Tercer Curso del Grado de Ingeniería Informática
Segundo Cuatrimestre
Metaheurísticas
Grupo de prácticas A2 (martes de 17:30 a 19:30)

Curso 2017-2018
Universidad de Granada

UNIVERSIDAD DE GRANADA

Práctica 1B
Aprendizaje de Pesos en Características (APC):
Técnicas de Búsqueda Local y Algoritmos *Greedy*

Alberto Lejárraga Rubio

DNI: 50618389N

Email: alejarragar@correo.ugr.es

Granada, 2 de abril de 2018

1. Índice

1. Índice 2
2. Descripción del Problema del APC 3
3. Descripción de los elementos comunes de los algoritmos 4
Representación de la solución 4
Representación de los datos 4
Clasificador 1-NN 5
Evaluar una solución 6
Distancia euclídea 6
4. Descripción de los algoritmos 7
Algoritmo simple de clasificación 7
Algoritmo de búsqueda local 9
5. Descripción del algoritmo de comparación 10
6. Breve manual de usuario 10
7. Experimentos y análisis de resultados 12
Algoritmo 1-NN 12
Algoritmo Relief 13
Algoritmo de Búsqueda Local 13
Comparación de resultados globales 14
8. Bibliografía 15

2. Descripción del Problema del APC

El problema del Aprendizaje de Pesos en Características (APC) trata de optimizar el rendimiento de un clasificador basado en la técnica de clasificación a partir de los vecinos más cercanos que se explica a continuación. Para ello intenta asignar a cada característica un peso que servirá para aumentar o disminuir la capacidad de dicha característica de influir en la clasificación final.

La técnica de clasificación mediante los vecinos más cercanos o *k*-NN (*k nearest neighbors*, *k* vecinos más cercanos) intenta decidir de qué tipo o clase es un ítem a clasificar a partir de los *k* ítems, de los que ya se conoce la clasificación, cuya distancia al elemento a clasificar sea menor de todo el conjunto de entrenamiento, siendo este conjunto de entrenamiento los elementos de los que se conoce el tipo o clase a la que pertenecen.

Es decir, dado que en la práctica el clasificador utilizado es el 1-NN, para clasificar un ítem habría que buscar en el conjunto de entrenamiento el otro ítem cuya distancia al primero sea menor. Encontrado dicho elemento que tiene una distancia mínima, el que se quería clasificar se clasificará con la clase de este. Como se ha dicho, el APC intentará asignar un peso a cada característica, de modo que la distancia entre los vecinos más cercanos se modifique en función de este peso.

Estos pesos habrá que almacenarlos en una lista para luego aplicarlo a la hora de obtener la distancia entre dos elementos. La calidad de esta lista de pesos dependerá no solo de lo bien que clasifique sino también de cuántas características se tendrán en cuenta para clasificar, por tanto se debe medir esta calidad mediante la siguiente función:

$$Calidad(W) = \alpha * tasaDeClasificación(W) + (1-\alpha) * tasaDeReducción(W)$$

$$tasaDeClasificación(W) = 100 * \frac{Instacias\ bien\ Clasificadas\ de\ T}{Instancias\ totales\ en\ T}$$

$$tasaDeReducción(W) = 100 * \frac{valores<0.2\ en\ W}{número\ de\ características\ de\ W}$$

Los parámetros que aparecen en estas fórmulas y sus restricciones son:

- W, lista de pesos: es el vector de pesos que se utilizará para clasificar. Estará formado por tantos elementos como características tengan los elementos a clasificar, asignando un peso a cada una de ellas. Este peso debe estar normalizado en el intervalo real [0,1]
- α: es un número real entre 0 y 1 que determina la importancia que debe tener una buena clasificación o una reducción mayor. En el caso de la práctica será 0.5, asignando igual importancia al acierto obtenido y a la reducción de la solución.
- T, conjunto a clasificar: son los elementos sobre los que se aplica el clasificador.

Para terminar, comentar que el lenguaje de programación que he utilizado es Python. He elegido este por la gran facilidad con otorga al trabajar con listas.

3. Descripción de los elementos comunes de los algoritmos

Representación de la solución

Como se ha dicho anteriormente, el APC asigna un peso a cada característica que se utilizará para clasificar los elementos. Estos pesos se pueden representar utilizando una lista de Python. Dicha lista tendrá como tamaño el número de características que se tengan en cuenta en cada conjunto de datos. Los elementos de la lista de pesos deben estar normalizados para que no haya características que predominen sobre otras una vez obtenido el vector.

Por tanto los valores (w_i) que pueden aparecer en el vector de pesos (W) son:

- $w_i<0.2$: estos elementos que estén por debajo de 0.2 no se tendrán en cuenta para clasificar y serán los que hagan que la tasa de reducción aumente, pues es la tasa que mide precisamente esto, cuántos elementos se desechan del vector de pesos por no aportar lo suficiente a la clasificación.
- $w_i \in [0.2, 1]$: son los elementos que sí se tendrán en cuenta, resultando más importantes en la clasificación cuanto más se acerquen los valores a 1.

Según cada algoritmo, encontrada esta solución que es el vector de pesos se pasará a clasificar los ejemplos necesarios siguiendo sus ponderaciones.

Representación de los datos

Los datos se obtienen en la práctica a partir de varios set de datos en formato ".arff" y se vuelcan a listas utilizando la librería de Python "liac-arff" importada como "arff". Para ello primero se cargan los datos de un fichero en una variable con la función *load.* Esta función devuelve un diccionario de Python, permitiendo acceder fácilmente a los campos "attributes" y "data", que son el nombre de los atributos y los valores respectivamente.

A partir de estos dos campos genero una lista que podría representarse de la siguiente forma:

[[<lista_de_nombres>],[<partición_1>],[<partición_2>],…,[<partición_n>]]

A su vez, la lista <lista_de_nombres> es una lista de tuplas que contiene los nombres de cada característica y el tipo de esta:

[(<nombre1>,<tipo1>), (<nombre2>,<tipo2>), … , (<nombren>,<tipon>)]

Por su parte, las listas <partición_1,2,…,n> representan una partición del conjunto de datos que contienen ejemplos:

[[<ejemplo_1>], [<ejemplo_2>], … , [<ejemplo_m>]]

De nuevo, cada ejemplo de la partición es una lista con los valores de cada característica para cada ejemplo. La última característica de cada ejemplo indica la clase:

[<carac1>, <carac2>, … , <caracn-1>, <clase>]

La distribución de clases de las particiones se debe mantener con respecto al total de datos, es decir, si en el set de datos completo el 70% de los datos pertenecían a una clase, en cada partición se debe respetar ese 70% de la clase.

Para ello, se leen los datos obtenidos en la variable *datos* (obtenida a partir del arff, "data") y se va metiendo un dato en cada partición:

listaResul=[]

Añadir a *listaResul* una lista con los nombres de los atributos (obtenidos a partir del arff, "attributes")

Añadir a *listaResul* tantas listas vacías como particiones se desee crear

numClases=<numero_de_clases_del_set>

Crear lista *meterClase* de tamaño *numClases* y añadir 0 en sus *numClases* posiciones

#esta lista indicara en que partición se mete el siguiente ejemplo que venga de una de las clases

Para cada *dato* del set:

    Meter en *listaResul*[<sublista indicada por *meterClase*>] el *dato*

    Actualizar *meterClase* para que indique la siguiente sublista en la clase del dato

De esta forma todas las particiones tendrán el mismo número de elementos (como mucho uno de diferencia) y la distribución de clases se mantendrá.

Después de esto habrá que normalizar los valores para que estos estén en el intervalo [0,1], lo que se hará en función del máximo y el mínimo valor de las listas.

Como ejemplo, la lista de atributos y una de las particiones tras la normalización sería:

[(u'V2', u'NUMERIC') , (u'V3', u'NUMERIC') , … , (u'Class', [u'1', u'2'])]

[[0.8 , 0.6 ,…, 1], [1.1 , 0.9 ,…, 2] , … , [0.7 , 0.6 , 2]]

Clasificador 1-NN

Tal como se ha explicado, el método de clasificación utilizado es el 1-NN, que buscará cual es el vecino más cercano en el conjunto de entrenamiento para clasificar un nuevo ejemplo del que, en principio, no se sabe su clase. Se hará uso del vector de pesos encontrado siguiendo uno de los algoritmos. En pseudocódigo, el algoritmo es el siguiente:

Función clasificador (*vectorDePesos*, *ejemploAClasificar*, *conjuntoDeEntrenamiento*):

    distanciaMinima = valor elevado (*maxInt* de cada lenguaje)

    Para cada *ejemplo* de *conjuntoDeEntrenamiento*:

        distancia = distanciaPonderada (*ejemplo*, *ejemploAClasificar*, *vectorDePesos*)

        Si *distancia*<*distanciaMinima*:

            distanciaMinima = *distancia*

            ejemploMasCercano = *ejemplo*

    Devolver la clase del *ejemploMasCercano*

Como se puede observar el algoritmo no es para nada complejo. Solo habrá que recorrer el conjunto de entrenamiento y calcular la distancia ponderada de cada ejemplo de este con el ejemplo que se quiere clasificar y devolver la clase del que se haya encontrado con menor distancia. La distancia ponderada se calcula mediante el siguiente procedimiento:

```
Funcion distanciaPonderada (elemento1, elemento2, vectorDePesos):
    suma=0
    Para cada posición de elemento1:
        Si vectorDePesos[posición]>=0.2:
            Sumar a suma vectorDePesos[posición]*(elemento1[posición]-elemento2[posición])^2
    Devolver raizCuadrada(suma)
```

Como puntualización, aunque el algoritmo de clasificación sea el que se ha representado en pseudocódigo más arriba, en el código que yo entrego hago una pequeña modificación. Como lo que me interesa realmente es saber si se ha acertado o no la clasificación, en vez de devolver la clase del ejemplo más cercano compruebo si esta clase obtenida es igual a la clase del ejemplo a clasificar. Si el resultado es positivo devuelvo 1 y si no devuelvo 0.

Evaluar una solución

La evaluación de la solución se realiza en base a las tasas de acierto, reducción y la agregada que proviene de las dos primeras. Para hacerlo el pseudocódigo sería el siguiente, recordando que la función "clasificador" devuelve 1 si se acertó o 0 si se falló la clasificación:

```
Función evaluarSolucion(vectorDePesos, particionAEvaluar, conjuntoDeEntrenamiento, alfa):
   aciertos = 0
   Para cada ejemplo de la particionAEvaluar:
      Sumar a aciertos el resultado de clasificador(vectorDePesos,ejemplo,conjuntoDeEntrenamiento)
   tasaDeClasificacion=100*aciertos/<tamaño de particionAEvaluar>
   suma=0
   Para cada elemento en vectorDePesos:
      Si el elemento es <0.2: incrementar suma en una unidad
   tasaDeReduccion=100*suma/<tamaño de vectorDePesos>
   tasaAgregada=alfa * tasaDeClasificacion + (1-alfa) * tasaDeReduccion
   Devolver una lista con tasaDeClasificacion, tasaDeReduccion y tasaAgregada
```

Distancia euclídea

El algoritmo para obtener la distancia euclídea entre dos elementos es el mismo que para obtener la distancia ponderada pero sin multiplicar la distancia por el vector de pesos, por lo que no mostraré el pseudocódigo y me referiré al algoritmo ya explicado.

4. Descripción de los algoritmos

Algoritmo simple de clasificación

Se trata del algoritmo 1-NN sin tener en cuenta ninguna lista de pesos aprendidos, simplemente busca el vecino más cercano, que será la clasificación del nuevo ejemplo. En la práctica lo he implementado a la vez que el método *Relief*, que recibe un parámetro indicando si se deben "aprender" pesos a partir del algoritmo o si no se deben usar estos pesos. En caso de que se quiera ejecutar el algoritmo sin pesos aprendidos el vector de pesos generados está formado en todas sus posiciones por unos, con lo que se puede comprobar fácilmente que la distancia ponderada no se modifica:

suma=0

...

Se multiplica siempre por 1

Sumar a *suma* *vectorDePesos*[*posición*]*(*elemento1*[*posición*]-*elemento2*[*posición*])^2

Por tanto el pseudocódigo de este algoritmo es el mismo que el del *Relief* cambiando la línea de "obtenerPesos" por "llenar de unos el vector de pesos".

Algoritmo Relief

En este caso sí se aprenden pesos que se utilizarán para clasificar. El pseudocódigo del algoritmo que obtiene los pesos e imprime las tasas de clasificación es el siguiente:

```
Función algoritmoRelief(setEjemplos):

   vectorDePesos = obtenerPesos(setEjemplos)

  tasasDeClasificacion=evaluarSolucion(vectorDePesos, particionAClasificar, conjuntoDeEntrenamiento)

   Imprimir las tasas de tasasDeClasificacion
```

En Python, utilizando la forma de almacenar los datos mostrada antes, se corresponderán con:

```
particionAClasificar = setEjemplos[-1]

conjuntoDeEntrenamiento = setEjemplos[1:-1]
```

Esta forma de acceder a las listas se utilizará durante todo el programa y el presente documento.

Por su parte el pseudocódigo de "obtenerPesos", que es el algoritmo en sí, es el siguiente:

```
Funcion obtenerPesos (lista):
   Crear listaDePesos de tamaño el número de características de los ejemplos y llena de ceros
   Crear conjuntoDeEntrenamiento a partir de lista
   Para cada ejemplo en conjuntoDeEntrenamiento:
      Definir distanciaMinimaDeAmigo y distanciaMinimaDeEnemigo a maxInt
      Para cada ejemploOtraVez en conjuntoDeEntrenamiento:
         Si ejemplo != ejemploOtraVez:
            distancia=distanciaEuclidea(ejemplo,ejemploOtraVez)
            Si ejemplo y ejemploOtraVez son amigos(pertenecen a la misma clase):
               Si distancia < distanciaMinimaDeAmigo:
                  amigoMasCercano = ejemploOtraVez
                  distanciaMinimaDeAmigo = distancia
            En otro caso (son enemigos, de clases distintas):
               Si distancia < distanciaMinimaDeEnemigo:
                  enemigoMasCercano = ejemploOtraVez
                  distanciaMinimaDeEnemigo = distancia
      Para cada posición en listaDePesos:
         actualizacionEnemigo = valorAbsoluto(ejemplo[posición]-enemigoMasCercano[posición])
         actualizacionAmigo = valorAbsoluto(ejemplo[posición]-amigoMasCercano[posición])
         sumar a listaDePesos[posición] (acualizacionEnemigo – actualizacionAmigo)
   Encontrar maximoValorListaDePesos
   Para cada dato en listaDePesos:
      Si dato<0: dato=0
      En otro caso: dato=dato/maximoValorListaDePesos
   Retornar listaDePesos
```

"Leave one out"

Encontrar amigo y enemigo más cercano de cada ejemplo

Actualizar lista de pesos en base al amigo y enemigo más cercanos encontrados posición a posición

Normalizar vector de pesos

Como se ve en recuadro negro se utiliza la técnica de "leave one out". Esto consiste en que a la hora de encontrar el amigo más cercano de un ejemplo no se tenga en cuenta ese mismo ejemplo, ya que siempre sería él mismo al ser la distancia cero. Esta técnica debería aplicarse también si se quisieran clasificar las particiones de entrenamiento a la hora de encontrar el vecino más cercano, pues de nuevo el vecino más cercano sería él mismo, con lo que la clasificación no tendría sentido (siempre se encontraría la solución correcta pero sería irreal).

Algoritmo de búsqueda local

```
Función algoritmoBuscLocal(setEjemplos, generadorRandom,alfa,sigma):
    Definir numeroDeCaracteristicas con el número que corresponda
    Definir contador y numeroDeVecinosExplorados e inicializarlos a 0
    mejorSolucion = lista con numeroDeCaracteristicas posiciones con aleatorios en [0,1]
    calidadMejorSolucion = evaluarSolucion(mejorSolucion,particionAClasificar,conjuntoDeEntrenamiento,alfa)
    distNormal = lista con 15000 números con distribución normal de media 0 y varianza sigma^2
    Mientras (contador < 15000) y (vecinosExplorados < (20*numeroDeCaracteristicas)):
        Definir mejora a falso
        quedanPorModificar = lista con números secuenciales del 0 a numeroDeCaracterísticas-1
        modificadas = 0
        Mientras (mejora sea falso) y (modificadas < numeroDeCaracteristicas):
            Incrementar contador, numeroDeVecinosExplorados y modificadas en una unidad
            caracteristicaAModificar = elegirCaracteristica(quedanPorModificar,generadorRandom)
            solucionActual = generarVecino(mejorSolucion,distNormal,caracteristicaAModificar)
            calidadSolucionActual = evaluarSolucion(solucionActual,particionAClasificar,conjuntoDeEntrenamiento,alfa)
            Si calidadSolucionActual es mejor que calidadMejorSolucion:
                mejorSolucion = solucionActual
                calidadMejorSolucion = calidadSolucionActual
                Cambiar mejora a verdadero
                numeroDeVecinosExplorados = 0
    Imprimir las tasas de calidadMejorSolucion
```

Definir variables necesarias, encontrar la primera solución aleatoria y evaluarla

Encontrar soluciones nuevas y evaluarlas

Decidir si la solución encontrada es mejor y guardarla en ese caso

Como se ve en el pseudocódigo del algoritmo he empleado tres funciones exteriores para que fuera más legible, que son, además de la de "evaluarSolucion" ya explicada antes estas dos:

```
Función elegirCaracteristica (quedanPorModificar, generadorRandom):
    posicion=obtener un aleatorio del generadorRandom entre [0,<tamaño de quedanPorModificar>]
    Retornar quedanPorModificar[posición]
```

Utilizando esta función se consigue que cada vez que se modifique una característica, esta sea una cualquiera pero que en cada iteración no se modifique dos veces la misma característica. En Python al extraer con la función pop el valor de una posición, esta se eliminará y la lista disminuirá su tamaño en una unidad.

```
Función generarVecino(solucion, distNormal, caractAModificar):
   solucionAux=copia de solucion
   nuevoValor=extraer un numero de distNormal
   Modificar solucionAux[caractAModificar] sumándole nuevoValor
   Si el solucionAux[caractAModificar] es > 1:
      solucionAux[caractAModificar]=1
   Si el solucionAux[caractAModificar] es < 0:
      solucionAux[caractAModificar]=0
   Retornar solucionAux
```

Como *distNormal* es una lista con valores aleatorios basta con coger uno de ellos cualquiera (en mi caso el ultimo)

Se debe utilizar una copia de la solución, pues en Python al modificar una lista en una función se modifica la lista original no una copia de esta.

5. Descripción del algoritmo de comparación

Todos los métodos obtienen un vector de pesos, clasifican después la partición de validación con este vector de pesos y muestran los resultados por pantalla. Al terminar la ejecución se pueden observar las tasas devueltas y pasar a las tablas de Excel que se pueden ver en el apartado 7 de este documento. Este paso de la salida del programa a las tablas Excel se hace a mano, por lo que una posible mejora del programa sería que el paso fuera automático:

```
vectorDePesos = obtenerVectorDePesosSegunAlgoritmo( )
tasas = evaluarSolucion(vectorDePesos,particionAClasificar, conjuntoDeEntrenamiento, alfa)
Mostrar las tasas por pantalla
Pasar este resultado a Excel e interpretarlo
```

Para comparar los resultados se tendrán en cuenta tanto las tasas de clasificación, reducción y la agregada, así como el tiempo que se tarda en ejecutar los algoritmos.

6. Breve manual de usuario

Como ya se ha explicado anteriormente, la práctica la he desarrollado en el lenguaje de programación Python y ese es el código que entrego, como detallaré más adelante. Para realizarlo he utilizado el guion de la práctica así como las diapositivas del seminario dedicado a ella.

El código está dividido en 5 archivos, además de los archivos de datos descargados de la página web de la asignatura correspondientes a las temáticas del ozono en días normales o con una alta concentración de este, a datos que indican la presencia o ausencia de la enfermedad del Parkinson y a la fisiología correcta o incorrecta del corazón. Estos tres archivos se encuentran en la carpeta “data”.

Por otro lado, en la raíz de la carpeta que entrego aparecen los cinco archivos que comentaban y que paso a detallar a continuación:

- aleatorio.py: en este fichero creo una clase "Random" que es una adaptación a Python de la clase "Random" para C++ colgada en la web de la asignatura. El nombre de las funciones es prácticamente el mismo y simplemente he "traducido" este código original para poder utilizarlo con este lenguaje. Para utilizar esta clase, desde el "main" creo un objeto "Random" generado a partir de la semilla 15041995 y lo paso al algoritmo necesario para que se obtenga un número aleatorio utilizando las opciones que proporciona cada método
- utilidades.py: aquí introduzco los métodos que aparecen en el apartado 3 de este documento, es decir, los métodos comunes a los distintos algoritmos como pueden ser el cálculo de distancias, el clasificador o el evaluador de una solución, así como los métodos necesarios para leer los ficheros en el "main" e inicializar las listas de datos.
- relief.py: como es de esperar, en este archivo están implementados los métodos necesarios para ejecutar desde el "main" el algoritmo "Relief" y el algoritmo 1-NN simple.
- buscLocal.py: en este caso los métodos los métodos que aparecen son los utilizados para lanzar el algoritmo de búsqueda local.
- main.py: es el programa principal desde el que se llama a los métodos antes descritos. De forma resumida es:

```
algoritmos=["1nn","Relief","BusquedaLocal"](1)
Para cada set de datos:
   Introducir en lista repartirClase("<nombre set>")
   Introducir en listaNormalizada normalizar(lista)
generadorAleatorios = crear objeto Random a partir de una semilla
Para cada algoritmo en algoritmos:
   Para cada listaNormalizada:
      Para cada partición:
         Si no es la primera iteración: reordenar listaNormalizada (2)
         Lanzar algoritmo
```

A partir de este esquema, haré dos comentarios:

1. Esta lista de algoritmos esta al principio del "main" y es donde deben ponerse los algoritmos que se desea ejecutar, actualmente los tres que aparecen en el pseudocódigo de arriba
2. En la lista de cada set de datos normalizada ya están los datos para empezar a trabajar con ellos normalizados y divididos en cinco particiones (más una de los nombres), pero la reordeno para que cada vez la partición de validación sea una diferente de las cinco. Para ello Python permite hacer lo siguiente:

ozonoNormalizado[partición],ozonoNormalizado[5]=ozonoNormalizado[5],ozonoNormalizado[partición]

De esta forma se intercambian las listas en las posiciones <partición> y 5 de modo que cada partición será una vez la que esté en la posición 5 de la lista normalizada, siendo tratada en los algoritmos como partición de validación.

Para terminar, antes de lanzar el programa se deben instalar las librerías necesarias llamadas liac-arff y numpy, que pueden instalarse desde terminal con el comando:

```
Pip install liac-arff
```

```
Pip install numpy
```

Si estuviera instalada previamente la librería de nombre arff es probable que la ejecución falle por lo que habría que desinstalarla y dejar solo la anterior. Para ello el comando sería:

```
Pip uninstall arff
```

Ahora sí, para lanzar el programa se debe ejecutar el comando siguiente desde la carpeta raíz:

```
Python main.py
```

7. Experimentos y análisis de resultados

Antes de empezar a analizar los resultados obtenidos voy a explicar los conjuntos de datos que ya se han mencionado antes, sobre el ozono, el parkinson y la fisiología del corazón:

- "ozone-320.arff": 320 ejemplos con 73 características (incluida la clase) con una distribución de clases del 50%
- "parkinsons.arff": 195 ejemplos con 23 características (incluida la clase) con una distribución de clases de un 30-70% aproximadamente
- "spectf-heart.arff": 349 ejemplos con 45 características(incluida la clase) con una distribución de clases de un 73-27% aproximadamente

Además, decir que los tiempos reflejados en las tablas están en segundos y se han obtenido a partir de la librería "time" de Python e indican el tiempo de ejecución del algoritmo de búsqueda del vector de peso y el tiempo de comprobar la solución.

Analizados los conjuntos de datos y hecho este último apunte, paso a mostrar las tablas de resultados para cada uno de los algoritmos:

Algoritmo 1-NN

	Ozone				**Parkinsons**				**Spectf-heart**			
	%_clas	***%red***	***Agr.***	**T**	***%_clas***	***%red***	***Agr.***	**T**	***%_clas***	***%red***	***Agr.***	**T**
Partición 1	79,6875	0,0000	39,8438	0,7760	94,7368	0,0000	47,3684	0,0990	75,3623	0,0000	37,6812	0,5990
Partición 2	84,3750	0,0000	42,1875	0,7790	95,0000	0,0000	47,5000	0,1000	74,2857	0,0000	37,1429	0,5910
Partición 3	82,8125	0,0000	41,4063	0,7800	97,5000	0,0000	48,7500	0,1040	67,1429	0,0000	33,5714	0,5860
Partición 4	75,0000	0,0000	37,5000	0,7780	97,4359	0,0000	48,7179	0,0970	80,0000	0,0000	40,0000	0,6190
Partición 5	82,8125	0,0000	41,4063	0,8020	97,3684	0,0000	48,6842	0,0960	68,5714	0,0000	34,2857	0,5960
Media	80,9375	0,0000	40,4688	0,7830	96,4082	0,0000	48,2041	0,0992	73,0725	0,0000	36,5362	0,5982
Desv. Típica	3,7304	0,0000	1,8652	0,0107	1,4095	0,0000	0,7047	0,0031	5,2470	0,0000	2,6235	0,0126

Este algoritmo no tiene en cuenta pesos, aunque se puede comprobar como la media del conjunto "Parkinson" es realmente alta, si bien es cierto que en los tres algoritmos ejecutados este es el set de datos que mejor clasifica. Incluyo también en la tabla las desviaciones típicas de los valores para comprobar que no hay una gran diferencia al utilizar para clasificar unas u otras particiones, observando que los datos más dispares de clasificación se encuentran en las particiones 3 y 4 del conjunto de datos "Spectf-heart". La media de las clasificaciones con este método es del 83.4727%

Por último, comentar que el porcentaje de reducción es siempre 0 al utilizar como vector de pesos (como se ha comentado antes esto simula que no se utiliza vector de pesos como tal) una lista de unos.

Algoritmo Relief

	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	81,2500	26,3889	53,8194	3,2730	97,3684	4,5455	50,9569	0,4210	72,8571	0,0000	36,4286	2,4220
Partición 2	78,1250	8,3333	43,2292	3,2400	94,7368	4,5455	49,6411	0,4370	84,0580	0,0000	42,0290	2,4250
Partición 3	75,0000	16,6667	45,8333	3,2480	97,5000	0,0000	48,7500	0,4250	68,5714	0,0000	34,2857	2,3810
Partición 4	81,2500	18,0556	49,6528	3,2400	95,0000	4,5455	49,7727	0,4180	64,2857	0,0000	32,1429	2,3720
Partición 5	71,8750	25,0000	48,4375	3,2370	97,4359	4,5455	50,9907	0,4400	84,2850	0,0000	42,1425	2,3870
Media	77,5000	18,8889	48,1944	3,2476	96,4082	3,6364	50,0223	0,4282	74,8114	0,0000	37,4057	2,3974
Desv.Tipica	4,0745	7,2569	4,0017	0,0148	1,4095	2,0328	0,9536	0,0098	9,0663	0,0000	4,5332	0,0244

Aunque cabría esperar que este algoritmo mejorara al anterior en cuanto a la tasa de clasificación esto no es así, pues solo mejora en un 1% la clasificación del set "Spectf-heart". En cambio en el set "Ozono" se pierde un 3.5% de porcentaje de clasificación correcto, mientras que el set "Parkinson" se queda igual. En media, este algoritmo empeora al algoritmo 1-NN sin pesos pues es de un 82.9066% frente al 83.4727% del algoritmo anterior

Con respecto a las tasas de reducción en este caso no son cero, aunque los números no son muy elevados al no optimizarse esta cuestión en el algoritmo. Si se considera esta tasa junto con la de clasificación, la agregada, el algoritmo sí que mejora al 1-NN. De nuevo, la mayor dispersión de los datos de clasificación está en el set "Spectf-heart" lo que indicaría que es un conjunto de datos que en ocasiones puede alcanzar un número alto de aciertos pero en otras este número baja considerablemente, por lo que la fiabilidad no sería muy alta. Dicho esto, habrá que esperar al algoritmo de Búsqueda Local para ver como este "problema" se soluciona.

Algoritmo de Búsqueda Local

	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	81,2500	41,6666	61,4583	2683,48	94,8718	31,8182	63,3450	36,381	100,0000	36,3636	68,1818	45,134
Partición 2	87,5000	27,7778	57,6389	1622,52	97,3684	40,9091	69,1388	42,404	97,5000	27,2727	62,3864	61,479
Partición 3	90,6250	41,6667	66,1458	2463,46	94,7368	40,9091	67,8230	47,902	95,0000	40,9091	67,9545	68,486
Partición 4	84,3750	25,0000	54,6875	1776,11	95,0000	45,4545	70,2273	33,455	97,5000	36,3636	66,9318	40,678
Partición 5	82,8125	33,3333	58,0729	1151,23	97,5000	31,8182	64,6591	140,187	95,0000	22,7273	58,8636	44,741
Media	85,3125	33,8889	59,6007	1939,36	95,8954	38,1818	67,0386	60,0658	97,0000	32,7273	64,8636	52,104
Desv.Tipica	3,7630	7,7080	4,3764	627,81	1,4086	6,0984	2,9368	45,134	2,0917	7,4689	4,0920	12,141

Puede comprobarse como este algoritmo ya sí que obtiene unos porcentajes elevados en todos los set de datos, pues aunque pierde un 0.5% en la clasificación del set "Parkinsons" este sigue siendo muy alto, así como los otros dos que mejoran en gran medida con respecto a los anteriores algoritmos. En este caso la media de clasificación es de un 92.736%, cerca de un diez por ciento mejor que los anteriores algoritmos.

Además, como este algoritmo ya sí que busca reducir el número de características a considerar a la hora de clasificar, el porcentaje de reducción aumenta con respecto a los anteriores métodos, situándose entre un 30 y un 40 por ciento en los tres sets.

Con respecto a la desviación típica es bastante baja en los porcentajes de clasificación y mejora a los anteriores algoritmos, aunque la tasa de reducción sí que tiene una dispersión algo elevada en los tres sets.

Por último se puede observar como los tiempos de ejecución de este algoritmo son muy altos en comparación con los demás, pues se comprueban muchas más soluciones posibles, aunque bien es cierto que la calidad del vector de pesos que encuentra es mucho más alta si se tiene en cuenta la tasa agregada como es nuestro caso.

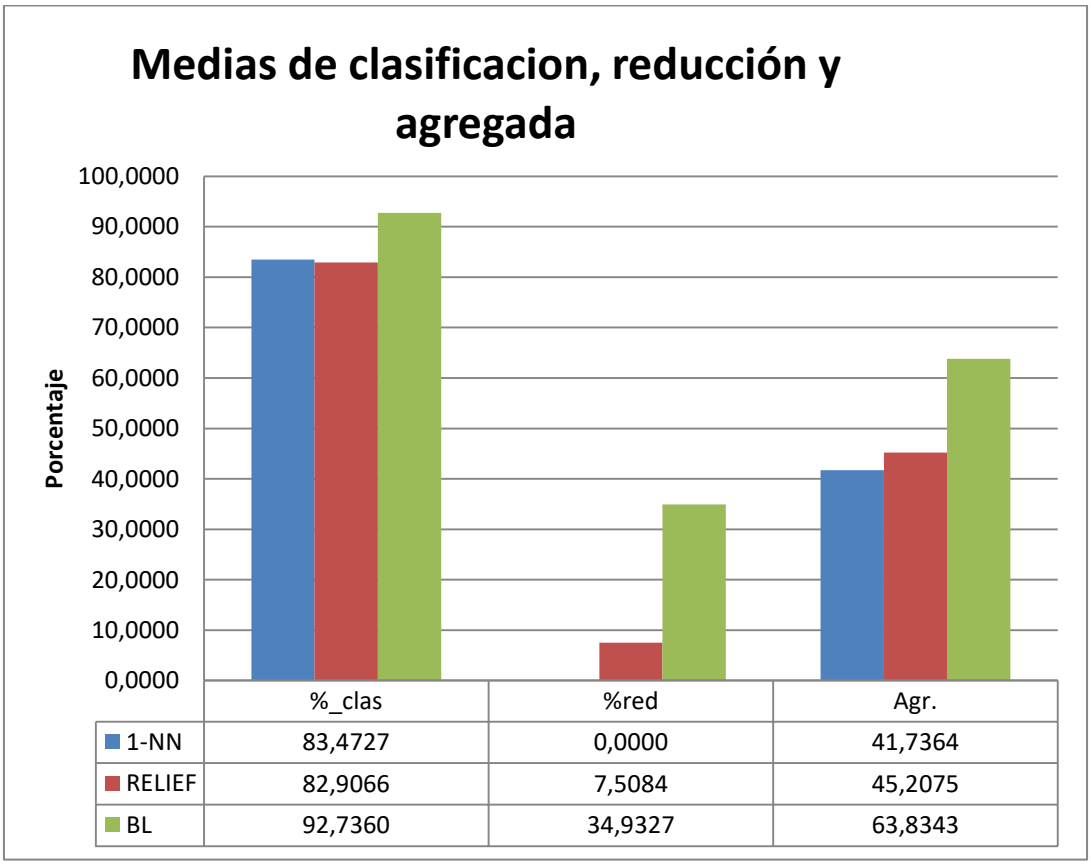
Comparación de resultados globales

En la tabla que muestro a continuación se pueden ver los resultados medios para todas las tasas y tiempos de todos los algoritmos en cada set de datos:

	Ozone				**Parkinsons**				**Spectf-heart**			
	%_clas	***%red***	***Agr.***	**T**	***%_clas***	***%red***	***Agr.***	**T**	***%_clas***	***%red***	***Agr.***	**T**
1-NN	80,9375	0,0000	40,4688	0,7830	96,4082	0,0000	48,2041	0,0992	73,0725	0,0000	36,5362	0,5982
RELIEF	77,5000	18,8889	48,1944	3,2476	96,4082	3,6364	50,0223	0,4282	74,8114	0,0000	37,4057	2,3974
BL	85,3125	33,8889	59,6007	1939,3594	95,8954	38,1818	67,0386	60,0658	97,0000	32,7273	64,8636	52,1036

Al ver la tabla, algo que llama la atención es la dependencia que existe del set de datos, pues los porcentajes de clasificación y reducción no son uniformes en los tres conjuntos. Por ello, se debe ser consciente de que los datos analizados van a influir en una alta medida en el resultado de la clasificación.

Por otro lado destaca la gran superioridad del algoritmo de Búsqueda Local frente a los otros dos, pues su tasa de acierto esta siempre por encima del 85% en media, aunque esto se puede ver mejor en el siguiente gráfico:



Como comentaba, el algoritmo de Búsqueda Local consigue, en media, mejores valores tanto en la tasa de clasificación como en la de reducción y por tanto en la agregada. Así, parece claro que el mejor algoritmo de los tres usados es sin duda el de Búsqueda Local aunque el tiempo de ejecución sea mucho mayor. Sin embargo, entre los otros dos podría haber más dudas. En cuanto a la tasas mostradas, la única diferencia existe en la de reducción, la de clasificación es solo medio punto menor, por lo que habría que decidir si este 7.5% de aumento merece la pena. Al estar tan igualadas podría decidirse a partir del tiempo medio de ejecución de los algoritmos:

	Totales
	T
1-NN	0,4935
RELIEF	2,0244

Como se puede comprobar, el algoritmo "Relief" tarda 1.5 segundos más, no siendo el tiempo excesivamente grande. Por tanto, si se desea optimizar la tasa de clasificación sin importar la de reducción podría optarse por el algoritmo 1-NN, aunque si ese 7.5% de diferencia fuera importante no debería de haber demasiado problema en función del caso concreto en gastar 1.5 segundos más.

8. Bibliografía

La bibliografía que he utilizado para desarrollar la práctica ha sido básicamente la que aparece en la web de la asignatura, tanto el guion como las diapositivas del seminario 2 de prácticas sobre los problemas QAP y APC.

Además, he utilizado también una práctica de otro año compartida en github en el enlace https://github.com/NestorRV/UGR-MH/tree/master/APC

Por otro lado, he estudiado también algunos conceptos como la varianza o la distribución normal en sus respectivas páginas de https://es.wikipedia.org y he utilizado varias entradas de la página https://stackoverflow.com para resolver dudas sobre Python.