UNIVERSIDAD DE GRANADA

Aprendizaje de Pesos en Características

29 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.

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)

VERIFICAR ESTO. NO ES COMUN A TODOS PORQUE HAY ALGUNOS QUE NO LOS USAN .

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:

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

Y calcularemos T_{red} así:

```
procedure T-red(weight)
    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

```
procedure greedyRelief(features, targets)
    w ← [0, ...., 0]
    distances ← matriz cuadrada de distancias (euclídeas)
loop: para cada muestra del conjunto de entrenamiento
    en_indices ← índices de ejemplos con distinto target a la muestra
    fr_indices ← índices de ejemplos con igual target a la muestra
    friends ← features[fr_indices]
    enemies ← features[en_indices]

    closestFriend ← Amigo mas cercano
    closestEnemy ← Enemigo mas cercano

    w ← w + |features(i) - closestEnemy| - |features(i)-closestFriend|
endloop
    w ← Truncamos valores negativos a 0
    w ← Normalizamos w
    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 Búsqueda Multiarranque Básica (BMB)

Es el algoritmo multiarranque más básico que podemos considerar. En él, las soluciones iniciales se generan al azar, y la etapa de búsqueda se realiza mediante el algoritmo de Búsqueda Local. Siguiendo el pseudocódigo de los seminarios y teoría, se ha implementado una versión concisa aprovechando la búsqueda local implementada en las sesiones anteriores.

Algorithm 3 BMB

```
1:  procedure BMB(data,classes, trainIndex, testIndex)
2:      ini_weights ← generamos la solución inicial
3:      bestF ← f(ini_weights)
4:
5:  for i in (range(15)):
6:      w ← localSearch(ini_weights, data, classes)          (15000 / 15 evaluaciones)
7:      currentF ← f(w)
8:
9:      if currentF > bestF then
10:         weights ← np.copy(w)
11:         bestF ← currentF
12:
13:     ini_weights ← generamos nueva solución aleatoria inicial
14: endloop
15:     Entrenamos el modelo, predecimos y calculamos las tasas
16:     return tasa clasificación, tasa reducción
```

3.4 Enfriamiento Simulado

Este algoritmo es bastante más complejo que el anterior. Empezamos comentando algunas particularidades. Para el cálculo de la temperatura inicial se sigue el siguiente esquema: $T_0 = 0.3 * f(inicial)/ - ln(0.3)$. Donde $f(x)$ es nuestra función fitness. La temperatura inicial se calcula como el mínimo entre 10^{-3} y T_0, para que nunca sea la temperatura final mayor que la inicial.

Por otro lado, existen varios criterios de parada. Para el bucle externo el criterio de parada es el número de evaluaciones de la función fitness y la temperatura actual. Mientras que en el bucle interno es el número de vecinos generados y el número de soluciones aceptadas (entre cada enfriamiento).

Finalmente el criterio de enfriamiento es Cauchy modificado donde tenemos que $T_{i+1} = T_i/(1 + \beta * T_i)$, para $\beta = (T_0 - T_f)/(M * T_0 * T_f)$. M es el número de enfriamientos a realizar, es decir el número máximo de evaluaciones a la función fitness, entre el número de vecinos que se genera en cada bucle interno.

Algorithm 4 ES

procedure ES(data,classes, trainIndex, testIndex)
    weights ← generamos la solución inicial
    bestF, currentF ← f(weights)
    best_weights ← weights.copy()

    max_neighbours ← 10 * nº features
    max_successes ← 0.1 * max_neighbours
    M ← nº de enfriamientos a realizar

    $T_0 \leftarrow \frac{0.3 * currentF}{-\log(0.3)}$
    $T_F \leftarrow 10^{-3}$
    $\beta \leftarrow \frac{T_0 - T_F}{M * T_0 * T_F}$
    $T_current \leftarrow T_0$

while num_successes != 0 and $T_{current} > T_F$ and iters ≥ max_iterations
    neighbours, num_successes ← 0
    while num_successes < max_successes and negihbours < max_neighbours
        w ← weights.copy()
        index ← componente aleatoria
        w ← mute(w, index)
        newF ← f(w)
        Δf ← newF - currentF

        if diff > 0 or U(0,1) ≤ $e^{\frac{\Delta f}{T_{current}}}$ **then**
            currentF ← newF
            num_successes ← +1
            weights ← w.copy()
            if currentF > bestF **then**
                best_weights ← weights
                bestF ← currentF
            neighbours ← +1
    endloop
    $T_{current} = \frac{T_{current}}{1 + \beta * T_{current}}$
    coolings ← +1
endloop
    Entrenamos el modelo, predecimos y calculamos las tasas
    return tasa clasificación, tasa reducción

3.5 Búsqueda Local Iterativa

La ILS está basada en la aplicación repetida de la Búsqueda Local a una solución inicial que se obtiene por mutación de un óptimo local previamente encontrado. Siguiendo el pseudocódigo de los seminarios y teoría, se ha implementado una versión concisa aprovechando la búsqueda local implementada en las sesiones anteriores.

Algorithm 5 ILS

```
1:  procedure ILS(data,classes, trainIndex, testIndex)
2:      weights ← generamos la solución inicial
3:      weights ← aplicamos búsqueda local para 15000 / 15 evaluaciones máximas
4:      bestF ← f(weights)
5:      best_weigths ← weights.copy()
6:      mutations ← range(0.1 * nº features)
7:
8:  for i in (range(14)):
9:          for i in mutations
10:             index ← índice característica aleatoria
11:             weigths ← mute(weights, index, 0.4)
12:         endloop
13:         weights ← aplicamos búsqueda local para 15000 / 15 evaluaciones máximas
14:         currentF ← f(weights)
15:         if currentF > bestF then
16:             bestF ← currentF
17:             best_weights ← weights
18:
19:         weights ← best_weights.copy()
20: endloop
21:     Entrenamos el modelo, predecimos y calculamos las tasas
22:     return tasa clasificación, tasa reducción
```

3.6 Híbrido ILS - ES

Basicamente se repite el mismo pseudocodigo pero cambiando la localSearch por el enfriamiento simulado.

Algorithm 6 ILS

```
1:  procedure ILS(data,classes, trainIndex, testIndex)
2:      weights ← generamos la solución inicial
3:      weights ← aplicamos enfriamiento simulado para 15000 / 15 evaluaciones máximas
4:      bestF ← f(weights)
5:      best_weigths ← weights.copy()
6:      mutations ← range(0.1 * nº features)
7:
8:  for i in (range(14)):
9:          for i in mutations
10:             index ← índice característica aleatoria
11:             weigths ← mute(weights, index, 0.4)
12:         endloop
13:         weights ← aplicamos enfriamiento simulado para 15000 / 15 evaluaciones máximas
14:         currentF ← f(weights)
15:         if currentF > bestF then
16:             bestF ← currentF
17:             best_weights ← weights
18:
19:         weights ← best_weights.copy()
20: endloop
21:     Entrenamos el modelo, predecimos y calculamos las tasas
22:     return tasa clasificación, tasa reducción
```

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 de la P1, y algunas funciones auxiliares programadas para la práctica.
- **P3_algorithms.py**: fichero con los algoritmos implementados en esta práctica.
- **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

BMB

	Ionosphere				*Parkinson*				*Spectf-Heart*			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
0	0.86	0.91	0.89	28.95681	0.77	1.00	0.88	10.89507	0.94	0.86	0.90	23.11825
1	0.84	0.91	0.88	23.07019	0.77	1.00	0.88	10.65365	0.91	0.89	0.90	19.02795
2	0.77	0.88	0.83	19.34851	0.95	1.00	0.97	10.32327	0.73	0.95	0.84	18.88538
3	0.91	0.97	0.94	19.68961	0.77	1.00	0.88	11.13045	0.80	0.86	0.83	19.26337
4	0.91	0.91	0.91	17.38932	0.62	1.00	0.81	10.89773	0.96	0.84	0.90	19.06175
Media	0.86	0.92	0.89	21.69089	0.77	1.00	0.89	10.78003	0.87	0.88	0.88	19.87134
Std	0.05	0.03	0.04	4.06673	0.11	0.00	0.05	0.27367	0.09	0.04	0.03	1.62794

ES

	Ionosphere				*Parkinson*				*Spectf-Heart*			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
0	0.92	0.94	0.93	5.56628	0.74	0.95	0.85	2.19462	0.96	0.89	0.92	3.77431
1	0.86	0.97	0.91	4.60594	0.82	0.95	0.89	2.13416	0.86	0.82	0.84	3.13027
2	0.81	0.94	0.88	3.79682	0.87	0.95	0.91	2.08432	0.79	0.91	0.85	3.02167
3	0.89	0.97	0.93	3.13974	0.67	0.95	0.81	2.05098	0.93	0.89	0.91	2.96716
4	0.91	0.97	0.94	3.11074	0.67	0.95	0.81	1.87637	0.97	0.80	0.88	3.05440
Media	0.88	0.96	0.92	4.04391	0.75	0.95	0.85	2.06809	0.90	0.86	0.88	3.18956
Std	0.04	0.01	0.02	0.93626	0.08	0.00	0.04	0.10741	0.07	0.04	0.03	0.29711

ILS

	Ionosphere				*Parkinson*				*Spectf-Heart*			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
0	0.77	1.00	0.89	20.86955	0.69	1.00	0.85	6.97767	0.99	0.91	0.95	19.04552
1	0.90	0.97	0.94	19.08801	0.77	1.00	0.88	7.97694	0.91	0.86	0.89	19.17284
2	0.87	1.00	0.94	23.22469	0.87	1.00	0.94	6.47391	0.73	0.93	0.83	18.66915
3	1.00	1.00	1.00	17.06149	0.77	1.00	0.88	7.57291	0.83	0.98	0.90	19.20604
4	0.91	0.97	0.94	16.45233	0.74	1.00	0.87	6.39850	0.97	0.93	0.95	19.32700
Media	0.89	0.99	0.94	19.33922	0.77	1.00	0.88	7.07999	0.89	0.92	0.90	19.08411
Std	0.07	0.01	0.04	2.49095	0.06	0.00	0.03	0.61477	0.10	0.04	0.04	0.22602

ILS - ES

	Ionosphere				*Parkinson*				*Spectf-Heart*			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
0	0.72	0.76	0.74	3.76587	0.82	0.91	0.86	2.03265	0.96	0.70	0.83	2.29400
1	0.89	0.88	0.88	2.57248	0.82	0.95	0.89	2.06782	0.87	0.68	0.78	2.30661
2	0.87	0.74	0.80	3.11919	0.92	0.86	0.89	2.03176	0.80	0.77	0.79	2.10852
3	0.91	0.76	0.84	2.64675	0.82	1.00	0.91	2.10896	0.91	0.66	0.79	2.09741
4	0.93	0.85	0.89	2.75514	0.72	0.86	0.79	1.87049	0.99	0.73	0.86	2.52641
Media	0.86	0.80	0.83	2.97189	0.82	0.92	0.87	2.02234	0.91	0.71	0.81	2.26659

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
BMB	0.86	0.92	0.89	21.69089	0.77	1.00	0.89	10.78003	0.87	0.88	0.88	19.87134
ES	0.88	0.96	0.92	4.04391	0.75	0.95	0.85	2.06809	0.90	0.86	0.88	3.18956
ILS	0.89	0.99	0.94	19.33922	0.77	1.00	0.88	7.07999	0.89	0.92	0.90	19.08411
ILS-ES	0.86	0.80	0.83	2.97189	0.82	0.92	0.87	2.02234	0.91	0.71	0.81	2.26659

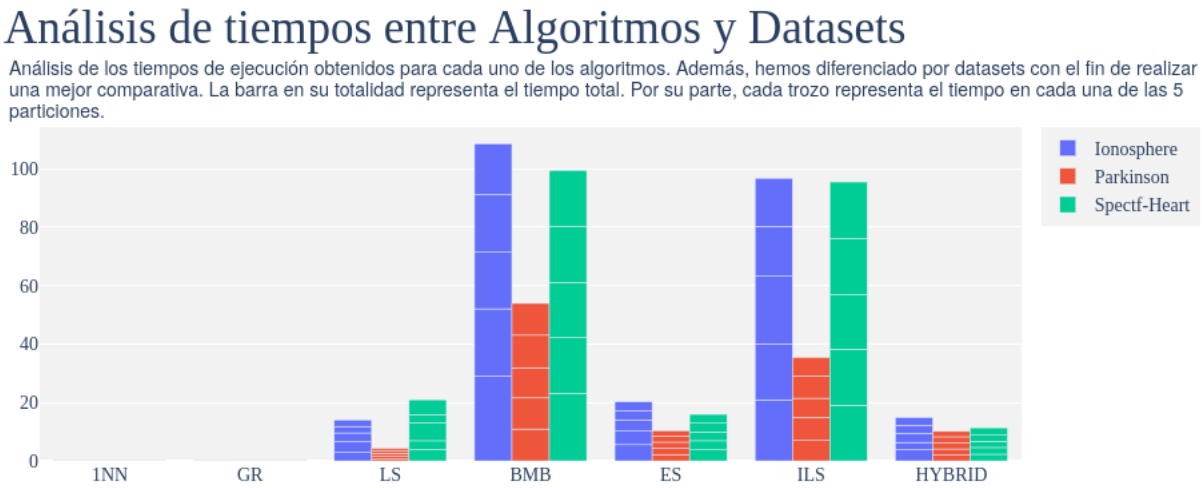
5.3 Análisis de los resultados

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

Tal y como podemos observar en el gráfico que incluyo a continuación, los tiempos de los algoritmos implementados en esta práctica no mejoran los de la búsqueda local.



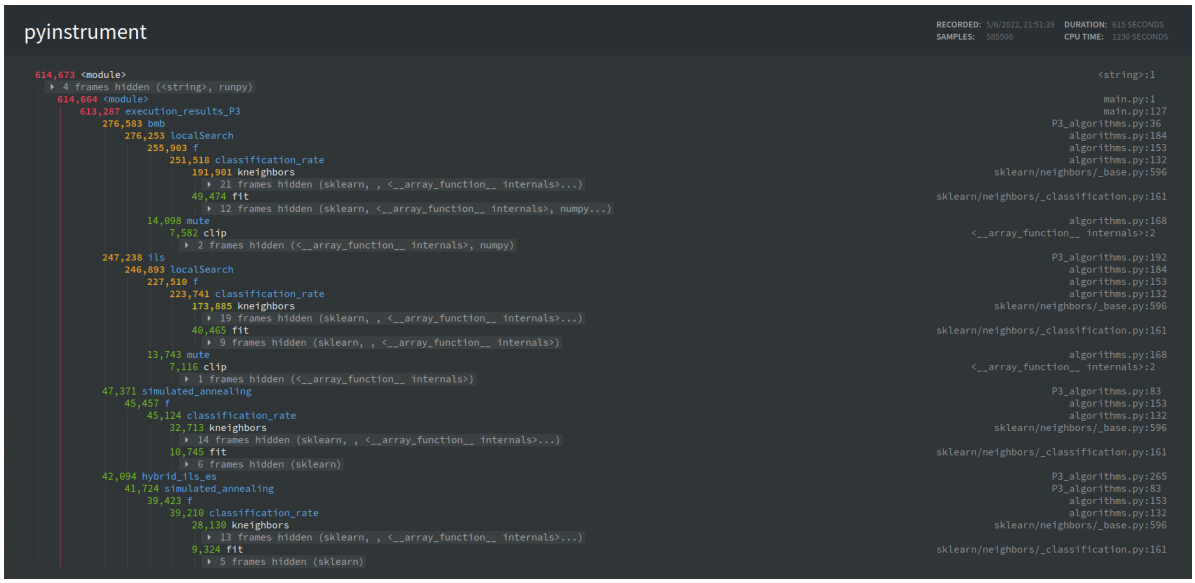
Vamos por partes. En cuanto a los **algoritmos BMB y ILS** podemos observar como presentan un runtime muchísimo mayor al de la búsqueda local. Teniendo en cuenta que estos algoritmos se basan en la aplicación reiterada de la búsqueda local, de primeras puede resultar un poco extraño. Sin embargo, esto tiene sentido debido a que en el algoritmo de la búsqueda local no se llegan a realizar todas las evaluaciones de la función objetivo.

```
(base) javier@javier-Lenovo-ideapad-510-15IKB:~/Escritorio/UGR/DGIIM/4º/Cuatri IMH/Prácticas/P3$ python main.py
Algorithms belonging to P1
File:  Datasets/ionosphere.arff
Máximas evaluaciones:  15000
Iteracion en la que se sale la LS 3295
Partition  1
Máximas evaluaciones:  15000
Iteracion en la que se sale la LS 2273
Partition  2
Máximas evaluaciones:  15000
Iteracion en la que se sale la LS 3121
Partition  3
Máximas evaluaciones:  15000
Iteracion en la que se sale la LS 2057
Partition  4
Máximas evaluaciones:  15000
Iteracion en la que se sale la LS 3702
Partition  5
```

Se puede observar que la búsqueda local no llega a gastar ni un 50 % de las evaluaciones máximas totales de las que dispone. Por el contrario, hemos comprobado que para BMB, por regla general, la búsqueda local suele completar las 1000 evaluaciones correspondientes cada vez que se aplica. En el caso de ILS, hay veces en las que una ejecución de la búsqueda local no llega a esas 1000 evaluaciones. Sin embargo, suele rondar un valor muy cercano (no suele bajar de 500 evaluaciones mínimo). En caso de que se quiera comprobar esto último, basta ejecutar el archivo *main.py* y comprobar las salidas por pantalla para los algoritmos de BMB y ILS.

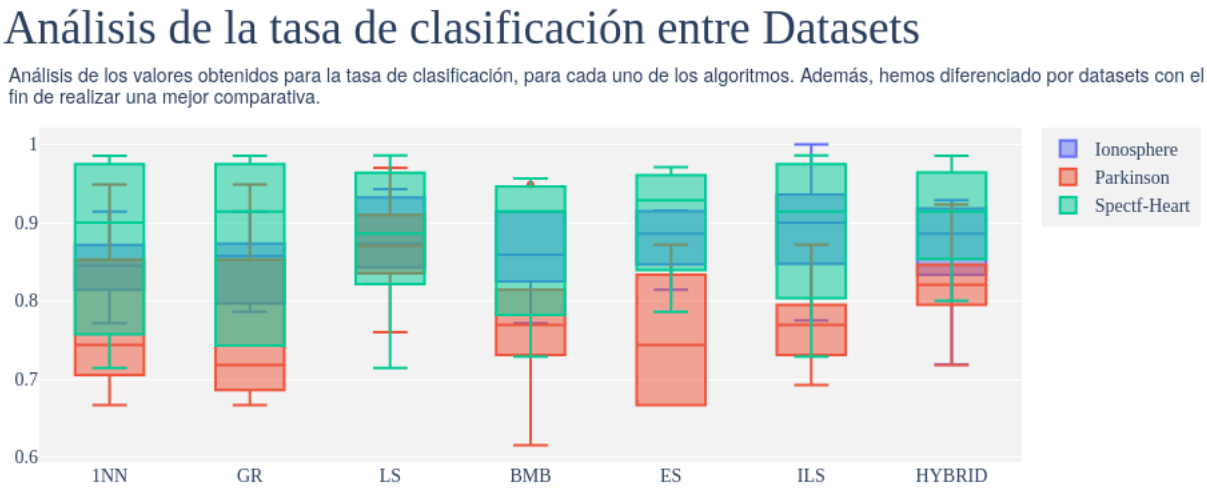
Pasemos a hablar de los **algoritmos ES y ILS-ES**. Podemos observar como los tiempos que obtiene el enfriamiento simulado se asemejan más a los de la búsqueda local. Para los dos primeros datasets vemos que son un poco superiores. Sin embargo, para el dataset de spectf-heart obtiene un tiempo menor. Esto tiene su explicación en que en ambos algoritmos no se llegan a completar las 15000 evaluaciones totales de las que se dispone. En el algoritmo de enfriamiento simulado, no se suelen superar las 3000 evaluaciones por ejecución. Por su parte, en el híbrido una aplicación del algoritmo ES con 1000 evaluaciones, tampoco llega a completar siquiera un 30-40 % de las evaluaciones disponibles.

De la práctica anterior sabemos que el cuello de botella es la función **classification_rate** la cual nos proporciona el valor de la tasa de clasificación. Es evidente entonces, que aquellos algoritmos que más evaluaciones consiguen realizar, sean los que mayor tiempo de ejecución tengan. A continuación adjunto una imagen con el profiling realizado sobre los algoritmos implementados en esta práctica:

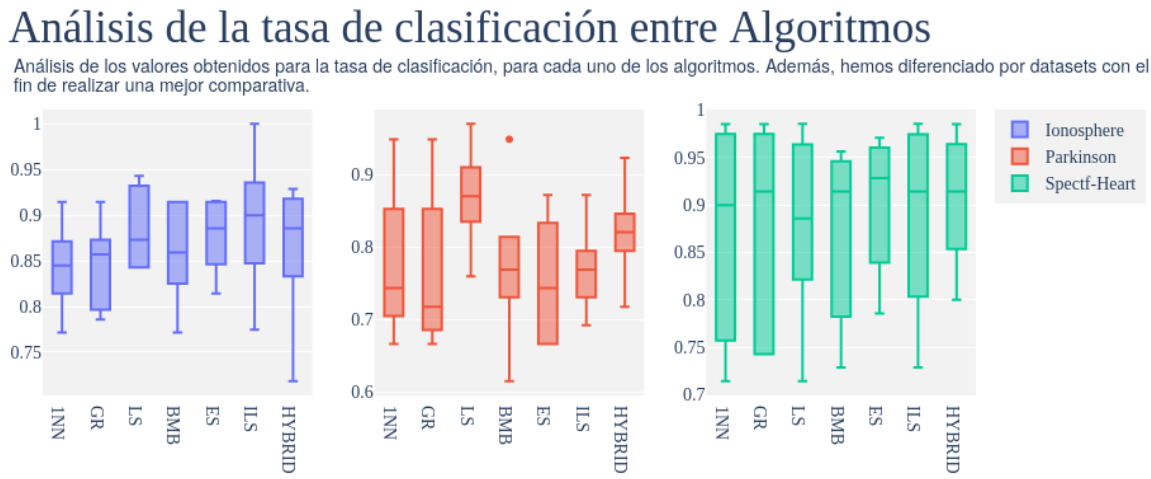


5.3.2 Tasa de Clasificación

Empezamos haciendo un análisis de la comparativa entre los datasets. Para ello observamos la siguiente gráfica. Podemos ver como por regla general, el dataset de Parkinson resulta ser el más complicado de clasificar. Esto puede concluirse debido a que para todos los algoritmos sobresale por debajo la caja roja, correspondiente a los valores obtenidos para este dataset. Así mismo, podemos ver como el dataset de Ionosphere obtiene los valores menos dispersos.



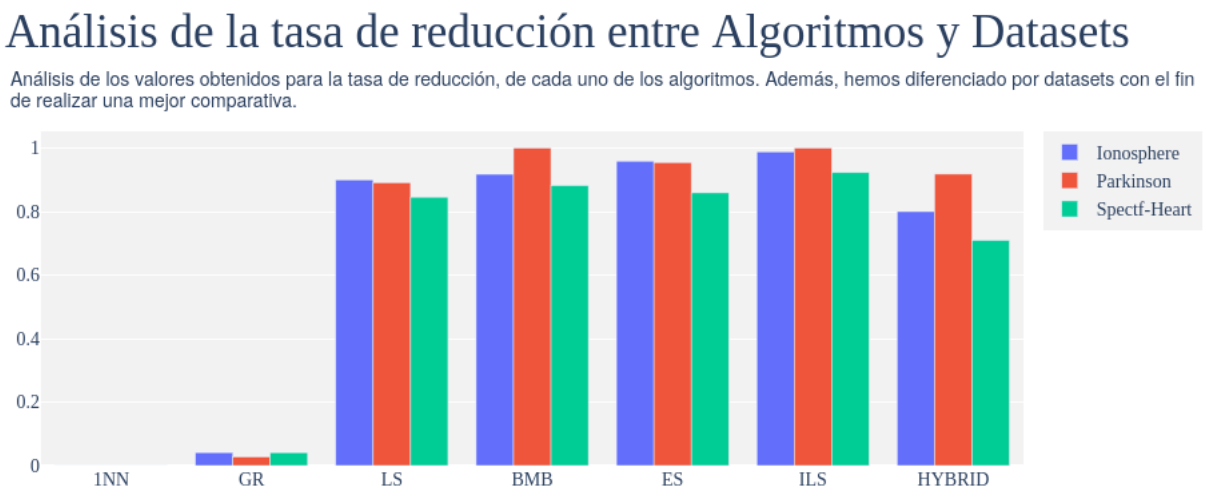
Pasemos ahora a estudiar la comparativa entre algoritmos. Se puede apreciar que entre los algoritmos implementados en esta práctica, son o bien el algoritmo ILS o el híbrido, ILS - ES, los que obtienen el mejor valor, por partición, en general.



Resulta llamativa, en Ionosphere, la diferencia existente entre el valor máximo obtenido por los algoritmos LS y ILS. Aquí, el algoritmo de basado en trayectorias múltiples, debido a su capacidad de explotación de soluciones, y la de salir de óptimos locales, ha sido capaz de encontrar una solución que predice de forma casi perfecta sobre una de las particiones del conjunto de datos.

5.3.3 Tasa de Reducción

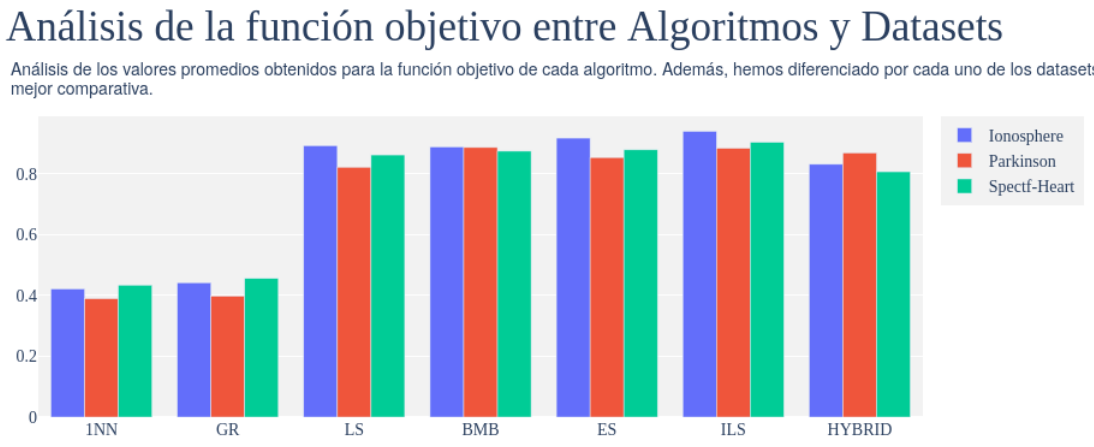
Entre los nuevos algoritmos, vemos como sigue predominando el dataset de Parkinson como el más sencillo para reducir el vector de pesos. Es el que obtiene el valor más alto, para cada uno de los algoritmos. Recordemos que era el dataset con menor número de características. En general, suele ser spectf-heart el dataset más complicado de reducir.



En cuanto a la comparativa entre algoritmos, vemos que los mejores valores promedio los obtienen los algoritmos con búsqueda multiarranque BMB y ILS. Podemos ver como la mayoría de los algoritmos mejoran al valor promedio obtenido por la búsqueda local.

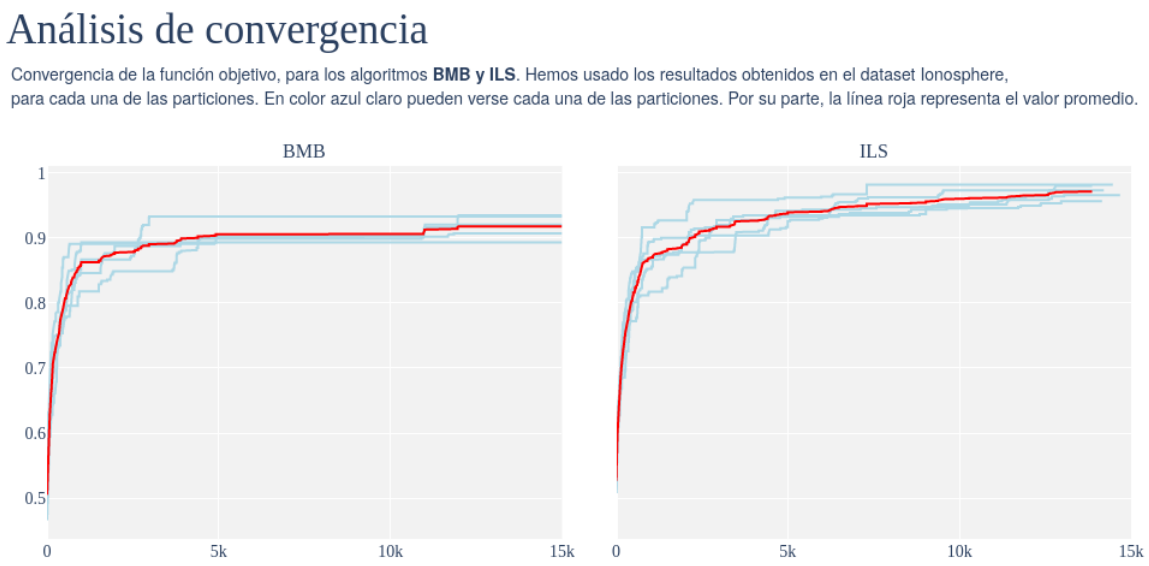
5.3.4 Función Objetivo

Por último, vamos a analizar los valores obtenidos para la función objetivo. En la siguiente gráfica tenemos los valores promedio para cada uno de los algoritmos. Podemos ver como los algoritmos BMB, ES y ILS han obtenido mejores valores que la búsqueda local, en todos y cada uno de los dataset. Por el contrario, el algoritmo híbrido solo ha conseguido superar a la búsqueda local en el dataset segundo, el de Parkinson.



5.3.5 Análisis de convergencia

En este último apartado vamos a estudiar como es la convergencia hacia la solución para algunos de los algoritmos implementados en esta práctica. En la siguiente gráfica tenemos la comparativa entre los algoritmos BMB y ILS:



Como sabemos, la primera aplicación de la búsqueda local es sobre una solución inicial generada aleatoriamente para ambos algoritmos. Esto puede verse claramente al comienzo de las gráficas. Ambos algoritmos presentan un crecimiento progresivo bastante significativo, en las primeras 1000 evaluaciones. Sin embargo, es a partir de aquí donde se aprecian las diferencias. Vemos como la gráfica del BMB presenta largos períodos de estancamiento, mientras que la de ILS sigue creciendo progresivamente (aunque de manera más lenta). Es aquí donde se puede apreciar bien como **la explotación de una solución optimizada (obtenida por una aplicación anterior de la BL) surte efecto para el algoritmo ILS.**