Práctica 1.b.
Técnicas de Búsqueda Local y Algoritmos Greedy para el Problema del Aprendizaje de Pesos en Características

UNIVERSIDAD
DE GRANADA

Curso 2022/23

Jesús Miguel Rojas Gómez. jesusjrg1400@correo.ugr.es
Subgrupo A2 — Martes 17:30-19:30

Contents

1 Descripción del problema

El problema de Aprendizaje de Pesos en Características (APC) se refiere a la selección y ponderación de características relevantes para mejorar la precisión de los modelos de aprendizaje automático. El objetivo es encontrar una combinación óptima de pesos para cada característica que maximice la precisión del modelo en la tarea de predicción. El problema APC está relacionado con el aprendizaje supervisado, en el que se utiliza un conjunto de datos etiquetados para entrenar un modelo y luego hacer predicciones sobre nuevos datos.

Utilizaremos como clasificador la técnica del vecino más cercano. El clasificador 1-NN, o más general, k-NN (del inglés, *k-Nearest Neighbors*) busca determinar la etiqueta de un objeto desconocido basándose en las etiquetas de los objetos más cercanos a él en el espacio de características (sin tener en cuenta al propio objeto). En un clasificador k-NN, el valor de k representa el número de vecinos más cercanos que se tomarán en cuenta para la clasificación del objeto desconocido. Para medir la similitud entre los objetos, el clasificador k-NN utiliza una función de distancia. En nuestro caso, utilizaremos la función distancia euclídea ponderada, que tiene la siguiente expresión:

$$d(e_1, e_2) = \sqrt{\sum_i w_i (e_1^i - e_2^i)},$$

donde e_j^i corresponde a la característica i del dato j, y w_i corresponde al valor del peso que pondera la característica i.

Evaluaremos el modelo combinando dos criterios: precisión y simplicidad:

- Para medir la precisión, utilizaremos el porcentaje de instancias correctamente clasificadas:

$$tasa\_clas = 100 \cdot \frac{\text{nº de instancias bien clasificadas}}{\text{nº total de instancias}}$$

- Para medir la simplicidad, utilizaremos el porcentaje de características descartadas:

$$tasa\_red = 100 \cdot \frac{\text{nº valores } w_i < 0.1}{\text{nº características}}$$

Para combinar ambos criterios, utilizaremos la siguiente función objetivo:

$$tasa\_fitness = \alpha \cdot tasa\_clas + (1 - \alpha) \cdot tasa\_red,$$

donde α es el parámetro que pondera precisión y reducción. En nuestro caso, utilizaremos $\alpha = 0.8$.

A lo largo de las distintas prácticas de la asignatura, estudiaremos e implementaremos diferentes algoritmos para obtener tales combinaciones de pesos, para después realizar una comparación entre ellos.

2 Descripción de la aplicación y algoritmos comúnes

Para realizar la aplicación y la implementación de los algoritmos utilizaremos Python, debido al manejo que hace este lenguaje de arrays y listas.

2.1 Representación de la solución

Utilizaremos arrays de la biblioteca numpy para representar la solución de APC. Concretamente, al ser un peso un elemento que pondera cada característica del conjunto de datos de entrada, la solución será representada por un array de pesos del tamaño del número de características de los datos, siendo la componente i del vector de pesos la ponderación correspondiente a la característica i de los datos de entrada. Cada componente del vector de pesos pertenecerá al intervalo $[0, 1]$.

2.2 Representación de los datos

Utilizaremos de nuevo arrays de numpy para representar los datos de entrada del problema APC. Para cada base de datos, disponemos de cinco ficheros con formato .arff. Dentro de cada fichero, hemos realizado tres divisiones: nombres de las características, datos, siendo estos las características, y clase. Ambos elementos del fichero serán almacenados en arrays de numpy, por separado. Estos, a su vez, se encontrarán en un array junto con los elementos del resto de ficheros, de nuevo por separado. He decidido realizar esta jerarquía para agilizar el proceso de ejecución y el número de llamadas al fichero main; de este modo, solo necesitaremos realizar una llamada por *dataset*. Por lo tanto, tenemos la siguiente estructura:

```
datosi = [ [...], [...], ..., [...] ]
clasesi = [ tipo1, tipo2, ..., tipo1 ]
...
datos_todos = [datos1, datos2, datos3, datos4, datos5]
clases_todos = [clases1, clases2, clases3, clases4, clases5]
```

Para leer los ficheros, he definido una función `leer_datos`, que operaría de la siguiente manera:

```
def leer_datos(ruta_fichero):
    abrimos ruta_fichero
    por cada linea en fichero:
        si no contiene @data la saltamos
        si contiene @atribute, guardamos el nombre del atributo
        si contiene @data: por cada linea sucesiva:
            dividimos la linea en datos y clase, y la almacenamos

    return nombres, datos, clases
```

Esta función toma como parámetros la ruta del fichero de datos y devuelve los nombres, los datos y las clases del fichero, atendiendo a su formato.

Una vez leídos todos los ficheros, se normalizan los datos de los mismos, de modo que toda característica de cualquier dato se encuentra en el intervalo $[0, 1]$. Para ello, he definido otra función `normalizar_datos`, que opera de la siguiente manera:

```
def normalizar_datos(datos):
    min = valor minimo de todas las caracteristicas
    max = valor maximo de todas las caracteristicas
    por cada dato en datos:
        por cada caracteristica en dato:
            caracteristica = (caracteristica-min) / (max-min)

    return datos_normalizados
```

Esta función toma como parámetro todos los datos y devuelve los datos normalizados. Destacar que esta normalización se realiza teniendo en cuenta todos los ficheros de un mismo *dataset*.

2.3 Funciones comunes

La implementación de esta práctica está organizada en dos archivos: `faux.py` y `P1.py`. El archivo `faux.py` contiene las funciones comunes a los algoritmos implementados, y el archivo `P1.py` contiene los algoritmos implementados y el main.

Dentro del fichero `faux.py`, encontramos varias funciones. Concretamente, están las funciones ya mencionadas `leer_datos` y `normalizar_datos`, y las funciones necesarias para el calculo de la distancia, del clasificador 1-NN y la función objetivo.

Clasificador 1-NN

Para definir el clasificador necesitaríamos definir una función distancia ponderada, pero gracias al lenguaje que estamos utilizando y al manejo de arrays que realiza, no necesitamos explícitamente definir dicha función. Por lo tanto, tenemos el siguiente pseudocódigo:

```
def clasificador1nn(pesos, dato, datos, clases):
    distancias_sin_sumar = pesos x (datos-dato)^2
    # en este punto tenemos todas las caracteristicas de los datos
        con el resultado de la operacion
    distancias_sumadas = suma por filas de distancias_sin_sumar
    distancias = raiz cuadrada de distancias_sumadas
    si hay alguna distancia que es 0, la sustituimos por mas
        infinito

    return de la clase en clases relativa al dato_elegido en datos
        que hace minima la distancia entre dato (parametro) y
        dato_elegido
```

Esta función toma como parámetros un vector de pesos, el dato que queremos clasificar, y los datos y las clases del conjunto donde estamos realizando la comparación. Calcula la distancia ponderada al cuadrado del dato respecto a todos los datos, y devuelve la clase relativa al dato que hace mínima la distancia anterior.

Función objetivo

Como he mencionado en la introducción, la función objetivo es la función `tasa_fitness`. Para ello, necesitamos definir antes las funciones relativas a precisión y simplicidad:

```
def tasa_clas(pesos, datos_clasificar, clases_clasificar,
    datos_train, clases_train):
    por cada dato,clase en datos_clasificar,clases_clasificar:
        si clasificador1nn(pesos,dato,datos_train,clases_train) ==
            clase, contabilizamos un acierto

    return 100 x aciertos / no datos clasificar


def tasa_red(pesos):
    return 100 x no pesos con valor menor que 0.1 / n   pesos
```

La función `tasa_clas` aplica el `clasificador1nn` que hemos definido previamente a cada dato a clasificar, y compara la clase obtenida con la clase verdadera del propio dato. Una vez hecho esto para cada dato, devuelve el porcentaje de aciertos. La función `tasa_red` devuelve el porcentaje del número de pesos con valor menor que 0.1.

Una vez definidas ambas funciones, podemos definir la función objetivo, realizando una ponderación según un valor alpha, que será fijado en el main a 0.8:

```
def tasa_fitness(alpha, pesos, datos_clasificar, clases_clasificar
    , datos_train, clases_train):
    return alpha x tasa_clas(pesos, datos_clasificar,
        clases_clasificar, datos_train, clases_train) + (1-alpha)
        x tasa_red(pesos)
```

Main

Todo lo relativo a la ejecución de los algoritmos y obtención de los resultados se encuentra en el main del archivo `P1.py`.

En primer lugar, leo los datos de los cincos ficheros, los junto en un mismo array, los normalizo y los vuelvo a dividir. En cuanto a las clases, dependiendo del *dataset* las clases tenian un formato diferente, por lo que decidí transformarlas a un valor numérico, ayudándome en la transformación de un diccionario. Una vez hecho esto, fijamos la semilla y con ayuda del módulo tabulate construimos la tabla con los resultados. Por cada algoritmo, realizamos cinco ejecuciones, cada una con un fichero de test distinto, cronometramos la ejecución y calculamos las métricas respecto al fichero test correspondiente. Una vez realizadas las cinco ejecuciones, imprimimos los resultados y pasamos al siguiente algoritmo.

Para cronometrar cada ejecución hemos utilizado la librería timeit.

3 Descripción de los algoritmos implementados

En total, hemos implementado tres algoritmos: *1-Nearest Neighbors*, *Greedy-Relief* y Búsqueda Local.

3.1 *1-Nearest Neighbors*

Según la implementación realizada, este algoritmo ya está implícitamente definido. 1-NN opera realizando la clasificación al vecino más cercano, sin tener en cuenta los pesos al realizar la distancia ponderada. Realizando una llamada a `tasa_fitness` con vector de pesos inicializado al vector cuyas componentes son todas 1 tendríamos el resultado.

3.2 Relief

Este y el siguiente algoritmo tienen como objetivo calcular un vector de pesos que maximice la función objetivo, por lo que el resultado de ambos será dicho vector. Para implementar este algoritmo necesitamos dos conceptos nuevos: amigo más cercano y enemigo más cercano. El amigo más cercano de un dato respecto a un conjunto de datos es el dato de la clase del primero con distancia mínima. El enemigo más cercano de un dato respecto a un conjunto de datos es el dato de la clase contraria al primero con distancia mínima. Para la implementación, he definido dos funciones:

```
def amigo_mas_cercano(dato, clase, datos, clases):
    amigos = datos cuya clase coincide con clase

    distancias_sin_sumar = (amigos-dato)^2
    distancias_sumadas = suma por filas de distancias_sin_sumar
    distancias = raiz cuadrada de distancias_sumadas
    si hay alguna distancias que es 0, la sustituimos por mas
        infinito

    return dato en amigos con distancia minima

def enemigo_mas_cercano(dato, clase, datos, clases):
    enemigos = datos cuya clase no coincide con clase

    distancias_sin_sumar = (enemigos-dato)^2
    distancias_sumadas = suma por filas de distancias_sin_sumar
    distancias = raiz cuadrada de distancias_sumadas

    return dato en enemigos con distancia minima
```

Estas funciones toman como parámetros el dato y la clase de referencia, y el conjunto de datos y clases de donde obtener el amigo o enemigo más cercano. En primer lugar, separamos los datos según necesitemos el amigo o el enemigo más cercano, calculamos las distancias y elegimos el dato con distancia mínima (teniendo en cuenta que si estamos buscando el amigo más cercano, no tengamos en cuenta el propio dato). Destacar también que en estas funciones usamos implícitamente la función distancia, que gracias al lenguaje que estamos utilizando podemos evitar definirla explícitamente.

Una vez definidas, ya podemos implementar el algoritmo:

```
def relief(datos, clases):
    w = vector de ceros
    por cada dato en datos:
        ee = enemigo_mas_cercano(dato, clase del dato, datos,
            clases)
        ea = amigo_mas_cercano(dato, clase del dato, datos, clases
            )
        w = w + |dato - ee| - |dato - ea|

    si w[i] < 0 entonces w[i] = 0
    normalizamos w
    return w
```

Este algoritmo toma como parámetros los datos y las clases relativas a los datos, y devuelve el vector de pesos calculado. Para calcular el vector de pesos, a cada componente se suma la diferencia entre el enemigo mas cercano y el amigo mas cercano. Si en la componente i la diferencia es próxima a cero, significa que esa característica no influye a la hora de clasificar el dato, por lo que el peso no cambiaría de valor. Si la diferencia es considerable, significa que esa característica si es determinante a la hora de realizar la clasificación. Realizando dicha operación por componente y por cada dato, se obtienen qué características podrían ser relevantes y cuáles no.

Una vez calculados los pesos según el algoritmo descrito, estos se utilizan para calcular el desempeño según la función objetivo.

3.3 Búsqueda local

Para la implementación de este algoritmo, necesitamos el concepto de mutación o movimiento. Una mutación de una componente del vector de pesos según un parámetro delta consiste en, a la componente indicada del vector, sumarle una cantidad obtenida según una distribución normal de media cero y desviación típica delta. Una vez sumada dicha cantidad, como los pesos se encuentran en el intervalo $[0, 1]$, se aproxima a los extremos los valores fuera del intervalo. Por lo tanto:

```
def mov(pesos, indice, delta):
    pesos[indice] = pesos[indice] + distribucion_normal(0,delta)
    si pesos[indice] > 1 entonces pesos[indice] = 1
    si pesos[indice] < 0 entonces pesos[indice] = 0
```

Esta función toma como parámetro el vector de pesos, el índice a modificar y el valor de delta, y realiza tal mutación.

Una vez definida esta función, podemos implementar el algoritmo:

```
def busqueda_local(n_max_vecinos, n_max_evaluaciones, delta, datos
    , clases):
```

```
    pesos = vector inicializado segun una distribucion uniforme en
        [0,1]
    vecinos_consecutivos = no de mutaciones consecutivas
        realizadas = 0
    evaluaciones = no de evaluaciones de la funcion objetivo = 0
    tasa_ultima_mejora = tasa_fitness de los pesos inicializados =
        tasa_fitness(0.8, pesos, datos, clases, datos, clases)

    mientras vecinos_consecutivos < n_max_vecinos y evaluaciones <
        n_max_evaluaciones:
        hacemos una copia de los pesos := copia
        elegimos un indice aleatorio := indice
        mov(copia,indice,delta)
        tasa_actual = tasa_fitness con el nuevo vector de pesos
        si se produce mejora:
            actualizamos el valor de los pesos, la tasa de la
                ultima mejora, y el numero de vecinos consecutivos
        vecinos_consecutivos = vecinos_consecutivos+1
        evaluaciones = evaluaciones+1

    return pesos
```

Este algoritmo toma como parámetros el número máximo de vecinos consecutivos a generar, el número máximo de evaluaciones de la función objetivo, el parámetro delta de la distribución normal, los datos y las clases, y devuelve el vector de pesos calculado. Para calcular el vector de pesos, primero se inicializan según una distribución uniforme. Calculamos el valor obtenido de la función objetivo y comenzamos a explorar soluciones. Para ello, elegimos un índice aleatorio sin repetición y realizamos la mutación. Si no se produce mejora, sobre el vector de pesos original, elegimos otro índice y realizamos el mismo proceso hasta que se produce mejora o hasta que no haya índices que mutar. En el caso de que no haya índices que mutar, empezamos de nuevo el proceso de selección de índices. En el caso de que se produzca una mejora, entonces actualizamos el valor de la función objetivo obtenido y comenzamos de nuevo el proceso mutando ahora sobre el último vector de pesos obtenido. Repetimos este proceso hasta que se hayan producido un número máximo de evaluaciones fijado o se hayan obtenido un número fijo de vecinos consecutivos sin mejora.

De nuevo, una vez calculados los pesos, obtenemos el desempeño del algoritmo según la función objetivo.

4 Desarrollo de la práctica y manual de usuario

En mi caso, he desarrollado la práctica desde cero, sin necesidad de utilizar ningún código proporcionado en prácticas.

Para obtener correctamente los resultados, realizaremos la siguiente ejecución:

```
<interprete-python> P1.py <fichero-1> <fichero-2> <fichero-3> <
    fichero-4> <fichero-5> <semilla>
```

El orden de los ficheros determina el orden en el que aparecen los resultados de cada algoritmo. Si el orden es $[1, 2, 3, 4, 5]$, es decir, el que aparece en el recuadro, entonces en cada tabla, la primera linea corresponderá a la partición 1, la segunda a la partición 2, etc.

Para realizar tal ejecución deberemos tener instalado las librerías numpy, tabulate y timeit. Destacar también que he usado un script de windows para realizar las ejecuciones en los tres *datasets*, pero simplemente realizando tres llamadas por separado, una con cada *dataset*, es suficiente.

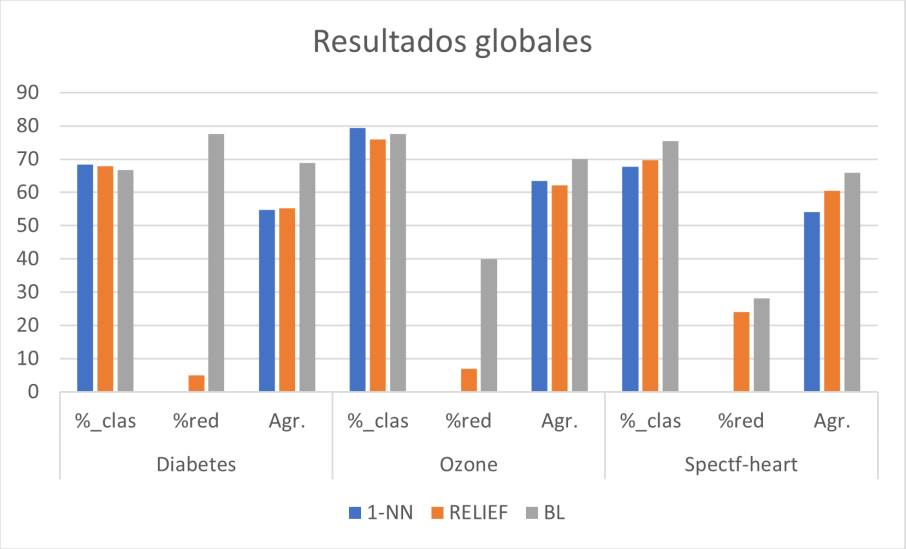
5 Experimentos y análisis de resultados

En esta práctica, el único hiperparámetro utilizado, que no está indicado en el guión, es la semilla, inicializada a 42. A continuación vemos una tabla y dos gráficas con los resultados globales de los algoritmos.

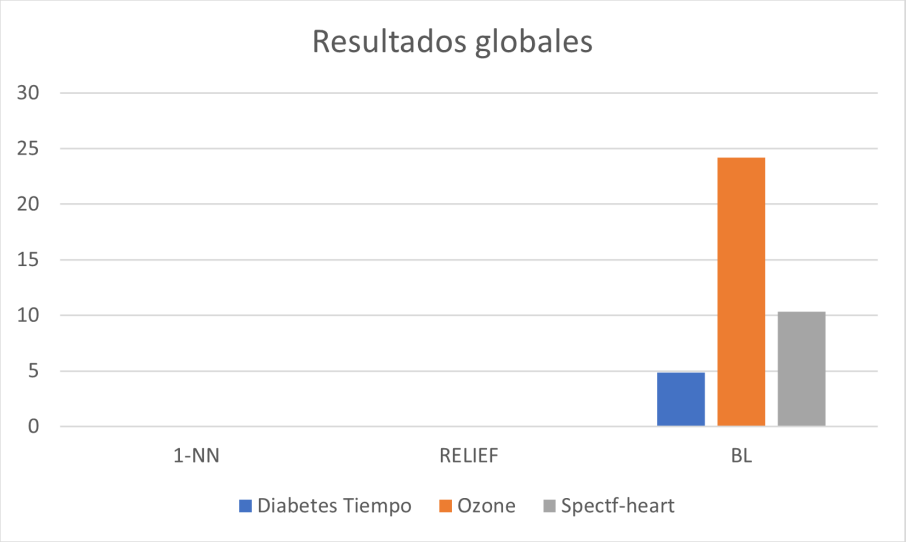
	Diabetes			
	%_clas	***%red***	***Agr.***	**T**
1-NN	68,36	0,00	54,69	0,0034
RELIEF	67,84	5,00	55,27	0,0259
BL	66,66	77,50	68,83	4,8321
	Ozone			
	%_clas	***%red***	***Agr.***	**T**
1-NN	79,38	0,00	63,50	0,0016
RELIEF	75,94	6,94	62,14	0,0114
BL	77,50	40,00	70,00	24,1734
	Spectf-heart			
	%_clas	***%red***	***Agr.***	**T**
1-NN	67,63	0,00	54,10	0,0016
RELIEF	69,62	24,09	60,52	0,0107
BL	75,35	28,18	65,92	10,3418

Table 1: Resultados globales en el problema APC.

En una primera instancia, y como era lógico pensar, 1-NN es más rápido que el resto de algoritmos, debido a su sencillez. Después, el más rápido es Relief, aunque este es aproximadamente diez veces más lento que 1-NN. Y por último, y con bastante diferencia, está Búsqueda Local, que es muchísimo más lento en comparación con el resto. Esto se debe a que Búsqueda Local está constantemente explorando entornos de soluciones para encontrar mejoras en la función objetivo, mientras que el resto simplemente aplican una heurística sobre los pesos, sin tener en cuenta si dicha heurística proporciona un buen resultado en la función objetivo.



(a) Tasas.



(b) Tiempo.

Figure 1: Gráficas resultados globales obtenidos, divididos en tasas de evaluación y tiempo.

Fijándonos en los tiempos de ejecución dentro de cada algoritmo, vemos que tanto en 1-NN como en Relief el tiempo de ejecución en el *dataset* diabetes es mayor que en el resto de *datasets*. Esto se debe a que el número de datos en diabetes es mucho mayor y ambos algoritmos son lineales respecto al número de datos. Por el contrario, el algoritmo de Búsqueda Local depende del número máximo de vecinos a explorar, que este, a su vez, depende del número de características de cada *dataset*. Por lo tanto, el *dataset* diabetes es el más

rápido, le sigue el *dataset* heart y por último está el *dataset* ozone.

Por otro lado, vemos que en cuanto a porcentaje de clasificación, dependiendo del *dataset*, Búsqueda Local puede o no superar al resto. Sin embargo, en cuanto a la tasa fitness, Búsqueda Local siempre es mejor. Esto es debido a que Búsqueda Local obtiene soluciones que mejoren la solución anterior respecto a la función objetivo, es decir, la tasa fitness. La tasa fitness, al depender un 20% del porcentaje de reducción, hace que la búsqueda de soluciones mejores en el algoritmo tienda a encontrar pesos menores que 0.1. Por lo tanto, al ser este algoritmo el único que tiene realmente en cuenta la tasa de reducción, en cuanto a tasa fitness es el mejor. Por el contrario, la tasa de reducción del algoritmo 1-NN siempre es cero (ya que es como si estuviesen todos los pesos inicializados a uno), y la tasa de reducción en el algoritmo Relief es baja (realmente, para que un peso calculado en el algoritmo Relief sea menor que 0.1, debe ocurrir que la característica relacionada al peso en relación al número de datos que tenemos influya muy poco. Explico esto último con un ejemplo. Si para una característica x, de media, el amigo más cercano y el enemigo más cercano distan 0.001, dependiendo del número de datos de los que dispongamos, al realizar el sumatorio del algoritmo, ese peso se tendrá en cuenta para la tasa de reducción).

En cuanto a las ejecuciones por separado de cada algoritmo (las tablas y las gráficas se encontrarán al final del documento), vemos que, en la Tabla 2, relativa al algoritmo 1-NN, no hay prácticamente variabilidad en los resultados. Esto lo podemos comprobar viendo la desviación típica de cada columna de la tabla. Lo mismo ocurre con los resultados de la Tabla 3, relativa al algoritmo Relief. Excepto en la tasa de reducción del *dataset* diabetes, y en la tasa de clasificación del *dataset* ozone, no hay prácticamente variabilidad. Esto se debe a que estos algoritmos únicamente dependen de la partición de los datos que se haya realizado.

Por el contrario, respecto a la Tabla 4, relativa al algoritmo de Búsqueda Local, si hay más variabilidad. Esto se debe a que el algoritmo depende de fenómenos aleatorios, que son, la inicialización de la solución, la elección de qué índice mutar y qué cantidad mutar. Por ejemplo, si la inicialización de la solución no es buena, el algoritmo tardará más en estabilizarse, ya que más a menudo se encontrarán vecinos que mejoren la solución. Sin embargo, si la inicialización ya es muy buena, tardará poco en recorrer los vecinos consecutivos necesarios sin mejora. Y en este ejemplo también influye si el índice seleccionado aleatoriamente realmente influye en la clasificación, y si, a la hora de calcular la cantidad, aleatoriamente se le otorga más o menos peso. Por ejemplo, en el *dataset* ozone, en la partición 4, vemos que el tiempo empleado es muy superior al resto, que hace que se eleve la desviación típica. Esto puede haber ocurrido por lo que acabo de explicar, parte de una solución que no es muy buena y ha coincidido que los índices y la cantidad de la mutación no ha sido beneficiosa (de hecho, en cuanto a fitness, vemos que dicha partición obtiene los peores resultados).

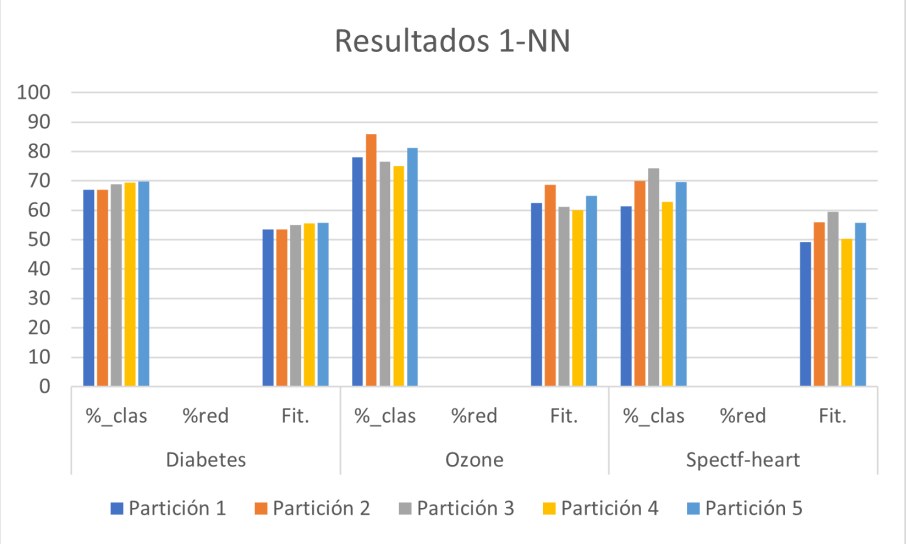
Por último, destacar lo que ocurre con este algoritmo en el *dataset* diabetes. Vemos que, en general, en Búsqueda Local el porcentaje de reducción siempre es mayor, debido a que buscamos soluciones que mejoren la función objetivo. En el *dataset* diabetes, el porcentaje de reducción es mucho mayor que el resto

(hasta el punto que en una ejecución la reducción es del 100%). En realidad, en cuanto a tasa de clasificación, todos los algoritmos obtienen un desempeño parecido. Lo que realmente diferencia la tasa fitness es la tasa de reducción. En el algoritmo 1-NN, los pesos están inicializados a 1. Si en lugar de estar inicializados a 1, estuviesen inicializados todos a un mismo número menor que 0.1 (pero distinto de 0), en cuanto a porcentaje de clasificación el resultado sería el mismo pero la tasa de reducción sería del 100%, incrementando la tasa fitness en un 20%. Esta podría ser la explicación de por qué en diabetes la tasa de reducción es mayor; al ser el número de características menor que el resto y obtener un buen resultado estabilizando los pesos al mismo valor, lo que maximizaría entonces la función objetivo sería estabilizar estos pesos a valores menores que 0.1. En el resto de *datasets* no ocurre esto ya que el número de características es mayor. Al ser mayor el número de características, es menos probable que todas tiendan a ir a valores menores que 0.1.

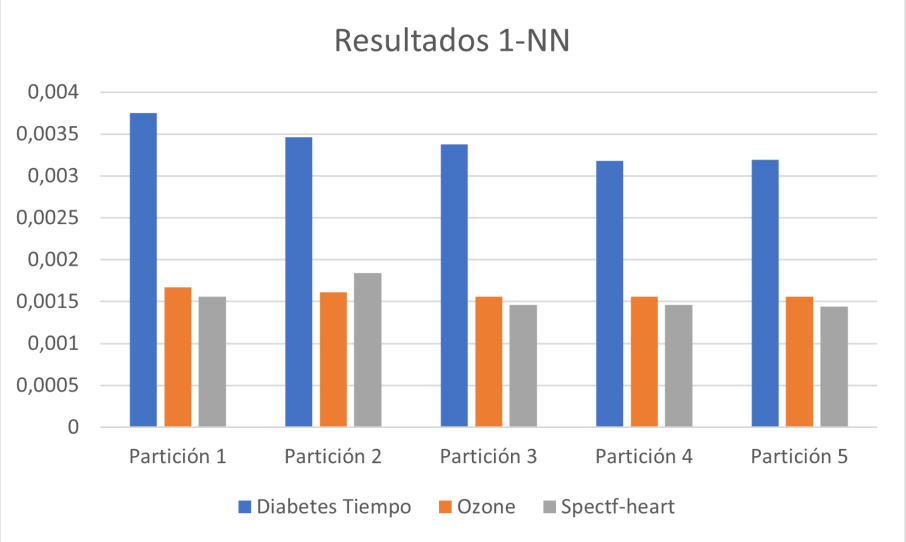
Nota: Soy consciente de que los resultados en el *dataset* Spectf-heart son menores que los que obtienen el resto de mis compañeros, pero no le he encontrado explicación. He repasado las operaciones una a una y con dicho *dataset* no hago nada distinto al resto. El formato del fichero es el mismo que el resto y los datos los lee del mismo modo, y sin embargo en este *dataset* los resultados difieren de mis compañeros y en el resto de *datasets* el resultado es el mismo.

	Diabetes			
	%_clas	***%red***	***Fit.***	**T**
Partición 1	66,88	0,00	53,51	0,00375
Partición 2	66,88	0,00	53,51	0,00346
Partición 3	68,83	0,00	55,06	0,00338
Partición 4	69,48	0,00	55,58	0,00318
Partición 5	69,74	0,00	55,79	0,00319
Media	68,36	0,00	54,69	0,00339
Std	1,24	0,00	1,00	0,00021
	Ozone			
	%_clas	***%red***	***Fit.***	**T**
Partición 1	78,13	0,00	62,50	0,00167
Partición 2	85,94	0,00	68,75	0,00161
Partición 3	76,56	0,00	61,25	0,00156
Partición 4	75,00	0,00	60,00	0,00156
Partición 5	81,25	0,00	65,00	0,00156
Media	79,38	0,00	63,50	0,00159
Std	3,88	0,00	3,10	0,00005
	Spectf-heart			
	%_clas	***%red***	***Fit.***	**T**
Partición 1	61,43	0,00	49,14	0,00156
Partición 2	70,00	0,00	56,00	0,00184
Partición 3	74,29	0,00	59,43	0,00146
Partición 4	62,86	0,00	50,29	0,00146
Partición 5	69,57	0,00	55,65	0,00144
Media	67,63	0,00	54,10	0,00155
Std	4,79	0,00	3,83	0,00015

Table 2: Resultados algoritmo 1NN.



(a) Tasas.

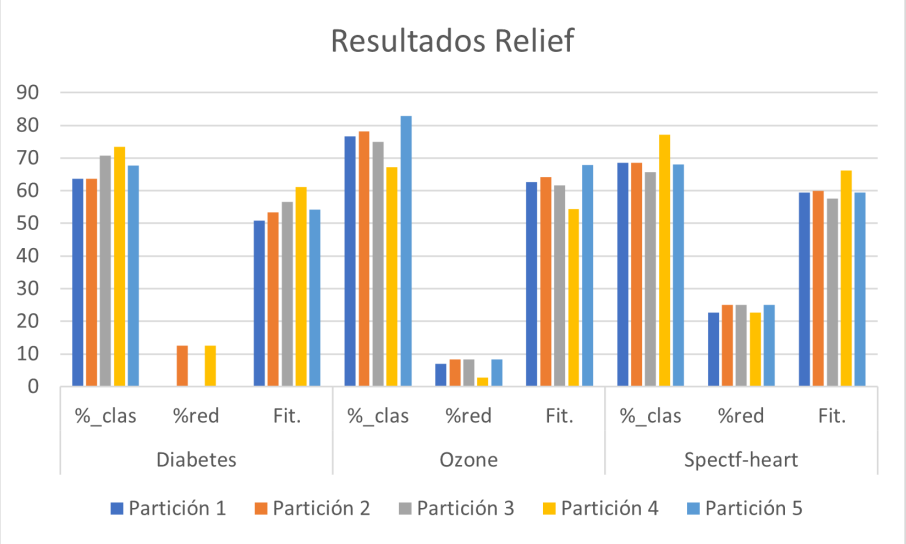


(b) Tiempo.

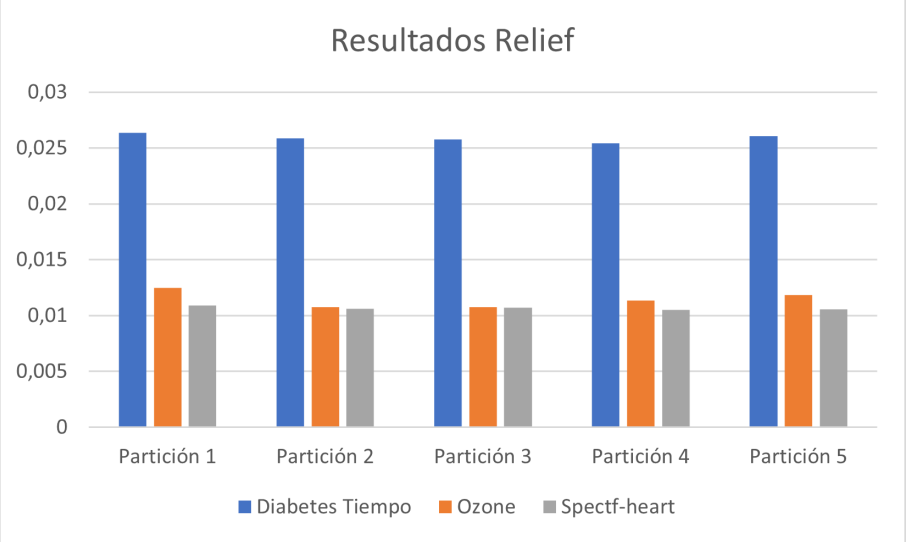
Figure 2: Gráficas resultados 1-NN, divididos en tasas de evaluación y tiempo.

	Diabetes			
	%_clas	***%red***	***Fit.***	**T**
Partición 1	63,64	0,00	50,91	0,02638
Partición 2	63,64	12,50	53,41	0,02589
Partición 3	70,78	0,00	56,62	0,02577
Partición 4	73,38	12,50	61,20	0,02544
Partición 5	67,76	0,00	54,21	0,02605
Media	67,84	5,00	55,27	0,02591
Std	3,86	6,12	3,48	0,00031
	Ozone			
	%_clas	***%red***	***Fit.***	**T**
Partición 1	76,56	6,94	62,64	0,01246
Partición 2	78,13	8,33	64,17	0,01077
Partición 3	75,00	8,33	61,67	0,01076
Partición 4	67,19	2,78	54,31	0,01137
Partición 5	82,81	8,33	67,92	0,01182
Media	75,94	6,94	62,14	0,01144
Std	5,10	2,15	4,46	0,00065
	Spectf-heart			
	%_clas	***%red***	***Fit.***	**T**
Partición 1	68,57	22,73	59,40	0,01090
Partición 2	68,57	25,00	59,86	0,01063
Partición 3	65,71	25,00	57,57	0,01073
Partición 4	77,14	22,73	66,26	0,01052
Partición 5	68,12	25,00	59,49	0,01056
Media	69,62	24,09	60,52	0,01067
Std	3,91	1,11	2,98	0,00014

Table 3: Resultados algoritmo Relief.



(a) Tasas.

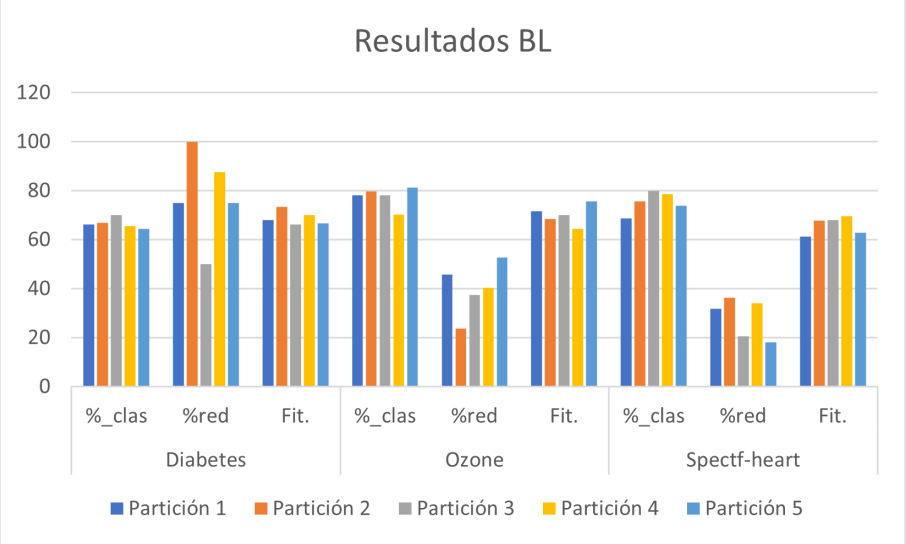


(b) Tiempo.

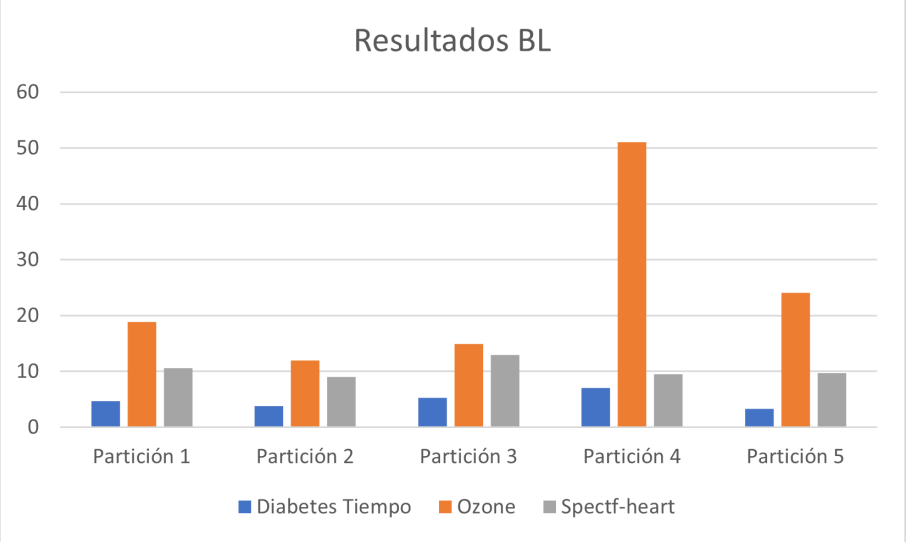
Figure 3: Gráficas resultados Relief, divididos en tasas de evaluación y tiempo.

	Diabetes			
	%_clas	***%red***	***Fit.***	**T**
Partición 1	66,23	75,00	67,99	4,729190
Partición 2	66,88	100,00	73,51	3,786630
Partición 3	70,13	50,00	66,10	5,310150
Partición 4	65,58	87,50	69,97	7,047100
Partición 5	64,47	75,00	66,58	3,287450
Media	66,66	77,50	68,83	4,832100
Std	1,91	16,58	2,70	1,313340
	Ozone			
	%_clas	***%red***	***Fit.***	**T**
Partición 1	78,13	45,83	71,67	18,87460
Partición 2	79,69	23,61	68,47	11,99490
Partición 3	78,13	37,50	70,00	14,91680
Partición 4	70,31	40,28	64,31	51,01420
Partición 5	81,25	52,78	75,56	24,06640
Media	77,50	40,00	70,00	24,17340
Std	3,78	9,72	3,70	14,01800
	Spectf-heart			
	%_clas	***%red***	***Fit.***	**T**
Partición 1	68,57	31,82	61,22	10,57080
Partición 2	75,71	36,36	67,84	8,98582
Partición 3	80,00	20,45	68,09	12,90320
Partición 4	78,57	34,09	69,68	9,53438
Partición 5	73,91	18,18	62,77	9,71486
Media	75,35	28,18	65,92	10,34180
Std	4,00	7,41	3,30	1,37816

Table 4: Resultados algoritmo Búsqueda Local.



(a) Tasas.



(b) Tiempo.

Figure 4: Gráficas resultados 1-NN, divididos en tasas de evaluación y tiempo.