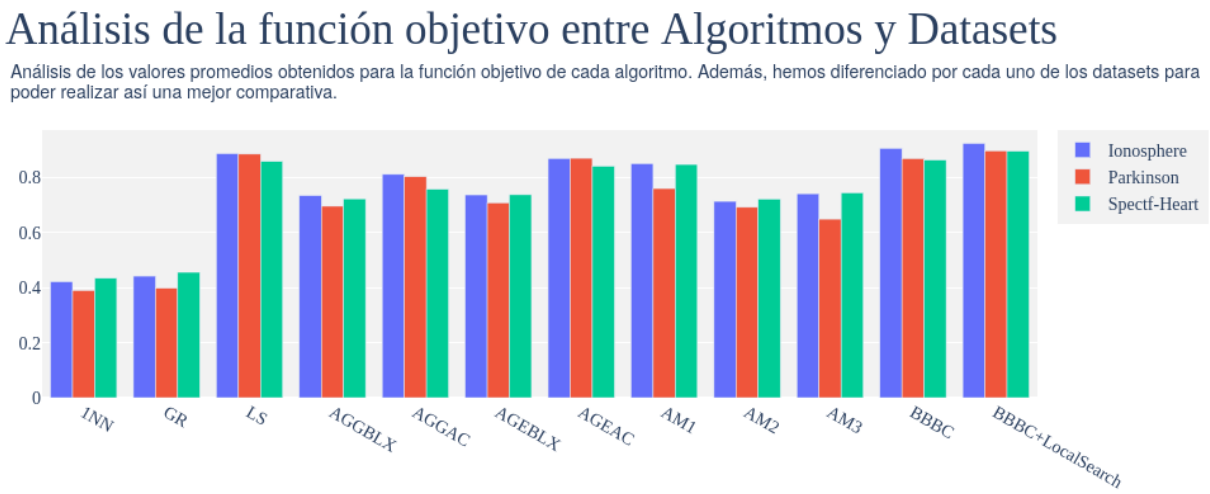
5.3.3 Tasa de Reducción

Pasemos a analizar ahora como de capaz es el algoritmo de reducir los pesos. En la siguiente gráfica podemos apreciar como los algoritmos que hemos implementado para este proyecto son los que obtiene los mejores resultados. Entre ellos mismos, podemos apreciar como la búsqueda local no implica una diferencia significativa en los valores de esta tasa. Esto es porque dichos resultados son prácticamente idénticos.



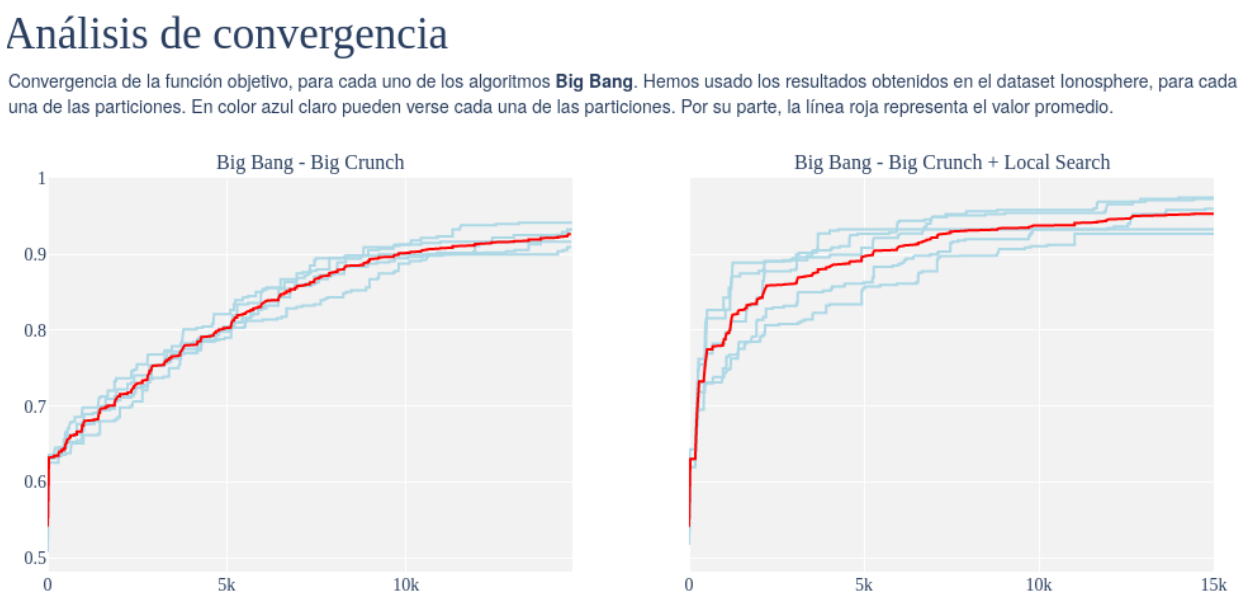
5.3.4 Función Objetivo

Ahora, estudiaremos la comparativa para la función objetivo, que es aquella cuyo valor buscamos maximizar. Podemos observar como los valores obtenidos por los algoritmos del Big Bang son de los mejores, a la par de la Local Search, aunque un poco por encima. Así mismo, vemos como superan por una amplia diferencia a los algoritmos genéticos y meméticos. Entre BB-BC y su híbrido, la diferencia es mínima. Sin embargo esta se decanta a favor del híbrido. Obtiene de media 2-3 puntos más que el BB-BC. Para comprobar dicho análisis de forma más efectiva ir al notebook .ipynb que adjunto en la entrega



5.3.5 Análisis de la convergencia

Por último, vamos a analizar como ha sido la convergencia hacia la solución de los dos algoritmos. Para ello, nos hemos creado varios archivos CSV mediante el uso de la librería *Pandas*, los cuales usamos en el notebook para hacer la gráfica.



Como podemos observar en la gráfica el algoritmo BB - BC presenta una convergencia mucho más progresiva que el hibrído. Esto se debe a que en el híbrido cuando se aplica la búsqueda local al principio, se mejora mucho el valor objetivo. Podemos ver como de media el algoritmo sin búsqueda local necesita 10k evaluaciones para llegar al 0.9 de valor en la función objetivo. Por su parte, el híbrido requiere de la mitad aproximadamente. Sin embargo, podemos ver a su vez como el híbrido presenta largos períodos de estancamiento en la parte media / final de las evaluaciones.

6 Estudio Experimental

Antes de empezar con el estudio experimental, quiero comentar que los resultados vistos hasta ahora corresponden a la mejor de las variantes que vamos a ver a continuación. Debido a las similitudes entre algunos resultados, hemos priorizado los resultados del híbrido. Al fin y al cabo, es el algoritmo que mejor resultado obtiene.

6.1 Tamano del conjunto de élite. Nº de vecinos

Procedemos a mostrar los resultados del análisis experimental llevado a cabo. Este, ha consistido en un **hyperparameter tuning** (ajuste de los parámetros). Los parámetros que hemos intentado ajustar han sido el tamaño del conjunto de élite, y el número de vecinos que se generan. Los diferentes tamaños considerados para el estudio han sido:

- Tamaños del conjunto de élite: 5, 10
- Nº de vecinos generados: 3, 5, 6

A continuación, mostramos los resultados promedio obtenidos. Hemos dividido en dos tablas. En la primera tabla encontramos los resultados obtenidos para el BB-BC sin búsqueda local. En la segunda, los resultados obtenidos para la hibridación.

6.1.1 BB-BC Hyperparameter Tuning

	Ionosphere				*Parkinson*				*Spectf-Heart*			
Alg	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
EP - 10. Vec - 5	0.88	0.94	0.91	22.31425	0.74	1.00	0.87	14.09645	0.87	0.86	0.86	22.36760
EP - 5. Vec - 3	0.85	0.95	0.90	18.96379	0.82	0.99	0.90	11.84648	0.88	0.90	0.89	20.18627
EP - 10. Vec - 6	0.89	0.95	0.92	18.73483	0.77	1.00	0.88	12.33951	0.88	0.85	0.86	20.63002
EP - 5. Vec - 5	0.88	0.95	0.91	17.88934	0.78	1.00	0.89	11.44364	0.89	0.84	0.86	18.89408

6.1.2 BB-BC + Local Search Hyperparameter Tuning

	Ionosphere				*Parkinson*				*Spectf-Heart*			
Alg	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
EP - 10. Vec - 5	0.88	0.96	0.92	22.75897	0.79	1.00	0.90	14.55609	0.91	0.89	0.90	22.70724
EP - 5. Vec - 3	0.89	0.95	0.92	18.57567	0.76	1.00	0.88	11.87192	0.89	0.89	0.89	19.57296
EP- 10. Vec - 6	0.87	0.94	0.90	19.09527	0.80	1.00	0.90	13.04603	0.87	0.87	0.87	20.70465
EP - 5. Vec - 5	0.88	0.98	0.93	18.36273	0.77	1.00	0.88	11.86581	0.87	0.90	0.89	19.08534