PRÁCTICA 1: TÉCNICAS DE BÚSQUEDA LOCAL y ALGORITMOS GREEDY

Problema: Aprendizaje de Pesos en Características
Algoritmos: Búsqueda Local, Greedy RELIEF

Juan Manuel Rodríguez Gómez
49559494Z
juanmaarg6@correo.ugr.es

Metaheurísticas, Grupo 1 (Lunes 17:30 – 19:30)
Curso 2022 – 2023

UNIVERSIDAD
DE GRANADA

Índice

1. Descripción del Problema

El **Problema del Aprendizaje de Pesos en Características (APC**) es un problema que consiste en obtener un vector de pesos que permita ponderar las características asociadas a un dato con la intención de obtener un mejor porcentaje de clasificación de datos futuros. Tenemos una muestra de datos X = {x_1, x_2, ..., x_n}, donde cada dato de dicha muestra está formado por un conjunto de características {c_1, c_2, ..., c_m,} y su CATEGORÍA, de forma que buscamos obtener un vector de pesos W = {w_1, w_2, ..., w_m} donde el peso w_i pondera la característica c_i y que ese vector de pesos nos permita optimizar el rendimiento del clasificador k-NN. Nosotros usaremos el clasificador 1-NN, por lo que vamos a considerar el vecino más cercano (con el que se tiene menor **distancia euclídea ponderada por pesos**) para predecir la clase de cada dato.

A lo largo de nuestro proceso vamos a realizar dos tareas: primero obtendremos el vector de pesos (aprendizaje) y a continuación valoraremos la calidad del mismo (validación). Usaremos la técnica de validación cruzada ***5-fold cross validation***. El conjunto de datos ya está dividido en 5 particiones disjuntas al 20% desordenadas, y con la misma distribución de clases. Aprenderemos un clasificador utilizando el 80% de los datos disponibles (4 particiones de las 5) y validaremos con el 20% restante (la partición restante). Así obtendremos un total de 5 valores de porcentaje de clasificación en el conjunto de prueba, uno para cada partición empleada como conjunto de validación. La calidad del método de clasificación se medirá con un único valor, correspondiente a la media de los 5 porcentajes de clasificación del conjunto de prueba.

Para la función de evaluación se combinan dos criterios:

- **Precisión →** $\boldsymbol{tasa\_clas = 100 * \frac{Nº\ de\ instancias\ bien\ clasificadas\ en\ T}{Nº\ de\ instancias\ en\ T}}$

- **Simplicidad →** $\boldsymbol{tasa\_red = 100 * \frac{Nº\ de\ valores\ w_i < 0.1}{Nº\ de\ características}}$

Utilizaremos una agregación sencilla que combine ambos objetivos en un único valor. Así, la **función objetivo** será $\boldsymbol{F(w) = \alpha * tasa\_clas(w) + (1 - \alpha) * tasa\_red(w)}$. El valor de α pondera la importancia entre el acierto y la reducción de características de la solución encontrada (el clasificador generado). Usaremos α = 0.8, dando más importancia al acierto. El objetivo es obtener el conjunto de pesos W que maximiza esta función, es decir, que maximice el acierto del clasificador 1-NN y, a la vez, que considere el menor número de características posible.

Se ha implementado el **clasificador 1-NN**, el **algoritmo Greedy RELIEF** y el **algoritmo de Búsqueda Local del Primer Mejor**. Los distintos algoritmos se van a ejecutar sobre **tres conjuntos de datos distintos: *diabetes*, *ozone-320* y *spectf-heart***. Hay que tener en cuenta que debemos normalizar los datos.

2. Descripción de la Aplicación de los Algoritmos Empleados al Problema

2.1. Esquema de Representación de Soluciones

Para representar los datos en el programa se emplea un **struct Ejemplo** que recoge toda la información necesaria: un vector<double> con los valores de cada una de las n características del ejemplo concreto, así como un string que representa su clase o categoría.

```
struct Ejemplo {
   vector<double> val_caracts;
   string categoria;
   int num_caracts;
}
```

Cada **conjunto de datos** se almacena en un **vector<Ejemplo>** y se emplean las funciones *leerFicheroARFF* y *normalizarValores* para leer los datos del conjunto, normalizarlos y almacenarlos en dicho vector.

Por otro lado, la **solución** a nuestro problema es un vector de pesos, W, representando mediante **vector<double>**, de n componentes (siendo n el número de características de los ejemplos de nuestro conjunto de datos). En cada componente del vector tendremos un valor real $w_i \in [0, 1]$, que representa el peso asociado a la característica i-ésima.

2.2. Operadores Comunes

Todos los algoritmos hacen uso del cálculo de la distancia. Como dijimos, para este cálculo se emplea **la distancia euclídea ponderada mediante un vector de pesos**. En el caso de que la distancia deseada sea la estándar (es decir, sin ponderación de pesos), se asume que los pesos valen siempre uno. Se han realizado dos implementaciones de la distancia euclídea, una con pesos y otra sin pesos.

Como simplemente usamos la distancia para comparar, la vamos a implementar como si la eleváramos al cuadrado, es decir, eliminando la raíz cuadrada (podemos hacerlo sin problema ya que se mantiene el orden.). Hacemos esto para ahorrar tiempo de cálculo (ya que esta es la función más llamada a lo largo del programa).

```
función distanciaEuclidea(e1, e2)
   distancia = 0
   for i:=0 to n-1 do              (n es el número de características)
      distancia += (e2[i] – e1[i]) * (e2[i] – e1[i])
```

```
función distanciaEuclideaPesos(e1, e2, w)
   distancia = 0
   for i:=0 to n-1 do              (n es el número de características)
      if w[i] >= 0.1 then
         distancia += w[i] * (e2[i] – e1[i]) * (e2[i] – e1[i])
```

También tenemos la función ***clasificador1NNPesos***, que clasifica un ejemplo basándose en la técnica del vecino más cercano. Debemos pasarle también el conjunto de entrenamiento (donde hay otros ejemplos ya están clasificados), y el vector de pesos. Al igual que pasaba con la distancia euclídea, si queremos que el clasificador no tenga en cuenta los pesos, hacemos que todas las componentes del vector de pesos sean 1.

```
función clasificador1NNPesos(e, pos_e, entrenamiento, w)
   pos_vecino_mas_cercano = 0
   distancia_min = +infinito
   for i:=0 to n-1 do        (n es el número de ejemplos de entrenamiento)
      if i != pos_e then
         distancia = distanciaEuclideaPesos(e, entrenamiento[i], w)
         if distancia < distancia_min then
            distancia_min = distancia
            pos_vecino_mas_cercano = i
   return entrenamiento[pos_vecino_mas_cercano].categoria
```

2.3. Función Objetivo

La función objetivo que queremos maximizar se implementa tal y como se dijo en la descripción del problema, donde el valor de *alpha* es de 0.8, dando más importancia al acierto que a la reducción. De igual forma, para la tasa de clasificación y la tasa de reducción, usamos las expresiones dadas en la descripción del problema.

```
función fitness(tasa_clas, tasa_red)
   return (alpha * tasa_clas) + ( (1 – alpha) * tasa_red )
```

```
función tasaClasificacion(clasificacion, test)
   num_instancias_bin_clas = 0
   for i:=0 to n-1 do           (n es el número de ejemplos clasificados)
      if clasificación[i] == test[i].categoria then
         num_instancias_bin_clas++
   return 100.0 * num_instancias_bin_clas / n
```

```
función tasaReduccion(clasificacion, test)
   num_caracts_descartadas = 0
   for i:=0 to n-1 do              (n es el tamaño del vector de pesos)
      if w[i] < 0.1 then
         num_caracts_descartadas++
   return 100.0 * num_caracts_descartadas / n
```

3. Descripción de los Algoritmos Implementados

En esta sección se describen los dos algoritmos implementados en esta primera práctica para el problema del Aprendizaje de Pesos en Características: Greedy RELIEF y Búsqueda Local. En ambos lo que se pretende es rellenar un vector de pesos para maximizar la función objetivo.

3.1. Algoritmo de Comparación: Greedy RELIEF

Este es un algoritmo voraz muy sencillo, que **servirá como caso base para comparar las diferentes metaheurísticas desarrolladas**. Se trata de buscar, para cada ejemplo, su amigo (otro ejemplo que tenga la misma clase) y su enemigo (otro ejemplo que tenga distinta clase) más cercano. Después, componente a componente, se suma al vector de pesos la distancia a su enemigo y se resta la distancia a su amigo.

En este proceso es posible que los pesos se salgan del intervalo [0, 1], por lo que al finalizar la suma componente a componente es necesario normalizar. Además, el vector de pesos inicialmente tiene todas sus componentes a 0 (para realizar esta inicialización disponemos de una función *inicializarVectorPesos*).

A continuación, se encuentra la función que se encarga de encontrar el amigo y enemigo más cercano a un ejemplo dado (hay que tener en cuenta que el amigo más cercano no puede ser el propio ejemplo).

```
función amigoEnemigoMasCercanos(e, pos_e, entrenamiento)
   distancia_min_amigo = +infinito
   distancia_min_enemigo = +infinito
   for i:=0 to n-1 do        (n es el número de ejemplos de entrenamiento)
      if i != pos_e then
         distancia = distanciaEuclidea(e, entrenamiento[i])
         if (entrenamiento[i].categoria != e.categoria and
             distancia < distancia_min_enemigo) then
            pos_enemigo = i
            distancia_min_enemigo = distancia
         else if (entrenamiento[i].categoria == e.categoria and
                  distancia < distancia_min_enemigo) then
            pos_amigo = i
            distancia_min_amigo = distancia
   return pos_enemigo, pos_amigo
```

Utilizando dicha función, el **algoritmo Greedy RELIEF** sería el siguiente:

```
función greedy(entrenamiento)
   w = inicializarVectorPesos()
   for i:=0 to n-1 do        (n es el número de ejemplos de entrenamiento)
     pos_enemigo, pos amigo = amigoEnemigoMasCercanos(entrenamiento[i],
                              i, entrenamiento)
      for j:=0 to m-1 do            (m es el tamaño del vector de pesos)
         w[j] = w[j]
+ |entrenamiento[i].val_caracts[j] - entrenamiento[pos_enemigo].val_caracts[j]|
- |entrenamiento[i].val_caracts[j] - entrenamiento[pos_amigo].val_caracts[j]|
   w_max = max(w)
   for j:=0 to m-1 do
      if w[j] < 0 then
         w[j] = 0
      else
         w[j] = w[j] / w_max
   return w
```

3.2. Algoritmo de Búsqueda Local

Empleamos la técnica de búsqueda local del **primer mejor** para rellenar el vector de pesos. La idea es mutar en cada iteración una componente aleatoria y **distinta** del vector de pesos, sumándole un valor extraído de una normal de *media* = 0 y desviación típica *sigma* =sqrt(3). Si tras esta mutación se mejora la función objetivo, nos quedamos con este nuevo vector, y si no lo desechamos. Si algún peso se sale del intervalo [0, 1] tras la mutación, directamente lo truncamos a 0 (si es menor que 0) o a 1 (si es mayor que 1).

Para la **generación de la solución inicial** sobre la que iterar, consideramos valores extraídos de una distribución uniforme U[0,1], que obtenemos gracias al tipo *uniform_real_distribution<double>* de la librería *random*. Generamos así una solución inicial aleatoria.

(*)

```
// Inicializamos el vector de índices y el vector de pesos (este último
// se hace aleatoriamente)
for i:=0 to m-1 do                    (m es el tamaño del vector de pesos)
   índices.push_back(i)
   w[i] = valorDistribucionUniforme(0,1)
```

Para escoger el valor con el que se muta cada componente utilizamos esta vez el tipo predefinido *normal_distribution<double>* de la librería *random*. De esta forma, el **operador de generación de vecino** sería de la siguiente forma:

```
// Suponemos que comp_mutada es la componente del vector de pesos que
// vamos a modificar
w_mutado = w
w_mutado[comp_mutada] += valorDistribucionNormal(0, sqrt(3))
// Truncamos si es necesario
if w_mutado[comp_mutada] > 1 then
   w_mutado[comp_mutada] = 1
else if w_mutado[comp_mutada] < 0 then
   w_mutado[comp_mutada] = 0
```

Por otro lado, el **método de exploración del entorno** es el siguiente: Para escoger qué componente del vector de pesos vamos a mutar, tenemos un vector de índices del mismo tamaño que el vector de pesos, que barajamos de forma aleatoria y recorremos secuencialmente. Si llegamos al final, volvemos a barajarlo para seguir generando nuevas soluciones. Para **comprobar si se mejora el fitness con una nueva solución**, se realiza lo siguiente:

(**)

```
for i:=0 to n-1 do          (n es el número de ejemplos de entrenamiento)
   clasificacion.push_back( clasificador1NNPesos(entrenamiento[i], i,
                            entrenamiento, w) )
fitness_calculado = fitness( tasaClasificacion(clasificación,
                    entrenamiento), tasaReduccion(w) )
clasificación.clear()
```

Con todo lo anterior, el algoritmo de Búsqueda Local sería el siguiente:

```
función busquedaLocal(entrenamiento)
  clasificacion          (vector<string>)
  indices                (vector<int>)
  mejor_fitness
  num_iteraciones = 0
  num_vecinos = 0
  hay_mejora = false
  w, indices = (*)       (Código de inicialización descrito anteriormente)
  shuffle(indices)
  mejor_fitness = (**) con w (Código de evaluación descrito anteriormente)
  while(num_iteraciones < MAX_ITER and num_vecinos < m*COEF_MAX_VECINOS) do
                                        (m es el tamaño del vector de pesos,
                                         MAX_ITER = 15000,
                                         COEF_MAX_VECINOS = 20)
    comp_mutada = index[num_iteraciones % m]
    w_mutado = w
    w_mutado[comp_mutada] += valorDistribucionNormal(0, sqrt(3))
    // Truncamos si es necesario
    if w_mutado[comp_mutada] > 1 then
      w_mutado[comp_mutada] = 1
    else if w_mutado[comp_mutada] < 0 then
      w_mutado[comp_mutada] = 0
    fitness_actual = (**) con w_mutado
    if fitness_actual > mejor_fitness then
      num_vecinos = 0
      w = w_mutado
      mejor_fitness = fitness_actual
      hay_mejora = true
    else
      num_vecinos++
    if (num_iteraciones % m == 0 or hay_mejora) then
      shuffle(índices)
      hay_mejora = false
    num_iteraciones++
  return w
```

4. Procedimiento Considerado para Desarrollar la Práctica

Todo el código de la práctica se ha desarrollado en **C++** siguiendo el estándar 2011. Utilizamos diferentes librerías como iostream, vector, cmath, random... Además, hacemos uso también del archivo **random.hpp** proporcionado por el profesor para trabajar de forma más cómoda con números aleatorios. **No se ha hecho uso de ningún framework de metaheurísticas.**

Respecto a la **estructura de directorios y archivos**, Los ficheros de datos se encuentran en la carpeta *BIN/DATA* y los ejecutables en la carpeta *BIN*. Por otro lado, el código fuente se encuentra en la carpeta *FUENTES*, en la cual encontramos el archivo *CMakeLists.txt* (este contiene una serie de instrucciones que al lanzar el comando *cmake* para ejecutarlo, nos generará un makefile que nos permitirá compilar el proyecto mediante la orden *make*), la carpeta *INCLUDE* con los archivos de cabecera *.h* y la carpeta *SRC* con los archivos de código *.cpp*. Respecto a los diferentes archivos de código:

- **util.h/util.cpp** → Contienen funciones auxiliares que se utilizan en las diferentes prácticas, tales como aquellas destinadas a la lectura y normalización de datos de los archivos, las funciones de la distancia euclídea, la función de evaluación, etc.
- **p1.h/p1.cpp** → Contienen los algoritmos implementados en esta primera práctica (clasificador1NN, Greedy RELIEF y Búsqueda Local), así como una función para ejecutar cada algoritmo con los diferentes conjuntos de datos y mostrar los resultados obtenidos.
- **random.hpp** → Archivo proporcionado por el profesor para trabajar de forma más cómoda con números aleatorios.
- **main.cpp** → Archivo que contiene la función principal para ejecutar el programa.

Para **compilar y ejecutar el proyecto** hay que hacer lo siguiente:

1) Situarnos en la carpeta ***software/FUENTES***.
2) Ejecutar el comando ***cmake***, generándose así el makefile correspondiente.
3) Ejecutar el comando ***make*** para compilar el proyecto y obtener el ejecutable *practica1_MH* en la carpeta *software/BIN*.
4) Situarnos en la carpeta ***software/BIN***.
5) Como ya hemos dicho, trabajamos con números aleatorios, luego, necesitamos especificar una semilla mediante la línea de comandos a la hora de ejecutar el programa. Por ello, el comando para ejecutar el programa es ***./practica1_MH {semilla}*.**

5. Experimentos y Análisis de Resultados

Ejecutamos los algoritmos sobre cada uno de los tres conjuntos de datos siguientes:

- **Diabetes**: Contiene atributos calculados a partir de mediciones de pacientes. La tarea consiste en determinar si desarrollarán diabetes en los próximos 5 años. Consta de 768 ejemplos, 8 atributos numéricos y la clase (puede ser de dos tipos, negativa o positiva).
- **Ozone**: Base de datos para la detección del nivel de ozono según las mediciones realizadas a lo largo del tiempo. Consta de 320 ejemplos, 72 atributos numéricos y la clase (puede ser de dos tipos, día normal o día con una alta concentración en ozono).
- **Spectf-heart**: Contiene atributos calculados a partir de imágenes médicas de tomografía computerizada (SPECT) del corazón de pacientes humanos. La tarea consiste en determinar si la fisiología del corazón analizado es correcta o no. Costa de 267 ejemplos, 44 atributos entero y la clase (puede ser de dos tipos, paciente sano o paciente con patología cardíaca).

5.1. Resultados Obtenidos

A continuación, se muestran las tablas de los resultados obtenidos para cada uno de los algoritmos aplicados sobre los diferentes conjuntos de datos con la semilla 49559494. En la tabla se observan los resultados de 5 ejecuciones diferentes para cada conjunto de datos de acuerdo a la técnica de validación cruzada *5-fold cross validation*. Se muestran los valores de la tasa de clasificación (*%_clas*), la tasa de reducción (*%red*), la función objetivo (*Fit.*) y el tiempo de ejecución en milisegundos (*T*). Finalmente, también se muestre una tabla global con los resultados medios obtenidos de cada algoritmo para los diferentes conjuntos de datos.

Tabla 5.1: Resultados obtenidos por el algoritmo 1-NN en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	*%red*	*Fit.*	T	*% clas*	*%red*	*Fit.*	T	*% clas*	*%red*	*Fit.*	T
Partición 1	66,88	0,00	53,51	1,34	70,31	0,00	56,25	1,79	77,14	0,00	61,71	1,39
Partición 2	66,88	0,00	53,51	1,28	75,00	0,00	60,00	1,80	88,57	0,00	70,86	1,38
Partición 3	68,83	0,00	55,06	1,33	79,69	0,00	63,75	1,83	91,43	0,00	73,14	1,38
Partición 4	69,48	0,00	55,58	1,52	73,44	0,00	58,75	1,80	85,71	0,00	68,57	1,38
Partición 5	69,74	0,00	55,79	1,44	81,25	0,00	65,00	1,80	84,06	0,00	67,25	1,38
Media	68,36	0,00	54,69	1,38	75,94	0,00	60,75	1,80	85,38	0,00	68,31	1,38

Tabla 5.2: Resultados obtenidos por el algoritmo Greedy RELIEF en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	*%red*	*Fit.*	T	*% clas*	*%red*	*Fit.*	T	*% clas*	*%red*	*Fit.*	T
Partición 1	64,94	98,70	71,69	3,56	75,00	14,06	62,81	3,80	85,71	32,86	75,14	2,77
Partición 2	35,06	98,70	47,79	3,55	64,06	25,00	56,25	4,08	87,14	34,29	76,57	2,78
Partición 3	35,06	98,70	47,79	3,66	71,88	9,38	59,38	3,91	91,43	35,71	80,29	2,67
Partición 4	35,06	98,70	47,79	3,52	62,50	9,38	51,88	4,35	90,00	35,71	79,14	2,77
Partición 5	34,21	98,03	46,97	3,85	81,25	9,38	66,88	3,83	82,61	30,43	72,17	2,86
Media	40,87	98,57	52,41	3,63	70,94	13,44	59,44	3,99	87,38	33,80	76,66	2,77

Tabla 5.3: Resultados obtenidos por el algoritmo BL en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	*%red*	*Fit.*	T	*% clas*	*%red*	*Fit.*	T	*% clas*	*%red*	*Fit.*	T
Partición 1	68,83	94,16	73,90	55796,63	79,69	40,62	71,88	28665,43	74,29	54,29	70,29	10380,18
Partición 2	66,88	94,81	72,47	76162,13	68,75	32,81	61,56	11672,03	85,71	60,00	80,57	15275,71
Partición 3	66,88	94,81	72,47	54218,16	71,88	45,31	66,56	17233,68	92,86	44,29	83,14	19553,47
Partición 4	65,58	96,10	71,69	56756,61	64,06	35,94	58,44	16439,11	84,29	52,86	78,00	12989,44
Partición 5	64,47	94,74	70,53	50671,95	79,69	46,88	73,13	17888,69	81,16	66,67	78,26	10190,96
Media	66,53	94,92	72,21	58721,09	72,81	40,31	66,31	18379,79	83,66	55,62	78,05	13677,95

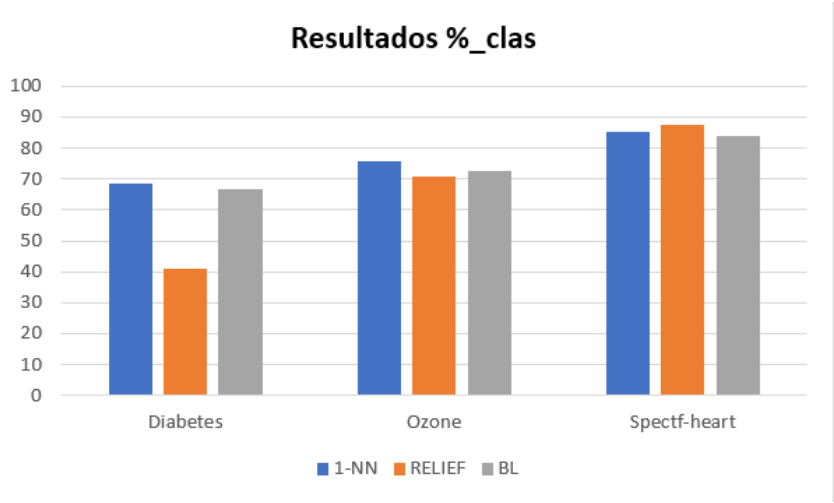
Tabla 5.4: Resultados globales en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	%_clas	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T	*%_clas*	*%red*	*Fit.*	T
1-NN	68,36	0,00	54,69	1,38	75,94	0,00	60,75	1,80	85,38	0,00	68,31	1,38
RELIEF	40,87	98,57	52,41	3,63	70,94	13,44	59,44	3,99	87,38	33,80	76,66	2,77
BL	66,53	94,92	72,21	58721,09	72,81	40,31	66,31	18379,79	83,66	55,62	78,05	13677,95

5.2. Análisis de los Resultados Obtenidos

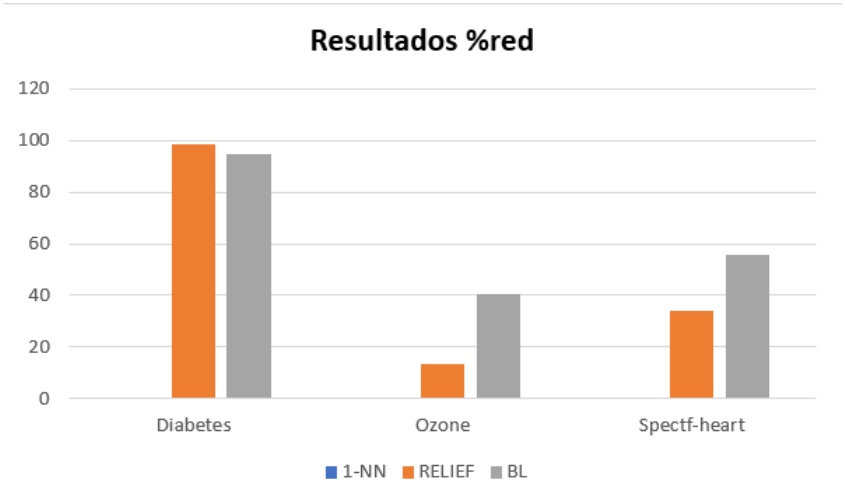
Vamos a comparar los diferentes resultados obtenidos por los diferentes algoritmos para cada conjunto de datos. Vamos a ir columna no por columna (primero comparamos la tasa de clasificación media, luego la tasa reducción, después el fitness y finalmente el tiempo de ejecución).

- **Tasa de clasificación:**



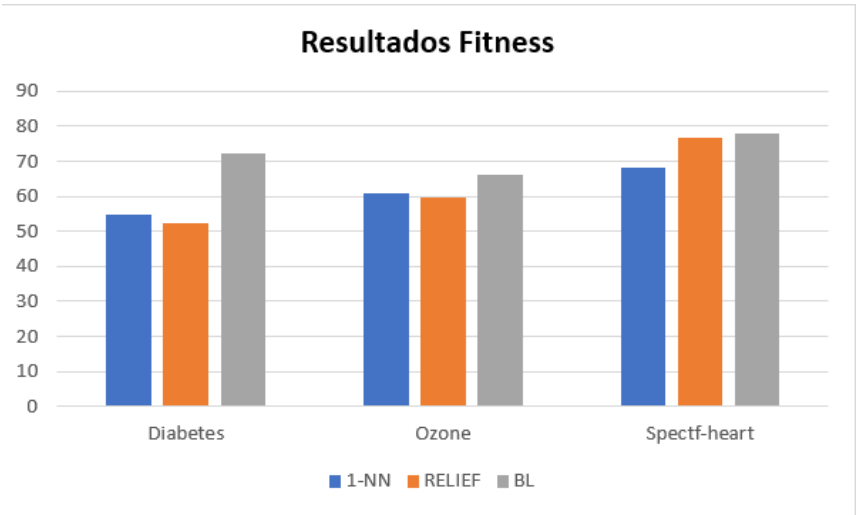
Observamos que, por lo general, la tasa de clasificación media obtenido por los tres algoritmos en cada conjunto de datos es muy **similar** (excepto en el algoritmo Greedy RELIEF para el conjunto de datos de la Diabetes, la cual es muy inferior).

- **Tasa de reducción:**



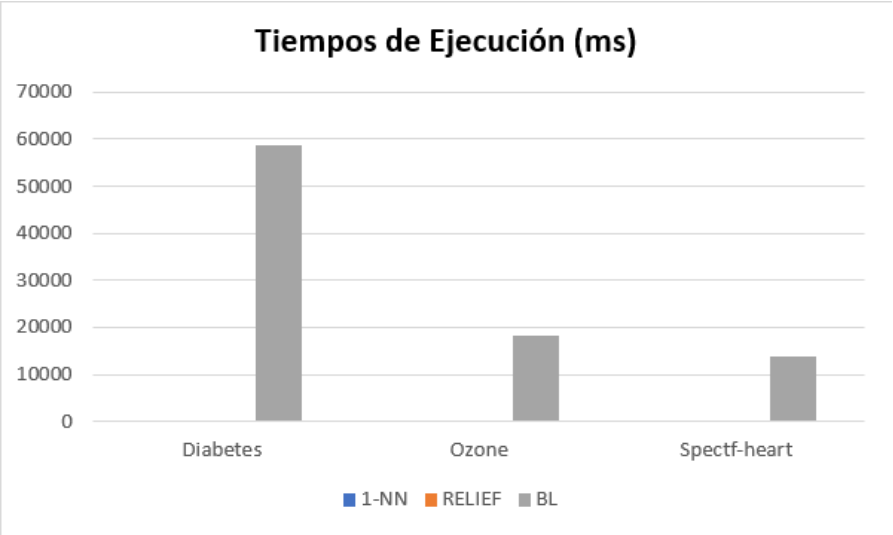
Vemos que se obtienen resultados esperados. Excepto en el conjunto de datos de la Diabetes (donde la tasa de reducción media obtenida por el algoritmo Greedy RELIEF es ligeramente superior a la obtenida por el algoritmo de Búsqueda Local), en los otros dos conjuntos de datos **es superior la obtenida por el algoritmo de Búsqueda Local.**

- **Fitness:**



Se puede observar que el **algoritmo de Búsqueda Local nos proporciona mejores resultados medios de fitness** para los diferentes conjuntos de datos. Esto se debe en gran parte, como hemos visto anteriormente, a la mayor tasa de reducción que nos proporciona este algoritmo con respecto a los otros.

- **Tiempo de ejecución:**



Se puede ver fácilmente que el **algoritmo de Búsqueda Local tarda de media muchísimo más** en ejecutarse con respecto a los otros algoritmos. Este es uno de los mayores inconvenientes que observamos en la Búsqueda Local, ya que, aunque nos proporcione datos muy buenos, el tiempo de ejecución puede llegar a ser desmesurado si tratamos con conjuntos de datos mucho más grandes de los que tenemos.

5.3. Conclusiones

En general, la **Búsqueda Local** nos proporciona mejores resultados (aunque sean óptimos locales, los resultados son suficientemente buenos) en comparación con los otros algoritmos debido a su gran tasa de reducción, pero a cambio de tener un tiempo de ejecución mucho mayor con respecto a los otros algoritmos (aunque siguen siendo tiempos de ejecución viables). Si tuviésemos que reducir el número de características sin importar si la ejecución tarda más (sin llegar a ser un tiempo desmesurado), la mejor opción sería la Búsqueda Local. Sin embargo, si tuviésemos un conjunto de datos muy grande en el que no sea muy necesario la reducción de características, no elegiría la Búsqueda Local por su gran tiempo de ejecución.

Con respecto al **algoritmo Greedy RELIEF**, podemos observar que el fitness que proporciona es muy similar con respecto al proporcionado por el clasificador 1-NN para los conjuntos de datos de Diabetes y de Ozone (al ser un algoritmo voraz, no está garantizada siempre una mejor solución). Sin embargo, para el conjunto de datos de Spectf-heart, este produce una gran mejora del fitness con respecto al clasificador 1-NN. Al permitir que la tasa de reducción no sea 0 (lo cual es lo que hace que varíe con respecto al clasificador 1-NN, donde la tasa de reducción siempre es 0), podemos aumentar la simplicidad (aunque no tanto como con la Búsqueda Local) y, por tanto, mejorar el valor de la función objetivo. Una de las grandes ventajas de este algoritmo, como se puede observar es que su tiempo de ejecución es ínfimo (similar al del clasificador 1-NN).

En conclusión, el algoritmo Greedy RELIEF nos proporciona resultados similares o ligeramente superiores a los del clasificador 1-NN y su tiempo de ejecución es ínfimo (parecido al del clasificador 1-NN), mientras que el algoritmo de Búsqueda Local nos proporciona destacablemente resultados mejores con respecto al clasificador 1-NN y al Greedy RELIEF pero una de sus grandes desventajas es el considerable aumento que produce del tiempo de ejecución.