Universidad de Granada

Práctica 1.b

Técnicas de Búsquedad Local y Algoritmos Greedy para el Problema del aprendizaje de Pesos en Características

Metaheurísticas

Grupo 2 (Lunes)

Pedro Ramos Suárez

76591270M

pedrors@correo.ugr.es

Doble Grado de Ingeniería Informática y Matemáticas

4 de abril de 2022

Índice

1. Problema de Aprendizaje de Pesos en Características

1.1. Descripción del problema

El problema consiste en optimizar el rendimiento de un clasificador basado en el vecino más cercano a partir de la inclusión de pesos asociados a las características del problema.

El clasificador utilizado es el 1-NN (1 vecino más cercano), el cuál consiste en asignar a cada nuevo la misma etiqueta que a su vecino más cercano. Para ello, utilizamos la distancia euclídea modificada por unos pesos asociados a cada característica:

$$d(e_1, e_2) = (\sum_{i=1}^{n} w_i \cdot (e_1^i - e_2^i)^2)^{\frac{1}{2}}$$

donde n es el número de características de los datos, y $W = (w_1, \ldots, w_n)$ el vector de números reales entre 0 y 1 que define el peso que pondera cada una de las características.

La variante que intentaremos optimizar será la agregación, que combina tanto la precisión como la complejidad del clasificador, definida como:

$$\text{Agregación}(W) = \alpha \cdot \text{Tasa\_clas}(W) + (1 - \alpha) \cdot \text{Tasa\_red}(W) \tag{1}$$

donde:

$$\text{Tasa\_clas} = 100 \cdot \frac{\text{nº de instancias bien clasificadas en T}}{\text{nº de instancias en T}} \tag{2}$$

$$\text{Tasa\_red} = 100 \cdot \frac{\text{nº de valores } w_i < 0{,}1}{\text{nº de características}} \tag{3}$$

siendo T el conjunto de datos sobre los que se evalúa el clasificador, y α es un número real entre 0 y 1 que pondera la importancia entre acierto y la reducción de la solución.

1.2. Datos utilizados

Los datos utilizados serán:

1. **Ionosphere**: Conjunto de datos de radar, formado por 351 ejemplos con 34 características clasificados en dos clases.
2. **Parkinsons**: Conjunto de datos orientados a distinguir entre la presencia y ausencia de la enfermedad de Parkinson en una serie de pacientes, formado por 195 ejemplos con 22 características clasificados en dos clases.
3. **Spectf-heart**: Conjunto de datos de detección de enfermedades cardíacas a partir de imágenes médicas de termografía computerizada del corazón, formada por 349 ejemplos con 44 características clasificados en dos clases.

2. Aplicación de los algoritmos

Todos los algoritmos utilizados para resolver este problema tienen como entrada la matriz de datos de las características $D = (k \times n)$, y el vector de etiquetas asociadas a dicha matriz $y = (y_1, \ldots, y_k)$, donde k es el número de ejemplos, y n el número de características. Dependiendo del algoritmo, estos datos pueden ser todos los leídos, o sólo un subconjunto de entrenamiento.

Para representar las soluciones, utilizaremos un vector de números reales entre 0 y 1, $W = (w_1, ;w_n)$, que define el peso que pondera cada una de las características.

La evaluación de la calidad de una solución se hará como indicada en (1), tomando como $\alpha = 0{,}5$, es decir, dándole la misma importancia al acierto y a la reducción de las características, por lo que será de la forma:

$$\text{Agregación}(W) = 0{,}5 \cdot \text{Tasa\_clas}(W) + 0{,}5 \cdot \text{Tasa\_red}(W) = \frac{\text{Tasa\_clas}(W) + \text{Tasa\_red}(W)}{2}$$

2.1. Algoritmos para evaluar las soluciones

El pseudocódigo de la función para calcular la tasa de clasificación es:

Algorithm 1: TASA_CLAS calcula la tasa de clasificación de una solución.

Input: El conjunto de etiquetas reales Y.
Input: El conjunto de etiquetas obtenidas de la predicción del vecino más cercano P.
Output: La tasa de clasificación como se describe en (2).

```
clas ← 0 ;
for i = 0 to length(Y) do
    if Y[i] = P[i] then
        clas ← clas + 1 ;
    end
end
output ← clas/length(Y) ;
return output ;
```

El pseudocódigo de la función para calcular la tasa de reducción es:

Algorithm 2: TASA_RED calcula la tasa de reducción de una solución.

Input: El conjunto de pesos W.
Input: El conjunto de etiquetas obtenidas de la predicción del vecino más cercano P.
Output: La tasa de clasificación como se describe en (3).

```
red ← 0 ;
for w in W do
    if w < 0,1 then
        red ← red + 1 ;
    end
end
output ← red/length(W) ;
return output ;
```

Por último, el pseudocódigo de la función de agregación obtenido a partir de la tasa de clasificación y la tasa de reducción es:

Algorithm 3: AGREGACION calcula la agregación de una solución.

Input: El conjunto de pesos W.
Output: La agregación como se describe en (1).

```
clas ← tasa_clas(W) ;
red ← tasa_red(W) ;
output ← (clas + red)/2 ;
return output ;
```

2.2. Algoritmos para calcular las distancias

Utilizaremos la distancia euclídea o una variante que modifica la importancia de cada componente según los pesos.

El pseudocódigo para la distancia euclídea es:

Algorithm 4: EULCIDEAN calcula la distancia entre dos puntos.

Input: Un vector con las coordenadas del primer punto P.
Input: Un vector con las coordenadas del segundo punto Q.
Output: La distancia entre los dos puntos.
$dist \leftarrow 0$;
for $i = 0$ ***to*** $length(P)$ **do**
  $dist \leftarrow dist + (P[i] - Q[i])^2$;
end
$output \leftarrow sqrt(dist)$;
return *output* ;

El pseudocódigo para la distancia euclídea modificada por el vector de pesos es:

Algorithm 5: WEIGHTED_EULCIDEAN calcula la distancia entre dos puntos.

Input: Un vector de pesos W.
Input: Un vector con las coordenadas del primer punto P.
Input: Un vector con las coordenadas del segundo punto Q.
Output: La distancia entre los dos puntos.
$dist \leftarrow 0$;
for $i = 0$ ***to*** $length(W)$ **do**
  $dist \leftarrow dist + W[i](P[i] - Q[i])^2$;
end
$output \leftarrow sqrt(dist)$;
return *output* ;

2.3. Algoritmos para predecir las etiquetas

Tenemos dos casos:

1. El conjunto de datos está dividido en entrenamiento y test. Entonces predeciremos las etiquetas del conjunto de test a partir de los resultados obtenidos en entrenamiento.
2. Los datos no están divididos en entrenamiento y test. Entonces aplicaremos el algoritmo *leave-one-out*, que predice una etiqueta a partir de todos los demás datos.

Para el pseudocódigo utilizaremos la versión de la distancia euclídea con pesos. Sin embargo, el caso que no los utiliza sería casi idéntico, pero sin tomar el vector de pesos como entrada, y utilizando la función *euclidean* en lugar de *weighted_euclidean*.

El pseudocódigo para el primer caso, en el que tenemos los datos separados en entrenamiento y test, es:

Algorithm 6: PREDICT_LABEL predice las etiquetas.

Input: Una matriz de datos de test X_test.
Input: Una matriz de datos de entrenamiento X.
Input: Un vector con las etiquetas de entrenamiento Y.
Input: El vector de pesos W.
Output: Un vector con la predicción de etiquetas de los datos de test.

```
Y_pred ← {} ;
for x in X_test do
    min_dist ← length(x) ;
    neighbour ← 0 ;
    for i = 0 to length(X) do
        dist ← weighted_euclidean(x, X[i], W) ;
        if dist < min_dist then
            min_dist ← dist ;
            neightbour ← i ;
        end
    end
    Y_pred ← Y_pred ∪ {neighbour} ;
end
return Y_pred ;
```

Nótese que inicializamos la distancia mínima como el la longitud de x, es decir, el número de características. Esto se debe a que como los datos están normalizados, la distancia siempre será menor.

El pseudocódigo para el segundo caso, *leave-one-out*, en el que tenemos un único conjunto de datos, es:

Algorithm 7: LEAVE_ONE_OUT predice las etiquetas.

Input: Una matriz de datos X.
Input: Un vector con las etiquetas Y.
Input: El vector de pesos W.
Output: Un vector con la predicción de etiquetas.

```
Y_pred ← {} ;
for x in X do
    min_dist ← length(x) ;
    neighbour ← 0 ;
    for i = 0 to length(X) do
        if x ≠ X[i] then
            dist ← weighted_euclidean(x, X[i], W) ;
            if dist < min_dist then
                min_dist ← dist ;
                neightbour ← i ;
            end
        end
    end
    Y_pred ← Y_pred ∪ {neighbour} ;
end
return Y_pred ;
```

2.4. Otros algoritmos

Por último, sólo nos queda un algoritmo que merece la pena mencionar, que es el que separa los datos en entrenamiento:

Algorithm 8: TRAIN_TEST_SPLIT divide los datos en entrenamiento y test.

Input: Una matriz de datos X.
Input: Un vector con las etiquetas Y.
Input: Un entero que indica el número de conjuntos en el que dividir los datos n.
Input: Un entero que indica qué conjunto es de test y cuáles de entrenamiento k.
Output: Un vector con los datos de entrenamiento.
Output: Un vector con los datos de test.
Output: Un vector con las etiquetas de entrenamiento.
Output: Un vector con las etiquetas de test.

```
X_train ← {} ;
X_test ← {} ;
Y_train ← {} ;
Y_test ← {} ;
size ← length(X)/n ;
remain ← length(X) %n ;
if k < remain then
    begin ← (size + 1) ;
    end ← begin + (size + 1) ;
else if k == remain then
    begin ← (size + 1) ;
    end ← begin + size ;
else
    begin ← size ;
    end ← begin + size ;
end
for i = 0 to length(X) do
    if i < begin or i > begin then
        X_train ← X_train ∪ X[i] ;
        Y_train ← Y_train ∪ Y[i] ;
    else
        X_test ← X_test ∪ X[i] ;
        Y_test ← Y_test ∪ Y[i] ;
    end
end
return X_train ;
return X_test ;
return Y_train ;
return Y_test ;
```

En caso de que el número de datos sea múltiplo del número de particiones, todas estas tendrán el mismo tamaño. En caso de que no lo sea, los primeros conjuntos de test tendrán un elemento extra.

3. Descripción de los algoritmos

3.1. Búsqueda Local

Procedemos al algoritmo de Búsqueda Local, el objetivo de esta práctica. Generamos nuevas soluciones modificando de manera aleatoria el vector de pesos. Debido a esto, hay infinitas posibles nuevas soluciones, por lo que utilizamos la técnica del Primero Mejor, con la cuál tomamos una nueva solución si es mejor que la actual.

Iniciaremos el algoritmo con una solución aleatoria generando cada componente a partir de una distribución uniforme entre 0 y 1, utilizando la función:

Algorithm 9: RANDOM_SOL genera una solución aleatoria.

Input: El tamaño del vector de pesos n.
Output: Un vector con una solución aleatoria.
$W \leftarrow \{\}$;
for $i = 0$ ***to*** $length(X[0])$ **do**
    $e \leftarrow$ elemento aleatorio de una distribución uniforme ;
    if $e < 0{,}1$ **then**
        $e \leftarrow 0$;
    end
    $W \leftarrow W \cup \{e\}$;
end
return W ;

En cada paso de la exploración modificamos una componente del vector de pesos distinta sin repetición hasta encontrar mejora (técnica del Primero Mejor) o hasta modificar todas las componentes sin conseguir una mejora, momento en el cual se comienza de nuevo la exploración. Para ello, necesitamos una función que nos de el orden en el que aplicar las modificaciones de forma aleatoria:

Algorithm 10: PERMUTATION genera una permutación.

Input: El tamaño del vector de pesos n.
Output: Un vector con la permutación.
$perm \leftarrow \{0, \ldots, n-1\}$;
$perm \leftarrow \text{shuffle}(perm)$;
return $perm$;

Continuaremos generando candidatos hasta realizar un total de 15000 iteraciones (contando como iteración cada vez que modificamos una componente del vector de pesos), o 20 iteraciones sin obtener una solución mejor (contando como iteración modificar todas las componentes del vector de pesos).

Cabe mencionar que para obtener el número de características utilizamos *length(X[0])*, ya que, a menos que usemos un conjunto vacío (caso que no tiene sentido), siempre tendremos un elemento en esa posición.

Con todo esto, el algoritmo de búsqueda local queda de la forma:

Algorithm 11: LocalSearch

Input: Una matriz de datos X.
Input: Un vector de etiquetas Y.
Output: Un vector de pesos.

```
iteration ← 0 ;
iteration_mod ← 0 ;
W ← random_sol(length(X[0])) ;
while iteration < 15000 and iteration_mod < 20 do
    modified ← false ;
    perm ← permutation(length(W)) ;
    for i == 0 to length(W) and not modified and iteration < 15000 do
        s ← elemento aleatorio de una distribución normal de media 0 y varianza 0.3 ;
        neighbour ← copy(W) ;
        neighbour[i] ← neighbour[i] + s ;
        if neighbour[i] > 1 then
            neighbour[i] ← 1 ;
        else if neighbour[i] < 0,1 then
            neighbour[i] ← 0 ;
        end
        if agregation(neighbour) > agregation(W) then
            W ← neighbour ;
            modified ← true ;
        end
        iteration ← iteration + 1 ;
    end
    if modified then
        iteration_mod ← 0 ;
    else
        iteration_mod ← iteration_mod + 1 ;
    end
end
return W ;
```

4. Algoritmos de comparación

4.1. 1-NN

El primer algoritmo que utilizaremos para comparar la eficacia del algoritmo es *1-NN*, que es el algoritmo **k-NN** (k nearest neighbours) con $k = 1$, es decir, con un sólo vecino.

Los demás algoritmos son una versión modificada de este, en los que modificamos la distancia euclídea con un vector de pesos. Debido a ello, como no tenemos vector de pesos, no estamos realizando ninguna optimización, y los resultados solo dependen de los datos de entrada, por lo que no tiene sentido dividir los datos en entrenamiento y test (ya que no "entrenamos"), así que usamos la técnica *leave-one-out*.

El funcionamiento del algoritmo ya ha sido explicado en el algoritmo 7.

4.2. Greedy: RELIEF

El principal algoritmo que utilizaremos para comparar la eficacia es el *greedy* **RELIEF**, que genera un vector de pesos a partir de las distancias de cada ejemplo a su *enemigo*, que es el ejemplo de diferente clase más cercano, y a su *amigo* más cercano, que es el ejemplo de la misma clase más cercano.

El algoritmo aumentará el peso de las características que mejor separan a ejemplos enemigos, y reduce el peso en las que separan a amigos. El incremento es proporcional a la distancia entre los ejemplos en cada característica.

Algorithm 12: RELIEF

Input: Una matriz de datos X.
Input: Un vector de etiquetas Y.
Output: Un vector de pesos.
$W \leftarrow \{0, 0, \ldots, 0\}$ `// Tantos ceros como carcaterísticas tengan los datos.`
for $i = 0$ ***to*** $length(X)$ **do**
  $ind\_enemy \leftarrow i$;
  $ind\_friend \leftarrow i$;
  $dist\_enemy \leftarrow length(X[0])$ `// Al normalizar los datos, la distancia máxima`
  $dist\_friend \leftarrow length(X[0])$ `// es` $\sqrt{k} < k$`, siendo` k `el número de características`
  for $j = 0$ ***to*** $length(X)$ **do**
    if $i \neq j$ **then**
      if $X[i] == X[j]$ ***and*** weighted_euclidean$(X[i], X[j], W) < dist_friend$ **then**
        $ind\_friend \leftarrow j$;
        $dist\_friend \leftarrow$ weighted_euclidean$(X[i], X[j], W)$;
      else if $X[i] \neq X[j]$ ***and*** weighted_euclidean$(X[i], X[j], W) < dist_enemy$ **then**
        $ind\_enemy \leftarrow j$;
        $dist\_enemy \leftarrow$ weighted_euclidean$(X[i], X[j], W)$;
      end
    end
  end
  for $j = 0$ ***to*** $length(X[0])$ **do**
    $w[j] \leftarrow |X[i][j] - x[ind\_enemy][j]| - |X[i][j] - X[ind\_friend][j]|$;
  end
end
$W \leftarrow$ normalize_weight(W) ;
return W ;

donde *normalize_weight* es la función que normaliza los valores del vector de pesos, haciendo que estén entre 0 y 1, y aplica la reducción, eliminando aquellos valores que están por debajo de 0.1, es decir:

Algorithm 13: NORMALIZE_WEIGHT normaliza y aplica reducción al vector de pesos.

Input: El vector de pesos W.
Output: El vector de pesos normalizado.

```
max_w ← w[0] ;
for k in W do
    if k > max_w then
        max_w ← w ;
    end
end
for k in W do
    if k < 0,1 then
        k ← 0 ;
    else
        k ← k/max_w ;
    end
end
return W ;
```

5. Desarrollo de la práctica

La práctica ha sido implementada en C++, utilizando las siguientes bibliotecas:

- **iostream**: Para salida de datos por pantalla.
- **fstream**: Para lectura y escritura en ficheros.
- **sstream**: Para lectura y escritura en ficheros.
- **string**: Para lectura y escritura en ficheros.
- **vector**: Para almacenar los datos en un vector de vectores (que funciona como una matriz), y las etiquetas y los pesos en un vector.
- **math.h**: Para el calculo de la raíz cuadrada para la distancia euclídea.
- **stdlib.h**: Para la generación de distribuciones aleatorios.
- **random**: Para la generación de aleatorios.
- **chrono**: Para calcular el tiempo de ejecución utilizando *system_clock*.
- **regex**: Para modificar strings. Utilizado únicamente en la búsqueda local para poder exportar datos y representarlos en gráficas.

Los ficheros están almacenados en los siguientes directorios:

- **bin**: Contiene los archivos ejecutables.
- **src**: Contiene los archivos con el código.
- **data**: Contiene los ficheros con los datos.
- **graph**: Contiene un archivo de *python*, y dos carpetas. En la primera de ellas, *data*, se almacenan datos para representarlos. El archivo *graphPlotter.py* lee todos los archivos que haya en este carpeta, y crea sus gráficas, guardándolos en la segunda carpeta, *plots*. Esto está más desarrollado en el siguiente apartado.

Dentro de los ficheros, tenemos las constantes:

- **SEED**: Usado en los tres algoritmos. Contiene la semilla usada para las generaciones aleatorias. Por defecto, 0.
- **FOLDS**: Usado en *knn* y *localSearch*. Contiene el número de particiones en las que dividir el conjunto de datos para entrenamiento y test usando *k-folds cross validation*. Tomará uno de estos conjuntos como test, y el resto como entrenamiento. Por defecto, 5.
- **REDUCCTION**: Usado en *knn* y *localSearch*. Contiene el umbral por debajo del cual los pesos se consideran como 0. Por defecto, 0.1.
- **EXPORT**: Usado exclusivamente en *localSearch*. Determina si se almacenan datos para su posterior representación en gráficas (explicado más en detalle en el siguiente apartado). Por defecto, para reducir el tiempo de ejecución, no almacena datos, esto es, inicializamos a *false*.

5.1. Representación en gráficas

Un resultado interesante que se puede observar es como evolucionan la tasa de clasificación, reducción y agregación con las iteraciones en el algoritmo de búsqueda local. Sin embargo, en C++ no existe ninguna función o biblioteca que facilite la representación en gráficas.

Para solucionar esto, definimos una nueva función llamada *export_localSearch*, que tiene el mismo funcionamiento que *localSearch*, pero cada vez que se encuentra una mejor solución, almacena la iteración y las tasas en un fichero en el directorio *graph*$\,$*data*. Esta definida como una nueva función ya que, si añadiésemos comprobaciones o salida a un fichero, aumentaríamos el tiempo de ejecución.

En el directorio *graph*, tenemos además un archivo llamado *graphPlotter.py*, que es un archivo en *Python* que lee todos los ficheros de *graph_data*, crea las gráficas usando *matplotlib*, y las almacena como imágenes en *graph_plots*.

Con esto, podemos tener gráficas como la siguiente:

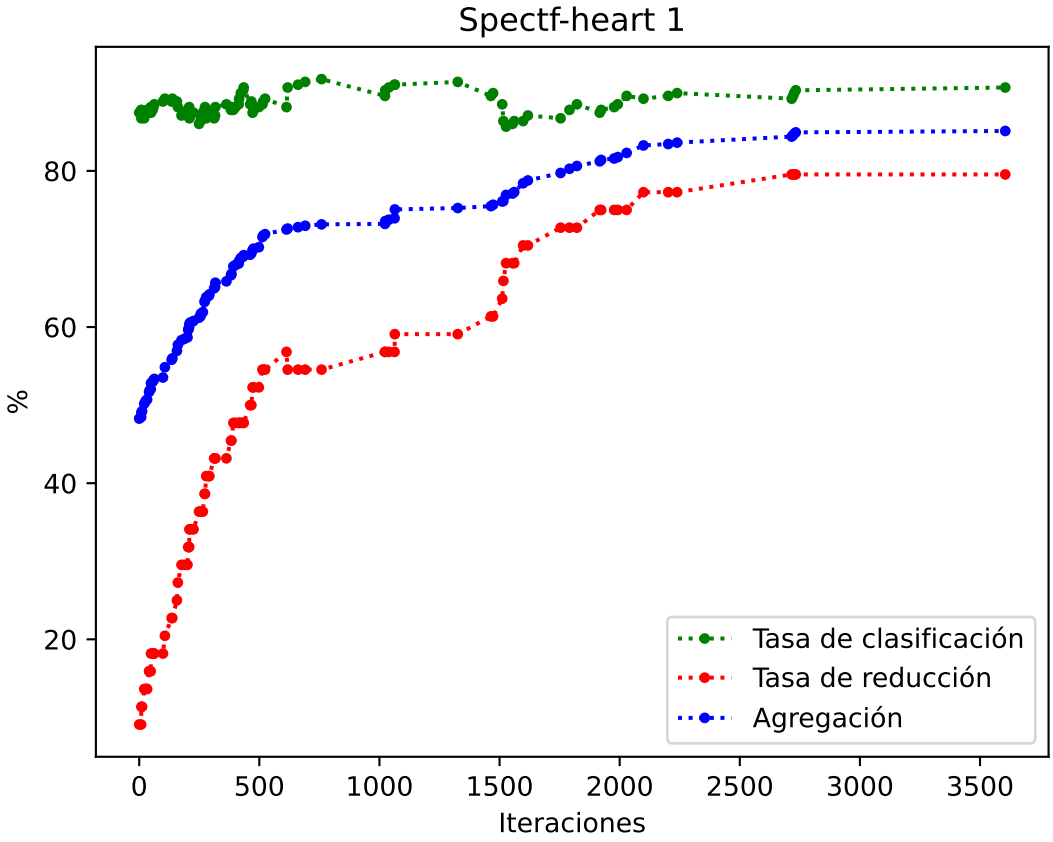

Figura 1: Ejemplo de gráfica obtenida

Nótese que esta gráfica es un suavizado de la gráfica real, ya que en la gráfica real las tasas no cambiaría hasta encontrar una mejor solución, momento en el que ocurriría un salto, por lo que tendríamos funciones a saltos, en lugar de continuas. Sin embargo, para una "mejor" representación, las tratamos como funciones continuas.

Todas las gráficas, incluido este ejemplo, están incluidas en la sección de resultados.

Para eliminar todos los datos y las gráficas obtenidas, podemos ejecutar *make fullclean*, que elimina tanto los ejecutables como los datos en las carpetas de *graph*.

5.2. Manual de usuario

Para obtener los ficheros ejecutables, sólo tenemos que realizar el comando *make*, el cuál compila el código fuente usando optimización *-O2*.

Los ficheros ejecutables son:

- **knn**: Implementación del algoritmo *k-NN*, con $k = 1$.
- **Greedy**: Implementación del algoritmo greedy *RELIEF*.
- **LocalSearch**: Implementación del algoritmo Búsqueda Local.

Los archivos de código contienen una explicación de todas las funciones, así como de los parámetros que toman de entrada y de salida. Además, al comienzo de cada uno de ellos, están definidas las constantes en caso de que se deseen modificar.

Toda la parte de generación de gráficas crea los archivos necesarios automáticamente, y con *make fullclean*, todos son eliminados (excepto el archivo de *Python*).

Todos requieren el uso de un fichero de entrada, e imprimen por pantalla los resultados obtenidos. En el caso de *RELIEF* y la Búsqueda Local, como separamos los datos en entrenamiento y test, imprimen los resultados para cada uno de los conjuntos de test.

Por ejemplo, si queremos ejecutar el algoritmo *localSearch* con los datos de *ionosphere.arff*, sería:

```
bin/localSearch data/ionosphere.arff
```

6. Experimentos y análisis de resultados

Todos los resultados se han obtenido en un ordenador con las siguientes especificaciones:

- Sistema operativo: macOS Montery.
- Procesador: 1,6 GHz Intel Core i5 de doble núcleo.
- Memoria: 16 GB 2133 MHz LPDDR3.
- Gráficos: Intel UHD Graphics 617 1536 MB.

6.1. Resultados

En todos los resultados, el tiempo es en segundos.

Los resultados obtenidos con el algoritmo *greedy RELIEF* son:

	Ozone				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
Partición 1	90.14	2.94	46.54	0.02	94.87	0.00	47.44	0.01	77.14	0.00	38.57	0.02
Partición 2	70.00	2.94	36.47	0.02	100.00	0.00	50.00	0.01	84.29	0.00	42.14	0.02
Partición 3	67.14	2.94	35.04	0.02	94.87	0.00	47.44	0.01	77.14	0.00	38.57	0.02
Partición 4	68.57	2.94	35.76	0.02	97.44	0.00	48.72	0.01	92.86	0.00	46.43	0.02
Partición 5	67.14	2.94	35.04	0.02	94.87	0.00	47.44	0.01	86.96	0.00	43.48	0.02
Media	72.60	2.94	37.77	0.02	96.41	0.00	48.21	0.01	83.68	0.00	41.84	0.02

Tabla 1: Tabla con los resultados del algoritmo greedy RELIEF.

Los resultados obtenidos con el algoritmo *localSearch* son:

	Ozone				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
Partición 1	92.96	88.24	90.60	25.71	87.18	83.36	86.77	6.61	74.29	79.55	76.92	59.97
Partición 2	88.67	79.41	84.04	38.19	89.74	68.18	78.96	4.59	85.71	81.82	83.77	80.71
Partición 3	84.29	88.24	86.26	45.87	89.75	81.82	85.78	3.86	80.00	72.73	76.36	41.19
Partición 4	78.57	61.76	70.17	27.53	92.31	72.73	82.52	2.97	87.14	79.55	83.34	37.09
Partición 5	80.00	85.29	82.65	24.81	87.18	86.36	86.77	5.44	82.61	75.00	78.80	42.40
Media	84.90	80.59	82.74	32.42	89.23	79.09	84.16	4.69	81.95	77.73	79.84	52.27

Tabla 2: Tabla con los resultados del algoritmo localSearch.

Finalmente, los resultados globales son:

	Ozone				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
1-NN	86.89	0.00	43.45	0.02	96.41	0.00	48.21	0.00	86.25	0.00	43.12	0.02
RELIEF	72.60	2.94	37.77	0.02	96.41	0.00	48.21	0.01	83.68	0.00	41.84	0.02
BL	84.90	80.59	82.74	32.42	89.23	79.09	84.16	4.69	81.95	77.73	79.84	52.27

Tabla 3: Tabla con los resultados globales en el problema APC.

Hay que tener en cuenta que en los algoritmos *RELIEF* y *localSearch*, el tiempo es por cada partición, es decir, por cada cuatro quintos de los datos (los conjuntos de entrenamiento), mientras que el tiempo en el algoritmo *1-NN* es con el total de los datos.

6.2. Gráficas del algoritmo localSearch

Como explicado en la sección 5.1, es interesante ver como evolucionan la tasa de clasificación, reducción y agregación con las iteraciones en el algoritmo *localSearch*. En esta sección añado todas las gráficas obtenidas.

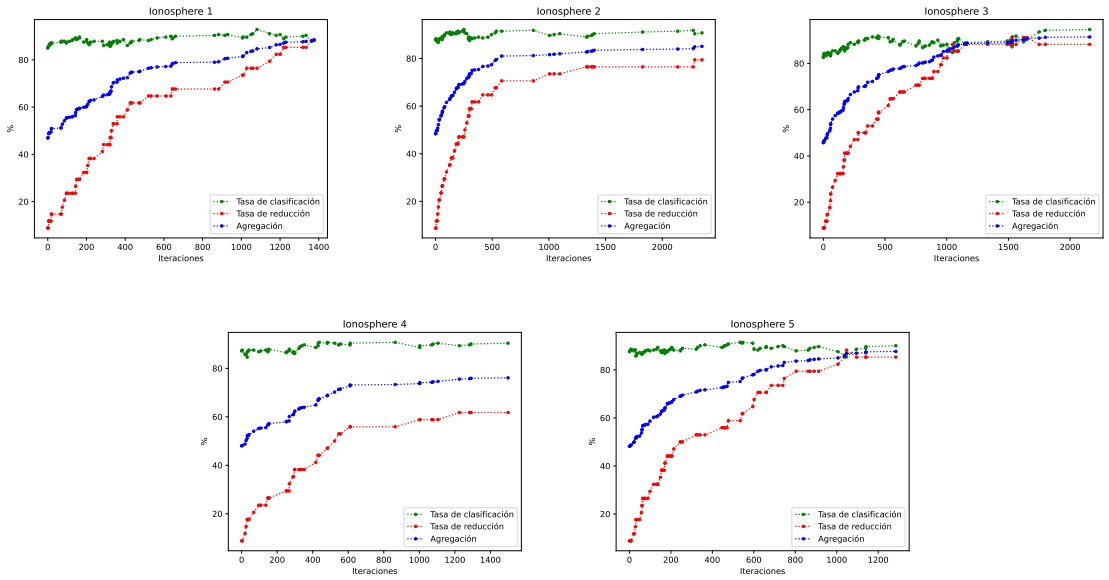

Figura 2: Gráficas de la tasa de clasificación, reducción y agregación respecto al número de iteraciones con los datos de train de Ionosphere

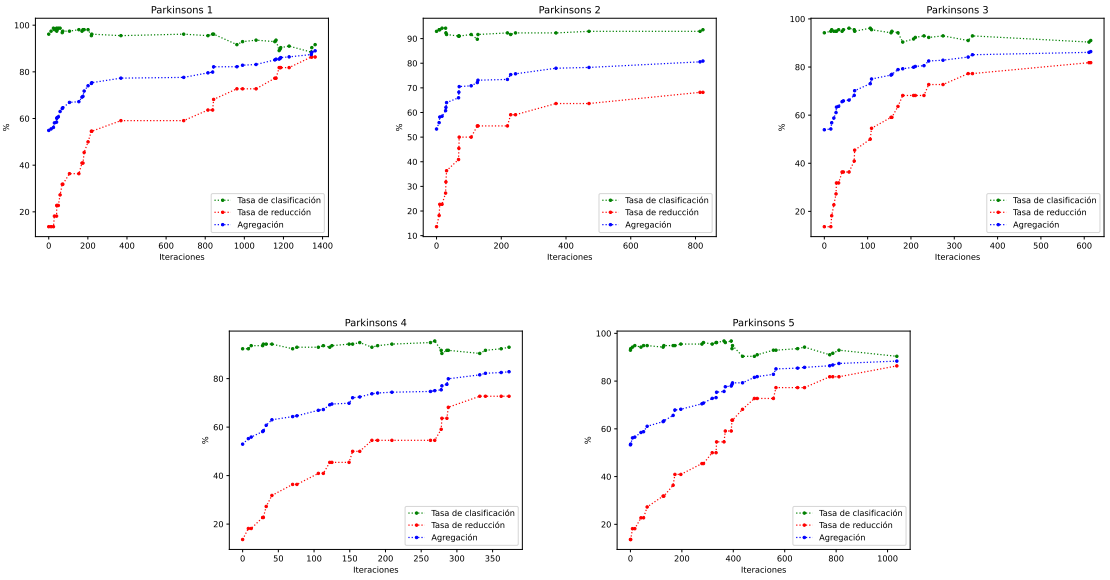

Figura 3: Gráficas de la tasa de clasificación, reducción y agregación respecto al número de iteraciones con los datos de train de Parkinsons

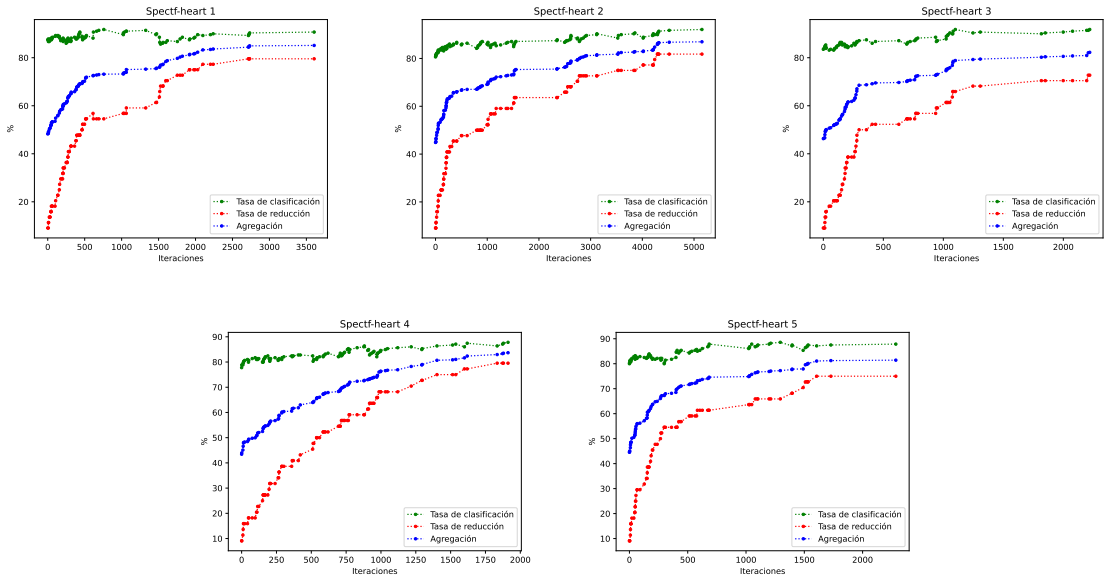

Figura 4: Gráficas de la tasa de clasificación, reducción y agregación respecto al número de iteraciones con los datos de train de Spectf-heart

En todas las gráficas podemos observar los mismos resultados:

- Inicialmente, la tasa de reducción comienza sobre el 10 %. Esto se debe a que los pesos iniciales siguen una distribución uniforme entre 0 y 1, y eliminamos aquellos que están por debajo de 0.1 por la reducción, los cuáles serán aproximadamente el 10 %.
- La tasa de evaluación comienza por encima del 50 %, aunque varía en cada caso. Eso se debe a que, aunque esta modificado por unos pesos aleatorios, se sigue usando el algoritmo *1-NN*, por lo que los resultados serán mejor que si los tomásemos completamente al azar (en cuyo caso estarían sobre el 50 % de media).
- Según aumentan las iteraciones, podemos ver un gran aumento en la tasa de reducción. Esto se debe a que, al empezar cerca del 10 %, tiene un gran margen de mejora, mientras que la tasa de clasificación, al empezar con valores más altos, es más difícil que mejore. Por lo que es mucho más probable una mejora en la tasa de reducción que en la tasa de clasificación.
- Al aumentar las iteraciones, también vemos como disminuye la pendiente de las gráficas. Esto se debe a que cada vez tenemos mejores soluciones, y es más difícil encontrar una mejora. Inicialmente, hay muchas soluciones mejores, mientras que al final, hay muchas menos al estar cerca del máximo local. Además, no sólo mejora menos en cada paso, si no que es más difícil encontrar una mejor solución, por lo que aumenta el número de iteraciones entre cada punto.
- Obviamente, al ser tomar $\alpha = 0{,}5$, la agragación funciona como una media entre la tasa de clasificación y la tasa de reducción, por lo que siempre se encuentra en el punto medio.

6.3. Análisis de los resultados

Todo este análisis se hace a partir de la tabla 6.1.

El algoritmo *1-NN*, como cabe esperar, es con diferencia el más rápido de los tres. Sin embargo, los resultados obtenidos no son muy buenos, principalmente porque no realiza ninguna reducción.

El algoritmo *greedy RELIEF*, aunque siendo mucho más rápido que el algoritmo *localSearch*, es algo más lento que el algoritmo *1-NN*, ya que obtiene tiempos similares con un quinto menos de datos. Sin embargo, los resultados no son mucho mejores, e incluso son peores en algunos casos. Debido a esto, no es un algoritmo que tenga mucho sentido usar en nuestro problema.

El algoritmo *localSearch*, aunque aumenta mucho los tiempos de ejecución, es el algoritmo que da mejores resultados con diferencia. Además, podemos ver como el tiempo crece muy rápido al aumentar las características: teniendo casi el mismo número de ejemplos, pasamos de aproximadamente 32 segundos en *ionosphere.arff* con 34 características a cerca de 52 en *spectf-heart* con 44. Sin embargo, gran parte de estos buenos resultados se debe a la gran reducción que realiza. Debido a esto, hay que tener cuidados de que no produzca un "sobreajuste" eliminando demasiadas características.

En resumen, si lo que nos interesa es un algoritmo que tarde poco tiempo, es mejor usar el algoritmo *1-NN*, pero si lo que nos interesa es obtener buenos resultados sin importar el tiempo de ejecución, es mejor el algoritmo *localSeach*.