Metaheurísticas

Departamento de Ciencias de la Computación e Inteligencia Artificial

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Práctica 1: Técnicas de Búsqueda Local y Algoritmos Greedy. Problema de Aprendizaje de Pesos en Características

Álvaro Rodríguez Gallardo. 77034155W.

alvaro155w@correo.ugr.es

Grupo 2 (17.30-19.30)

Curso 2023-2024

Índice general

Descripción del problema

El problema afrontado, Aprendizaje de Pesos en Características, consiste en obtener un vector de pesos que permita ponderar aquellas características de una muestra, intentando optimizar el porcentaje de clasificación para futuras muestras con mismas características. Si se tiene que la muestra tiene $n \in \mathbb{N}$ características, entonces el vector $c_i = (c_1, ..., c_n)$,$0 \leq c_i \leq 1$, representa las características de la i-ésima componente del dato $X = (x_1, ..., x_m)$, con $m \in \mathbb{N}$ y $1 \leq i \leq m$. Además, cada componente tendrá asociada una clase, es decir, para un vector de datos como X, para la componente i, que se llamará "muestra i-ésima", se le asociará un vector de características y una clase. Fijado X anterior, se define para cierto conjunto A de clases

$$p : \mathbb{R} \to \mathbb{R}^n \times A$$

$$p(x_i) = (c_i, \text{clase}_i)$$

para $1 \leq i \leq m$ y $x_i \in X = (x_1, ..., x_m) \subset \mathbb{R}^m$.

Con lo anterior, se quiere encontrar un vector de pesos $W = (w_1, ..., w_n)$, con el peso j-ésimo ponderando a la característica j-ésima del vector de características. Para ello, se usará un clasificador k-NN, pero en el caso k= 1, con el que se considera el vecino más cercano (muestra con menor distancia euclídea).

Haciendo uso de 5-fold cross validation, se buscará tanto encontrar el vector de pesos correspondiente como evaluar la calidad del mismo (primero aprendizaje, luego validación). Puesto que el dataset se divide en cinco particiones, se separa una partición para validación y se usa el resto de entrenamiento (todas deben ser para validación en algún momento). A continuación, se usan varios estadísticos para evaluar la calidad mencionada, con la media de los cinco porcentajes de clasificación obtenidos con 5-fold cross validation.

Se mezclan dos estadísticos:

- Precisión:

$$\text{tasa\_clas} = 100 \cdot \frac{\text{Nº instancias bien clasificadas en T}}{\text{Nº instancias en T}}$$

- Simplicidad:

$$\text{tasa\_red} = 100 \cdot \frac{\text{Nº valores } w_j < 0{,}1}{\text{Nº características}}$$

donde T es la partición para validación. Se usará la siguiente función objetivo, $F(W) = \alpha \cdot \text{tasa\_clas}(W) + (1 - \alpha) \cdot \text{tasa\_red}(W)$, que se busca maximizar, con $\alpha = 0{,}75$, importante por ser el que pondera la importancia entre acierto y reducción de características de una solución, dando en este caso más importancia al acierto.

Se implementa el algoritmo 1-NN, junto a los algoritmos Greedy Relief y búsqueda local el primer mejor, sobre los datasets *breast-cancer*,*ecoli* y *parkinsons*, previa normalización de los mismos.

Aplicación de algoritmos empleados

2.1. Esquema y representación de las soluciones

Se ha buscado representar de manera consistente todos y cada uno de los datos presentes en los datasets. Por ello, se declará la estructura *Muestra*, que representa una de las componentes de X:

```
struct Muestra {
    vector<double> caracteristicas;
    string clase;
}
```

con *caracteristicas* representando los valores de las características para una muestra y *clase*, que representa su clase. Nótese que, volviendo a la función p anterior, una instancia de *Muestra* es la tupla (c_i, clase) que le asocia a cierto $x_i \in X$.

Además, se almacenan los atributos, con el tipo correspondiente, con la siguiente estructura, buscando almacenar los datos dados.

```
struct Atributo {
    string nombre;
    string tipo;
}
```

donde, para cierto atributo o característica, se almacena el nombre y su tipo como cadenas de texto.

Para terminar de representar los datos recogidos en un archivo .arff, donde se almacenan las particiones de los datasets, se utiliza la siguiente estructura de datos (sería la representación de X):

```
struct Dataset {
        vector<Atributo> atributos;
        vector<Muestra> muestras;
}
```

con *atributos* hace referencia a la colección de nombre y tipo de las características recogidas, y *muestras* es una colección de muestras. Volviendo a la notación de la descripción del problema, una instancia inicializada de *Dataset* implica que *muestras* tendrá n muestras, correspondiendo a cada una (cada x_i) una clase y vector de características. Por último, se considera necesario hacer una representación consistente de las soluciones de los algoritmos (vectores de pesos). Se usa la siguiente estructura:

```
struct Pesos {
        vector<double> valores;
}
```

donde *valores* es el vector (de n componentes) de valores de los pesos, todos ellos normalizados a $[0, 1]$.

2.2. Operaciones auxiliares

Se mencionó anteriormente que se usará el algoritmo 1-NN para clasificar una muestra. Por ello, son necesarias funciones para el cálculo de distancias. Como los valores son numéricos, se escoge la métrica euclídea. Se implementan *distanciaEuclidea* y *distanciaEuclideaPonderada*, siendo la primera un caso especial de la segunda (todos los pesos valen 1,0):

Algorithm 1 Calcula la distancia Euclídea entre dos muestras

```
1: function DISTANCIAEUCLIDEA(Muestra m1, Muestra m2)
2:     sum ← 0,0
3:     for i ← 0 to m1.caracteristicas.size() − 1 do
4:         sum ← sum + (m1.caracteristicas[i] − m2.caracteristicas[i])^2
5:     end for
6:     return √sum
7: end function
```

El siguiente código generaliza la distancia euclídea a un vector de pesos cualquiera:

Algorithm 2 Calcula la distancia Euclídea ponderada entre dos muestras

```
1: function DISTANCIAEUCLIDEAPONDERADA(Muestra m1, Muestra m2, Pesos pesos)
2:     sum ← 0,0
3:     for i ← 0 to m1.caracteristicas.size() − 1 do
4:         if pesos.valores[i] > 0,1 then
5:             sum ← sum + pesos.valores[i] ∗ (m1.caracteristicas[i] −
   m2.caracteristicas[i])^2
6:         end if
7:     end for
8:     return √sum
9: end function
```

Se debe remarcar que, aunque se use la raíz cuadrada en el cálculo de la distancia, para el uso que se hace de la distancia euclídea se podría eliminar, pues la función raíz cuadrada sobre los números positivos es estrictamente creciente, y no se pierde información si se suprime.

Estas implementaciones se han realizado para simplificar la programación de la función que hará de clasificador 1-NN. Esta última función, dado un *Dataset* de entrenamiento y otro de validación. La implementación hará uso de la distancia euclídea ponderada, y si fuese necesario usar la distancia euclídea normal, se usarían todos los pesos con valor 1,0.

Algorithm 3 Algoritmo 1-NN

```
1: function ALGORITMO1NN(Muestra m, Dataset entrenamiento, Pesos pesos)
2:     index_min_distancia ← 0
3:     min_distancia ← DISTANCIAEUCLIDEAPONDERADA(m,
4:     entrenamiento.muestras[index_min_distancia], pesos)
5:     distancia ← min_distancia
6:     for i ← 1 to entrenamiento.muestras.size() − 1 do
7:         distancia ← DISTANCIAEUCLIDEAPONDERADA(m, entrenamiento.muestras[i], pesos)
8:         if distancia < min_distancia then
9:             index_min_distancia ← i
10:            min_distancia ← distancia
11:        end if
12:    end for
13:    return entrenamiento.muestras[index_min_distancia].clase
14: end function
```

Por último, se implementa una función auxiliar para simplificar la implementación de los algoritmos, *buscarMuestraMenorDistancia*, con el siguiente pseudocódigo:

Algorithm 4 Buscar Muestra a Menor Distancia

```
1: function BUSCARMUESTRAAMENORDISTANCIA(ds, m)
2:     index_buscado ← 0
3:     distancia_m_buscado ← DISTANCIAEUCLIDEA(ds.muestras[index_buscado], m)
4:     for i ← 1 to ds.muestras.size() − 1 do
5:         distancia_m_probable ← DISTANCIAEUCLIDEA(ds.muestras[i], m)
6:         if distancia_m_probable < distancia_m_buscado then
7:             index_buscado ← i
8:             distancia_m_buscado ← distancia_m_probable
9:         end if
10:    end for
11:    return index_buscado
12: end function
```

2.3. Función objetivo y estadísticos

Tal y como se mencionó en la descripción del problema, se busca maximizar el valor de la función objetivo, llamada *fitness*, con $\alpha = 0{,}75$. Además, se usan estadísticos complementarios para calcular la reducción y la tasa de acierto según unos datos de entrenamiento, validación y vector de pesos. A continuación se muestra el pseudocódigo correspondiente.

Algorithm 5 Tasa de Clasificación

```
1: function TASACLASIFICACION(Dataset entrenamiento, Dataset test, Pesos pesos)
2:     n_bien_clasificada ← 0
3:     prediccion ←
4:     for i ← 0 to test.muestras.size() − 1 do
5:         prediccion ← ALGORITMO1NN(test.muestras[i], entrenamiento, pesos)
6:         if prediccion = test.muestras[i].clase then
7:             n_bien_clasificada ← n_bien_clasificada + 1
8:         end if
9:     end for
10:    return (100,0 × convertir a double(n_bien_clasificada))/
   convertir a double(test.muestras.size())
11: end function
```

Algorithm 6 Tasa de Reducción

```
1: function TASAREDUCCION(Pesos pesos)
2:     UMBRAL ← 0,1
3:     n_menor_umbral ← 0
4:     for i ← 0 to pesos.valores.size() − 1 do
5:         if pesos.valores[i] < UMBRAL then
6:             n_menor_umbral ← n_menor_umbral + 1
7:         end if
8:     end for
9:     return (100,0 × n_menor_umbral)/(1,0 × pesos.valores.size())
10: end function
```

Algorithm 7 Calcula el valor de fitness

```
1: function FITNESS(tasa_cla, tasa_red)
2:     return α × tasa_cla + (1,0 − α) × tasa_red
3: end function
```

Descripción de algoritmos implementados

A continuación se muestran los pseudocódigos de los algoritmos implementados para la resolución del problema: búsqueda local y greedy RELIEF. Se busca un vector de pesos que maximice la función objetivo.

3.1. Búsqueda local

Se escoge la estrategia de el primero mejor para obtener las distintas soluciones para el vector de pesos. Por iteración se muta una única componente del vector de pesos de forma aleatoria sin repetir hasta que no se hayan mutado todas las componentes del vector. Más específicamente, para n pesos (inicializados con valores de una distribución uniforme sobre $[0,1]$, $U[0,1]$), se escoge aleatoriamente un índice $0 \leq i \leq n-1$ con $i \notin H$, donde H es el conjunto de índices ya mutados (este conjunto, cuando no se ha empezado a iterar, o se han mutado todos los elementos del vector de pesos, es $H = \{\}$. Ahora, escogido un índice i, si $pesos.valores[i]$ es el peso correspondiente, entonces la mutación es $pesos.valores[i] + X$ con $X \sim \mathcal{N}(0,0{,}3)$ (media 0 varianza 0.3). En pseudocódigo se tiene

Algorithm 8 Mutación con Búsqueda Local

```
1: function MUTACIONBL(pesos, index, normal)
2:     pesos.valores[index] ← pesos.valores[index] + RANDOM::GET(normal)
3:     if pesos.valores[index] > 1,0 then
4:         pesos.valores[index] ← 1,0
5:     end if
6:     if pesos.valores[index] < 0,0 then
7:         pesos.valores[index] ← 0,0
8:     end if
9: end function
```

donde, tras mutar, se trunca a 1,0 si $pesos.valores[i] > 1{,}0$ o a 0,0 si $pesos.valores[i] <$ 0,0.

La forma de tomar los índices de forma aleatoria sin repetir se muestra en la llamada a la siguiente función, donde para optimizar el tiempo solo se recorre una vez el vector de índices, haciendo siempre una mezcla aleatoria.

Con *Random::shuffle* se baraja aleatoriamente el vector de índices según la semilla introducida en la llamada al programa.

El pseudocódigo completo de la función de búsqueda local se muestra a continuación, desde la creación de la solución inicial hasta la mutación y elección de nuevas

Algorithm 9 Mezclar Índices

```
1: function MEZCLARINDICES(n_pesos, indices, inicializar)
2:     if inicializar then
3:         for i ← 0 to n_pesos − 1 do
4:             indices[i] ← i
5:         end for
6:     end if
7:     RANDOM::SHUFFLE(indices)
8: end function
```

soluciones, con criterios de parada $MAXIMO_{=}15000$ evaluaciones de la función objetivo o la generación de $20 \cdot n\_caracteristicas = LIMITE$ vecinos sin mejora:

Algorithm 10 Búsqueda Local

```
1: function BusquedaLocal(entrenamiento)
2:     solucion_actual ← new Pesos
3:     distribution ← new uniform_real_distribution(0,0, 1,0)
4:     normal ← new normal_distribution(0,0, √VARIANZA)
5:     for i ← 0 to entrenamiento.muestras[0].caracteristicas.size() − 1 do
6:         solucion_actual.valores.push_back(Random::get(distribution))
7:     end for
8:     tasa_cla_sol_actual ← tasaClasificacionLeaveOneOut(entrenamiento,
9:     solucion_actual)
10:    tasa_red_sol_actual ← tasaReduccion(solucion_actual)
11:    fitness_actual ← fitness(tasa_cla_sol_actual, tasa_red_sol_actual)
12:    n_vecinos_generados_sin_mejora ← 0
13:    num_caracteristicas ← entrenamiento.muestras[0].caracteristicas.size()
14:    indexes ← new vector<int(num_caracteristicas)
15:    hubo_mejora ← false
16:    mezclarIndices(num_caracteristicas, indexes, true)
17:    for i ← 0 to MAXIMO_ITERACIONES and n_vecinos_generados_sin_mejora <
       LIMITE × num_caracteristicas do
18:        hubo_mejora ← false
19:        if i > 0 then
20:            i ← i − 1
21:        end if
22:        for j ← 0 to indexes.size() − 1 and not hubo_mejora and i <
           MAXIMO_ITERACIONES do
23:            vecino ← solucion_actual
24:            mutacionBL(vecino, indexes[j], normal)
25:            tasa_cla_vecino ← tasaClasificacionLeaveOneOut(entrenamiento, vecino)
26:            tasa_red_vecino ← tasaReduccion(vecino)
27:            fitness_vecino ← fitness(tasa_cla_vecino, tasa_red_vecino)
28:            i ← i + 1
29:            if fitness_vecino > fitness_actual then
30:                fitness_actual ← fitness_vecino
31:                solucion_actual ← vecino
32:                hubo_mejora ← true
33:            else
34:                n_vecinos_generados_sin_mejora ←
               n_vecinos_generados_sin_mejora + 1
35:            end if
36:        end for
37:        if hubo_mejora then
38:            n_vecinos_generados_sin_mejora ← 0
39:            mezclarIndices(num_caracteristicas, indexes, false)
40:        end if
41:    end for
42:    return make_pair(solucion_actual, fitness_actual)
43: end function
```

3.2. Greedy RELIEF

El algoritmo Greedy RELIEF es un algoritmo sencillo, clasificado como voraz. Dada una muestra, se busca un enemigo más cercano (muestra más cercana con clase distinta) y un amigo más cercano (muestra más cercana con misma clase distinta a la muestra). Con ellos, se aplica una fórmula con las distancias de las características para calcular la respectiva coordenada del vector de pesos (el vector de pesos inicialmente tiene todas sus componentes a 0,0). Tras este cálculo, se normalizan los valores del vector de pesos (pues se pueden obtener valores fuera del intervalo $[0, 1]$).

Se muestran los correspondientes pseudocódigos para la implementación del algoritmo:

Algorithm 11 Buscar Enemigo Más Cercano

```
1: function buscarEnemigoMasCercano(entrenamiento, m)
2:     candidatosEnemigo ← new Dataset
3:     for i ← 0 to size(entrenamiento.muestras) − 1 do
4:         if entrenamiento.muestras[i].clase ≠ m.clase then
5:             candidatosEnemigo.muestras.push_back(entrenamiento.muestras[i])
6:         end if
7:     end for
8:     index_enemigo_buscado ←
9:     buscarMuestraAMenorDistancia(candidatosEnemigo, m)
10:    return candidatosEnemigo.muestras[index_enemigo_buscado]
11: end function
```

Algorithm 12 Buscar Amigo Más Cercano

```
1: function buscarAmigoMasCercano(entrenamiento, m)
2:     candidatosAmigo ← nuevo Dataset vacío
3:     amigo ← nueva Muestra vacía
4:     index_amigo_buscado ← −1
5:     for i ← 0 to entrenamiento.muestras.size() − 1 do
6:         if entrenamiento.muestras[i].clase = m.clase and m.caracteristicas ≠
   entrenamiento.muestras[i].caracteristicas then
7:             candidatosAmigo.muestras.push_back(entrenamiento.muestras[i])
8:         end if
9:     end for
10:    if candidatosAmigo.muestras.size() > 0 then
11:        index_amigo_buscado ← buscarMuestraAMenorDistancia(candidatosAmigo, m)
12:        amigo ← candidatosAmigo.muestras[index_amigo_buscado]
13:    end if
14:    return amigo
15: end function
```

Se usa una función auxiliar, *buscarMuestraMenorDistancia*, para hacer más legible el código. Su implementación se mostró anteriormente.

En la búsqueda del amigo más cercano, puede ocurrir que en el conjunto de entrenamiento haya una muestra con una clase que no tengan otras muestras, lo que hace que no se pueda encontrar ese amigo (ocurre en el dataset *ecoli*). Para solventar el problema, en el siguiente código (del algoritmo Greedy RELIEF) se hace la comprobación para evitar un error de ejecución.

Algorithm 13 Algoritmo Greedy Relief

```
1: function GreedyRelief(entrenamiento)
2:     solucion_actual ← nuevo Pesos
3:     for i ← 0 to entrenamiento.muestras[0].caracteristicas.size() − 1 do
4:         solucion_actual.valores.push_back(0,0)
5:     end for
6:     for i ← 0 to entrenamiento.muestras.size() − 1 do
7:         enemigoMasCercano ← buscarEnemigoMasCercano(entrenamiento, entrenamiento.muestras[i])
8:         amigoMasCercano ← buscarAmigoMasCercano(entrenamiento, entrenamiento.muestras[i])
9:         if amigoMasCercano.caracteristicas.size() > 0 then
10:            for j ← 0 to amigoMasCercano.caracteristicas.size() − 1 do
11:                solucion_actual.valores[j] ← solucion_actual.valores[j] + |entrenamiento.muestras[i].caracteristicas[j]−enemigoMasCercano.caracteristicas[j]|− |entrenamiento.muestras[i].caracteristicas[j]−amigoMasCercano.caracteristicas[j]|
12:            end for
13:        end if
14:    end for
15:    w_max ← máximo elemento en solucion_actual.valores
16:    for j ← 0 to solucion_actual.valores.size() − 1 do
17:        if solucion_actual.valores[j] < 0,0 then
18:            solucion_actual.valores[j] ← 0,0
19:        else
20:            solucion_actual.valores[j] ← solucion_actual.valores[j]/w_max
21:        end if
22:    end for
23:    tasa_cla ← tasaClasificacionLeaveOneOut(entrenamiento, solucion_actual)
24:    tasa_red ← tasaReduccion(solucion_actual)
25:    valor_fitness ← fitness(tasa_cla, tasa_red)
26:    return make_pair(solucion_actual, valor_fitness)
27: end function
```

Procedimiento para el desarrollo de la práctica

En primer lugar, la implementación de la práctica se ha realizado en C++, con el apoyo de librerías como *vector*,*utility*,*cmath*,..., además de usar *random.hpp*, sin hacer uso de ningún framework.

Por otro lado, se ha hecho uso de las siguientes carpetas y archivos (más adelante se mencionan y explican los archivos desarrollados para implementar la práctica):

- *BIN*: Se almacena el ejecutable y la carpet *BIN/DATA*, en la que se almacenan los archivos con los datos usados en la práctica.
- *FUENTES*: Todo el código desarrollado se almacena en esta carpeta:
  - *FUENTES/INCLUDE*: Se almacenan los archivos .h y .hpp con la declaración de las funciones y constantes a usar. Además, el archivo random.hpp también aparece aquí.
  - *FUENTES/SRC*: Aquí aparecen los archivos donde se implementan las funciones declaradas en los archivos .h y .hpp.
  - *FUENTES/CMakeLists.txt*: Archivo con las órdenes correspondientes para la creación del archivo Makefile, con el que se genera el programa. Para reducir el tiempo de ejecución, se usa el parámetro -O3 para optimizar el código compilado.

Además, los archivos realizados por el alumno para la práctica son:

- *aux.h/aux.cpp*: En estos archivos se declaran e implementan funciones para el preprocesamiento de datos (lectura y normalización), el cálculo de distancias, la declaración de constantes que se consideran generales, la declaración de las estructuras de datos y algunas funciones de depuración.
- *practica1.hpp/practica1.cpp*: Declaración e implementación de los algoritmos de la práctica. Aquí está la implementación del clasificador 1-NN, los algoritmos de búsqueda lineal y Greedy RELIEF y la función para mostrar los resultados.
- *main.cpp*: Establece la semilla para trabajar con números aleatorios y se llama a la función para las soluciones indicando el algoritmo a ejecutar.

Los pasos a seguir para la creación y ejecución del programa son los siguientes:

- **Paso 1**: Abrir la terminal, o ir, a la carpeta *software/FUENTES*,
- **Paso 2**: Ejecutar el comando çmake .".

- **Paso 3**: Ejecutar el comando "make". Esto creará el ejecutable en la carpeta *software/BIN*.
- **Paso 4**: Ir a la carpeta *software/BIN*.
- **Paso 5**: Ejecutar el comando "./practica1 <semilla>".

Experimentación y análisis de resultados

Se han realizado las mediciones de los estadísticos sobre tres conjuntos de datos: *breast-cancer*,*ecoli* y *parkinsons*. El dataset *breast-cancer* son datos para identificar la gravedad del cáncer de mama a partir de ciertas características, con 569 muestras, 31 características (incluida su clasificación) y 2 clases. El dataset *ecoli* contiene datos para identificar la posición de proteinas, con 366 muestras, 8 características (clase incluida) y clasificación en 8 clases. Por último, *parkinsons* almacena datos que identifican la presencia de la enfermedad de Parkinson según medidas obtenidas por la voz de pacientes, con 195 muestras, 23 características (clase incluida) y 2 posibles clases; este dataset está desbalanceado, con más enfermos que sanos.

5.1. Exposición de resultados

Los resultados obtenidos se obtienen mediante el uso de la semilla 60. No se usan otras debido a la gran cantidad de muestras que hay en los conjuntos de datos, aunque para algoritmos probabilísticos como búsqueda local es recomendable la comparación de resultados entre varias semillas. Aparecen, según la técnica 5-fold cross validation, los resultados de los estadísticos para cada ejecución del algoritmo, mostrándose la tasa de clasificación para la partición usada para test, la tasa de reducción de la solución, el valor de fitness y el tiempo de ejecución en segundos (s).

Tabla 5,1: Resultados obtenidos por el algoritmo 1NN en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
Partición 1	80.00	0.00	60.00	9×10-5	97.50	0.00	73.12	7×10-5	97.39	0.00	73.04	8.8×10-4
Partición 2	72.85	0.00	54.64	9×10-5	87.50	0.00	65.62	7×10-5	96.52	0.00	72.39	1.03×10-3
Partición 3	82.35	0.00	61.76	1×10-4	97.50	0.00	73.12	7×10-5	92.17	0.00	69.13	1.42×10-3
Partición 4	83.82	0.00	62.86	9×10-5	100.00	0.00	75.00	7×10-5	98.26	0.00	73.69	7.8×10-4
Partición 5	85.00	0.00	63.75	7×10-5	91.42	0.00	68.57	6×10-5	93.57	0.00	70.18	7.2×10-4
Media	80.80	0.00	60.60	9×10-5	94.78	0.00	71.08	7×10-5	95.58	0.00	71.68	9.6×10-4

Tabla 5,2: Resultados obtenidos por el algoritmo BL en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
Partición 1	77.14	42.85	68.57	0.44	100.00	77.27	94.31	0.92	93.04	76.66	88.94	10.15
Partición 2	54.28	71.42	58.57	0.62	92.50	72.72	87.55	1.12	94.78	86.66	92.75	21.00
Partición 3	77.94	57.14	72.74	0.56	90.00	77.27	86.81	0.65	93.91	86.66	92.10	14.85
Partición 4	79.41	57.14	73.84	0.56	92.50	86.36	90.96	0.85	94.78	80.00	91.08	11.37
Partición 5	86.66	42.85	75.71	1.12	91.42	68.18	85.61	1.12	94.49	83.33	91.70	11.66
Media	75.08	54.28	69.88	0.66	93.28	76.36	89.05	0.93	94.20	82.66	91.31	13.80

Tabla 5,3: Resultados obtenidos por el algoritmo Greedy RELIEF en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
Partición 1	77.14	28.57	65.00	3.16×10-3	97.50	0.00	73.12	1.07×10-3	97.39	3.33	73.87	1.014×10-2
Partición 2	74.28	28.57	62.85	3.07×10-3	90.00	0.00	67.50	1.17×10-3	97.39	3.33	73.87	1.275×10-2
Partición 3	79.41	28.57	66.70	3.13×10-3	95.00	0.00	71.25	0.11× 10-3	89.56	13.33	70.50	9.93×10-3
Partición 4	80.88	28.57	67.80	3.3×10-3	100.00	0.00	75.00	1.06×10-3	97.39	0.00	73.04	8.78×10-3
Partición 5	86.66	28.57	72.14	3.59×10-3	91.42	0.00	68.57	1.12×10-3	94.49	0.00	70.87	9.21×10-3
Media	79.67	28.57	66.90	3.25×10-3	94.78	0.00	71.08	1.1×10-3	95.24	4.00	72.43	1.016×10-2

Además, en la siguiente tabla se recogen los estadísticos medios en la ejecución de cada algoritmo:

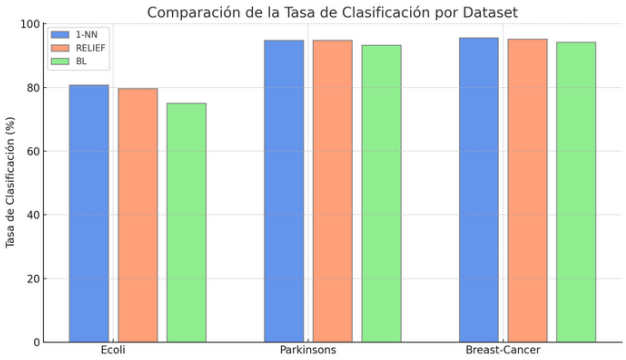
Tabla 5,4: Resultados globales en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
1-NN	80.80	0.00	60.60	9×10-5	94.78	0.00	71.08	7×10-5	95.58	0.00	71.68	9.6×10-4
RELIEF	79.67	28.57	66.90	3.25×10-3	94.78	0.00	71.08	1.1×10-3	95.24	4.00	72.43	1.016x10-2
BL	75.08	54.28	69.88	0.66	93.28	76.36	89.05	0.93	94.20	82.66	91.31	13.80

5.2. Análisis de resultados

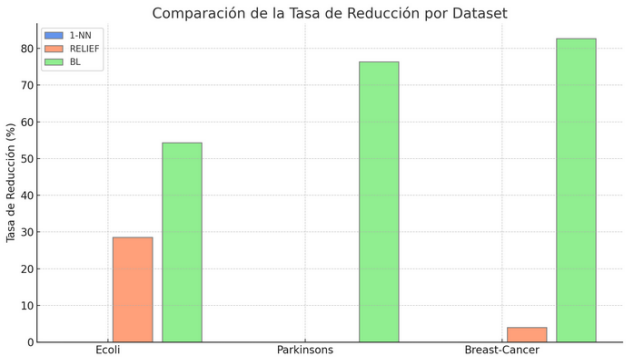
Se ha considerado el uso de gráficos de barras, ya que se estima recogen y muestran mejor los estadísticos medios para comparar entre algoritmos. Se hace la restricción a la media de cada estadístico.

- Tasa de clasificación:



Los resultados de la tasa de clasificación en test media arroja resultados similares entre los tres algoritmos, siendo la diferencia más notable que el dataset *ecoli* da una tasa de clasificación menor al resto de datasets. Mirando cada algoritmo, 1-NN siempre es algo mejor, y esto se debe a que están todos los pesos a 1.0, siendo el resto valores entre 0.0 y 1.0.

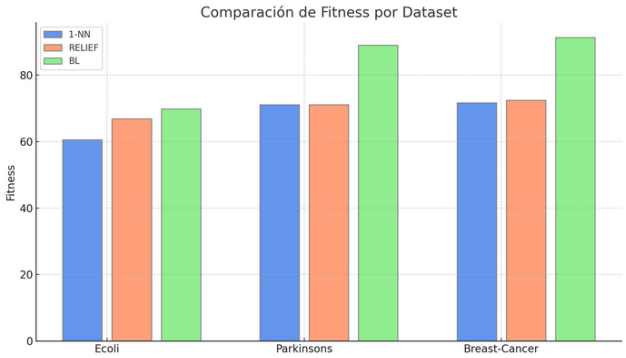
- Tasa de reducción:



Tal y como se define la tasa de reducción, es normal que para 1-NN no haya, pues todos los pesos están con valor 1.0. Respecto al resto de algoritmos, se desprenden resultados esperables, teniendo en todos los conjuntos de datos menor tasa de reducción en el algoritmo greedy RELIEF que en la búsqueda local, puesto que 1-NN no descarta características y greedy RELIEF es menos

agresivo. En general, búsqueda local elimina un mayor número de características irrelevantes en los conjuntos de datos usados, lo que lleva a pensar que muchas características tomadas en cuenta en las mediciones de las muestras no son de mucha importancia para la clasificación en el modelo, sobre todo en el dataset *breast-cancer* (aunque *parkinsons* no da una tasa de reducción mucho menor).

- Fitness:

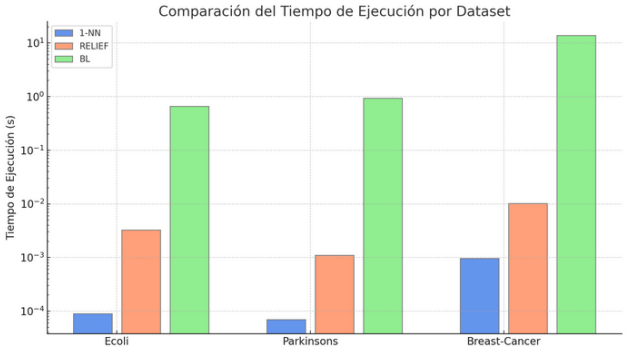


Usando $\alpha = 0{,}75$, dando mayor peso a las muestras bien clasificadas que a la reducción de características, como la expresión es $\alpha \cdot \text{Tasa\_clas}(W) + (1 - \alpha) \cdot \text{Tasa\_red}(W)$, los resultados mostrados son normales. Como α no es muy pequeño, se parte de que en general la tasa de clasificación es parecida, lo que hace que $\alpha \cdot \text{Tasa\_clas}(W)$ no varíe mucho entre algoritmos, teniendo búsqueda local algo menos. Por otro lado, si se suma $(1 - \alpha) \cdot \text{Tasa\_red}(W)$, dado que $\text{Tasa\_red}(W)$ es mucho más alto en búsqueda local, resulta en un fitness más alto que el resto de algoritmos. Además, en 1-NN la tasa de reducción es nula, y en consecuencia el valor de fitness coincide con el de la tasa de clasificación.

Se ha remarcado que si α no fuese muy pequeño porque en tal caso, como $\lim_{\alpha \to 0} \text{fitness}(W) = \text{Tasa\_red}(W)$, entonces fitness para 1-NN es nulo, y el de búsqueda local es mucho mayor al de greedy RELIEF (notándose mucho más en el dataset *parkinsons*, ya que volviendo a la gráfica de la tasa de reducción media, dicho dataset tiene tasa de reducción nula). Si $\alpha \to 1$, se tendría que 1-NN tiene más fitness (pasando al límite), aunque no con una diferencia tan notable como cuando $\alpha \to 0$. Esto da a entender que la elección de α no es trivial, y debe tomarse teniendo en cuenta cuánta importancia se le quiera dar a la cantidad de muestras bien clasificadas en test o a la cantidad de características que el algoritmo considera no relevantes.

A grandes rasgos, el algoritmo de búsqueda local obtiene mejor fitness medio que el resto de algoritmos por lo expuesto anteriormente, aunque en *ecoli* no es mucho mejor al algoritmo greedy RELIEF.

- Tiempo de ejecución (segundos):



Para una correcta comparación se ha pasado a una escala logarítmica, manteniendo las proporciones, ya que el tiempo de ejecución medio de búsqueda local en *breast-cancer* es mucho mayor al del resto de ejecuciones.

Se recuerda que *ecoli* tiene 366 muestas con 7 características (excluyendo la clase), *parkinsons* tiene 195 muestras con 22 características (excluyendo la clase) y *breast-cancer* tiene 569 muestras con 30 características (excluyendo la clase). A primera vista, se observa que el dataset *breast-cancer* es mucho mayor que los otros dos, y entre *ecoli* y *parkinsons*, el primero tiene más muestras, pero el segundo más características. Esto puede traducirse en un objeto *Dataset* de más filas (más muestras) en *ecoli* y un objeto *Dataset* de más columnas (más características) en *parkinsons*. Por lo tanto, pensando en el objeto como una matriz (exluyendo la clase y el nombre y tipo de atributos), se tiene que *ecoli* tiene $366 \times 7 = 2562$ elementos y *parkinsons* tiene $195 \times 22 = 4290$ elementos, lo que lleva a pensar que *parkinsons* sea más grande que *ecoli*.

Como *breast-cancer* es el dataset más grande, se observa que es el que más tarda. Si se mira un dataset de los tres, se observa que el que más tarda es búsqueda local, y el que menos 1-NN. Esto es normal puesto que búsqueda local puede caer en óptimos locales, intentando mejorar hasta que se da la condición de parada en cuanto a iteraciones o vecinos generados sin mejora, que conlleva bastante más tiempo que simplemente observar si la clase de una muestra en test coincide con la clase de su vecino más cercano.

Por otra parte, el tiempo de greedy RELIEF se encuentra acotado entre los tiempos de 1-NN (inferiormente) y búsqueda local (superiormente). Tarda más que 1-NN ya que debe evaluar y seleccionar características, pero menos que búsqueda local ya que no tiene una condición de parada de tantas iteraciones y generaciones como búsqueda local, además de que búsqueda local es un algoritmo probabilístico, encontrando una solución mejor o no encontrándola en función de la mutación, mientras que greedy RELIEF no lo es.

5.2.1. Conclusiones

Tras realizar el análisis de cada estadístico medio, se llega a la conclusión de que el algoritmo de búsqueda local ofrece mejores resultados en cuanto a que obtiene mayor valor en fitness. Sin embargo, cabe mencionar que tarda bastante más que el resto de algoritmos. Era de esperar también que 1-NN fuese el que menos tardase.

Si se tuviese que escoger un algoritmo con el fin de resolver un problema, la elección del mismo dependerá de los requisitos del mismo. Si el tiempo no es un inconveniente, o el dataset no es muy grande, se escogería el algoritmo de búsqueda local, vistos los resultados medios dados en fitness, además de la capacidad de reducción de características que no considera relevantes. Podría decirse que consigue

obviar características no relevantes para una correcta clasificación (aprende características más robustas). Sin embargo, si el tiempo fuese crucial en el problema a resolver, habría que discutirse si escoger 1-NN o greedy RELIEF. Este último da unos resultados similares a 1-NN, aunque mejora un poco en el dataset *ecoli*. Si se observan los valores de las tablas (los de las gráficas han sido escalados logarítmicamente), se haría la elección del algoritmo greedy RELIEF, ya que aunque no mejora mucho el valor de fitness, y pese a tardar ligeramente más, en general reduce algunas características (aunque no ocurre en el dataset *parkinsons*.

Una consideración crucial en nuestro análisis es el desequilibrio de clases presente en el dataset *parkinsons*. Este desequilibrio puede influir significativamente en la fiabilidad y el rendimiento de los modelos evaluados. En particular, puede llevar a una sobreestimación de la importancia de ciertas características, dado que las métricas de evaluación podrían estar sesgadas hacia la clase predominante. Este fenómeno se observa en la tasa de reducción nula reportada para el algoritmo Greedy Relief en este dataset, lo que sugiere que el algoritmo podría estar priorizando incorrectamente las características en función de su prevalencia en la clase mayoritaria en lugar de su verdadera relevancia para la clasificación. Para solucionar esto se podrían usar otras métricas (más potentes con respecto el desbalanceo de clases) u obtener muestras tales que aumente la calidad de los datos, entre otros. Esto demuestra que la calidad y fiabilidad de un modelo dependen en gran medida de la calidad de los datos con los que se entrene.