# METAHEURÍSTICAS

## Problema del Aprendizaje en Pesos de Características


UNIVERSIDAD DE GRANADA
E.T.S.I. INFORMÁTICA Y TELECOMUNICACIÓN

UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial


## Grado en Ingeniería Informática. Curso 2023-2024

Algoritmos 1-NN, Greedy-Relief y Búsqueda Local.


*Miguel Ángel Fernández Alonso*
*76065289-B*
*mfernandez1@correo.ugr.es*
*Grupo 3 - Jueves 17:30h*

# Índice

# 1. Descripción del problema

El problema del APC consiste en la optimización en base al rendimiento de un clasificador de vecinos cercanos a partir de la inclusión de pesos asociados a las características del problema, las cuales modifican su valor en el momento de calcular las distancias entre los ejemplos.
La variante a afrontar busca optimizar tanto la precisión como la complejidad del clasificador.
Se formula como:

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

Partimos de un vector de datos T = {t1,...tn} donde cada dato contiene un conjunto de características, o, “traits” {x1,...,xn}.
Este problema se reduce en calcular un vector de pesos W = {w1,...,wn} dónde el peso wi ∈ [0,1] pondera la característica xi.
Este vector es el vector de aprendizaje, el cual nos permitirá clasificar otro conjunto de datos llamado vector de validación.

Para aumentar la fiabilidad de este proceso usamos el método “5-fold cross validation”, el cual consistirá en crear 5 particiones diferentes de datos repartidos equitativamente según la clase correspondiente.
Para clasificar estos datos implementamos el algoritmo **k-NN** siendo k = 1. Este algoritmo asigna a cada dato su vecino más cercano obtenido a partir de la distancia euclídea.

Calculamos el vector de pesos con los distintos algoritmos y aplicaremos el clasificador **k-NN** para valorar el rendimiento de dichos resultados.
Se define:

$$tasa\_clas = 100 \cdot \frac{n^o\ instancias\ bien\ clasificadas\ en\ T}{n^o\ instancias\ en\ T}$$

$$tasa\_red = 100 \cdot \frac{n^o\ valores\ w_i < 0.2}{n^o\ caracteristicas}$$

Para poder seguir con los cálculos debemos partir de unos datos normalizados. En esta práctica implementamos el algoritmo **Greedy (Relief)** junto al de **Búsqueda Local** y los ejecutaremos sobre un conjunto de datos contenidos en los archivos *breast_cancer, ecoli, parkinsons* todos ellos han sido pasados a formato .csv para una mejora en la lectura.

# 2.Descripción de los Algoritmos Empleados

El esquema de representación de la solución será un vector de pesos W el cual nos ayudará a valorar la bondad de los datos en clasificaciones futuras.
El algoritmo **k-NN** medirá dicha bondad de los pesos calculados mediante cada algoritmo. Este, recibe como parámetro un vector de datos de entrenamiento, un vector de datos de test y el vector de pesos calculados por el algoritmo seleccionado. El clasificador devolverá un *struct Resultados* el cual contendrá las tasas de clasificación y reducción calculadas a partir del número de aciertos entre ambos conjuntos, así como el tiempo empleado para la ejecución.
La tasa de clasificación se calcula como la media de los porcentajes de acierto obtenidos por cada método en cada partición del conjunto de datos. Mayor tasa de clasificación supondrá un mejor vector de pesos *W* generador por el algoritmo.

La tasa de reducción corresponde al porcentaje de reducción obtenido en la selección del subconjunto de características respecto del total. Una tasa de reducción alta indica que necesitaremos menos atributos para clasificar los datos en un futuro.

El pseudocódigo del algoritmo **k-NN** es el siguiente:

```
function KNN(train, test, w)
   for i=0, i < tamaño test, i++ do
      pos = nearestNeighbour(train, test[i], w)
      if train[pos].clase = test[i].clase then
          aciertos++
      end if
   end for
   for i=0, i < tamaño w, i++ do
     if w[i] < 0.2 then
       num_w_menor++
     end if
   end for
  tasa_clas = 100.0*(aciertos / tamaño test)
  tasa_red = 100.0*(num_w_menor / tamaño w)
end function
```

Para calcular el vecino más cercano, calculamos el que tenga menor distancia euclídea respecto del que está siendo valorado. Ha sido implementado aplicando *leave-one-out*, ya que al ser utilizado en la **Búsqueda Local**, no podemos calcular el mejor vecino sin tener en cuenta esto. Estamos aplicando el **k-NN** sobre el mismo conjunto de entrenamiento y podríamos obtener como mejor vecino el mismo datos que estamos comparando.

```
function nearestNeighbour(train, actual, w)
   mejor_distancia = n traits primera partición de train + 1
   for i=0, i < tamaño train, i++ do
      // leave-one-out
      if actual != train[i] then
         // .t accede al vector de caracteristicas
         distancia_actual = euclideanDistance(train[i].t, actual.t, w)
         if distancia_actual < mejor_distancia then
            mejor_distancia = distancia_actual
            pos = i
         end if
      end if
   end for
end function
```

Utilizamos la función *euclideanDistance* para calcular la distancia euclídea entre los diferentes vectores de características teniendo en cuenta los pesos mayores de 0.2

```
function euclideanDistance(v1, v2, w)
   for i=0, i < tamaño v1, i++ do
      if w[i] >= 0.2 then
         dist = dist + w[i]*(v2[i]-v1[i])*(v2[i]-v1[i])
      end if
   end for
end function
```

# 3. Pseudocódigo de los Algoritmos

## 3.1 Algoritmo Greedy (Relief)

El algoritmo se basa en incrementar el peso de las características que mejor separan a ejemplos de enemigos entre sí, y, reducir el peso de aquellas características que separan ejemplos que son amigos entre sí.

El pseudocódigo del algoritmo **Greedy (Relief)** es el siguiente:

```
function Relief(train, w)
   w = {0,...,0}
   for i=0, i < tamaño train, i++ do
      nearestFriendEnemy(train, train[i], friend, enemy)
      for j=0, j < tamaño w, j++ do
         w[j]= w[j]+|train[i].t[j]-train[i].t[enemy]|
               -|train[i].t[j]-train[i].t[friend]|
      end for
   end for
   w_max = máximo del vector de pesos w


    for i=0, i < tamaño w, i++ do
       if w[i] < 0 then
          w[i] = 0
       else
          w[i] = w[i] / w_max
       end if
    end for
 end function
```

La función *nearestFriendEnemy* calcula el amigo y el enemigo más cercano a un dato dado en función de la distancia euclídea. Un dato es considerado como enemigo si tiene la clase distinta y amigo si tiene la misma clase del dato actual. La función emplea *leave-one-out* para evitar comparar distancias con el dato actual.

```
function nearestFriendEnemy(train, actual, friend, enemy)
   for i=0, i < tamaño train, i++ do
      if actual != train[i] then
         distancia_actual = euclideanDistance(train[i].t, actual.t)
         if train[i].category = actual.category then
            if distancia_actual < mejor_distancia_a then
               mejor_distancia_a = distancia_actual
               friend = i
            end if
         end if
      else
         if distancia_actual < mejor_distancia_e then
            mejor_distancia_e = distancia_actual
            enemy = i
         end if
      end if
   end for
end function
```

### 3.2 Búsqueda Local

La búsqueda local implementa una búsqueda de primero el mejor. El vector index nos indica en qué orden se van a modificar las componentes, modificando así, una componente aleatoria en cada paso que no se haya modificado antes.
El vector de pesos se generará de forma aleatoria con valores comprendidos entre [0,1] utilizando una distribución uniforme real.

Para poder generar soluciones nuevas debemos modificar el vector de pesos añadiendo a cada elemento un valor que siga una distribución normal de media 0 y varianza al cuadrado. Esto puede proporcionar soluciones negativas, por lo que debemos truncar los valores negativos a 0.

El pseudocódigo del algoritmo **Búsqueda Local** es el siguiente:

```
function BL(train, w, semilla)
   w = distribucion_uniforme(0,1)
   index = {0,...,w.size()}

   mezcla los valores de index

   // clasifica el vector de pesos w con KNN y
   // calcula su agregado como tasa de evaluación
   antiguo = KNN(train, train, w)
   agr_ant = agregado(antiguo.clas,antiguo.red)

   while iter < MAX_ITER and neighbour < tamaño w*MAX_NEIGHBOUR do
      aux = index[iter % w.size()]
      w_mut = w
      w_mut[aux] = w_mut[aux] + normal(generator)
      truncar el vector de pesos mutado
      //clasifica el vector de pesos mutado
      // y calcula su agregado
      nuevo = KNN(train, train, w_mut)
      agr_new = agregado(nuevo.clas,nuevo.red)
      iter++
      if agr_new > agr_ant then
         w = w_mut
         agr_ant = agr_new
         neighbour = 0
      else
         neighbour++
      end if
      if iter % tamaño w = 0 then
         mezcla los valores de index
      end if
   end while
end function
```

La función agregado calcula la tasa de agregado de los resultados obtenidos al clasificar los datos de entrenamiento con el vector de pesos en cada caso, y, utiliza este resultado para evaluar la bondad de la solución obtenida. Usamos alpha = 0.5.

```
function agregado(t_clas, t_red)
  alpha*t_clas+(1.0-alpha)*t_red
end function
```

# 4. Descripción en Pseudocódigo de los Algoritmos de Comparación

El proceso para comparar resultados obtenidos para cada algoritmo es similar y se realiza en la función *ejecutar* para cada algoritmo implementado.

Primero obtenemos los pesos usando el algoritmo a comparar, y, seguidamente clasificamos los datos de entrenamiento y test con **k-NN** sobre el vector de pesos obtenido. Devolvemos finalmente usando un *struct Resultados* las tasas de la clase, reducción, agregado y el tiempo que ha tardado en ejecutar y repetimos el proceso cambiando el índice de la partición de test.

```
function ejecutarAlgoritmo(particion, i)
   test = toma la partición i
   train = toma la suma de datos de las particiones !=
   w = Algoritmo(train, w)
   resultados = KNN(train, test, w)
end function
```

# 5. Procedimiento del Desarrollo de la Práctica y Manual del Usuario

La práctica ha sido desarrollada en el lenguaje C++. Para el desarrollo de la práctica he necesitado leer los archivos que proporcionan los datos, para esto, he optado por pasar los archivos .arff a .csv, un formato más manipulable. Con la ayuda de la función *read_csv*, guardo los datos en memoria en un vector de estructuras denominado *FicheroARFF.* Cada fichero contiene la información de cada línea del fichero csv original incluyendo un vector de los datos y un string que indica la clase de los mismos. Consideramos, por tanto, cada conjunto de datos como un vector<*FicheroARFF*>.

Para aplicar *5-fold cross validation* creo un vector con 5 particiones que contienen punteros (se reduce el costo de memoria) a estructuras *FicheroARFF*. Los datos son introducidos en la partición ya normalizados.

Una vez tenemos el vector<vector<FicheroARFF*>> ya repartimos los datos en train y test y procedemos.

Primero fue desarrollada una versión del **k-NN** sin *leave-one-out*, luego implementé el algoritmo **Greedy (Relief)** y finalmente la **Búsqueda Local**, tuve que modificar el algoritmo **k-NN** para poder aplicar *leave-one-out*.

Dentro del archivo *LEEME.txt* viene el proceso para replicar las ejecuciones del programa explicado.

Existe un make el cual genera el ejecutable del programa. Este programa ha sido compilado con g++ y optimizado con la opción -O2 para reducir tiempos de ejecución.

./bin/p1 ./data/archivo.csv seed
Ejecuta los tres algoritmos implementados en esta práctica sobre el conjunto de datos contenido en el archivo .csv y usando como semilla el número *seed* pasado como parámetro en el **Búsqueda Local**.

En un principio implementé también para poder ser ejecutado el programa con el comando ./bin/p1 seed pero daba *Exit Code 1* y no logré identificar el error. Este comando creaba un archivo *tablas_seed.csv* sobre los conjuntos de datos y volcaba su contenido en una tabla.

# 6. Experimentos y Análisis de Resultados

El algoritmo de **Búsqueda Local** depende de un parámetro que especifica la semilla para la generación de números aleatorios. He utilizado para el análisis de resultados y para los experimentos la semilla SEED=23.

Para la generación de vecinos y mutación se van a usar los datos generados por una distribución normal de media 0 y varianza = 0.3. Como criterio de parada en la **Búsqueda Local** se va a usar el número de evaluaciones realizadas así como el número de vecinos por características explorados. Los valores serán 15000 y 20 · *tam_vector_pesos* respectivamente.

A continuación se muestran las tablas para cada fichero y cada algoritmo utilizado:

Tabla 5.2: Resultados globales en el problema del APC

| | Ecoli | | | | Parkinsons | | | | Breast-cancer | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *% clas* | *%red* | *Agr,* | T | *% clas* | *%red* | *Agr,* | T | *% clas* | *%red* | *Agr,* | T |
| 1-NN | 75,926 | 0,00 | 37,964 | 1,00E-04 | 77,858 | 0,00 | 38,928 | 1,00E-04 | 92,684 | 0,00 | 46,396 | 1,00E-04 |
| RELIEF | 74,21 | 25 | 49,61 | 1,00E-04 | 77,43 | 0,17 | 38,8 | 1,00E-04 | 92,96 | 0,13 | 46,55 | 1,00E-04 |
| BL | 66,22 | 73,00 | 69,61 | 5,16E-03 | 79,64 | 81,04 | 80,34 | 1,53E-02 | 92,11 | 85,03 | 88,57 | 3,12E-01 |

Analizando los resultados podemos observar que los mejores resultados son obtenidos por el algoritmo **Greedy (Relief)** igualando al **1-NN** sobre el conjunto de datos *ecoli*. Mientras que el **Búsqueda Local** es el que obtiene en los tres casos la mejor tasa de reducción, esto indica que se necesitan menos atributos para clasificar los datos con respecto al resto de algoritmos.

A continuación muestro los resultados totales obtenidos para cada conjunto de datos:

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

| | Ecoli | | | | Parkinsons | | | | Breast-cancer | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *% clas* | *%red* | *Agr,* | T | *% clas* | *%red* | *Agr,* | T | *% clas* | *%red* | *Agr,* | T |
| Partición 1 | 78,57 | 0,00 | 39,29 | 1,00E-04 | 72,50 | 0,00 | 36,25 | 1,00E-04 | 97,39 | 0,00 | 48,70 | 1,00E-04 |
| Partición 2 | 67,14 | 0,00 | 33,57 | 1,00E-04 | 77,50 | 0,00 | 38,75 | 1,00E-04 | 96,52 | 0,00 | 48,26 | 1,00E-04 |
| Partición 3 | 76,59 | 0,00 | 38,30 | 1,00E-04 | 80,00 | 0,00 | 40,00 | 1,00E-04 | 82,09 | 0,00 | 41,30 | 1,00E-04 |
| Partición 4 | 80,66 | 0,00 | 40,33 | 1,00E-04 | 85,00 | 0,00 | 42,50 | 1,00E-04 | 95,65 | 0,00 | 47,83 | 1,00E-04 |
| Partición 5 | 76,67 | 0,00 | 38,33 | 1,00E-04 | 74,29 | 0,00 | 37,14 | 1,00E-04 | 91,77 | 0,00 | 45,89 | 1,00E-04 |
| Media | 75,93 | 0,00 | 37,96 | 1,00E-04 | 77,86 | 0,00 | 38,93 | 1,00E-04 | 92,68 | 0,00 | 46,40 | 1,00E-04 |

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

| | Ecoli | | | | Parkinsons | | | | Breast-cancer | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *%_clas* | *%red* | *Agr,* | T | *%_clas* | *%red* | *Agr,* | T | *%_clas* | *%red* | *Agr,* | T |
| **Partición 1** | 74,29 | 25,00 | 49,64 | 1,00E-04 | 70,00 | 0,00 | 35,00 | 1,00E-04 | 96,52 | 0,00 | 48,26 | 1,00E-04 |
| **Partición 2** | 62,86 | 25,00 | 43,93 | 1,00E-04 | 77,50 | 0,87 | 39,18 | 1,00E-04 | 97,39 | 0,65 | 49,02 | 1,00E-04 |
| **Partición 3** | 75,16 | 25,00 | 50,08 | 1,00E-04 | 85,00 | 0,00 | 42,50 | 1,00E-04 | 82,61 | 0,00 | 41,30 | 1,00E-04 |
| **Partición 4** | 82,09 | 25,00 | 53,54 | 1,00E-04 | 77,50 | 0,00 | 38,75 | 1,00E-04 | 96,52 | 0,00 | 48,26 | 1,00E-04 |
| **Partición 5** | 76,67 | 25,00 | 50,83 | 1,00E-04 | 77,14 | 0,00 | 38,57 | 1,00E-04 | 91,77 | 0,00 | 45,89 | 1,00E-04 |
| **Media** | 74,21 | 25,00 | 49,61 | 1,00E-04 | 77,43 | 0,17 | 38,80 | 1,00E-04 | 92,96 | 0,13 | 46,55 | 1,00E-04 |

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

| | Ecoli | | | | Parkinsons | | | | Breast-cancer | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *% clas* | *%red* | *Agr,* | T | *% clas* | *%red* | *Agr,* | T | *% clas* | *%red* | *Agr,* | T |
| **Partición 1** | 62,86 | 72,50 | 67,68 | 6,40E-03 | 75,00 | 81,74 | 78,37 | 1,64E-02 | 93,91 | 85,16 | 89,54 | 2,95E-01 |
| **Partición 2** | 58,57 | 72,50 | 65,54 | 4,60E-03 | 75,00 | 81,74 | 78,37 | 1,50E-02 | 92,17 | 85,16 | 88,67 | 2,95E-01 |
| **Partición 3** | 60,33 | 72,50 | 66,41 | 5,40E-03 | 80,00 | 79,13 | 79,57 | 1,64E-02 | 86,96 | 82,58 | 84,77 | 3,33E-01 |
| **Partición 4** | 80,99 | 70,00 | 75,49 | 5,80E-03 | 82,50 | 80,87 | 81,68 | 1,54E-02 | 94,78 | 83,87 | 89,33 | 3,41E-01 |
| **Partición 5** | 68,33 | 77,50 | 72,92 | 3,60E-03 | 85,71 | 81,74 | 83,73 | 1,34E-02 | 92,73 | 88,39 | 90,56 | 2,94E-01 |
| **Media** | 66,22 | 73,00 | 69,61 | 5,16E-03 | 79,64 | 81,04 | 80,34 | 1,53E-02 | 92,11 | 85,03 | 88,57 | 3,12E-01 |

Vemos que el que mayor tiempo de ejecución necesita es el **Búsqueda Local**, lo cual, es evidente ya que realiza más evaluaciones e iteraciones. El conjunto de datos *breast_cancer* es el que tarda más pero es el que mayor tasa de clasificación obtiene mientras que el segundo con mayor tiempo son ambos *ecoli* y *parkinsons*, este último supera su tasa de clasificación en un 13%.

# 7. Bibliografía

Guión de Prácticas y Seminarios de la Asignatura de Metaheurísticas