# APRENDIZAJE DE PESOS EN CARACTERÍSTICAS

Javier Sáez Maldonado

*77448344F*
*fjaviersaezm@correo.ugr.es*
*Universidad de Granada*

2019

## Nota previa

La ejecución del programa que he escrito en python es *demasiado* lenta. El tiempo medio de ejecución completa es de 25 minutos en mi ordenador portátil que dispone de *Intel Core i7-4510U CPU @ 2.00GHz x 4* y el sistema operativo Manjaro Linux. Es por ello que se recomienda que no se ejecute el programa o se modifique si se quiere comprobar el funcionamiento.

# 1 Introducción

El problema de clasificación consiste en, dado un conjunto $A = \{(a, b) : a \in R^n, b \, es \, una \, clase\}$ de datos ya clasificados, obtener un sistema que permita clasificar un objeto nuevo de forma automática.

Un ejemplo de clasificador, y el que utilizaremos en esta práctica,es el $k-NN$ , $k$ vecinos más cercanos. Este toma la clase que más se repita entre los $u_i \in A$ tales que su distancia al nuevo elemento $u$ sea mínima. En nuestro caso, en una versión sencilla del problema, consideraremos el clasificador $1 - NN$.

Consideraremos como distancias la distancia trivial si las características son discretas (esto es, la distancia será 1 si las características son diferentes, y 0 si son iguales. La denotamos como $d_n$), y la distancia euclídea para características que sean continuas. Además, cada característica tendrá un peso asociado, por lo que dado un vector de pesos $w$, la distancia entre dos vectores $u$ y $v$ será de la forma:

$$d(u, v) = \sqrt{\sum_i w_i(u_i - v_i)^2 + \sum_j w_j d_n(u_j, v_j)}$$

El aprendizaje de pesos en características consiste en hallar un vector de pesos que maximice la siguiente función:

$$F(w) = \alpha T_{clas}(w) + (1 - \alpha)T_{red}(w)$$

Donde

- $T_{clas}$ es la función que indica cómo de bueno es nuestro clasificador, es decir, cuántos casos ha clasificado correctamente si entrenamos el clasificador usando el resto de datos ,la técnica *k-fold cross validation*, y dejando un elemento fuera (leave one out).
- $T_{red}$ que es la función que nos indica cuántas características de un dato tienen un peso menor que un valor establecido, en nuestro caso 0.2.

# 2 Descripción de la aplicación de los algoritmos

Vamos a describir las consideraciones, tipos y operaciones comunes en los algoritmos de nuestra práctica.

## 2.1 Esquemas de representación

Los datos de entrada que tenemos para nuestro problema tienen los siguientes elementos:

- Vector de características, que es un vector de valores reales que trataremos de normalizar al intervalo $[0, 1]$ para trabajar con ellos.
- Clase del elemento, que es la categoría a la que corresponde el mismo
- Ejemplo, que es un par que tiene un vector de características y una clase
- Conjunto de datos, que contendrá una lista de ejemplos

De ellos , obtendremos una solución que será un vector $w$ de pesos, valores reales también en el intervalo $[0, 1]$

## 2.2 Operadores comunes

Vamos a describir ahora dos funcionalidades comunes a todos los algoritmos que hemos utilizado para la realización de la práctica.

La función *normalizeData* se encarga de dado un vector de datos (que puede ser de tamaño 1), normalizarlos al intervalo $[0, 1]$

La función *weighted_onenn* es la función que nos da el vecino más cercano a otro utilizando el algoritmo $1-NN$, usando un vector de pesos dado. La utilizaremos dando diferentes vectores de pesos $w$ para obtener diferentes resultados de clasificación.

El resto de funcionalidades que se han utilizado se han hecho en cada algoritmo concreto, debido a la facilidad que nos da el lenguaje de programación escogido para la programación de estos operadores.

## 2.3 Función objetivo

En nuestro caso, se nos indica que tomemos como *alpha* el valor 0.5, así que en realidad lo que estamos haciendo es:

$$F(w) = 0.5(T_{clas}(w) + T_{red}(w))$$

Calcularemos $T_{class}$ de la siguiente manera:

```
1: procedure T-CLASS(guess,classes)
2:     n = 0
3: loop : i = 0, ..., guess.size
4:     if classes(i) = guess(i) then n ← n + 1
5: endloop
6:     return n/guess.size
```

Y calcularemos $T_{red}$ así:

1: **procedure** T-RED(w)
2: n = 0
3: *loop* : $i = 0, ..., w.size$
4: **if** w(i) $< 0.2$ **then** $n \leftarrow n + 1$
5: *endloop*
6: **return** n/w.size

# 3 Estructura del método de búsqueda

## 3.1 Algoritmo Greedy Relief

Este algoritmo recorre todo el conjunto de datos punto por punto y , en cada uno, modifica el vector de pesos según el ejemplo enemigo y amigo más cercanos. Lo que haremos será , por cada punto, partir el conjunto de datos en un conjunto de datos "amigos" (que tienen su misma clase), y "enemigos" (que no tienen su misma clase), y luego obtendremos de cada uno el "vecino más cercano". El procedimiento para obtener el conjunto de enemigos(y el de amigos, pues son idénticos) es el siguiente:

1: **procedure** GETENEMIES(data,dClasses,example,exampleClass)
2: *loop* : $i = 0, ..., data.size$
3: **if** dClasses(i) = exampleClass **then** $enemies \leftarrow data(i)$
4: **return** enemies

Al final, implementamos la función *greedyRelief*:

**Algorithm 1** Greedy Relief

1: **procedure** GREEDYRELIEF(data, classes)
2: $w \leftarrow 0$
3: *loop*: $i = 0, ..., data.size$
4: $friends \leftarrow getFriends$
5: $enemies \leftarrow getEnemies$
6: $closestFriend \leftarrow min(distance(friends, data(i))$
7: $closestEnemy \leftarrow min(distance(enemies, data(i))$
8: $w \leftarrow w + |data(i) - closestEnemy| - |data(i) - closestFriend|$
9: *endloop*
10: $w \leftarrow normalize(w)$
11: **return** w

### 3.2 Búsqueda local

Para este algoritmo, debemos definir el algoritmo que hemos usado para obtener una mutación de un ejemplo. Al mutar, sumaremos a la componente $j$ un valor aleatorio y truncaremos al intervalo $[0, 1]$ si el nuevo valor se nos escapara del intervalo:

1: **procedure** MOV(w,sigma,j)
2: $w(j) \leftarrow w(j) + random(0, 1)$
3: $w(j) \leftarrow Normalize(w(j))$
4: **return** w

En el procedimiento de cálculo de pesos mediante la búsqueda local intervendrá la función de evaluación, que será notada por $f(w)$.
Así, el procedimiento general para la generación de pesos para la búsqueda local sería:

**Algorithm 2** Local Search

1: **procedure** LOCALSEARCH(initialWeight,data,classes)
2: $w \leftarrow initialWeight$, $initialF \leftarrow f(w)$,
3: *loop* i = 0,...,data.size
4: $copy \leftarrow w$
5: $copy \leftarrow mov(copy, i, sigma)$
6: $newF \leftarrow f(copy)$
7: **if** newF > initialF **then**
$initialF \leftarrow newF$
$w \leftarrow copy$
8: *endloop*
9: **return** w

Para completar este algoritmo, habría que añadir un contador para que contara un número de iteraciones máximo que se pudiera hacer para generar un número máximo de vecinos , pero esto no se ha incluido en el pseudocódigo.

## 4 Procedimiento considerado para desarrollar la práctica

Para desarrollar la práctica, he usado **Python** como lenguaje de programación, sin usar ningún framework de metaheurísticas.
Para poder ejecutar el código , hace falta tener instalado *Numpy,Scipy* y *Sklearn*. Este último es muy útil para la realización de prácticas de este estilo pues trae implementaciones de muchas funcionalidades básicas para *Machine learning*.
El fichero que hay que ejecutar dentro de la carpeta es el fichero ***apc.py***. Para ello, basta con escribir en la terminal:

*python apc.py*

Tras la ejecución, comenzará a ejecutar los algoritmos sobre los 3 ficheros de datos que tenemos, que se explicarán más adelante.
La lista de archivos que contiene la práctica son los siguientes:

- **apc.py**, el fichero principal a ejecutar para la ejecución de nuestro programa.
- **prepareData.py**, el fichero con las funciones que se encargan de la manipulación de los datos y la lectura para poder trabajar con ellos cómodamente.
- **algorithms**, el fichero en el que se encuentran los algoritmos y algunas funciones auxiliares programadas para la práctica.
- **Datasets**, una carpeta en la que se encuentran los datasets almacenados.

# 5 Experimentos y análisis de resultados

## 5.1 Casos del problema

Vamos a comentar primero los ficheros que tenemos a analizar mediante nuestros algoritmos.

### 5.1.1 Colposcopy

La colposcopia es un procedimiento ginecológico que consiste en la exploración del cuello uterino. El conjunto de datos fue adquirido y anotado por médicos profesionales del Hospital Universitario de Caracas. Las imágenes fueron tomadas al azar de las secuencias colposcópicas. Este archivo tiene 287 ejemplos, de 62 atributos cada uno y dos clases (positivo o negativo)

### 5.1.2 Ionosphere

Son los datos de radar recogidos por un sistema en Goose Bay, Labrador. Este sistema consiste en un conjunto de fases de 16 antenas de alta frecuencia con una potencia total transmitida del orden de 6.4 kilovatios Los objetivos son electrones libres en la ionosfera. Los "buenos" retornos de radar son aquellos que muestran evidencia de algún tipo de estructura en la ionosfera.
Así, este conjunto de datos consta de 352 ejemplos, con 34 atributos cada uno y acompañados de dos clases, buenos y malos.

### 5.1.3 Texture

El objetivo de este conjunto de datos es distinguir entre 11 texturas diferentes, caracterizándose cada pixel por 40 atributos construidos mediante la estimación de momentos modificados de cuarto orden en cuatro orientaciones: 0,45,90 y 135 grados.
Tiene este archivo 550 ejemplos de 40 atributos y consta, como ya hemos mencionado, de 11 clases diferentes, los tipos de textura.

## 5.2 Resultados

Vamos a comentar los resultados que se han obtenido. Hay que indicar que he redondeado a dos decimales. Introduzco ahora las siglas de la tabla:

- $T_{class}$ es la tasa de clasificación, porcentaje de acierto

- $T_{red}$ es la tasa de reducción
- $Agr$ es el valor de la función objetivo con $\alpha = 0.5$
- $T$ es el tiempo en segundos

Ahora, los resultados obtenidos son los siguientes:

**1nn**

| Partition | Colposcopy | | | | Ionosphere | | | | Texture | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | T_clas | T_red | Agr | T | T_clas | T_red | Agr | T | T_clas | T_red | Agr | T |
| 0 | 0.8 | 0 | 0.4 | 0 | 0.85 | 0 | 0.42 | 0 | 0.95 | 0 | 0.48 | 0.01 |
| 1 | 0.72 | 0 | 0.36 | 0 | 0.77 | 0 | 0.39 | 0 | 0.94 | 0 | 0.47 | 0.01 |
| 2 | 0.74 | 0 | 0.37 | 0 | 0.83 | 0 | 0.41 | 0 | 0.92 | 0 | 0.46 | 0.01 |
| 3 | 0.77 | 0 | 0.39 | 0 | 0.91 | 0 | 0.46 | 0 | 0.93 | 0 | 0.46 | 0.01 |
| 4 | 0.67 | 0 | 0.33 | 0 | 0.86 | 0 | 0.43 | 0 | 0.9 | 0 | 0.45 | 0.01 |
| Media | 0.74 | 0 | 0.37 | 0 | 0.844 | 0 | 0.422 | 0 | 0.928 | 0 | 0.464 | 0.01 |

**Greedy**

| Partition | Colposcopy | | | | Ionosphere | | | | Texture | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | T_clas | T_red | Agr | T | T_clas | T_red | Agr | T | T_clas | T_red | Agr | T |
| 0 | 0.66 | 0.56 | 0.66 | 0.01 | 0.86 | 0.03 | 0.35 | 0.02 | 0.97 | 0.05 | 0.5 | 0.03 |
| 1 | 0.7 | 0.42 | 0.6 | 0.01 | 0.79 | 0.03 | 0.35 | 0.01 | 0.93 | 0.07 | 0.51 | 0.03 |
| 2 | 0.77 | 0.27 | 0.53 | 0.01 | 0.8 | 0.09 | 0.39 | 0.01 | 0.94 | 0.03 | 0.48 | 0.03 |
| 3 | 0.77 | 0.32 | 0.54 | 0.01 | 0.91 | 0.03 | 0.35 | 0.02 | 0.95 | 0.05 | 0.5 | 0.03 |
| 4 | 0.72 | 0.21 | 0.48 | 0.01 | 0.86 | 0.03 | 0.36 | 0.01 | 0.92 | 0.15 | 0.55 | 0.03 |
| Media | 0.724 | 0.356 | 0.562 | 0.01 | 0.844 | 0.042 | 0.36 | 0.014 | 0.942 | 0.07 | 0.508 | 0.03 |

**Búsqueda Local**

| Partition | Colposcopy | | | | Ionosphere | | | | Texture | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | T_clas | T_red | Agr | T | T_clas | T_red | Agr | T | T_clas | T_red | Agr | T |
| 0 | 0.75 | 0.81 | 0.8 | 74.7 | 0.9 | 0.94 | 0.83 | 61 | 0.97 | 0.95 | 0.96 | 185.81 |
| 1 | 0.68 | 0.76 | 0.78 | 93.65 | 0.8 | 0.79 | 0.74 | 33.41 | 0.95 | 0.85 | 0.9 | 91.58 |
| 2 | 0.75 | 0.74 | 0.77 | 94.8 | 0.83 | 0.91 | 0.81 | 44.27 | 0.94 | 1 | 0.98 | 258.92 |
| 3 | 0.74 | 0.81 | 0.78 | 94.8 | 0.9 | 0.91 | 0.81 | 64.26 | 0.9 | 0.9 | 0.92 | 144.56 |
| 4 | 0.72 | 0.85 | 0.82 | 108.26 | 0.9 | 0.88 | 0.81 | 41.84 | 0.94 | 0.8 | 0.87 | 109.92 |
| Media | 0.728 | 0.794 | 0.79 | 93.242 | 0.866 | 0.886 | 0.8 | 48.956 | 0.94 | 0.9 | 0.926 | 158.158 |

Podemos ver que nuestro algoritmo que clasifica por el **vecino más cercano** tiene una efectividad considerable para clasificar objetos en estas bases de datos, con una media de aicerto que está por encima del 74% en todos los conjuntos de datos que hemos usado como conjuntos de prueba. Además, es claramente el más rápido debido a la la simplicidad de este lgoritmo, no demorándose más de una centésima en ninguno de los casos. Además, como este algoritmo siempre considera todos los datos, la tasa de reducción es siempre cero.

Los siguientes resultados que hemos reflejado en las tablas son los del algoritmo **Greedy Relief**. Sorprendentemente, estos resultados han resultado tener una tasa de acierto muy similiar al algoritmo del vecino más cercano. Sin embargo, en este caso tenemos tasas de reducción que ya no se anulan y los tiempos son mayores. Además, las tasas de agregación son más altas que en el algoritmo anterior, como era de de esperar.
Por último, tenemos nuestro algoritmo más pesado, **búsqueda local**. Este algoritmo nos ha dado unos resultados bastante buenos en la clasificación y sobre todo nos ha dado unos valores de nuestra función objetivo bastante altos, lo que nos hace ver que nuestros vectores de pesos son bastante buenos a la hora de clasificar datos en los conjuntos de datos que se nos presentan. Sin embargo, los tiempos de ejecución han sido muy altos, mucho más de lo esperados. Esto puede ser debido a la suma de la probable ineficiencia de programación del algoritmo y la ineficiencia que nos aporta el lenguaje de programación escogido para programar estos algoritmos. A pesar del tiempo tan alto, llegando a una media de casi 3 minutos de ejecución por cada partición del conjunto de datos "Textura", los resultados obtenidos son satisfactorios.
De los resultados también podemos obtener las siguientes conclusiones:

- *Colposcopy* es un conjunto de datos difíciles de clasificar comparado con los otros dos. Podría ser por tener menos ejemplos que los otros dos conjuntos o quizá porque los datos estén más dispersos y los ejemplos no representen bien a los representantes de cada clase.

- *Texture* nos da un porcentaje de clasificación muy alto en todos los algoritmos, a pesar de ser el que más tipos de clases diferentes tiene. Probablemente esto sea porque las clases tienen características bien diferenciadas.

- Todos los algoritmos han dado buenos resultados de clasificación en los *Datasets* proporcionados, así que lo que los diferencia es el valor de la función objetivo en cada caso.

Como última comparativa, podemos mostrar la tabla de medias :

| Colposcopy | | | | Ionosphere | | | | Texture | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| T_clas | T_red | Agr | T | T_clas | T_red | Agr | T | T_clas | T_red | Agr | T |
| 0.74 | 0 | 0.37 | 0 | 0.844 | 0 | 0.422 | 0 | 0.928 | 0 | 0.464 | 0.01 |
| 0.724 | 0.356 | 0.562 | 0.01 | 0.844 | 0.042 | 0.36 | 0.014 | 0.942 | 0.07 | 0.508 | 0.03 |
| 0.728 | 0.794 | 0.79 | 93.242 | 0.866 | 0.886 | 0.8 | 48.956 | 0.94 | 0.9 | 0.926 | 158.158 |

En esta tabla podemos ver claramente las diferencias de tiempo que hay y la gran brecha que crea la **búsqueda local**(la última fila) respecto a Greedy($2^a$ fila) y 1-nn($1^a$ fila). Además, se muestra también cómo aumentan la tasa de reducción y el valor de la función objetivo cuando aumenta la complejidad de nuestro algoritmo, lo que lo hace en el caso general mucho más efectivo aunque el coste computacional y de tiempo sea mucho mayor.