UNIVERSIDAD
DE GRANADA

Metaheurísticas

grado en ingeniería informática

PRÁCTICA 1

Problema del Aprendizaje de Pesos en Características (APC)

Autor
Vladislav Nikolov Vasilev

NIE
X8743846M

E-Mail
vladis890@gmail.com

Grupo de prácticas
MH3 Jueves 17:30-19:30

Rama
Computación y Sistemas Inteligentes

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Curso 2018-2019

Índice

1. Descripción del problema

El problema que se aborda en esta práctica es el Aprendizaje de Pesos en Características (APC). Es un problema típico de *machine learning* en el cuál se pretende optimizar el rendimiento de un clasificador basado en vecinos más cercanos. Esto se consigue mediante la ponderación de las características de entrada con un vector de pesos W, el cuál utiliza codificación real (cada $w_i \in W$ es un número real), con el objetivo de modificar sus valores a la hora de calcular la distancia. Cada vector W se expresa como $W = \{w_1, w_2, \ldots, w_n\}$, siendo n el número de dimensiones del vector de características, y cumpliéndose además que $\forall w_i \in W,\ w_i \in [0, 1]$.

El clasificador considerado para este problema es el 1-NN (genéricamente, un clasificador k-NN, con k vecinos, siendo en este caso $k = 1$), es decir, aquél que clasifica un elemento según su primer vecino más cercano utilizando alguna medida de distancia (en este caso, utilizando la distancia Euclídea). Cabe destacar que no en todos los casos se usará el clasificador 1-NN ya que se pueden dar casos en los que el vecino más cercano de un elemento sea él mismo. Por ese motivo, en algunas técnicas/algoritmos se usará un 1-NN con el criterio de *leave-one-out*, es decir, que se busca el vecino más cercano pero excluyéndose a él mismo.

El objetivo propuesto es aprender el vector de pesos W mediante una serie de algoritmos, de tal forma que al optimizar el clasificador se mejore tanto la precisión de éste como su complejidad, es decir, que se considere un menor número de características. Estos dos parámetros, a los que llamaremos $tasa\_clas$ y $tasa\_red$, respectivamente, se pueden expresar de la siguiente forma:

$$tasa\_clas = 100 \cdot \frac{n^{\text{o}}\ instancias\ bien\ clasificadas\ en\ T}{n^{\text{o}}\ instancias\ en\ T}$$

$$tasa\_red = 100 \cdot \frac{n^{\text{o}}\ valores\ w_i < 0.2}{n^{\text{o}}\ caracteristicas}$$

siendo T el tamaño del conjunto de datos sobre que el que se evalúa el clasificador.

Por tanto, al combinarlos en una única función a la que llamaremos $F(W)$, la cuál será nuestra función objetivo a optimizar (maximizar), tenemos que:

$$F(W) = \alpha \cdot tasa\_clas(W) + (1 - \alpha) \cdot tasa\_red(W)$$

siendo α la importancia que se le asigna a la tasa de clasificación y a la de reducción, cumpliendo que $\alpha \in [0, 1]$. En este caso, se utiliza un $\alpha = 0.5$ para dar la misma importancia a ambos, con lo cuál se pretende que se reduzcan al máximo el número de características conservando una $tasa\_clas$ alta.

2. Descripción de los algoritmos

2.1. Consideraciones previas

Primeramente, antes de empezar con la descripción formal de los algoritmos implementados, vamos a describir algunos aspectos comunes, como por ejemplo cómo se representan las soluciones, como se inicializan en algunos de los casos y algunas funciones utilizadas en muchas partes del código, como por ejemplo la función objetivo. Cabe destacar que muchos de los pseudocódigos que aparezcan a continuación, no se encuentren como tal en la implementación, ya que ya hay funciones que se encargan de hacerlos.

Como se dijo al principio, cada solución es un vector W en el que $\forall w_i \in W,\ w_i \in [0, 1]$. Por tanto, para evitar que las soluciones se salgan de este intervalo, se ha implementado una función que se encarga de normalizar los valores de W en el rango. La función se ha usado, por ejemplo, en la búsqueda local, para hacer que al aplicar el operador de generación de un nuevo vecino la solución siguiese siendo válida. La implementación de esta función es la siguiente:

Algorithm 1: Función de normalización de W

```
1 NormalizarW (W)
    input  : Vector de pesos W
    output: Vector de pesos W normalizado en [0, 1]
2   foreach w_i ∈ W do
3     if w_i < 0 then
4       w_i ← 0
5     else if w_1 > 1 then
6       w_i ← 1
7   return W
```

Como se ha mencionado anteriormente, también se tienen que generar las soluciones iniciales. En el caso del algoritmo greedy *RELIEF*, se ha tenido que generar un W inicial en el que todos sus elementos son 0. Conceptualmente, en el siguiente

pseudocódigo se puede ver cuál es la idea que hay detrás:

Algorithm 2: Inicialización de un vector de pesos W en *RELIEF*

```
1 GenerarWRelief (N)
    input  : Número de características N
    output: Vector de características W con valores 0
2   W ← vector[N]
3   foreach w_i ∈ W do
4     w_i ← 0
5   return W
```

En el caso de la búsqueda local, para inicializar los valores de W se han generado valores distribuidos uniformemente en el rango $[0, 1]$. Para mostrar como sería eso conceptualmente, aquí se muestra un pequeño pseudocódigo:

Algorithm 3: Inicialización de un vector de pesos W en BL

```
1 GenerarWBL (N)
    input  : Número de características N
    output: Vector de características W con valores aleatorios
2   W ← vector[N]
3   foreach w_i ∈ W do
4     w_i ← ValorAleatorioUniformeRango0-1()
5   return W
```

Una vez que sabemos como se representa e inicializa el vector de pesos W, vamos a comentar algunos detalles extra. Es importante saber como se calcula la distancia a un vecino, ya que esto juega un factor muy importante a la hora de encontrar cuál es el vecino más cercano a un elemento (o el vecino más cercano por el criterio *leave-one-out*). En la implementación de la práctica se ha utilizado un KDTree, que es una estructura de datos parecida a un árbol binario, solo que de K dimensiones. Por dentro, esta estructura utiliza la distancia Euclídea (distancia en línea recta entre dos elementos) para determinar cuál es el elemento más próximo a otro. No hace falta conocer como se implementa esta estructura de datos, pero sí es importante conocer cómo se realiza el cálculo de la distancia Euclídea. En el

siguiente pseudocódigo se puede ver el cálculo:

Algorithm 4: Cálculo de la distancia Euclídea entre dos puntos

```
1 DistanciaEuclidea (e_1, e_2)
      input  : e_1, e_2 dos puntos entre los que calcular la distancia
      output: Distancia Euclídea
2     distancia ← sqrt(sum_{i=1}^{N} (e_1^i - e_2^i)^2)
3     return distancia
```

Y ya sabiendo todo esto, solo nos queda comentar la función objetivo, $F(W)$, que es lo que se pretende optimizar. Para evaluar la función objetivo, necesitamos calcular *tasa_clas* y *tasa_red*. Para calcular lo primero, podemos seguir la idea detrás del siguiente pseudocódigo:

Algorithm 5: Cálculo de la tasa de clasificación

```
1 CalculoTasaClas (y, pred, N)
      input  : y las etiquetas originales, pred las etiquetas predichas y N el
               número de etiquetas
      output: Tasa de elementos bien clasificados
2     bien_clasificados ← 0
3     for i ← 1 to N do
4         if y_i = pred_i then
5             bien_clasificados ← bien_clasificados + 1
6     tasa_clas ← bien_clasificados / N
7     return tasa_clas
```

Para calcular *tasa_red*, suponiendo que queremos saber el número de características por debajo de 0.2 podemos seguir un esquema como el siguiente:

Algorithm 6: Cálculo de la tasa de reducción

```
1 CalculoTasaRed (W, N)
      input  : W el vector de pesos N el número de pesos
      output: Tasa de reducción
2     carac_red ← 0
3     foreach w_i ∈ W do
4         if w_i < 0.2 then
5             carac_red ← carac_red + 1
6     tasa_red ← carac_red / N
7     return tasa_red
```

Y finalmente, para poder calcular la función a optimizar (nuestra función *fitness* u objetivo), teniendo en cuenta que usamos un $\alpha = 0.5$ para ponderar las dos tasas, y que anteriormente hemos calculado ambas tasas, podemos seguir el siguiente esquema:

Algorithm 7: Cálculo de la función objetivo o *fitness*

```
1 CalculoFuncionFitness (tasa_clas, tasa_red, α)
    input : Recibe tasa_clas y tasa_red y calcula la agrupación de ambas
            con un factor α
    output: Valor fitness o agrupación
2   fitness ← α · tasa_clas + (1 − α) · tasa_red
3   return fitness
```

Una vez sabiendo todo esto, podemos comenzar a describir las implementaciones de los algoritmos utilizadas.

2.2. Algoritmos de comparación

Para la comparación de la metaheurística implementada, utilizaremos dos algoritmos/técnicas:

- Un clasificador 1-NN clásico, sin ponderar las características.
- Un algoritmo greedy llamado *RELIEF* para el cálculo de los pesos con los que ponderar las características, y posteriormente utilizar un 1NN con los valores ponderados.

Comencemos por la base: el 1-NN. Este clasificador lo que hace es, dado un conjunto de valores X que pertenecen a una muestra y un elemento e, decir cuál es el $x \in X$ más cercano a e, y por tanto, decir que e pertenece a la misma clase x. Para determinar cuál es el elemento más cercano se puede usar alguna métrica de distancia, como por ejemplo la distancia Euclídea, descrita anteriormente. A la hora de implementarlo, para poder acelerar los cálculos, se puede usar un KDTree, ya que permite realizar una consulta rápida (en los casos más favorables su complejidad temporal es $\mathcal{O}(\log n)$, mientras que en el peor caso es $\mathcal{O}(n)$) utilizando la distancia Euclídea para determinar el vecino más cercano, con la penalización de que tarda un tiempo $\mathcal{O}(n)$ en ser construido. Para poder ver un esquema de su

funcionamiento, se ofrece el siguiente pseudocódigo:

Algorithm 8: Clasificador 1-NN

```
1 1-NN (X, y, e)
      input : X un conjunto de elementos que pertenecen a una muestra, y
              las etiquetas correspondientes a cada x_i ∈ X, e el elemento
              del que buscar su vecino más cercano
      output: Clase a la que pertence el vecino más cercano de e
2     arbol_kd ← KDTree(X)
3     x ← arbol_kd.VecinoMasCercano(e)
4     etiqueta ← y[x]
5     return etiqueta
```

Una vez habiendo explicado el 1-NN, vamos a hablar del algoritmo para el cálculo de pesos *RELIEF*. Es un algoritmo greedy que, comenzando con un W cuyos pesos valen 0, actualiza W para cada $x_i \in X$, buscando para cada x_i cuál es su aliado más cercano (elemento que tiene la misma etiqueta que x_i con el criterio de *leave-one-out*, ya que él mismo podría ser su vecino más cercano) y su enemigo más cercano (elemento que tiene diferente etiqueta a la que tiene x_i).

A la hora de implementarlo, vamos a utilizar 2 KDTree en cada iteración que se van a construir sobre la marcha. En uno se encontrarán todos los aliados de e y en el otro estarán todos sus enemigos. Esto puede suponer una gran penalización por el tiempo de creación de los árvoles, pero es un tiempo insignificante ya que el algoritmo es muy rápido. Después de construir los árboles, se buscará en el caso del aliado más cercano, por el criterio de *leave-one-out*, cuál es el índice de este aliado. En el caso del enemigo más cercano, como este no puede ser él mismo, se buscará el índice del vecino más cercano en ese árbol. Una vez hecho eso, obtendremos los respectivos aliado y enemigo del conjunto de aliados y enemigos. Una vez teniéndolos, ya se puede actualizar el valor de W.

Cuando se ha terminado de iterar sobre todos los elementos de X, se normaliza W para que esté en el rango $[0, 1]$ eligiendo el $w_i \in W$ que sea más grande. Todos aquellos valores por debajo de 0 se truncan a 0, y el resto se normaliza dividiéndolos entre w_m (el w_i más grande).

Una vez dicho esto, veamos cómo sería la implementación en pseudocódigo:

Algorithm 9: Cálculo de los pesos mediante *RELIEF*

```
1  RELIEF (X, Y)
       input  : X un conjunto de elementos que pertenecen a una muestra,
                Y las etiquetas de cada x_i ∈ X
       output: Vector de pesos W
2      N ← num. elementos Y
3      n_carac ← num. caracteristicas X
4      W ← GenerarWRelief(n_carac)
5      for i ← 1 to N do
6          x, y ← X[i], Y[i]
7          aliados ← e_a ⊂ X : etiqueta(e_a) = y
8          enemigos ← e_e ⊂ X : etiqueta(e_e) ≠ y
9          arbol_aliados ← KDTree(aliados)
10         arbol_enemigos ← KDTree(enemigos)
11         aliado_cercano ← arbol_aliados.ObtenerVecinoMasCercanoL1O(x)
12         enemigo_cercano ← arbol_enemigos.ObtenerVecinoMasCercano(x)
13         aliado ← aliados[aliado_cercano]
14         enemigo ← enemigos[enemigo_cercano]
15         W ← W + |x − enemigo| − |x − aliado|
16     w_m ← máx(W)
17     foreach w_i ∈ W do
18         if w_i < 0 then
19             w_i ← 0
20         else
21             w_i ← w_i / w_m
22     return W
```

2.3. Algoritmo del método de búsqueda

A continuación, vamos a estudiar el algoritmo de búsqueda que se ha pedido implementar en esta primera práctica. Se trata de un algortimo de búsqueda local, que partiendo de un vector W con valores aleatorios busca optimizarlo mediante la exploración de los vecinos. Esta exploración se realiza modificando con un valor aleatorio generado a partir de una distribución normal con $\mu = 0$ y $\sigma = 0.3$ un $w_i \in W$, quedándose con el cambio en caso de mejorar la función objetivo, o descartándolo en otro caso.

Para realizar lo explicado anteriormente, se genera una permutación del con-

junto $\{1, 2, \ldots, N\}$, donde N es el número de características, y se van escogiendo las características según el orden de la permutación, aplicándoles el cambio anteriormente descrito. Si no se produce una mejora, se descarta el cambio realizado. Si no se ha producido mejora en la función objetivo con la permutación, se escoge una nueva permutación y se repite el proceso, hasta un máximo de $20 \cdot N$ evaluaciones sin éxito seguidas de la función objetivo. Si se produce mejora, se acepta el cambio y se genera una nueva permutación, repitiendo el proceso. Todo esto se realiza hasta que se hayan realizado 15000 evaluaciones de la función objetivo, o hasta que se dé la condición anterior (demasiadas iteraciones sin mejora).

La búsqueda local utiliza una función para evaluar lo bueno que es un vector W. Esta función lo que hace es aplicar W sobre X en aquellas características donde $w_i > 0.2$, y sobre estos crea un KDTree para buscar los índices de los vecinos más cercanos según el criterio *leave-one-out*. Esta búsqueda, para una mayor velocidad, se aplica sobre todos los elementos de X. Una vez calculados, se obtienen las etiquetas correspondientes, y se realiza la evaluación de la función objetivo. El pseudocódigo de esta función se puede ver aquí:

Algorithm 10: Función para evaluar W en la búsqueda local

```
1 Evaluar (X, Y, W)
    input : X un conjunto de elementos que pertenecen a una muestra,
            Y las etiquetas de cada x_i ∈ X, W vector de pesos
    output: Valor fitness para los datos de entrada
2   X_pesos ← aplicar w_i ∈ W sobre los x_i ∈ X donde w_i > 0.2
3   arbolkd ← KDTree(X_pesos)
4   vecinos ← arbolkd.ObtenerVecinosMasCercanoL1O(X_pesos)
5   pred ← Y[vecinos]
6   tasa_clas ← CalcularTasaClas(Y, pred, num. etiquetas)
7   tasa_red ← CalcularTasaRed(W, num. caracteristicas)
8   fitness ← CalculoFuncionFitness(tasa_clas, tasa_red)
9   return fitness
```

Con todo esto explicado, ya podemos ver el pseudocódigo de la búsqueda local:

Algorithm 11: Cálculo de los pesos mediante la Búsqueda Local

```
1  BusquedaLocal (X, Y)
       input  : X un conjunto de elementos que pertenecen a una muestra,
                Y las etiquetas de cada x_i ∈ X
       output: Vector de pesos W
2      Inicializar semilla aleatoria
3      N ← num. caracteristicas X
4      W ← GenerarWBL(N)
5      evaluaciones ← 0
6      evaluacions_malas ← 0
7      fitness ← Evaluar(X, Y, W)
8      while evaluaciones < 15000 do
9          W_actual ← W
10         orden_caracteristicas ← Permutacion(1 to N)
11         foreach carac ∈ orden_caracteristicas do
12             W[carac] ← W[carac]+ GenerarValorDistribucionNormal(μ, σ)
13             W ← NormalizarW(W)
14             evaluaciones ← evaluaciones + 1
15             nuevo_fitness ← Evaluar(X, Y, W)
16             if nuevo_fitness > fitness then
17                 fitness ← nuevo_fitness
18                 evaluaciones_malas ← 0
19                 break
20             else
21                 evaluaciones_malas ← evaluaciones_malas + 1
22                 W[carac] ← W_actual[carac]
23             if evaluaciones > 15000 or evaluaciones_malas > 20 · N then
24                 return W
25     return W
```

3. Desarrollo de la práctica

La práctica se ha implementado en **Python3** y ha sido probada en la versión 3.7.1. Por tanto, se recomienda encarecidamente utilizar un intérprete de Python3 al ejecutar el código y no uno de la versión 2.X, debio a problemas de compatibilidad con ciertas funciones del lenguaje. Se ha probado el código sobre Linux Mint 19 y al estar basado en Ubuntu 18 no debería haber problemas de compatibilidad con otros sistemas, además de que Python es un lenguaje muy portable. No se ha probado en el entorno **conda**, pero si se consiguen instalar los módulos necesarios, no debería haber problemas.

A la hora de implementar el software, se han utilizado tanto módulos ya incluidos en Python, como el módulo **time** para la medición de tiempos, como módulos científicos y para *machine learning*, como por ejemplo **numpy** y **sklearn**. Este último se ha utlizado para poder dividir los datos para el **5 Fold Cross Validation** y para obtener un clasificador KNN que poder utilizar para poder probar los resultados obtenidos por cada uno de los algoritmos. Para la visualización de datos se ha utilizado **pandas**, ya que permite conseguir una visualización rápida de estos gracias a los DataFrames.

Adicionalmente, la estructura de **KDTree** utilizada ha sido sacada de un módulo externo llamado **pykdtree**[1]. Este módulo está implementado en **Cython** y **C** y también utiliza **OMP**, con lo cuál su rendimiento va a ser muy superior a otras implementaciones como por ejemplo el **cKDTree** de **scipy**[1]. En cuanto a su uso, las funciones y la forma de construirlo son las mismas que las de cKDTree, con lo cuál se puede consultar su documentación[2] para obtener más información sobre su uso.

Siendo ahora más concretos en cuanto a la implementación, se ha creado una función para cada uno de los algoritmos que se han implementado. Estas funciones se encargan de recorrer las particiones creadas y de ejecutar los respectivos algoritmos pasándoles los datos, además de encargarse de recopilar información estadística para mostrarla luego por pantalla. Se han utilizado dos semillas aleatorias las cuáles están fijas en el código: una para dividir los datos, y otra en la búsqueda local, que se fija al principio, justo antes de inicializar los W, como se puede ver en el pseudocódigo. Los ficheros ARFF proporcionados se han convertido al formato CSV con un script propio, con el objetivo de facilitar la lectura de los datos. Estos archivos también se proporcionan junto con el código fuente implementado.

[1] De hecho, pykdtree está basado en cKDTree y libANN, cogiendo lo mejor de cada implementación y paralelizando el código con OMP para conseguir unos rendimientos muy superiores a ambos, tanto a la hora de crear el árbol como para hacer consultas.

4. Manual de usuario

Para poder ejecutar el programa, se necesita un intérprete de **Python3**, como se ha mencionado anteriormente. Además, para poder satisfacer las dependencias se necesita el gestor de paquetes **pip** (preferiblemente **pip3**).

Se recomienda instalar las dependencias, las cuáles vienen en el archivo **requirements.txt**, ya que sin ellas, el programa no podrá funcionar. Se recomienda utilizar el script de bash incluido para realizar la instalación, ya que se encarga de instalarlo en un entorno virtual para no causar problemas de versiones con paquetes que ya se tengan instaladas en el equipo o para no instalar paquetes no deseados. Una vez instalados[2], para poder utilizar el entorno creado se debe ejecutar el siguiente comando:

```
$ source ./env/bin/activate
```

Para desactivar el entorno virtual, simplemente basta con ejecutar:

```
(env) $ deactivate
```

Para ejecutar el programa basta con ejecutar el siguiente comando:

```
$ python3 practica1.py [archivo] [algoritmo]
```

Los argumentos **archivo** y **algoritmo** son obligatorios, y sin ellos el programa lanzará una excepción. En cuanto a sus posibles valores:

- **archivo** puede ser **colposcopy**, **ionosphere** o **texture**.
- **algoritmo** puede ser **knn**, **relief** o **local**.

A continuación, para ilustrar mejor lo explicado hasta el momento, se ofrece una captura de un ejemplo de ejecución del programa. En la imagen se puede ver la siguiente información:

- Se muestra primero el algoritmo que se va a ejecutar, el conjunto de datos sobre el que se ejecuta y el tiempo total.

[2]Si se produce algún error durante la instalación de los paquetes, puede ser debido a pykdtree, ya que al necesitar un compilador que soporte OMP puede fallar en los sistemas OSX. Para evitar estos problemas, el programa puede utilizar un cKDTree de scipy en caso de que a la hora de importar pykdtree se produzca un error, suponiendo a cambio una penalización en el tiempo de ejecución.

- Se puede ver una tabla en la que aparecen los datos referentes a cada partición (tasa de clasificación, tasa de reducción, agrupación y tiempo).
- Se muestran valores estadísticos para cada variable (valores máximo, mínimo, medio, mediana y desviación típica).

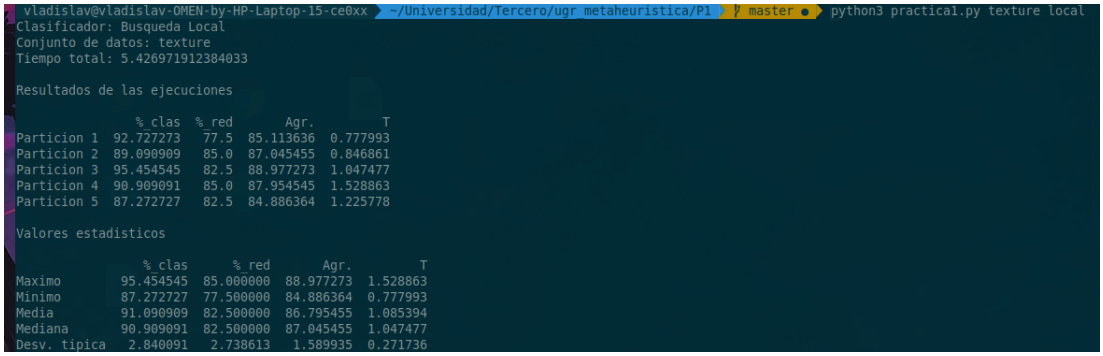


Figura 1: Ejemplo de salida de la ejecución con los datos **texture** y el algoritmo **local**.

5. Análisis de resultados y experimentación

5.1. Descripción de los casos del problema

Para analizar el rendimiento de los algoritmos, se han realizado pruebas sobre 3 conjuntos de datos:

- **Colposcopy**: Conjunto de datos de colposcopias adquirido y anotado por médicos profesionales del Hospital Universitario de Caracas. Las imágenes fueron tomadas al azar de las secuencias colposcópicas. 287 ejemplos con 62 características que deben ser clasificados en 2 clases.
- **Ionosphere**: Conjunto de datos de radar que fueron recogidos por un sistema en *Goose Bay*, Labrador. 352 ejemplos con 34 características que deben ser clasificados en 2 clases.
- **Texture**: Conjunto de datos de extracciones de imágenes para distinguir entre 11 texturas diferentes (césped, piel de becerro prensada, papel hecho a mano, rafia en bucle a una pila alta, lienzo de algodón,...). 550 ejemplos con 40 características que deben ser clasificados en 11 clases.

5.2. Análisis de los resultados

Se han realizado distintas ejecuciones con los datos para cada uno de los algoritmos. Cada conjunto de datos se ha dividido con una función de **sklearn**, y se han mezclado con la **semialla aleatoria** 40. Para el caso de la búsqueda local, tanto para generar el W inicial como para generar las posteriores mutaciones se ha utlizado la **semilla** 8912374. Ambas están fijadas en el código y se pueden comprobar en sus respectivas funciones.

A continuación se muestran los resultados obtenidos para cada algoritmo y para cada uno de los conjuntos de datos, midiendo sus valores de tasa de clasificación, tasa de reducción, agrupación (función objetivo) y tiempo que ha tardado (en segundos). Además se ofrece información extra sobre los valores máximo, mínimo, medio, mediana y desviación típica de cada uno de los datos, para poder comparar los datos más fácilmente. Y, como extra, se ha añadido una tabla en la que aparecen los valores medios de cada algoritmo, para facilitar la comparación:

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	79,66	0	39,83	0,00049	85,92	0	42,96	0,00045	91,82	0	45,91	0,00056
Partición 2	66,67	0	33,33	0,00037	85,71	0	42,86	0,00037	91,82	0	45,91	0,00048
Partición 3	71,93	0	35,96	0,00035	88,57	0	44,29	0,00036	93,64	0	46,82	0,00044
Partición 4	82,46	0	41,23	0,00035	87,14	0	43,57	0,00039	93,64	0	46,82	0,00046
Partición 5	73,68	0	36,84	0,00035	84,29	0	42,14	0,00039	90,91	0	45,45	0,00049
Media	74,88	0	37,44	0,00038	86,33	0	43,16	0,00039	92,36	0	46,18	0,00049
Máximo	82,46	0	41,23	0,00049	88,57	0	44,29	0,00048	93,64	0	46,82	0,00056
Mínimo	66,67	0	33,33	0,00035	84,29	0	42,14	0,00036	90,91	0	45,45	0,00044
Mediana	73,68	0	36,84	0,00035	85,92	0	42,96	0,00039	91,82	0	45,91	0,00048
Desv. Típica	5,62	0	2,81	0,000056	1,44	0	0,72	0,000033	1,09	0	0,55	0,00004

Cuadro 1: Resultados obtenidos por el algoritmo 1-NN en el problema del APC.

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	72,88	29,03	50,96	0,04223180	87,32	2,94	45,13	0,04617286	93,64	7,50	50,57	0,09011912
Partición 2	70,18	25,81	47,99	0,03592849	91,43	2,94	47,18	0,04701781	94,55	15,00	54,77	0,08130646
Partición 3	68,42	48,39	58,40	0,03983450	91,43	2,94	47,18	0,03659153	95,45	7,50	51,48	0,08078218
Partición 4	77,19	64,52	70,85	0,03201556	90,00	2,94	46,47	0,03484225	97,27	5,00	51,14	0,08749294
Partición 5	82,46	29,03	55,74	0,03293490	84,29	2,94	43,61	0,03472829	93,64	2,50	48,07	0,09723902
Media	74,23	39,35	56,79	0,03658905	88,89	2,94	45,92	0,03987055	94,91	7,50	51,20	0,08738794
Máximo	82,46	64,52	70,85	0,04223180	91,43	2,94	47,18	0,04701781	97,27	15,00	54,77	0,09723902
Mínimo	68,42	25,81	47,99	0,03201556	84,29	2,94	43,61	0,03472829	93,64	2,50	48,07	0,08078218
Mediana	72,88	29,03	55,74	0,03592849	90,00	2,94	46,47	0,03659153	94,55	7,50	51,14	0,08749294
Desv. Típica	5,07	14,91	7,91	0,00392631	2,75	0,00	1,37	0,00553680	1,36	4,18	2,15	0,00608498

Cuadro 2: Resultados obtenidos por el algoritmo *RELIEF* en el problema del APC.

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	81,36	83,87	82,61	2,47677159	88,73	85,29	87,01	0,74606109	92,73	77,50	85,11	0,81571221
Partición 2	71,93	82,26	77,09	2,88534594	88,57	79,41	83,99	0,83731246	89,09	85,00	87,05	0,86508775
Partición 3	70,18	82,26	76,22	2,59485793	80,00	94,12	87,06	0,42965746	95,45	82,50	88,98	1,02096820
Partición 4	71,93	82,26	77,09	2,90626860	88,57	91,18	89,87	0,75856829	90,91	85,00	87,95	1,56289935
Partición 5	64,91	75,81	70,36	2,81053543	85,71	91,18	88,45	0,62555575	87,27	82,50	84,89	1,22837043
Media	72,06	81,29	76,68	2,73475590	86,32	88,24	87,28	0,67943101	91,09	82,50	86,80	1,09860759
Máximo	81,36	83,87	82,61	2,90626860	88,73	94,12	89,87	0,83731246	95,45	85,00	88,98	1,56289935
Mínimo	64,91	75,81	70,36	2,47677159	80,00	79,41	83,99	0,42965746	87,27	77,50	84,89	0,81571221
Mediana	71,93	82,26	77,09	2,81053543	88,57	91,18	87,06	0,74606109	90,91	82,50	87,05	1,02096820
Desv. Típica	5,31	2,81	3,89	0,16968432	3,35	5,26	1,95	0,14206914	2,84	2,74	1,59	0,27312796

Cuadro 3: Resultados obtenidos por el algoritmo BL en el problema del APC.

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
1-NN	74,88	0	37,44	0,00039	86,33	0	43,16	0,00039	92,36	0	46,18	0,00049
RELIEF	74,23	39,35	56,79	0,03659	88,89	2,94	45,92	0,03987	94,91	7,50	51,20	0,08739
BL	72,06	81,29	76,68	2,73476	86,32	88,24	87,28	0,67943	91,09	82,50	86,80	1,09861

Cuadro 4: Valores medios de los tres algoritmos en el problema del APC.

Para comparar los resultados podemos tomar como referente la tasa de clasificación, ya que comparar por tiempo (el 1-NN y el algoritmo *RELIEF* son infinitamente más rápidos que la búsqueda local, pero eso no significa que los resultados obtenidos sean mejores) o por tasa se reducción (1-NN no reduce, ya que es simplemente el clasificador normal) o por agrupación (una agrupación alta

no significa que se clasifiquen mejor los elementos; puede que se hayan reducido mucho el número de características) no sería justo.

Hace falta recalcar que, a pesar de haber usado ciertas semillas para intentar hacer que todo el proceso fuese más determinista, los elementos que se encuentran en los conjuntos de entrenamiento y test, así como la forma de generar una solución inicial y de obtener vecinos en la búsqueda local hacen que todo este proceso sea influido por el azar, ya que a lo mejor con una distribución diferente de los datos se hubiesen conseguido resultados distintos, más favorables para un método o para otro. Por tanto, estos resultados obtenidos no pueden ser considerados concluyentes al 100 % a la hora de decidir qué técnica es mejor, ya que hay un factor no determinista por debajo que influirá de una u otra forma en el desarrollo de las operaciones. Una vez habiendo matizado esto, comencemos con el análisis.

Siguiendo los valores medios obtenidos, podemos ver que para el conjunto de datos **Colposcopy** el algoritmo 1-NN es el que mejores resultados ofrece, seguido por muy cerca de *RELIEF*, y un poco más lejos, la búsqueda local. Sin embargo, aunque dijésemos que solo se consideraría la tasa de clasificación, si nos fijamos en la tasa de reducción, podemos ver que la búsqueda local tiene una tasa media de reducción muy superior a los otros dos (hay que considerar que, en los datos de *RELIEF*, aunque aparezca una tasa de reducción, no se aplica sobre los datos; solo se ha dejado para que se vea qué porcentaje de los $w_i \in W$ eran menores a 0.2), y por tanto, si se dispusieran de más datos, a lo mejor la búsqueda local generalizaría mejor. Es importante destacar, además, que *RELIEF* tiene una desviación típica menor, con lo cuál la tasa de variabilidad de los resultados es menor que la de los otros algoritmos.

Lo descrito anteriormente se puede ver en el siguiente *boxplot*, donde se puede ver que uno de los resultados obtenidos por la búsqueda local se salen de los "bigotes", lo cuál hace que el valor de la desviación típica sea mucho más grande:

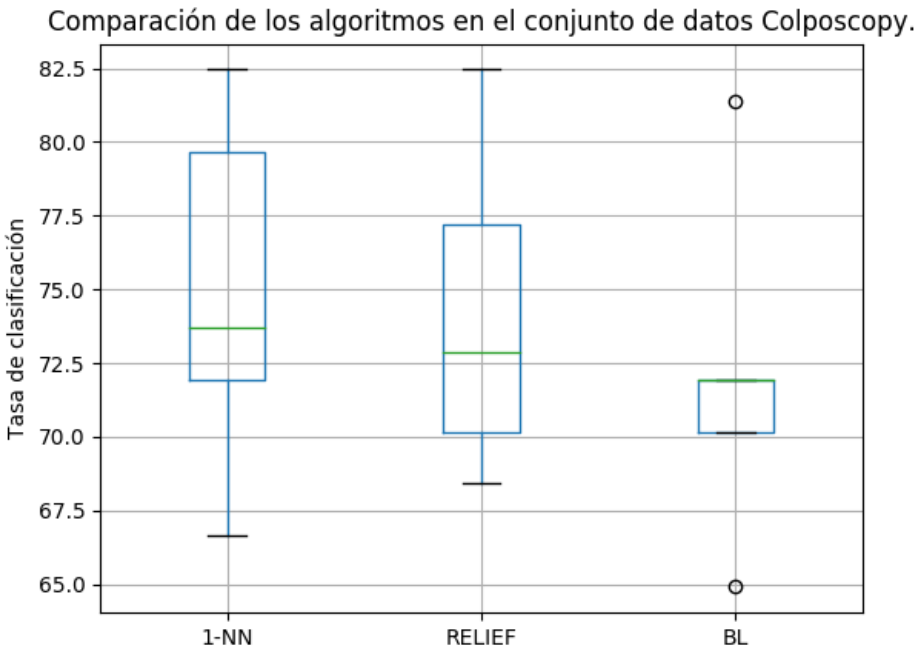


Figura 2: *Boxplot* que compara los algoritmos en Colposcopy.

Pasando a analizar los datos de **Ionosphere**, podemos ver que *RELIEF* es el que obtiene la tasa media de clasificación más alta, dejando casi igualados al 1-NN y a la búsqueda local. Además, es el que ofrece un valor de tasa de clasificación máximo más elevado con un 91.43 %. Sin embargo, el que obtiene una menor desviación típica es el 1-NN, con lo cuál, en ese caso, los datos no estarán tan dispersos. La búsqueda local se queda muy atrás en cuanto a desviación típica, siendo la que tiene un valor más alto, significando esto que hay una mayor variabilidad de resultados dentro del conjunto. Sin embargo, siguiendo de nuevo el mismo esquema, por la tasa de reducción obtenida en la búsqueda local, puede que para más datos sea la que mejor generalice, pero como el conjunto es finito y no se disponen de más, no se puede llegar a determinar con ciencia cierta cuál sería su comportamiento.

Esto se puede ver en el siguiente gráfico, donde se puede ver que hay un valor en la búsqueda local se queda fuera del rango de los "bigotes", haciendo que la desviación típica tenga un valor más elevado:

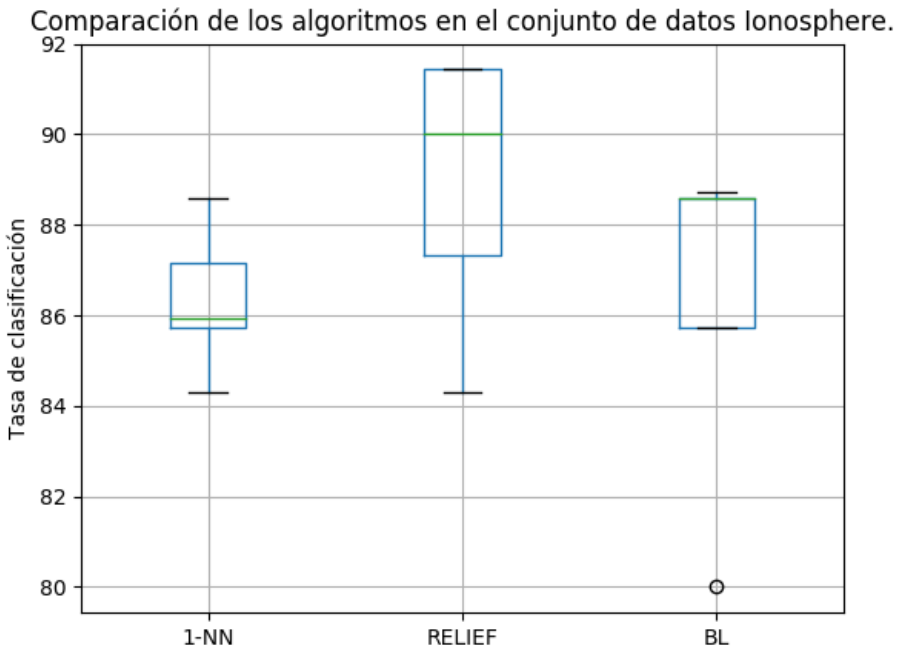


Figura 3: *Boxplot* que compara los algoritmos en Ionosphere.

Para finalizar el análisis, observemos que sucede con el conjunto de datos **Texture**. En este caso, de nuevo, nos encontramos que *RELIEF* ofrece la mayor tasa de clasificación, seguido del 1-NN, y en último lugar la búsqueda local. A pesar de estar en último lugar, los resultados de la búsqueda local no dejan de ser bastante buenos, ya que están por encima del 90 % de tasa de clasificación. Y, como sucedió en el caso anterior, las desviaciones típicas sitúan al 1-NN en mejor posición, seguido de *RELIEF*, y teniendo la búsqueda local en último lugar. Por tanto, la búsqueda local tendrá una variabilidad de los resultados muy alta, mientras que en el caso del 1-NN casi no habrá, y en el *RELIEF* habrá cierta variabilidad, pero de forma muy moderada. De nuevo, insistiendo en lo que ya se ha dicho hasta ahora : la búsqueda local reduce mucho la cantidad de características utilizadas, y por tanto, si llegan a haber más datos, podría darse el caso de que generalizase mejor.

En el siguiente *boxplot* se puede ver lo descrito anteriormente de forma gráfica:

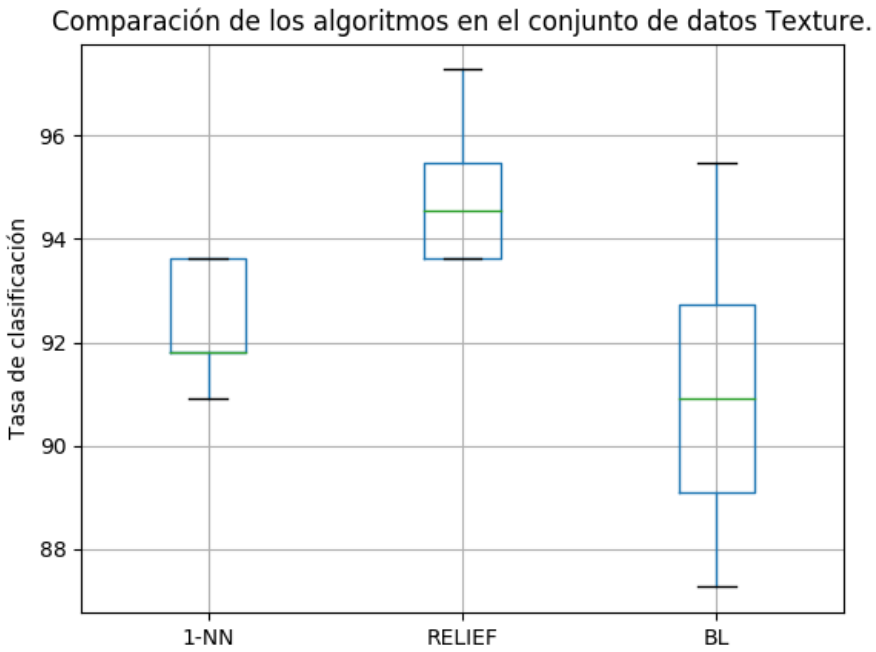
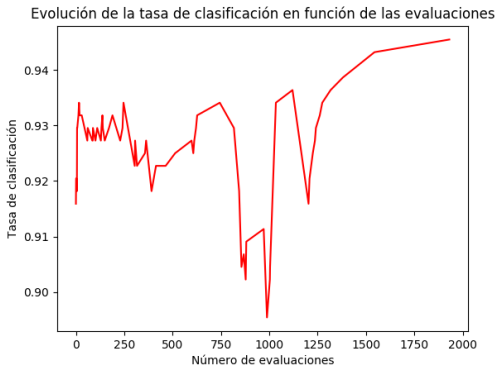
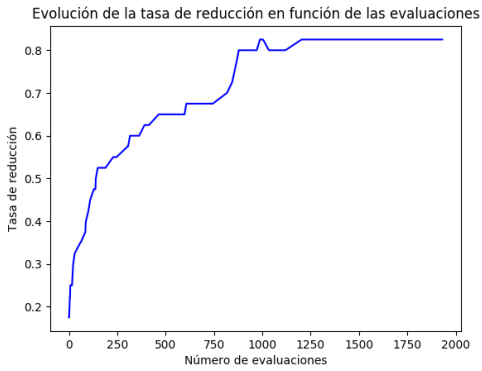
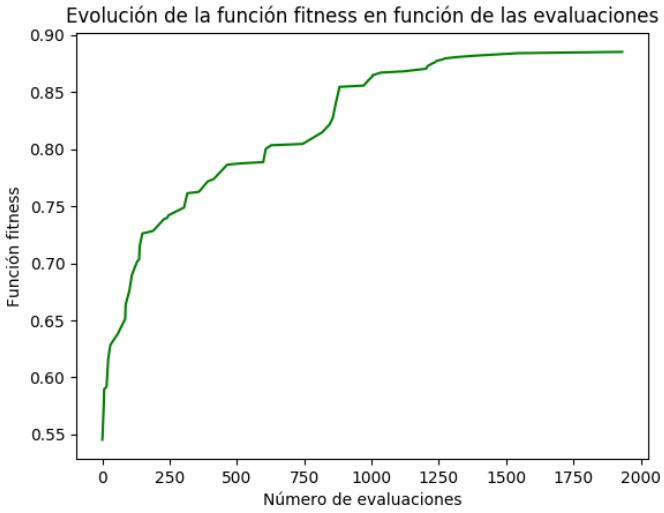


Figura 4: *Boxplot* que compara los algoritmos en Texture.

Para ver el comportamiento de la búsqueda local, vamos a observar los siguientes gráficos, que describen la evolución de las tasas de crecimiento y reducción, y del valor *fitness* en la Partición 3 del conjunto de datos **Texture**:







Se puede observar que la tasa de clasificación evoluciona de manera muy abrupta, ya que a medida que van pasando las evaluaciones su valor se ve, o bien incrementado, o bien decrementado, pero sin seguir ninguna tendencia fija, influenciado posiblemente por la tasa de reducción, la cuál sí que crece de una forma más suavizada. Como el objetivo es maximizar la función objetivo, es posible que sea mucho más fácil ir mejorándola a medida que se eliminan características que intentado como tal mejorar la tasa de clasificación. Por eso se puede ver su evolución tan poco natural al compararla por ejemplo con la tasa de reducción, la cuál si que crece de manera maś suave. La agrupación de ambos elementos se puede ver claramente que siempre va evolucionando, hasta que llega un momento en el cuál no se puede mejorar más. Hay que prestar atención en que la búsqueda local no llega a completar 15000 iteraciones si no que para mucho antes, dándose la condición de que se han explorado $20 \cdot N$ vecinos sin éxito.

5.3. Experimentación

En esta sección queremos presentar una moficación realizada para ver si se consiguen mejores resultados.

Esta modificación consiste en mezclar el algoritmo *RELIEF* y la búsqueda local. Es decir, se pretende que los valores iniciales de W no sean valores aleatorios, si no más bien que reciba los valors del algoritmo greedy, y que a partir de ahí, intente mejorarlos. Por tanto, queremos combinar los buenos resultados del algoritmo greedy con la capacidad de explorar que ofrece la búsqueda local.

Los resultados se pueden ver en las tablas de a continuación:

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	64,41	80,65	72,53	1,16521	85,92	88,24	87,08	0,74191	87,27	85,00	86,14	0,98343
Partición 2	78,95	82,26	80,60	1,73750	92,86	85,29	89,08	0,92928	90,00	85,00	87,50	0,96770
Partición 3	73,68	88,71	81,20	0,53926	87,14	85,29	86,22	1,22533	92,73	85,00	88,86	0,87373
Partición 4	73,68	87,10	80,39	0,66275	95,71	85,29	90,50	1,10265	92,73	82,50	87,61	0,72119
Partición 5	80,70	91,94	86,32	0,93132	80,00	85,29	82,65	0,74525	86,36	82,50	84,43	0,84195
Media	74,28	86,13	80,21	1,00721	88,33	85,88	87,10	0,94888	89,82	84,00	86,91	0,87760
Máximo	80,70	91,94	86,32	1,73750	95,71	88,24	90,50	1,22533	92,73	85,00	88,86	0,98343
Mínimo	64,41	80,65	72,53	0,53926	80,00	85,29	82,65	0,74191	86,36	82,50	84,43	0,72119
Mediana	73,68	87,10	80,60	0,93132	87,14	85,29	87,08	0,92928	90,00	85,00	87,50	0,87373
Desv. Típica	5,68	4,16	4,42	0,42466	5,51	1,18	2,69	0,19223	2,66	1,22	1,51	0,09494

Cuadro 5: Resultados obtenidos por el algoritmo BL+*RELIEF* en el problema del APC.

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
1-NN	74,88	0	37,44	0,00039	86,33	0	43,16	0,00039	92,36	0	46,18	0,00049
RELIEF	74,23	39,35	56,79	0,03659	88,89	2,94	45,92	0,03987	94,91	7,50	51,20	0,08739
BL	72,06	81,29	76,68	2,73476	86,32	88,24	87,28	0,67943	91,09	82,50	86,80	1,09861
BL+RELIEF	74,28	86,13	80,21	1,00721	88,33	85,88	87,10	0,94888	89,82	84,00	86,91	0,87760

Cuadro 6: Valores medios de los cuatro algoritmos en el problema del APC.

Como se puede ver a simple vista, partiendo de la búsqueda local como base, inicializar el vector W con los valores obtenidos por el algoritmo *RELIEF* presenta ciertas mejoras en algunos datos, como por ejemplo en **Colposcopy** y en **Ionosphere**, llegando a superar en el primer caso a *RELIEF* y quedándose muy cerca de hacerlo en el segundo. Sin embargo, a estas mejoras también le deben seguir algunos empeoramientos. Por ejemplo, en el conjunto de datos **Texture** se queda muy lejos de los tres algoritmos, siendo el peor detodos. Además, en todos los casos la desviación típica obtenida es mayor que la del resto de algoritmos, con lo cuál la variabilidad de los resultados será mucho mayor que en los otros casos.

En el siguiente *boxplot* se puede ver como son los resultados para cada conjunto de datos:

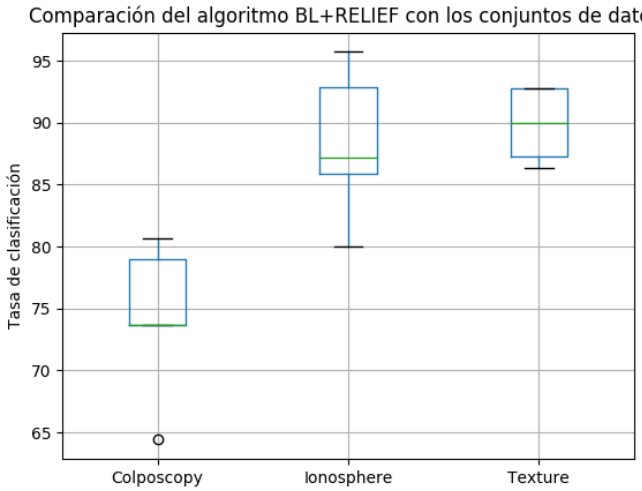
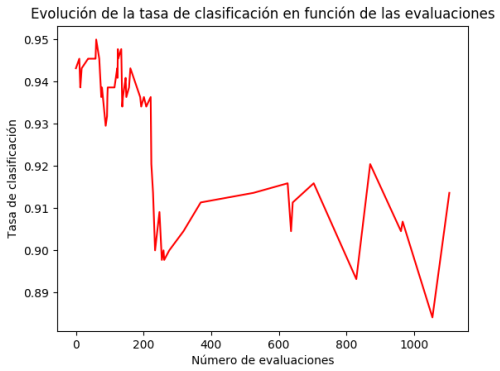
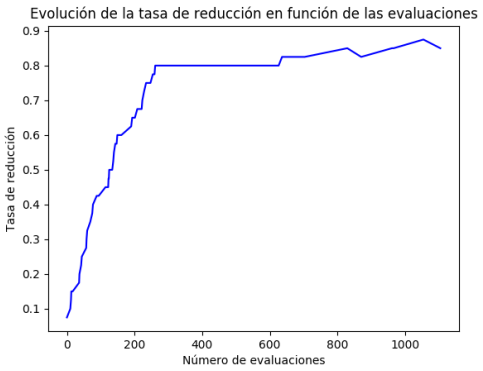
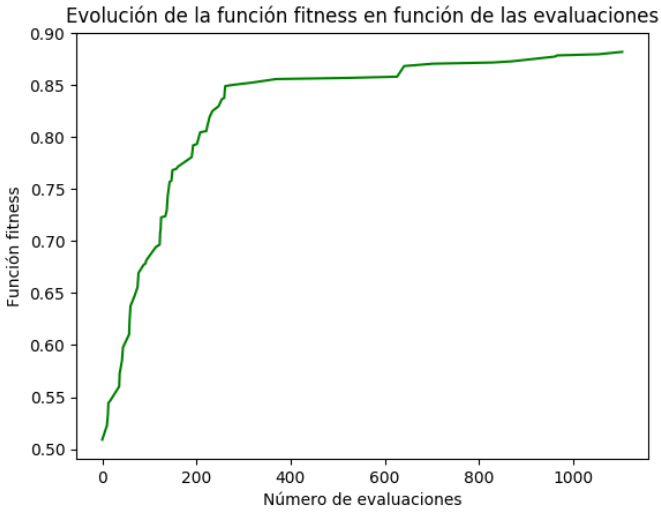


Figura 5: *Boxplot* de los distintos datos con BL+*RELIEF*.

Y en cuanto al proceso de búsqueda local, cogiendo de nuevo la Partición 3 del conjunto **Texture**, obtenemos las siguientes gráficas de la evolución de las tasas de clasificación y reducción y del valor de la función *fitness* a medida que se van realizando evaluaciones de la función objetivo:







Como se puede ver, esta vez la tasa de clasificación comienza en un valor muy elevado, pero a medida que van pasando evaluaciones de la función objetivo, se puede ver que su valor va bajando, siendo el motivo seguramente que muchas características se veían reducidas. Esto se debe, de nuevo, a que es más fácil mejorar la función objetivo reduciendo el número de características que mejorando la tasa de clasificación como tal. Comparando la tasa de clasificación con la tasa de reducción y la agrupación, se puede ver muy claramente como estas dos crecen de una forma mucho más natural, por el motivo comentado anteriormente.

De este pequeño estudio podemos concluir que combinar las dos técnicas nos ha permitido mejorar las tasas de clasificación en algunos conjuntos de datos, con el precio de empeorar mucho en otros y de aumentar la variabilidad de los resultados que se obtienen, ya que las desviaciones típicas eran muy elevadas. Por tanto, aunque a corto plazo parece que se haya mejorado, mirando los resultados obtenidos con una perspectiva más amplia podemos concluir que no merece la pena combinar estas dos técnicas esperando conseguir algún tipo de resultado milagroso que resuelva todos los casos con resultados casi perfectos. Además, nada garantiza que el W obtenido por el algoritmo *RELIEF* no sea un óptimo local, y que intentar mejorar a partir de este sea imposible, cosa que no pasaría si se iniciase desde un punto aleatorio. Por tanto, es mejor decantarse por alguna de las técnicas estudiadas anteriormente.

5.4. Conclusión

Como conlcusión de este análisis, podemos decir que en los casos estudiados, y de momento, el mejor algoritmo es *RELIEF*, ya que es el que ofrece mejores tasas de clasificación con unas desviaciones típicas bastante aceptables (no son tan buenas como las del 1-NN, pero son mucho mejores que las de la búsqueda local). La búsqueda local ha demostrado que no es la mejor solución en este caso, pero como se ha dicho anteriormente, a lo mejor disponiendo de más datos ésta hubiese mostrado otra tendencia. Asi que por el momento, nos quedamos con *RELIEF* hasta que encontremos una técnica mejor.

Referencias

[1] Repositroio de GitHub de pykdtree.
`https://github.com/storpipfugl/pykdtree`

[2] Documentación de cKDTree.
`https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.cKDTree.html`

[3] Información sobre la estructura de datos KDTree.
`https://en.wikipedia.org/wiki/K-d_tree`

[4] Documentación del StratifiedKFold para dividir las muestras.
`https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html`

[5] Documentación del clasificador KNeighborsClassifier.
`https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html`