Universidad de Granada

Ingeniería Informática

Computación y Sistemas Inteligentes

Problemas de optimización con técnicas basadas en búsqueda local

Autor: José Antonio Ruiz Millán
email: jantonioruiz@correo.ugr.es
Curso: 2017-2018
Problema: 1.b Aprendizaje de Pesos en Características (APC)
Algoritmos: Greedy (RELIEF) y BL
Grupo: A3
Asignatura: Metaheurísticas
20 de julio de 2018

Índice

1. Descripción del problema

El problema del APC consiste en optimizar el rendimiento de un clasificador basado en vecinos más cercanos a partir de la inclusión de pesos asociados a las características del problema que modifican su valor en el momento de calcular las distancias entre ejemplos. En nuestro caso, el clasificador considerado será el 1-NN (k-NN, k vecinos más cercanos, con k=1 vecino). La variante del problema del APC que afrontaremos busca optimizar tanto la precisión como la complejidad del clasificador. Así, se puede formular como:

Maximizar $F(W) = \alpha \cdot \text{tasa\_clas(W)} + (1-\alpha) \cdot \text{tasa\_red(W)}$

con:

- $\text{tasa\_clas} = 100 \cdot \frac{n^{\text{o}} instancias \quad bien \quad clasificadas \quad en \quad T}{n^{\text{o}} instancias \quad en \quad T}$
- $\text{tasa\_red} = 100 \cdot \frac{n^{\text{o}} valores \quad w_i < 0{,}2}{n^{\text{o}} caracteristicas}$

sujeto a que :

- $w_i = [0, 1], 1 \leq i \leq n$

donde:

- $W = (w_1, ..., w_n)$ es una solución al problema que consiste en un vector de números reales $w_1 \in [0, 1]$ de tamaño n que define el peso que pondera o filtra a cada una de las características f_i.
- 1-NN es el clasificador k-NN con $k = 1$ vecino generado a partir del conjunto de datos inicial utilizando los pesos en W que se asocian a las n características.
- T es el conjunto de datos sobre el que se evalúa el clasificador, ya sea el conjunto de entrenamiento (usando la técnica de validación leave-one-out) o el de prueba.
- $\alpha \in [0, 1]$ pondera la importancia entre el acierto y la reducción de la solución encontrada.

Para la distribución de los datos de entrada utilizaremos la técnica de validación cruzada ***5-fold cross validation*** de la siguiente manera:

- El conjunto de datos se divide en 5 particiones disjuntas al 20 %, con la distribución de clases equilibrada.
- Aprenderemos un clasificador utilizando el 80 % de los datos disponibles (4 particiones de las 5) y validaremos con el 20 % restante (la partición restante) $\rightarrow$ 5 particiones posibles al 80-20 %
- 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

2. Descripción de los algoritmos

2.1. k-NN

Este será el algoritmo básico que utilizaremos para obtener los resultados finales sobre la práctica. Aunque este algoritmo no nos proporciona unos pesos para mejorar la ejecución, lo mencionaré brevemente ya que es la base del problema.

Su funcionalidad es la siguiente:

- El proceso de aprendizaje de este clasificador consiste en almacenar una tabla con los ejemplos disponibles, junto a la clase asociada a cada uno de ellos.
- Dado un nuevo ejemplo a clasificar, se calcula su distancia (usaremos la euclídea) a los n ejemplos existentes en la tabla y se escogen los k más cercanos.
- El nuevo ejemplo se clasifica según la clase mayoritaria de esos k ejemplos más cercanos.
- El caso más simple es cuando k = 1 (1-NN).

Este algoritmo consiste en clasificar un elemento dado, mirando los elementos con los que hemos aprendido. Para ello utlizaremos una distancia que cuanto menor sea, más cercano será ese elemento al elemento que queremos clasificar. En nuestro caso vamos a utlizar la **distancia euclídea** ya que todos los elementos que manejamos son numéricos.

$$d_e(e_1, e_2) = \sqrt{\sum_{i=1}^{n} (e_1^i - e_2^i)^2}$$

Como queremos utilizar unos vectores W para mejorar los resultados, tenemos que unirlos a la funcion de distancia, quedando:

$$d_e(e_1, e_2) = \sqrt{\sum_{i=1}^{n} w_i(e_1^i - e_2^i)^2}$$

Debemos también mencionar que para ajustar aún más los datos, **vamos a normalizarlos** de la siguiente manera:

$$x_j^N = \frac{x_j - Min_j}{Max_j - Min_j}$$

Donde un valor x_j perteneciente al atributo j del ejemplo x y sabiendo que el dominio del atributo j es $[Min_j, Max_j]$.

2.2. Greedy RELIEF

El algoritmo escogido para ser comparado con las metaheurísticas implementadas en esta práctica (la BL) y en las siguientes es el greedy RELIEF, que también deberá ser implementado por el estudiante. El objetivo de este algoritmo será generar un vector de pesos a partir de las distancias de cada ejemplo a su enemigo más cercano y a su amigo más cercano. El enemigo más cercano a un ejemplo es aquel ejemplo de diferente clase más cercano a él. El amigo más cercano a un ejemplo es otro ejemplo de su misma clase más cercano a él.

La intuición del algoritmo es doble: primero se basa en incrementar el peso a aquellas características que mejor separan a ejemplos que son enemigos entre sí; también reduce el valor del peso en aquellas características que separan ejemplos que son amigos entre sí. Este/a incremento/reducción es

directamente proporcional a la distancia entre los ejemplos en cada característica. A continuación, se describe el algoritmo en los siguientes pasos:

- Se inicializa el vector de pesos W a cero.
- Para cada ejemplo del conjunto de entrenamiento, se identifica el enemigo y amigo más cercano. W se actualiza sumándole la distancia existente entre el ejemplo y su enemigo más cercano (considerando todas las características, suma componente a componente). Después, W se actualiza restándole la distancia existente entre el ejemplo y su amigo más cercano.
- Una vez hecho lo anterior para todos los ejemplos, W puede tener componentes que no estén en $[0, 1]$. Los valores negativos de W se truncarán a cero. El resto de valores se normalizarán dividiéndolo por el valor máximo encontrado en W.
- Nótese que el orden de actuación de los ejemplos no altera el resultado final.

Para calcular los pesos utilizando este algoritmo, utilizaremos la distancia euclídea sin pesos ya que si utilizamos los pesos, al estar inicializado a 0, siempre tendríamos 0, por lo que: $d_e(e_1, e_2) = \sqrt{\sum_{i=1}^{n} (e_1^i - e_2^i)^2}$

Tambien comentar que para el cálculo de los procentajes con Greedy, utilizo la función euclidea con pesos, pero teniendo en cuenta que no elimino las carácteristicas que tienen un valor menor que 0.2.

El algoritmo RELIEF, en nuestro caso, será ejecutado 5 veces para cada conjunto de datos de acuerdo a lo explicado anteriormente.

2.3. Búsqueda Local

El algoritmo de BL tiene las siguientes componentes:

- **Esquema de representación:** Se seguirá la representación real basada en un vector W de tamaño n con valores en $[0, 1]$ que indican el peso asociado a cada característica y la capacidad para eliminarla si su peso es menor que 0.2.
- **Función objetivo:** Será la combinación con pesos de las medidas de precisión (tasa de acierto sobre el conjunto de entrenamiento) y la complejidad (la tasa de reducción de características con respecto al conjunto original) del clasificador 1- NN diseñado empleando el vector W. Para calcular la tasa de acierto será necesario emplear la técnica de validación leave-one-out. El valor de α considerado será $\alpha = 0{,}5$, dándole la misma importancia a ambos criterios. El objetivo será maximizar esta función.

  $F(W) = \alpha \cdot \text{tasa\_clas(W)} + (1 - \alpha) \cdot \text{tasa\_red(W)}$
- **Generación de la solución inicial:** La solución inicial se generará de forma aleatoria utilizando una distribución uniforme en [0, 1] en todos los casos.
- **Esquema de generación de vecinos:** Se empleará el movimiento de cambio por mutación normal $Mov(W, \sigma)$ que altera el vector W sumándole otro vector Z generado a partir de una distribución normal de media 0 y varianza σ^2. En nuestro caso, utilizaremos $\sigma = 0{,}3$

$Mov(W, \sigma)$ verifica las restricciones si después de aplicarlo, truncamos el w_i modificado a $[0, 1]$. Así, si la solución original W es factible siempre genera una solución vecina W' factible

- **Criterio de aceptación:** Se considera una mejora cuando se aumenta el valor global de la función objetivo.
- **Exploración del vecindario:** En cada paso de la exploración se mutará una componente $i \in \{1, ..., n\}$ **distinta** sin repetición hasta que haya mejora o se hayan modificado todas las posiciones una vez sin conseguir una mejora. En ese momento, se comienza de nuevo la exploración sobre la nueva solución aceptada (si ha habido mejora) o sobre solución actual (en caso contrario).

En nuestro caso, utilizaremos el algoritmo de **búsqueda primero el mejor**: en cuanto se genera una solución vecina que mejora a la actual, se aplica el movimiento y se pasa a la siguiente iteración.

Para terminar el algoritmo, utilizaremos dos condiciones de parada que serán, o bien cuando hayamos generado 20n vecinos sin éxito sobre el mismo vector de pesos o bien cuando hallamos echo 15000 evaluaciones de la función objetivo.

3. Pseudocodigo algoritmos

3.1. Funciones comunes

Antes de nada, vamos a describir las funciones comunes a todos los algoritmos que vamos a usar.

Búsqueda del mejor vecino

Esta función, como su nombre indica, nos devolverá un vector compuesto por los k vecinos (en nuestro caso $k = 1$) mas cercanos a un elemento.

- Lo primero que hacemos es rellena el vector de vecinos con los primeros vecinos que encontremos que no sean él mismo.
- Una vez tenemos el vector lleno con K elementos, pasamos a recorrer toda la matriz de datos de test y para cada uno de ellos calculamos su distancia respecto al elemento que queremos clasificar.
- Almacenamos el índice del vector de vecinos el peor vecino (el vecino con mayor distancia respecto al elemento).
- Por último comprobamos si el nuevo vecino tiene una distacia menor (mejor) que la de peor vecino, y si es así, lo cambiamos. Teniendo en cuenta que la distancia no puede ser 0.0 porque sería él mismo.

Para el caso del Greedy en el que necesitamos el amigo mas cercano y el enemigo mas cercano, utilizamos la misma función, unicamente cambiamos las condiciones en los If para que la etiqueta

del los vecinos sean igual o distinta al del elemento.

```
input  : datosTraining,elemento,pesos,K
output: vecinos
vecinos ← [0, ..., 0];
while vecinos.size < K do
    datosTraining[i].distancia ← DistanciaEuclídea(datosTraining[i],elemento,pesos);
    if datosTraining[i].distancia ≠ 0,0 then
        vecinos.add(datosTraining[i]);
    end
end
for i ← K to datosTraining.size do
    datosTraining[i].distancia ← DistanciaEuclídea(datosTraining[i],elemento,pesos);
    peorVecino ← 0;
    for j ← 2 to K do
        if vecino[j].distancia > vecino[peorVecino].distancia then
            peorVecino ← j;
        end
    end
    if vecino[peorVecino].distancia > datosTraining[i].distancia and datosTraining[i].distancia
     ≠ 0,0 then
        vecinos[peorVecino] ← datosTraining[i];
    end
end
```

Algorithm 1: BusquedaVecinosCercanos

Clasifica

Dado un vector de vecinos, devuelve la etiqueta que corresponde al elemento que queremos clasificar.

En nuestro caso, como sabemos que $k = 1$ sabemos que el vector de vecinos sólo contendrá el vecino más proximo al elemento, por lo que sólo tenemos que **devolver la etiqueta de éste.**

3.2. k-NN

Dado unos datos de training, testing y unos pesos, nos devolverá los porcentajes y valores que nos pide la práctica sobre los datos de test.

- Primero, calculamos los k vecinos cercanos del elemento que estamos procesando.
- Cuando tenemos los vecinos, clasificamos la etiqueta perdicha y la asignamos al elemento
- Comprobamos si hemos acertado para luego calcular tasa_clas.
- Por último calculamos tasa_clas, tasa_red y el valor de la funcion objetivo para escribirlos en el fichero de salida y obtener los resultados.

```
input : datosTraining,datosTest,pesos,K
aciertos ← 0;
for i ← 1 to datosTest.size do
    elemento ← datosTest[i];
    vecinos ← BusquedaVecinosCercanos(datosTraining,elemento,pesos,K);
    etiqueta ← Clasifica(vecinos);
    elemento.etiquetaPredicha ← etiqueta;
    if elemento.etiquetaPredicha = elemento.etiqueta then
        aciertos ← aciertos+1;
    end
end
tasa_clas ← CalcularTasaClas(aciertos);
tasa_red ← CalcularTasaRed(pesos);
valor ← F(tasa_clas,tasa_red);
EscribirSalida(tasa_clas,tasa_red,valor);
```

Algorithm 2: k-NN

3.3. Greedy RELIEF

Para el greedy, asumimos que la funcion de vecino amigo cercano y vecino enemigo cercano son como he comentado anteriormente. A diferencia de los anteriores, greedy nos devuelve unos pesos para poder calcular los valores que nos pide la practica. También, la **funcion Clasifica** devuelve el vector de atributos en vez de la etiqueta.

He de indicar que aunque en el pseudocodigo haga referencia a datosTest, tanto para el greedy como para BL, **los datosTest son los mismos que datosTraining**.

- Como se indica en la practica, comenzamos con un bulce de 5 que dentro hará todo el proceso.
- Para cada uno de los elementos, buscamos sus k vecinos amigos y sus k vecinos enemigos.
- Casificamos esos elementos para obtener los vectores de caracteristicas de los vecinos mas cercanos.
- Actualizamos los pesos tal y como se indica en la practica.
- Escribimos el tiempo que hemos tardado en realizar este proceso y el propio vector de pesos que hemos obtenido.

```
input  : datosTraining,datosTest,pesos,K,etiqueta
output: pesos[x, ..., n]
for i ← 1 to 5 do
    for j ← 1 to datosTest.size do
        elemento ← datosTest[j];
        vecinosAmigos ← BusquedaVecinosAmigos(datosTraining,elemento,pesos,K,etiqueta);
        valorAmigo ← Clasifica(vecinosAmigos);
        vecinosEnemigos ← BusquedaVecinosEnemigos(datosTraining,elemento,pesos,K,etiqueta);
        valorEnemigo ← Clasifica(vecinosEnemigos);
        for k ← 1 to pesos.size do
            pesos[k] ← pesos[k]+|elemento.atributos[k] - valorEnemigo[k]|;
            pesos[k] ← pesos[k]-|elemento.atributos[k] - valorAmigo[k]|;
            if pesos[k] < 0,0 then
                pesos[k] ← 0;
            end
        end
        max ← Max(pesos);
        for k ← 1 to pesos.size do
            pesos[k] ← pesos[k]/max;
        end
    end
end
EscribirSalida(time,pesos);
return pesos;
```

Algorithm 3: Greedy

3.4. Búsqueda Local

En este algoritmo asumimos la existencia de una varialbe **maxValor** que es el valor de la funcion objetivo sobre los pesos que nos mandan al comienzo. Este paso se realiza antes pero no lo reflejaré en el pseudocodigo ya que es exactamente igual que cualquier paso, sólo que hay que realizarlo antes de comenzar con las iteraciones.

He de indicar que aunque en el pseudocodigo haga referencia a datosTest, tanto para el BL como para greedy, **los datosTest son los mismos que datosTraining**.

- Primero, calculamos los k vecinos cercanos del elemento que estamos procesando.
- Cuando tenemos los vecinos, clasificamos la etiqueta perdicha y la asignamos al elemento
- Comprobamos si hemos acertado para luego calcular tasa_clas.
- Por último calculamos tasa_clas, tasa_red y el valor de la funcion objetivo para escribirlos en el fichero de salida y obtener los resultados.

```
input  : datosTraining,datosTest,pesos,K
output: pesos[x, ..., n]
parada ← 20*pesos.size;
evaluaciones ← 0;
vecinosMal ← 0;
while evaluaciones < 15000 and vecinosMal < parada do
    mejorado ← false;
    for i ← 1 to pesos.size and not mejorado do
        nuevopeso ← calcularPesoNuevo(i,pesos);
        aciertos ← 0;
        for j ← 1 to datosTest.size do
            elemento ← datosTest[i];
            vecinos ← BusquedaVecinosCercanos(datosTraining,elemento,pesos,K);
            etiqueta ← Clasifica(vecinos);
            elemento.etiquetaPredicha ← etiqueta;
            if elemento.etiquetaPredicha = elemento.etiqueta then
                aciertos ← aciertos+1;
            end
        end
        tasa_clas ← CalcularTasaClas(aciertos);
        tasa_red ← CalcularTasaRed(pesos);
        valor ← F(tasa_clas,tasa_red);
        if valor > maxvalor then
            pesos ← nuevopeso;
            maxvalor ← valor;
            mal ← 0;
            mejorado ← true;
        else
            mal ← mal+1;
        end
        evaluaciones ← evaluaciones+1;
    end
end
EscribirSalida(tasa_clas,tasa_red,valor);
return pesos;
```

Algorithm 4: BL

Como hemos visto en el algoritmo anterior, utilizamos la funcion calcularPesoNuevo, que se define como:

```
input  : k,pesos
output: nuevopeso[x, ..., n]
nuevopeso ← pesos;
nuevopeso[k] ← pesos[k]+newGaussian();
nuevopeso[k] ← Truncar(pesos[k]) [0, 1];
return nuevopeso;
```

Algorithm 5: calcularPesoNuevo

4. Descripción para ejecución

En mi caso, he creado un proyecto con NetBeans para esta practica, por lo que adjuntado a este PDF irá un zip con el proyecto completo, con el .jar para poder ejecutarlo.

El proyecto esta compuesto por los siguientes elementos:

- **Carpeta src/calculoPesos**: Donde se encuentran todos los ficheros implementados.
- **Carpeta src/input**: En esta capeta tenemos los 3 ficheros de entrada de la práctica.
- **Carpeta src/output**: Donde se generan los ficheros de salida del programa con la siguiente estructura.

  Se genera un fichero con el nombre de "ficheroEntrada-Algoritmo-train/test-Particion.txt" donde podemos ver por el nombre a qué fichero de entrada pertenecen, el algoritmo utilizado, train o test, donde train indica que son datos de entrenamiento, es decir, devuelve el tiempo y el vector resultado. Y test, que indica que que son datos de test por lo que tendremos los porcentajes de acierto, etc... Por último vemos un numero que indica la partición utilizada como test, por lo que tendremos desde el 0 al 4.

  Para la **ejecución del programa**, debe descomprimir el zip que va adjuntado con la práctica y una vez descomprimido, moverse dentro de él. Podrá encontrar las distintas carpetas y un fichero llamado knn.jar. Abra un terminal y ejecute **java -jar knn.jar**. Comenzará la ejecución y cuando finalice, tendrá los ficheros de salida en **src/output**

5. Análisis de los resultados

5.1. Descripción

En este apartado discutiremos los distintos resultados obtenidos de clasificación. En mi caso, **la semilla utilizada** es 3395.

- **1-NN**

  Para este caso, he utilizado un vector de W inicializado a 1 en todas las componentes (y no cambiarán).

- **Greedy**

  Para este algoritmo, comenzamos con un vector W inicializado a 0 en todas las componentes, este vector se irá modificando como se explica en apartados anteriores y finalmente obtendremos un vector de pesos para ejecutar el K-NN utilizando este vector de pesos.

- **BL**

  Para este caso, se utiliza un vector de W inicializado al azar cada una de sus componentes (valores $\in [0, 1]$). El algoritmo comienza con este vector y lo va modificando hasta finalmente obtener el vector de pesos. Como he indicado anteriormente, **la semilla** que utilizo es 3395 y el valor de $\sigma = 0{,}3$.

5.2. Resultados obtenidos

5.2.1. 1-NN

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

	Ozone				**Parkinsons**				**Spectf-heart**			
	%_clas	**%_red**	**Agr.**	**T**	**%_clas**	**%_red**	**Agr.**	**T**	**%_clas**	**%_red**	**Agr.**	**T**
Particion 1	73.44	0.00	36.72	0.00	76.92	0.00	38.46	0.00	67.14	0.00	33.57	0.00
Particion 2	84.38	0.00	42.19	0.00	82.05	0.00	41.03	0.00	77.14	0.00	38.57	0.00
Particion 3	73.44	0.00	36.72	0.00	100.00	0.00	50.00	0.00	65.71	0.00	32.86	0.00
Particion 4	81.25	0.00	40.63	0.00	66.66	0.00	33.33	0.00	62.86	0.00	31.43	0.00
Particion 5	85.94	0.00	42.97	0.00	74.35	0.00	37.18	0.00	69.57	0.00	34.78	0.00
Media	79.69	0.00	39.84	0.00	80.00	0.00	40.00	0.00	68.48	0.00	34.24	0.00

Donde en mi caso, particion 1 esta compuesta por:

- **Train** → particiones 1,2,3,4.
- **Test** → particion 0.

Para la particion 2 se cambia el test por particion 1 y el train serían las demás. Y así sucesivamente. Esto se aplica a **todos los casos de todas las tablas**.

5.2.2. Greedy

Tabla 2: Resultados obtenidos por el algoritmo Greedy en el problema del APC

	Ozone				**Parkinsons**				**Spectf-heart**			
	%_clas	**%_red**	**Agr.**	**T**	**%_clas**	**%_red**	**Agr.**	**T**	**%_clas**	**%_red**	**Agr.**	**T**
Particion 1	73.44	0.00	36.72	0.21	82.05	0.00	41.03	0.15	67.14	0.00	33.57	0.21
Particion 2	85.94	0.00	42.97	0.08	74.36	0.00	37.18	0.01	78.57	0.00	39.29	0.07
Particion 3	73.44	0.00	36.72	0.07	100.00	0.00	50.00	0.01	72.86	0.00	36.43	0.06
Particion 4	75.00	0.00	37.50	0.07	69.23	0.00	34.62	0.02	72.86	0.00	36.43	0.07
Particion 5	71.88	0.00	35.94	0.07	84.62	0.00	42.31	0.01	65.22	0.00	32.61	0.06
Media	75.94	0.00	37.97	0.10	82.05	0.00	41.03	0.04	71.33	0.00	35.66	0.09

5.2.3. Búsqueda Local

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

	Ozone				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
Particion 1	65.63	81.94	73.78	64.60	66.67	81.82	74.24	2.50	82.86	81.82	82.34	27.69
Particion 2	87.50	70.83	79.17	42.13	84.62	72.73	78.67	1.63	70.00	81.82	75.91	29.86
Particion 3	70.31	83.33	76.8	83.91	97.44	86.36	91.90	1.35	81.43	86.36	83.90	31.50
Particion 4	76.56	86.11	81.34	68.04	56.41	81.82	69.11	1.76	78.57	65.91	72.24	19.14
Particion 5	84.38	77.77	81.07	55.07	76.92	90.90	83.91	1.53	84.06	79.55	81.80	23.81
Media	76.88	80.00	78.84	61.75	76.41	82.73	79.57	1.75	79.38	79.09	79.24	20.40

5.2.4. Búsqueda Local con K=3

Como extra, he decidio hacer un experimento utilizando el algoritmo BL con k = 3 para ver los resultados que obtenemos.

Tabla 4: Resultados obtenidos por el algoritmo BL con k=3 en el problema del APC

	Ozone				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
Particion 1	70.31	70.83	70.57	60.27	82.05	90.91	86.48	2.57	78.57	75.00	76.79	28.57
Particion 2	84.38	80.56	82.47	75.18	82.05	86.36	84.21	1.96	72.86	81.82	77.34	25.16
Particion 3	67.19	80.56	73.87	71.86	97.44	86.36	91.90	2.43	77.14	79.55	78.34	26.81
Particion 4	76.56	83.33	79.95	78.20	64.10	90.91	77.51	1.98	72.86	72.73	72.79	17.27
Particion 5	85.94	75.00	80.47	59.24	76.92	90.91	83.92	3.11	81.16	77.27	79.22	20.56
Media	76.88	78.06	77.47	68.97	80.51	89.09	84.80	2.41	76.52	77.27	76.89	23.67

5.2.5. Tabla resumen

Tabla 5: Resultados globales en el problema del APC

	Ozone				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
1-NN	79.69	0.00	39.84	0.00	80.00	0.00	40.00	0.00	68.48	0.00	34.24	0.00
RELIEF	75.94	0.00	37.97	0.10	82.05	0.00	41.03	0.04	71.33	0.00	35.66	0.09
BL	76.88	80.00	78.84	61.75	76.41	82.73	79.57	1.75	79.38	79.09	79.24	26.40
BL3	76.88	78.06	77.47	68.97	80.51	89.09	84.80	2.41	76.52	77.27	76.89	23.67

5.3. Análisis de resultados

Antes de nada comentar que T mide el tiempo en segundos que se tarda en calcular los pesos, por lo que es obvio que en el algoritmo 1-NN son 0 segundos ya que no tiene que calcular pesos.

Empezamos analizando la tabla 1 que se encuentra en la página 11 y que hace referencia al **algoritmo 1-NN**.

- Para este algoritmo, sabemos que el vector de pesos es 1 para todos los elementos, por lo que contará todas las características con el mismo valor.
- Vemos que para Ozone como para Parkinson tenemos un valor de media del 80 % de tasa_clas, lo que quiere decir que acertamos el 80 % de las etiquetas. Sin embargo, en el caso de Spectf-heart, sólo conseguimos un 68.48 % de media de acierto, que no es un valor demasiado bueno ya que fallamos demasiadas etiquetas. Por lo que finalmente podemos deducir que los elementos que disponemos sobre cada fichero, en Ozone y en Parkinson son algo mas representativos para el aprendizaje que los de Spectf-heart.
- Por otra parte, vemos que tenemos como tasa_res 0.0 %, que es claro ya que todos los elementos del vector de pesos son 1.
- El valor de la funcion objetivo, al tener $\alpha = 0{,}5$ y sabiendo que tasa_res es 0, será la mitad del valor de tasa_clas. Por lo que podemos concluir para la funcion objetivo que tenemos, que estos resultados no son nada favorables para nosotros.

Seguimos ahora analizando la tabla 2 que se encuentra en la página 11 y que hace referencia al **algoritmo Greedy RELIEF**.

- En este caso, comenzamos con los pesos inicializados a 0 pero se ivan actualizando con cada iteración respecto al vecino mas cercano amigo y enemigo. Para este caso, evaluamos todos los valores de los pesos entre $[0, 1]$ por lo que la tasa_red será 0.0 % en todos los casos ya que no la utilizaremos aunque el peso sea menor que 0.2.
- Podemos observar que el algoritmo Greedy mejora un poco el valor de la función objetivo respecto Parkinson y Spectf-heart, pero empeora un poco en Ozone. Estos cambios como podemos observar no son para nada significativos ya que estamos hablando de alrededor de un 2 % de media para cada caso. Podemos concluir, que en nuestro caso, con los datos de entrara que tenemos y los casos que tenemos, el algorimto Greedy RELIEF no mejora demasiado respecto al caso anterior. Pero hay que tener en cuenta que aunque no sean cambios significativos, conseguimos mejorar la tasa de acierto con un tiempo mínimo e insiginificante.
- Por otra parte, vemos que tenemos como tasa_res 0.0 %, como he comentado anteriormente, esto se debe a que no utilizamos esta restriccion para calcular el vector de pesos y por ello no la utilizamos para el calculo final.
- El valor de la funcion objetivo, al tener $\alpha = 0{,}5$ y sabiendo que tasa_res es 0, será la mitad del valor de tasa_clas. Por lo que podemos concluir para la funcion objetivo que tenemos, que estos resultados no son nada favorables para nosotros aunque mejoren minimamente el caso anterior.

El siguiente en analizar será el **algoritmo BL** que tenemos en la tabla 3 que se encuentra en la página 12.

- Para la búsqueda local, comenzamos con un vector de pesos inicializados a $[0, 1]$ cada componente. El algoritmo se encargará de buscar el mejor vector de pesos para mejorar la funcion objetivo. Tendremos en cuenta la restriccion de que una caracteristica con un peso menor que 0.2 "no existe" para la evaluacion de la distancia.
- Para este caso, podemos observar grandes cambios respecto a los algorimos anteriores. Esto es gracias a la propia eliminación de algunas de las caracteristicas, ya que respecto a la funcion objetivo nos dá muchos puntos. No obstante, podemos observar que respecto al Greedy o al 1-NN la tasa_clas no es tan diferente e incluso en algunos casos es peor que los algoritmos anteriores. Esto es simplemente porque buscamos el máximo en la función objetivo, no en tasa_clas, por lo que busca maximizar esa funcion tanto con tasa_clas como con tasa_res.
- En este caso podemos observar como los porcentajes de tasa_res crecen mucho respecto los anteriores (ya que eran nulos) y por ello conseguimos tener finalmente un valor de funcion objetivo razonable. Tenemos que tener en cuenta, que al visualizar tantas posibilidades distintas y tantos vecinos, el tiempo de cálculo aumenta considerablemente respesto a los algorimos anteriores. Es obvio que este método tiene un cálculo computacional más eleveado ya que tiene que explorar muchas posibilidades distintas. No obstante, en mi caso, tampoco es un tiempo excesivamente alto ya que en mi equipo (AMD FX8350 8-cores, 16GB RAM DDR3) tarda tan sólo 1 minuto y medio en hacer todos los cálculos.
- El valor de la funcion objetivo, crece considerablemente respecto los algorimos anteriores, esto se debe a que el algoritmo en cada evaluación que realiza, compara el valor de la función con el que ya tiene y se van quedando con el mejor que va encontrando. Al tener un $\alpha = 0{,}5$, da la misma importancia a la tasa_clas como a la tasa_red, y por eso aumenta tanto. Por lo que finalmente podemos concluir que utilizando el algoritmo BL respecto el algoritmo Greedy y 1-NN para esta función objetivo y estos datos, sería la mejor solución ya que obtenemos una media de 80 % en el valor de la función cuando en los algoritmos anteriores sólo conseguíamos alrededor de un 37 %.

Por último vamos a analizar el **algoritmo BL con k=3** que tenemos en la tabla 4 que se encuentra en la página 12.

- Este caso es exactamente igual que el anterior, con la diferencia de que en vez de obtener sólo en vecino mas cercano y coger su etiqueta, cogeremos los 3 vecinos más cercanos y escogeremos la etiqueta que más representativa sea.
- Podemos observar en la tabla 5 que hace referencia al resumen, que los cambios no son excesivamente altos, incluso vemos que respecto a la funcion evaluacion, sólo mejora en en caso del Parkinson. Ahí tenemos un cambio razonable ya que conseguimos que mejore un 5 % respecto al BL con k=1, pero vemos como en los demás casos empeora el resultado.
- El tiempo de cálculo aumenta mínimamente ya que tenemos que comprobar 3 vecinos en vez de 1, pero no es un cálculo elevado, por ello sólo aumenta un poco.
- Como conclusión, respecto al BL con k=1 podemos ver que en el fichero Parkinson si cojemos k=3 obtenemos mejores resultados, y el los demás no. Podemos deducir que en los datos de Parkinson las distancias entre los elementos de la misma clase son mas cercanas que los de distinta clase ya que si juntamos los 3 mas cercanos a un elemento, conseguimos mejorar le resultado. Sin embargo, en los otros dos casos no ocurre lo mismo y si obtenemos la clase predicha a traves de los 3 vecinos más cercanos, empeoramos el resultado.

Todas las conclusiones se pueden corroborar a través de la tabla 5 en la página 12 donde se podrá ver las distintas medias para cada uno de los algoritmos utilizados y comprobar todos los datos obtenidos.