



UNIVERSIDAD  
DE GRANADA

METAHEURÍSTICAS  
GRADO EN INGENIERÍA INFORMÁTICA

---

# PRÁCTICA 1

PROBLEMA DEL APRENDIZAJE DE PESOS EN  
CARACTERÍSTICAS (APC)

---

**Autor**

Vladislav Nikolov Vasilev

**NIE**

X8743846M

**E-Mail**

vladis890@gmail.com

**Grupo de prácticas**

MH3 Jueves 17:30-19:30

**Rama**

Computación y Sistemas Inteligentes



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE  
TELECOMUNICACIÓN

CURSO 2018-2019

# Índice

<b>1. Descripción del problema</b>	<b>2</b>
<b>2. Descripción de los algoritmos</b>	<b>3</b>
2.1. Algoritmos de comparación . . . . .	6
2.2. Algoritmo del método de búsqueda . . . . .	8
<b>3. Desarrollo de la práctica</b>	<b>11</b>
<b>4. Manual de usuario</b>	<b>12</b>
<b>5. Análisis</b>	<b>14</b>
<b>Referencias</b>	<b>15</b>

## 1. Descripción del problema

El problema que se aborda en esta práctica es el Aprendizaje de Pesos en Características (APC). Es un problema típico de *machine learning* en el cuál se pretende optimizar el rendimiento de un clasificador basado en vecinos más cercanos. Esto se consigue mediante la ponderación de las características de entrada con un vector de pesos  $W$ , el cuál utiliza codificación real (cada  $w_i \in W$  es un número real), con el objetivo de modificar sus valores a la hora de calcular la distancia. Cada vector  $W$  se expresa como  $W = \{w_1, w_2, \dots, w_n\}$ , siendo  $n$  el número de dimensiones del vector de características, y cumpliéndose además que  $\forall w_i \in W, w_i \in [0, 1]$ .

El clasificador considerado para este problema es el 1-NN (genéricamente, un clasificador  $k$ -NN, con  $k$  vecinos, siendo en este caso  $k = 1$ ), es decir, aquél que clasifica un elemento según su primer vecino más cercano utilizando alguna medida de distancia (en este caso, utilizando la distancia Euclídea). Cabe destacar que no en todos los casos se usará el clasificador 1-NN ya que se pueden dar casos en los que el vecino más cercano de un elemento sea él mismo. Por ese motivo, en algunas técnicas/algoritmos se usará un 1-NN con el criterio de *leave-one-out*, es decir, que se busca el vecino más cercano pero excluyéndose a él mismo.

El objetivo propuesto es aprender el vector de pesos  $W$  mediante una serie de algoritmos, de tal forma que al optimizar el clasificador se mejore tanto la precisión de éste como su complejidad, es decir, que se considere un menor número de características. Estos dos parámetros, a los que llamaremos *tasa\_clas* y *tasa\_red*, respectivamente, se pueden expresar de la siguiente forma:

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

$$tasa\_red = 100 \cdot \frac{n^\circ \text{ valores } w_i < 0.2}{n^\circ \text{ características}}$$

siendo  $T$  el tamaño del conjunto de datos sobre el que se evalúa el clasificador.

Por tanto, al combinarlos en una única función a la que llamaremos  $F(W)$ , la cuál será nuestra función objetivo a optimizar (maximizar), tenemos que:

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

siendo  $\alpha$  la importancia que se le asigna a la tasa de clasificación y a la de reducción, cumpliendo que  $\alpha \in [0, 1]$ . En este caso, se utiliza un  $\alpha = 0.5$  para dar la misma importancia a ambos, con lo cuál se pretende que se reduzcan al máximo el número de características conservando una *tasa\_clas* alta.

## 2. Descripción de los algoritmos

Primeramente, antes de empezar con la descripción formal de los algoritmos implementados, vamos a describir algunos aspectos comunes, como por ejemplo cómo se representan las soluciones, como se inicializan en algunos de los casos y algunas funciones utilizadas en muchas partes del código, como por ejemplo la función objetivo. Cabe destacar que muchos de los pseudocódigos que aparezcan a continuación, no se encuentren como tal en la implementación, ya que ya hay funciones que se encargan de hacerlos.

Como se dijo al principio, cada solución es un vector  $W$  en el que  $\forall w_i \in W, w_i \in [0, 1]$ . Por tanto, para evitar que las soluciones se salgan de este intervalo, se ha implementado una función que se encarga de normalizar los valores de  $W$  en el rango. La función se ha usado, por ejemplo, en la búsqueda local, para hacer que al aplicar el operador de generación de un nuevo vecino la solución siguiese siendo válida. La implementación de esta función es la siguiente:

---

**Algorithm 1:** Función de normalización de  $W$ 

---

```
1 Normalizar $W$  ( $W$ )  
   input : Vector de pesos  $W$   
   output: Vector de pesos  $W$  normalizado en  $[0, 1]$   
2 foreach  $w_i \in W$  do  
3   if  $w_i < 0$  then  
4      $w_i \leftarrow 0$   
5   else if  $w_i > 1$  then  
6      $w_i \leftarrow 1$   
7 return  $W$ 
```

---

Como se ha mencionado anteriormente, también se tienen que generar las soluciones iniciales. En el caso del algoritmo greedy *RELIEF*, se ha tenido que generar un  $W$  inicial en el que todos sus elementos son 0. Conceptualmente, en el siguiente

pseudocódigo se puede ver cuál es la idea que hay detrás:

---

**Algorithm 2:** Inicialización de un vector de pesos  $W$  en *RELIEF*

---

```

1 GenerarWRelief ( $N$ )
   input : Número de características  $N$ 
   output: Vector de características  $W$  con valores 0
2    $W \leftarrow \text{vector}[N]$ 
3   foreach  $w_i \in W$  do
4      $w_i \leftarrow 0$ 
5   return  $W$ 

```

---

En el caso de la búsqueda local, para inicializar los valores de  $W$  se han generado valores distribuidos uniformemente en el rango  $[0, 1]$ . Para mostrar como sería eso conceptualmente, aquí se muestra un pequeño pseudocódigo:

---

**Algorithm 3:** Inicialización de un vector de pesos  $W$  en BL

---

```

1 GenerarWBL ( $N$ )
   input : Número de características  $N$ 
   output: Vector de características  $W$  con valores aleatorios
2    $W \leftarrow \text{vector}[N]$ 
3   foreach  $w_i \in W$  do
4      $w_i \leftarrow \text{ValorAleatorioUniformeRango0-1}()$ 
5   return  $W$ 

```

---

Una vez que sabemos como se representa e inicializa el vector de pesos  $W$ , vamos a comentar algunos detalles extra. Es importante saber como se calcula la distancia a un vecino, ya que esto juega un factor muy importante a la hora de encontrar cuál es el vecino más cercano a un elemento (o el vecino más cercano por el criterio *leave-one-out*). En la implementación de la práctica se ha utilizado un KDTree, que es una estructura de datos parecida a un árbol binario, solo que de  $K$  dimensiones. Por dentro, esta estructura utiliza la distancia Euclídea (distancia en línea recta entre dos elementos) para determinar cuál es el elemento más próximo a otro. No hace falta conocer como se implementa esta estructura de datos, pero sí es importante conocer cómo se realiza el cálculo de la distancia Euclídea. En el

siguiente pseudocódigo se puede ver el cálculo:

---

**Algorithm 4:** Cálculo de la distancia Euclídea entre dos puntos

---

```

1 DistanciaEuclidea ( $e_1, e_2$ )
   input :  $e_1, e_2$  dos puntos entre los que calcular la distancia
   output: Distancia Euclídea
2    $distancia \leftarrow \sqrt{\sum_{i=1}^N (e_1^i - e_2^i)^2}$ 
3   return  $distancia$ 

```

---

Y ya sabiendo todo esto, solo nos queda comentar la función objetivo,  $F(W)$ , que es lo que se pretende optimizar. Para evaluar la función objetivo, necesitamos calcular *tasa\_clas* y *tasa\_red*. Para calcular lo primero, podemos seguir la idea detrás del siguiente pseudocódigo:

---

**Algorithm 5:** Cálculo de la tasa de clasificación

---

```

1 CalculoTasaClas ( $y, pred, N$ )
   input :  $y$  las etiquetas originales,  $pred$  las etiquetas predichas y  $N$  el
           número de etiquetas
   output: Tasa de elementos bien clasificados
2    $bien\_clasificados \leftarrow 0$ 
3   for  $i \leftarrow 1$  to  $N$  do
4     if  $y_i = pred_i$  then
5        $bien\_clasificados \leftarrow bien\_clasificados + 1$ 
6    $tasa\_clas \leftarrow bien\_clasificados / N$ 
7   return  $tasa\_clas$ 

```

---

Para calcular *tasa\_red*, suponiendo que queremos saber el número de características por debajo de 0.2 podemos seguir un esquema como el siguiente:

---

**Algorithm 6:** Cálculo de la tasa de reducción

---

```

1 CalculoTasaRed ( $W, N$ )
   input :  $W$  el vector de pesos  $N$  el número de pesos
   output: Tasa de reducción
2    $carac\_red \leftarrow 0$ 
3   foreach  $w_i \in W$  do
4     if  $w_i < 0.2$  then
5        $carac\_red \leftarrow carac\_red + 1$ 
6    $tasa\_red \leftarrow carac\_red / N$ 
7   return  $tasa\_red$ 

```

---

Y finalmente, para poder calcular la función a optimizar (nuestra función *fitness* u objetivo), teniendo en cuenta que usamos un  $\alpha = 0.5$  para ponderar las dos tasas, y que anteriormente hemos calculado ambas tasas, podemos seguir el siguiente esquema:

---

**Algorithm 7:** Cálculo de la función objetivo o *fitness*

---

```

1 CalculoFuncionFitness (tasa_clas, tasa_red,  $\alpha$ )
   input : Recibe tasa_clas y tasa_red y calcula la agrupación de ambas
           con un factor  $\alpha$ 
   output: Valor fitness o agrupación
2    $fitness \leftarrow \alpha \cdot tasa\_clas + (1 - \alpha) \cdot tasa\_red$ 
3   return fitness

```

---

Una vez sabiendo todo esto, podemos comenzar a describir las implementaciones de los algoritmos utilizadas.

### 2.1. Algoritmos de comparación

Para la comparación de la metaheurística implementada, utilizaremos dos algoritmos/técnicas:

- Un clasificador 1-NN clásico, sin ponderar las características.
- Un algoritmo greedy llamado *RELIEF* para el cálculo de los pesos con los que ponderar las características, y posteriormente utilizar un 1NN con los valores ponderados.

Comencemos por la base: el 1-NN. Este clasificador lo que hace es, dado un conjunto de valores  $X$  que pertenecen a una muestra y un elemento  $e$ , decir cuál es el  $x \in X$  más cercano a  $e$ , y por tanto, decir que  $e$  pertenece a la misma clase  $x$ . Para determinar cuál es el elemento más cercano se puede usar alguna métrica de distancia, como por ejemplo la distancia Euclídea, descrita anteriormente. A la hora de implementarlo, para poder acelerar los cálculos, se puede usar un KDTree, ya que permite realizar una consulta rápida (en los casos más favorables su complejidad temporal es  $\mathcal{O}(\log n)$ , mientras que en el peor caso es  $\mathcal{O}(n)$ ) utilizando la distancia Euclídea para determinar el vecino más cercano, con la penalización de que tarda un tiempo  $\mathcal{O}(n)$  en ser construido. Para poder ver un esquema de su

funcionamiento, se ofrece el siguiente pseudocódigo:

---

**Algorithm 8:** Clasificador 1-NN
 

---

```

1 1-NN ( $X, y, e$ )
   input :  $X$  un conjunto de elementos que pertenecen a una muestra,  $y$ 
           las etiquetas correspondientes a cada  $x_i \in X$ ,  $e$  el elemento
           del que buscar su vecino más cercano
   output: Clase a la que pertenece el vecino más cercano de  $e$ 
2    $arbol\_kd \leftarrow \text{KDTree}(X)$ 
3    $x \leftarrow arbol\_kd.\text{VecinoMasCercano}(e)$ 
4    $etiqueta \leftarrow y[x]$ 
5   return  $etiqueta$ 

```

---

Una vez habiendo explicado el 1-NN, vamos a hablar del algoritmo para el cálculo de pesos *RELIEF*. Es un algoritmo greedy que, comenzando con un  $W$  cuyos pesos valen 0, actualiza  $W$  para cada  $x_i \in X$ , buscando para cada  $x_i$  cuál es su aliado más cercano (elemento que tiene la misma etiqueta que  $x_i$  con el criterio de *leave-one-out*, ya que él mismo podría ser su vecino más cercano) y su enemigo más cercano (elemento que tiene diferente etiqueta a la que tiene  $x_i$ ).

A la hora de implementarlo, vamos a utilizar 2 KDTree en cada iteración que se van a construir sobre la marcha. En uno se encontrarán todos los aliados de  $e$  y en el otro estarán todos sus enemigos. Esto puede suponer una gran penalización por el tiempo de creación de los árboles, pero es un tiempo insignificante ya que el algoritmo es muy rápido. Después de construir los árboles, se buscará en el caso del aliado más cercano, por el criterio de *leave-one-out*, cuál es el índice de este aliado. En el caso del enemigo más cercano, como este no puede ser él mismo, se buscará el índice del vecino más cercano en ese árbol. Una vez hecho eso, obtendremos los respectivos aliado y enemigo del conjunto de aliados y enemigos. Una vez teniéndolos, ya se puede actualizar el valor de  $W$ .

Cuando se ha terminado de iterar sobre todos los elementos de  $X$ , se normaliza  $W$  para que esté en el rango  $[0, 1]$  eligiendo el  $w_i \in W$  que sea más grande. Todos aquellos valores por debajo de 0 se truncan a 0, y el resto se normaliza dividiéndolos entre  $w_m$  (el  $w_i$  más grande).



Una vez dicho esto, veamos cómo sería la implementación en pseudocódigo:

---

**Algorithm 9:** Cálculo de los pesos mediante *RELIEF*


---

```

1 RELIEF ( $X, Y$ )
   input :  $X$  un conjunto de elementos que pertenecen a una muestra,
            $Y$  las etiquetas de cada  $x_i \in X$ 
   output: Vector de pesos  $W$ 
2    $N \leftarrow$  num. elementos  $Y$ 
3    $n\_carac \leftarrow$  num. características  $X$ 
4    $W \leftarrow$  GenerarWRelief( $n\_carac$ )
5   for  $i \leftarrow 1$  to  $N$  do
6      $x, y \leftarrow X[i], Y[i]$ 
7      $aliados \leftarrow e_a \subset X : \text{etiqueta}(e_a) = y$ 
8      $enemigos \leftarrow e_e \subset X : \text{etiqueta}(e_e) \neq y$ 
9      $arbol\_aliados \leftarrow$  KDTree( $aliados$ )
10     $arbol\_enemigos \leftarrow$  KDTree( $enemigos$ )
11     $aliado\_cercano \leftarrow$   $arbol\_aliados$ .ObtenerVecinoMasCercanoL1O( $x$ )
12     $enemigo\_cercano \leftarrow$   $arbol\_enemigos$ .ObtenerVecinoMasCercano( $x$ )
13     $aliado \leftarrow aliados[aliado\_cercano]$ 
14     $enemigo \leftarrow enemigos[enemigo\_cercano]$ 
15     $W \leftarrow W + |x - enemigo| - |x - aliado|$ 
16   $w_m \leftarrow \text{máx}(W)$ 
17  foreach  $w_i \in W$  do
18    if  $w_i < 0$  then
19       $w_i \leftarrow 0$ 
20    else
21       $w_i \leftarrow w_i / w_m$ 
22  return  $W$ 

```

---

## 2.2. Algoritmo del método de búsqueda

A continuación, vamos a estudiar el algoritmo de búsqueda que se ha pedido implementar en esta primera práctica. Se trata de un algoritmo de búsqueda local, que partiendo de un vector  $W$  con valores aleatorios busca optimizarlo mediante la exploración de los vecinos. Esta exploración se realiza modificando con un valor aleatorio generado a partir de una distribución normal con  $\mu = 0$  y  $\sigma = 0.3$  un  $w_i \in W$ , quedándose con el cambio en caso de mejorar la función objetivo, o descartándolo en otro caso.

Para realizar lo explicado anteriormente, se genera una permutación del con-

junto  $\{1, 2, \dots, N\}$ , donde  $N$  es el número de características, y se van escogiendo las características según el orden de la permutación, aplicándoles el cambio anteriormente descrito. Si no se produce una mejora, se descarta el cambio realizado. Si no se ha producido mejora en la función objetivo con la permutación, se escoge una nueva permutación y se repite el proceso, hasta un máximo de  $20 \cdot N$  evaluaciones sin éxito seguidas de la función objetivo. Si se produce mejora, se acepta el cambio y se genera una nueva permutación, repitiendo el proceso. Todo esto se realiza hasta que se hayan realizado 15000 evaluaciones de la función objetivo, o hasta que se dé la condición anterior (demasiadas iteraciones sin mejora).

La búsqueda local utiliza una función para evaluar lo bueno que es un vector  $W$ . Esta función lo que hace es aplicar  $W$  sobre  $X$  en aquellas características donde  $w_i > 0.2$ , y sobre estos crea un KDTree para buscar los índices de los vecinos más cercanos según el criterio *leave-one-out*. Esta búsqueda, para una mayor velocidad, se aplica sobre todos los elementos de  $X$ . Una vez calculados, se obtienen las etiquetas correspondientes, y se realiza la evaluación de la función objetivo. El pseudocódigo de esta función se puede ver aquí:

---

**Algorithm 10:** Función para evaluar  $W$  en la búsqueda local

---

```

1 Evaluar ( $X, Y, W$ )
   input :  $X$  un conjunto de elementos que pertenecen a una muestra,
            $Y$  las etiquetas de cada  $x_i \in X$ ,  $W$  vector de pesos
   output: Valor fitness para los datos de entrada
2    $X\_pesos \leftarrow$  aplicar  $w_i \in W$  sobre los  $x_i \in X$  donde  $w_i > 0.2$ 
3    $arbolkd \leftarrow$  KDTree( $X\_pesos$ )
4    $vecinos \leftarrow$   $arbolkd$ .ObtenerVecinosMasCercanoL1O( $X\_pesos$ )
5    $pred \leftarrow Y[vecinos]$ 
6    $tasa\_clas \leftarrow$  CalcularTasaClas( $Y, pred$ , num. etiquetas)
7    $tasa\_red \leftarrow$  CalcularTasaRed( $W$ , num. características)
8    $fitness \leftarrow$  CalculoFuncionFitness( $tasa\_clas, tasa\_red$ )
9   return  $fitness$ 

```

---

Con todo esto explicado, ya podemos ver el pseudocódigo de la búsqueda local:

---

**Algorithm 11:** Cálculo de los pesos mediante la Búsqueda Local
 

---

```

1 BusquedaLocal ( $X, Y$ )
   input :  $X$  un conjunto de elementos que pertenecen a una muestra,
            $Y$  las etiquetas de cada  $x_i \in X$ 
   output: Vector de pesos  $W$ 
2   Inicializar semilla aleatoria
3    $N \leftarrow$  num. características  $X$ 
4    $W \leftarrow$  GenerarWBL( $N$ )
5    $evaluaciones \leftarrow 0$ 
6    $evaluaciones\_malas \leftarrow 0$ 
7    $fitness \leftarrow$  Evaluar( $X, Y, W$ )
8   while  $evaluaciones < 15000$  do
9      $W\_actual \leftarrow W$ 
10     $orden\_caracteristicas \leftarrow$  Permutacion(1 to  $N$ )
11    foreach  $carac \in orden\_caracteristicas$  do
12       $W[carac] \leftarrow W[carac] +$  GenerarValorDistribucionNormal( $\mu, \sigma$ )
13       $W \leftarrow$  NormalizarW( $W$ )
14       $evaluaciones \leftarrow evaluaciones + 1$ 
15       $nuevo\_fitness \leftarrow$  Evaluar( $X, Y, W$ )
16      if  $nuevo\_fitness > fitness$  then
17         $fitness \leftarrow nuevo\_fitness$ 
18         $evaluaciones\_malas \leftarrow 0$ 
19        break
20      else
21         $evaluaciones\_malas \leftarrow evaluaciones\_malas + 1$ 
22         $W[carac] \leftarrow W\_actual[carac]$ 
23      if  $evaluaciones > 15000$  or  $evaluaciones\_malas > 20 \cdot N$  then
24        return  $W$ 
25  return  $W$ 

```

---

### 3. Desarrollo de la práctica

La práctica se ha implementado en **Python3** y ha sido probada en la versión 3.7.1. Por tanto, se recomienda encarecidamente utilizar un intérprete de Python3 al ejecutar el código y no uno de la versión 2.X, debido a problemas de compatibilidad con ciertas funciones del lenguaje. Se ha probado el código sobre Linux Mint 19 y al estar basado en Ubuntu 18 no debería haber problemas de compatibilidad con otros sistemas, además de que Python es un lenguaje muy portable. No se ha probado en el entorno **conda**, pero si se consiguen instalar los módulos necesarios, no debería haber problemas.

A la hora de implementar el software, se han utilizado tanto módulos ya incluidos en Python, como el módulo **time** para la medición de tiempos, como módulos científicos y para *machine learning*, como por ejemplo **numpy** y **sklearn**. Este último se ha utilizado para poder dividir los datos para el **5 Fold Cross Validation** y para obtener un clasificador KNN que poder utilizar para poder probar los resultados obtenidos por cada uno de los algoritmos. Para la visualización de datos se ha utilizado **pandas**, ya que permite conseguir una visualización rápida de estos gracias a los DataFrames.

Adicionalmente, la estructura de **KDTree** utilizada ha sido sacada de un módulo externo llamado **pykdtree**, cuyo enlace puede ser encontrado en las referencias. Este módulo está implementado en **Cython** y **C** y también utiliza **OMP**, con lo cuál su rendimiento va a ser muy superior a otras implementaciones como por ejemplo el **cKDTree** de **scipy**<sup>1</sup>. En cuanto a su uso, las funciones y la forma de construirlo son las mismas que las de **cKDTree**, con lo cuál se puede consultar su documentación para obtener más información sobre su uso.

Siendo ahora más concretos en cuanto a la implementación, se ha creado una función para cada uno de los algoritmos que se han implementado. Estas funciones se encargan de recorrer las particiones creadas y de ejecutar los respectivos algoritmos pasándoles los datos, además de encargarse de recopilar información estadística para mostrarla luego por pantalla. Se han utilizado dos semillas aleatorias las cuáles están fijas en el código: una para dividir los datos, y otra en la búsqueda local, que se fija al principio, justo antes de inicializar los  $W$ , como se puede ver en el pseudocódigo. Los archivos ARFF proporcionados se han convertido al formato CSV con un script propio, con el objetivo de facilitar la lectura de los datos. Estos archivos también se proporcionan junto con el código fuente implementado.

---

<sup>1</sup>De hecho, **pykdtree** está basado en **cKDTree** y **libANN**, cogiendo lo mejor de cada implementación y paralelizando el código con **OMP** para conseguir unos rendimientos muy superiores a ambos, tanto a la hora de crear el árbol como para hacer consultas.

## 4. Manual de usuario

Para poder ejecutar el programa, se necesita un intérprete de **Python3**, como se ha mencionado anteriormente. Además, para poder satisfacer las dependencias se necesita el gestor de paquetes **pip** (preferiblemente **pip3**).

Se recomienda instalar las dependencias, las cuáles vienen en el archivo **requirements.txt**, ya que sin ellas, el programa no podrá funcionar. Se recomienda utilizar el script de bash incluido para realizar la instalación, ya que se encarga de instalarlo en un entorno virtual para no causar problemas de versiones con paquetes que ya se tengan instaladas en el equipo o para no instalar paquetes no deseados. Una vez instalados, para poder utilizar el entorno creado se debe ejecutar el siguiente comando:

```
$ source ./env/bin/activate
```

Para desactivar el entorno virtual, simplemente basta con ejecutar:

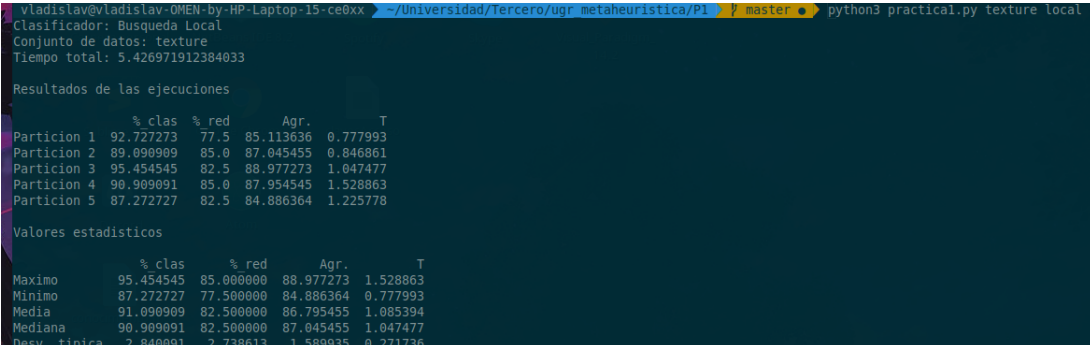
```
(env) $ deactivate
```

Para ejecutar el programa basta con ejecutar el siguiente comando:

```
$ python3 practica1.py [archivo] [algoritmo]
```

Los argumentos **archivo** y **algoritmo** son obligatorios, y sin ellos el programa lanzará una excepción. En cuanto a sus posibles valores:

- **archivo** puede ser **colposcopy**, **ionosphere** o **texture**.
- **algoritmo** puede ser **knn**, **relief** o **local**.



```
vladislav@vladislav-OMEN-by-HP-Laptop-15-ce0xx ~/Universidad/Tercero/ugr/metaheuristica/P1 $ python3 practica1.py texture local
Clasificador: Busqueda Local
Conjunto de datos: texture
Tiempo total: 5.426971912384033

Resultados de las ejecuciones

  %_clas  %_red  Agr.  T
Particion 1  92.727273  77.5  85.113636  0.777993
Particion 2  89.090909  85.0  87.045455  0.846861
Particion 3  95.454545  82.5  88.977273  1.047477
Particion 4  90.909091  85.0  87.954545  1.528863
Particion 5  87.272727  82.5  84.886364  1.225778

Valores estadísticos

  %_clas  %_red  Agr.  T
Maximo   95.454545  85.000000  88.977273  1.528863
Minimo   87.272727  77.500000  84.886364  0.777993
Media    91.090909  82.500000  86.795455  1.085394
Mediana  90.909091  82.500000  87.045455  1.047477
Desv. típica  2.840091  2.738613  1.589935  0.271736
```

Figura 1: Ejemplo de salida de la ejecución con los datos **texture** y el algoritmo **local**.

## **5. Análisis**

## Referencias

- [1] Texto referencia  
<https://url.referencia.com>