UNIVERSIDAD DE GRANADA

Práctica 3B
Aprendizaje de Pesos en Características:
Enfriamiento Simulado, Búsqueda Local Reiterada y Evolución Diferencial

Alberto Lejárraga Rubio

DNI: 50618389N

Email: alejarragar@correo.ugr.es

Granada, 9 de junio de 2018

1. Índice

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 los 'k' 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.

Estos valores pueden aparecer en el vector de pesos o cromosoma (W) de la siguiente forma:

- $w_i<0.2$: 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.

Además, como se hizo en la práctica anterior para algoritmos genéticos y meméticos, en el algoritmo de Differential Evolution se han añadido dos posiciones más a esta lista para intentar optimizar y acelerar el proceso de búsqueda de soluciones. Estas dos posiciones no se tendrán en cuenta a la hora de clasificar los ejemplos pero reducen el número de evaluaciones que se deben hacer en los algoritmos, que de realizarse provocarían un gran aumento en el tiempo de ejecución. Se añaden al final de la lista e indican si un cromosoma es el mejor de una generación mediante un booleano y la tasa agregada de dicho cromosoma, respectivamente.

En los otros dos algoritmos estas dos posiciones no se utilizan.

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 una lista con "sublistas". La lista principal contendrá una lista de los nombres de los atributos y las n (5 en la práctica) particiones realizadas al set completo de datos:

[[<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 su tipo:

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

Por su parte, las listas <partición_1,2,…,n> representan cada una de las particiones que se realizan al conjunto de datos.

[[<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 del fichero correspondiente, se normalizan y se va metiendo un dato en cada partición:

```
listaResul=[]
Añadir a listaResul una lista con los nombres de los atributos
numClases=<numero_de_clases_del_set>
Crear lista meterClase de tamaño numClases y añadir 0 en sus numClases posiciones
#esta lista indicara en qué partición se mete el siguiente ejemplo que venga de una de las clases
Añadir a listaResul tantas listas vacías como particiones se desee crear
Normalizar los datos en función del máximo y mínimo valor de cada atributo
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.

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

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

[[0.8 , 0.6 ,…, 1], [0.3 , 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 muy elevado
      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.

Función objetivo – Evaluación de la solución

La evaluación de la solución se realiza en base a las tasas de acierto y reducción, por lo que se tiene en cuenta 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
```

Generación de la población-solución inicial

Aunque la generación de una población inicial solo tenga sentido en el algoritmo *Differential Evolution(DE),* he decidido incluir aquí este apartado para seguir el mismo esquema que en prácticas anteriores. Por su parte, la solución inicial en los algoritmos *Simulated Annealing(SA)* e *Iterative Local Search(ILS)* se genera de manera completamente aleatoria.

El algoritmo que he desarrollado para generar la población utiliza siempre los mismos cromosomas iniciales en cada una de las llamadas al *Differential Evolution* para que, de esta forma, la ejecución dependa del algoritmo en sí y no tanto de la población inicial que se utilice.

Para ello, la primera vez que se ejecuta el programa se genera la población y se introduce en un fichero. A partir de la segunda, se comprobará que este fichero ya existe y se obtendrá la población de él. Con esto, además se consigue una reducción del tiempo de ejecución del algoritmos, al no tener que generar para cada partición de cada set de datos una nueva población con <tamaño de la población>*<número de cromosomas>* <número de genes> números aleatorios. Por otro lado, habría que evaluar también cada uno de los cromosomas para obtener su tasa agregada e introducirla en la última posición del cromosoma como se comentó anteriormente, lo que conllevaría un elevado gasto de tiempo.

La estructura de estos archivos (uno para cada set) es:

<tamaño de la población>
<número de genes>
<gen 1 del cromosoma 1>
…
<gen n del cromosoma 1>
<tasa agregada del cromosoma 1>
<gen 1 del cromosoma 2>

…
<tasa agregada del último cromosoma>

Para explicar la generación de la población inicial no tendré en cuenta lo comentado sobre los ficheros para que el pseudocódigo no sea demasiado complejo. En la primera ejecución, además del pseudocódigo aquí indicado, se escribirán los datos en un fichero y a partir de la segunda, en vez de generar valores aleatorios se obtendrán del archivo:

```
Función generarPoblaciónInicial:
      numeroDeGenes = <obtener este número a partir del número de atributos del set>
      Crear lista vacía de nombre población
      Desde 0 hasta <tamaño de la población>:
            Crear lista vacía de nombre cromosoma
            Desde 0 hasta numeroDeGenes:
                  Añadir a cromosoma <valor aleatorio en [0,1)>
            Añadir a población el cromosoma
      Retornar población
```

Algoritmo para generar vecino

Esto se utiliza tanto en ILS como SA. El funcionamiento se basa en que a partir de una solución dada y un atributo a modificar de esta solución, se genera otra igual pero modificando este atributo utilizando un número aleatorio perteneciente a una distribución normal de media 0 y varianza 0.3:

```
Función generarVecino(solución, distNormal, caractAModificar):
      soluciónAux=copia de solución
      nuevoValor=extraer un numero de distNormal
   Modificar caractAModificar de soluciónAux sumándole nuevoValor
      Si caractAModificar de soluciónAux es > 1:
            caractAModificar de soluciónAux = 1
      Si caractAModificar de soluciónAux es < 0:
            caractAModificar de soluciónAux =0
      Retornar soluciónAux
```

4. Descripción de los algoritmos

Iterative Local Search (ILS)

El algoritmo de Búsqueda Local Iterativa trata de encontrar soluciones realizando una búsqueda local sobre soluciones buenas encontradas que se van modificando. Al principio se genera una solución aleatoria y a partir de esta el mecanismo consiste en realizarle una búsqueda local. Tras esto se entrará en un bucle un número de veces que irá modificando la mejor solución encontrada y aplicándole una búsqueda local.

El pseudocódigo sería el siguiente:

```
Función ILS(setEjemplos, alfa, sigmaMutación, sigmaBuscLocal,iteraciones):
	Generar SoluciónInicial
	Aplicar la búsqueda local a la SoluciónInicial e introducir en mejorSolución
	numCaracteristicasAMutar=0.1 * número de atributos del problema
	veces=1;
	Mientras veces < iteraciones:
		SoluciónModificada = mejorSolución
		Para i = 0 hasta numCaracteristicasAMutar:
			SoluciónModificada = generarVecino(soluciónModificada, <distribución
normal con sigma =sigmaBuscLocal>,<numero aleatorio entre 0 y numero de características>)
		Aplicar búsqueda local a soluciónModificada
		Si calidad soluciónModificada es mejor que calidad de mejorSolución:
			mejorSolución = soluciónModificada
		veces = veces + 1
```

Operador de mutación

El operador de mutación empleado en el algoritmo se basa en modificar un diez por ciento de la mejor solución encontrada utilizando la función "generarVecino" con una distribución normal de varianza 0.4 y media 0. Debe apreciarse que las distribuciones normales utilizadas para la búsqueda local y para generar una mutación son distintas, pues en la búsqueda local se utiliza una varianza 0.3. Esto hace que la modificación sea mayor en este paso que en el operador de mutación de la búsqueda local, en el que se modifica una característica y con una diferencia menor (en media) debido a la varianza.

Con respecto a los valores de los parámetros que aparecían en el guion referidos al número de iteraciones del bucle y al número de evaluaciones a realizar en la búsqueda local he decidido reducirlos, al necesitar demasiado tiempo de ejecución (unas dos horas y media de media frente a algo menos de una hora con este número de iteraciones para el set de datos del ozono, por ejemplo). Además como explicaré en el análisis de resultados, las soluciones encontradas no siempre eran mejores al ejecutar todo completamente.

Estos nuevos valores han sido 10 para el número de iteraciones en vez de 15 y 500 a las evaluaciones a realizar en la búsqueda local en vez de 1000.

Simulated Annealing (SA)

El algoritmo de enfriamiento simulado se basa en una imitación a un proceso físico del campo de la termodinámica. La característica principal de este algoritmo es que permite aceptar una solución peor en base a una probabilidad que va disminuyendo a lo largo del algoritmo. El pseudocódigo del algoritmo es el siguiente:

```
Función SimulatedAnnealing(phi, mu, tempFinal,…):
	Generar SoluciónInicial e introducir en mejorSolucion
	CalidadMejorSolucion = evaluar mejorSolucion
	temperaturaActual = mu * calidadMejorSolucion/(-log(phi))
	maxVecinos = 10 * <número de características>
	maxExitos = 0.1 * maxVecinos
	numEvaluaciones = 1
	numExitos = 1
	numIteracionesMáximas = 15000/maxVecinos
	beta = (tempActual-tempFinal)/(numIteracionesMáximas*tempFinal*tempActual)
	solucionActual=mejorSolucion
	calidadSolActual=calidadMejorSol
	Mientras numEvaluaciones < 15000, numExitos > 0 y tempActual>tempFinal:
		Iteraciones=0
		numExitos=0
		Mientras iteraciones < maxVecinos y numExitos < maxExitos:
			soluciónNueva =generarVecino(solucionActual,<distNormal>,aleatorio)
```

Cálculo de la temperatura inicial

Cálculo de beta para realizar el enfriamiento

```
                    calidadSolNueva = evaluarSolucion(solucionNueva)
                    Si    calidadSolNueva    >    calidadSolActual    or    (<aleatorio>    <=
exp((calidadSolNueva-calidadSolActual)/tempActual)):
                          solucionActual = solucionNueva
                          calidadSoluciónActual = calidadSoluciónNueva
                          numExitos = numExitos +1
                          Si calidadSolActual > calidadMejorSolucion:
                                mejorSolución = soluciónActual
                                calidadMejorSolución = calidadSoluciónActual
                    iteraciones = iteraciones+1
              numEvaluaciones = numEvaluaciones + iteraciones
              tempActual = temperaturaActual/(1+beta*temperatura)
        Retornar calidadMejorSolución
```

Se permite empeorar la solución actual

Realizar el enfriamiento

En cuanto al cálculo de la temperatura inicial se puede ver recuadrado en azul como se hace en función de dos parámetros mu y phi, con valores ambos 0.3. Además influye también la calidad de la solución inicial generada aleatoriamente.

Por otro lado, el esquema de enfriamiento está dividido en dos partes (ambas señaladas en naranja). La primera de ellas define una "constante" beta que influye en lo rápido o lento que disminuye la temperatura, y se define así para realizar siempre 15000/10*<número de características> enfriamientos. Después en la segunda parte, al final del bucle externo, se realiza el enfriamiento en sí.

Differential Evolution (DE)

Este algoritmo es un modelo evolutivo que basa su potencial en el modelo de cruce que utiliza. En la práctica se han utilizado dos tipos, que se diferencian fundamentalmente dicho cruce. El CTB utiliza para cruzar dos padres aleatorios y cada uno de los cromosomas de la población y el aleatorio utiliza tres padres cualquiera de la población.

DE con cruce CTB

Este modelo realiza en cada iteración tantos cruces como cromosomas haya en la población. Para cada uno de ellos, se seleccionan dos padres aleatoriamente (distintos del cromosoma actual) y se realiza la mutación en base a los dos padres, al cromosoma actual y al mejor cromosoma encontrado globalmente. Una vez obtenidos los padres habrá que decidir si se muta cada gen con una probabilidad de cruce obteniendo un número aleatorio. Se recorrerá el cromosoma completo y si se decidiera no mutar el gen nuevo será el del cromosoma actual, mientras que si se muta, el nuevo gen será:

$$cromNuevo[i] = cromActual[i] + F * (mejorPoblación[i] - cromActual[i]) + F * (padre1[i] - padre2[i])$$

i ➔Variable que itera sobre cada uno de los genes de cada cromosoma
cromNuevo➔Cromosoma que reemplazará en la población a "cromActual"
mejorPoblación➔Mejor solución encontrada globalmente
F➔Parámetro pasado al algoritmo, en este caso 0.5
Padre1 y Padre2 ➔ Padres seleccionados aleatoriamente

Una vez realizado el cruce de todos los genes de todos los cromosomas de la población se introducirán si mejoran al cromosoma de la población actual situado en su misma posición. El pseudocódigo es el siguiente:

```
Función DE-CTB (poblaciónInicial,tamañoPoblación, numeroEvaluaciones, F, probCruce,…):
      Contador=tamañoPoblación
      poblaciónActual=poblaciónInicial
      tasaMejorSolución=0
      índiceMejorSol =10000
```

```
Buscar la mejor solución y actualizar tasaMejorSolución e índiceMejorSol
Mientras contador < numeroEvaluaciones:
        Generar lista vacia resultadoIteración
        Para cada cromosoma de poblaciónActual:
                Padre1,padre2=seleccionarPadres(cromosoma,población,tamanoPoblación)
                Generar lista vacía resultadoCruce
                Para cada gen de cromosoma:
                        Generar aleatorio
                        Si aleatorio < probCruce:
                                Añadir a resultadoCruce cromNuevo[gen]
                                Si resultadoCruce[gen]>1:resultadoCruce[gen]=1
                                Si resultadoCruce[gen]<0:resultadoCruce[gen]=0
                        En otro caso:
                                Añadir a resultadoCruce poblaciónActual[gen]
                Evaluar cromosoma añadido y almacenar su evaluación
                Contador=contador+1
                Añadir resultadoCruce a resultadoIteración
        tasaMejorSolución=0
        índiceMejorSol =10000
        Para cada cromosoma de poblaciónActual:
                Si resultadoIteración[cromosoma] es mejor que poblaciónActual[cromosoma]:
                        poblaciónActual[cromosoma]=resultadoIteración[cromosoma]
                Si poblaciónActual[cromosoma] es mejor que tasaMejorSolución:
                        tasaMejorSolución = valoración de poblaciónActual[cromosoma]
                        índiceMejorSol=cromosoma
                contador = contador+3
        Devolver poblaciónActual
```

Selección de los padres, algoritmo a continuación

CromNuevo[gen] se refiere a la fórmula indicada justo arriba

Comprobar si hay mejora cromosoma a cromosoma e introducirlo en ese caso

La forma de realizar la selección de los padres, marcada en verde, es la siguiente:

```
Función seleccionarPadres(prohibido,población,tamañoPoblación,…):
        índicePadre1=aleatorio entre 0 y tamañoPoblación
        Mientras índicePadre1==prohibido:
                índicePadre1=aleatorio entre 0 y tamañoPoblación
        índicePadre2=aleatorio entre 0 y tamañoPoblación
        Mientras índicePadre2==prohibido o indicePadre2==índicePadre1:
                índicePadre2=aleatorio entre 0 y tamañoPoblación
        Devolver población[padre1] y población[padre2]
```

Cromosoma distinto al actual

Cromosoma distinto al actual y al padre1

Como se puede comprobar, la función recibe el índice de un cromosoma de la población "prohibido" que se refiere al cromosoma actual del algoritmo. Con esto se consigue que los padres sean distintos al cromosoma que se está estudiando, pues este ya aparece en la fórmula de *cromNuevoGen.* Por otro lado, los padres deben ser siempre distintos entre ellos.

DE con cruce aleatorio

En este caso, el algoritmo es similar pero no utiliza para cruzar ni la mejor solución encontrada ni el cromosoma que se está modificando en cada iteración. Además, en vez de generar dos padres, genera tres, que serán los que realicen el cruce:

```
Función DE-CTB (poblaciónInicial,tamañoPoblación, numeroEvaluaciones, F, probCruce,…):
    Contador=tamañoPoblación
    poblaciónActual=poblaciónInicial
    Mientras contador < numeroEvaluaciones:
        Generar lista vacia resultadoIteración
        Para cada cromosoma de poblaciónActual:
            Padre1,padre2,padre3 =seleccionarPadres(cromosoma,población,tamanoPobl)
            Generar lista vacía resultadoCruce
            Para cada gen de cromosoma:
                Generar aleatorio
                Si aleatorio < probCruce:
                    Añadir a resultadoCruce padre1[gen]+F*(padre2[gen]-padre3[gen])
                    Si resultadoCruce[gen]>1:resultadoCruce[gen]=1
                    Si resultadoCruce[gen]<0:resultadoCruce[gen]=0
                En otro caso:
                    Añadir a resultadoCruce poblaciónActual[gen]
            Evaluar cromosoma añadido y almacenar su evaluación
            Contador=contador+1
            Añadir resultadoCruce a resultadoIteración
        Para cada cromosoma de poblaciónActual:
            Si resultadoIteración[cromosoma] es mejor que poblaciónActual[cromosoma]:
                poblaciónActual[cromosoma]=resultadoIteración[cromosoma]
            contador = contador+3
    Devolver poblaciónActual
```

Selección de los padres, algoritmo a continuación

Cruzar padres

Comprobar si hay mejora cromosoma a cromosoma e introducirlo en ese caso

En este modelo de algoritmo diferencial se seleccionan, como se ha dicho, tres padres por lo que en la función *seleccionarPadres* para este algoritmo habrá que seleccionar un padre más que antes. El algoritmo es igual pero seleccionando un tercero que tampoco sea igual a "prohibido", a "padre1" y a "padre2".

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:

```
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

Para 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
```

El programa se puede ejecutar tanto con los archivos de nombre poblaciónInicialDE<partición>.txt como sin ellos, pues estos se generan tras la primera llamada a los algoritmos evolutivos.

Para lanzar el programa se debe ejecutar el comando siguiente desde la carpeta raíz:

```
Python main.py
```

El programa solicitará la semilla que habrá que introducir. Yo he utilizado mi fecha de nacimiento 15041995

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 que después analizaré

Simulated Annealing

Tabla 5.1: Resultados obtenidos por el algoritmo Simulated Annealing en el problema del APC

	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	87,5000	35,1111	61,3056	1086,8160	94,7368	67,1818	80,9593	111,3190	92,7536	46,7273	69,7404	911,2150
Partición 2	90,6250	53,1667	71,8958	924,0030	95,0000	76,2727	85,6364	107,3510	88,5714	58,0909	73,3312	712,5710
Partición 3	85,9375	53,1666	69,5521	1012,3610	100,0000	62,6364	81,3182	114,6360	78,5714	39,9091	59,2403	802,4560
Partición 4	85,9375	37,8888	61,9132	1087,1000	94,8718	53,5455	74,2086	118,8360	82,8571	44,4545	63,6558	745,4550
Partición 5	85,9375	46,2222	66,0799	987,1480	97,3684	62,6364	80,0024	117,2590	88,5714	44,4545	66,5130	805,3570
Media	87,1875	45,1111	66,1493	1019,4856	96,3954	64,4545	80,4250	113,8802	86,2650	46,7273	66,4961	795,4108
Desv.Tipica	2,0373	8,4140	4,6372	69,4955	2,2891	8,2572	4,0946	4,6297	5,5576	6,8182	5,4304	75,6921

En cuanto a este primer algoritmo, el de Enfriamiento Simulado, destaca sobre todo el reducido tiempo de ejecución que emplea para conseguir resultados muy buenos. De hecho, se coloca entre los mejores algoritmos de todos los probados y es de los que menos tiempo utiliza, por lo que parece que es un algoritmo bastante bueno, por lo menos para los datos y el problema que se están utilizando en los experimentos.

Iterative Local Search

Tabla 5.1: Resultados obtenidos por el algoritmo ILS en el problema del APC

	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	85,9375	39,2777	62,6076	3269,1130	97,3684	39,9090	68,6387	383,2840	91,3040	37,6363	64,4702	2641,3130
Partición 2	87,5000	42,0555	64,7778	3269,5980	97,5000	49,0000	73,2500	364,2880	80,0000	30,8182	55,4091	2905,9400
Partición 3	84,3750	30,9444	57,6597	3691,2080	92,5000	30,8182	61,6591	392,6900	84,2857	44,4545	64,3701	2437,3490
Partición 4	82,8125	32,3333	57,5729	3597,2940	92,3077	53,5455	72,9266	323,4910	90,0000	49,0000	69,5000	2466,1040
Partición 5	85,9375	29,5555	57,7465	3735,3870	100,0000	53,5455	76,7727	347,2530	70,0000	49,0000	59,5000	2212,5720
Media	85,3125	34,8333	60,0729	3512,5200	95,9352	45,3636	70,6494	362,2012	83,1179	42,1818	62,6499	2532,6556
Desv.Tipica	1,7815	5,5032	3,3929	227,5103	3,3906	9,8543	5,7947	27,8403	8,6247	7,8730	5,3747	258,4152

Como se ha explicado en el apartado de implementación de esta metaheurística, he reducido el número tanto de evaluaciones para la búsqueda local (500) como el de las iteraciones del bucle de la búsqueda reiterada (10). Aunque el tiempo se ha reducido alrededor de un tercio creo que sigue siendo demasiado alto para los resultados que ofrece, que aunque no son malos, quedan por detrás de muchos otros algoritmos.

Además, comprobé que no en todos los casos mejoraba la calidad de los datos con la versión larga con respecto a la versión final que muestro aquí, de hecho el porcentaje de casos en los que la versión larga era mejor que la corta era "solo" de algo más de un 70% y no con una distancia muy grande. Esto creo que era así por la aleatoriedad presente en el algoritmo, pues como en todas estas metaheurísticas los números aleatorios generados juegan un papel fundamental.

Por ello creo que no merece la pena darle un mayor tiempo de ejecución al algoritmo, porque no mejora mucho aunque el número de evaluaciones de la función objetivo sea mayor.

Differential Evolution con cruce aleatorio

Tabla 5.1: Resultados obtenidos por el algoritmo DE CON CRUCE ALEATORIO en el problema del APC

	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	90,6250	73,3243	81,9747	2514,6720	100,0000	86,5000	93,2500	239,4640	91,3046	83,7826	87,5436	1887,5800
Partición 2	95,3125	74,6757	84,9941	2445,4820	100,0000	82,3330	91,1665	261,6120	91,4286	81,6087	86,5186	1826,4840
Partición 3	95,3125	73,3243	84,3184	2349,0480	95,0000	86,5000	90,7500	257,9270	91,4286	77,2609	84,3447	1965,8150
Partición 4	85,9375	78,7297	82,3336	2522,9410	92,3077	86,5000	89,4038	249,4830	91,4286	81,6087	86,5186	1671,9860
Partición 5	93,7500	78,7297	86,2399	2448,5090	100,0000	86,5000	93,2500	242,9130	95,7143	88,1304	91,9224	1590,1370
Media	92,1875	75,7568	83,9721	2456,1304	97,4615	85,6666	91,5641	250,2798	92,2609	82,4783	87,3696	1788,4004
Desv.Tipica	3,9836	2,7694	1,8015	69,8729	3,6039	1,8635	1,6712	9,4708	1,9312	3,9491	2,7998	154,6153

Con respecto a este algoritmo sorprenden mucho sus porcentajes de reducción, ya que la mayoría de algoritmos probados hasta ahora no conseguían tal reducción para todas las particiones de todos los sets de datos. Además, consigue esto con porcentajes de clasificación que tampoco se habían obtenido hasta ahora, aunque es cierto que en este caso está más parejo con el resto de algoritmos.

Obviamente, la tasa agregada también es la mejor de todas las metaheurísticas estudiadas

Differential Evolution con cruce CTB

Tabla 5.1: Resultados obtenidos por el algoritmo DE CON CRUCE CTB en el problema del APC

	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	92,1875	55,7568	73,9721	2090,0780	97,3684	78,1666	87,7675	222,8620	95,6522	66,3913	81,0217	1719,9700
Partición 2	96,8750	67,9189	82,3970	1932,2340	97,5000	78,1666	87,8333	242,5840	90,0000	68,5652	79,2826	1914,6350
Partición 3	87,5000	66,5676	77,0338	1970,3220	95,0000	82,3333	88,6667	238,1390	92,8571	66,3913	79,6242	1883,3890
Partición 4	89,0625	63,8649	76,4637	2204,8630	92,3077	82,3330	87,3203	238,0700	91,4286	62,0435	76,7360	2003,0110
Partición 5	93,7500	69,2703	81,5101	2166,0980	97,3684	78,1667	87,7675	243,0570	91,4286	64,2174	77,8230	1868,5620
Media	91,8750	64,6757	78,2753	2072,7190	95,9089	79,8332	87,8711	236,9424	92,2733	65,5217	78,8975	1877,9134
Desv.Tipica	3,7304	5,3715	3,5635	119,0632	2,2686	2,2821	0,4897	8,2185	2,1420	2,4786	1,6595	102,5444

En este caso se consiguen de nuevo muy buenos resultados, solo por detrás del otro modelo de Evolución Diferencial, el de cruce aritmético. De hecho, sorprende que este obtenga unos resultados peores (por lo menos a mí) ya que este utiliza para generar nuevas soluciones a la mejor solución encontrada y el anterior es completamente aleatorio, aunque después solo reemplace soluciones si son mejores que las de la población anterior.

Esto seguramente se deba a que la exploración del algoritmo con cruce aritmético sea mayor que la del de cruce CTB y la explotación siga siendo buena, aunque no se utilice a esta mejor solución encontrada como "guía".

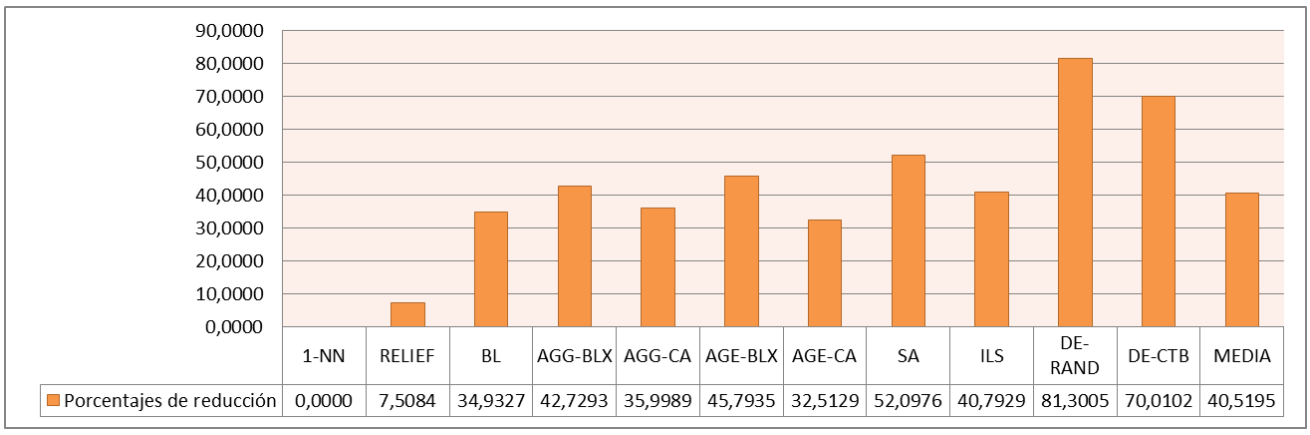
Comparación de resultados globales

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:

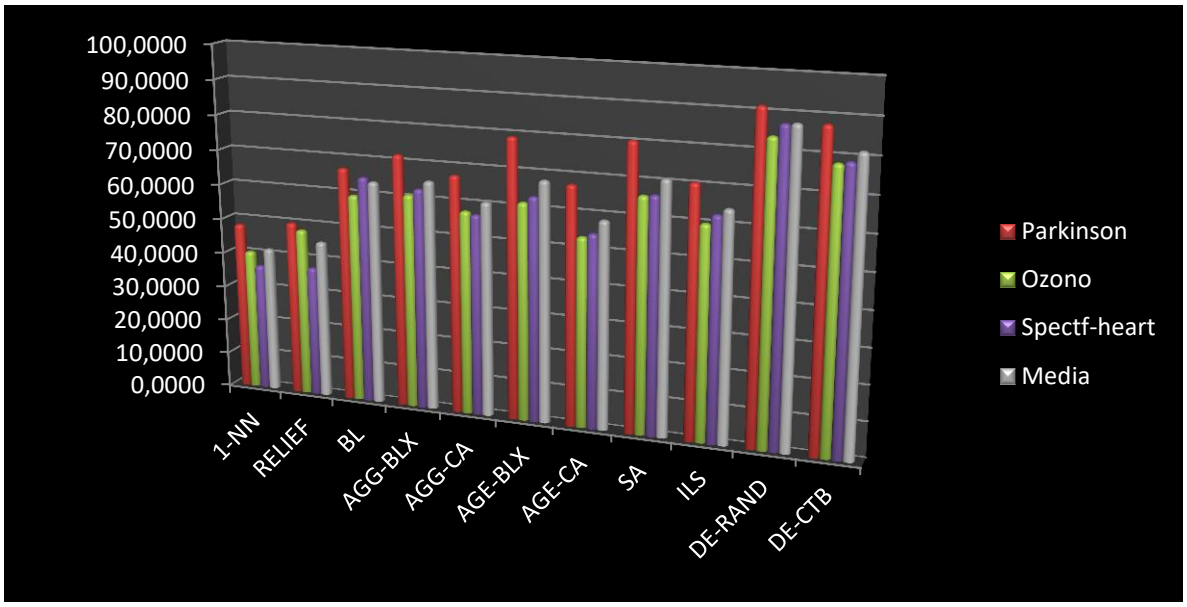
	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
1-NN	80,9375	0,0000	40,4688	0,7830	96,4082	0,0000	48,2041	0,0992	73,0725	0,0000	36,5362	0,5982
RELIEF	77,5000	18,8889	48,1944	3,2476	96,4082	3,6364	50,0223	0,4282	74,8114	0,0000	37,4057	2,3974
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
AGG-BLX	85,0000	37,1081	61,0541	2334,2976	95,8826	48,1666	72,0246	225,9868	82,8199	42,9130	62,8665	1488,4454
AGG-CA	83,4375	32,2432	57,8404	1532,9354	94,3826	40,6666	67,5246	142,0154	79,0932	35,0870	57,0901	976,4634
AGE-BLX	87,8125	35,4384	61,6254	543,8290	96,9217	61,6087	79,2652	55,5718	86,5590	40,3333	63,4462	383,6780
AGE-CA	80,6250	26,5676	53,5963	314,4846	94,3826	40,6667	67,5246	37,7880	79,1139	30,3043	54,7091	230,5196
AM-(10,1)	86,3281	40,8919	63,6100	6567,8645								
AM-(10,0.1N)	89,0625	39,8784	64,4704	4190,0100								
SA	87,1875	45,1111	66,1493	1019,4856	96,3954	64,4545	80,4250	113,8802	86,2650	46,7273	66,4961	795,4108
ILS	85,3125	34,8333	60,0729	3512,5200	95,9352	45,3636	70,6494	362,2012	83,1179	42,1818	62,6499	2532,6556
DE-RAND	92,1875	75,7568	83,9721	2456,1304	97,4615	85,6666	91,5641	250,2798	92,2609	82,4783	87,3696	1788,4004
DE-CTB	91,8750	64,6757	78,2753	2072,7190	95,9089	79,8332	87,8711	236,9424	92,2733	65,5217	78,8975	1877,9134

Para empezar, se debe comentar que sin duda las dos versiones de Evolución Diferencial son las dos mejores de todas las metaheurísticas que se han probado, tanto en clasificación como en reducción, donde marcan una gran diferencia con respecto al resto de algoritmos.

En el siguiente gráfico muestro esta situación, la gran superioridad en la tasa de reducción:



En la siguiente se muestran las tasas agregadas para cada set de datos en cada algoritmo, así como la media de los tres sets:



Como se puede ver, en todos los algoritmos, el set de datos que mejor clasifica es el del Parkinson. Esto parece que es debido a que es el set con una dimensión menor, al tener solo 22 atributos frente a los 72 y 44 de los algoritmos ozono y el de enfermedades del corazón, respectivamente. Aunque sigue habiendo una pequeña diferencia, esto parece no ser un problema para el "Differential Evolution", que clasifica casi igual los tres sets.

Una vez estudiados todos los algoritmos, muestro su clasificación:

Posición	Algoritmo	Tasa agregada	Tiempo de ejecución
1	DE-Cruce Rand	87,6352639	1498,2702
2	DE-Cruce CTB	81,6813072	1395,85826
3	Simulated Annealing	71,0234697	642,925532
4	Genético Estacionario-cruce BLX	68,1122694	327,692933
5	Genético Generacional-cruce BLX	65,3150426	1349,5766
6	MEDIA	64,5594516	1031,10561
7	Iterative Local Search	64,4573946	2135,79226
8	Búsqueda local	63,8343124	683,842933
9	Genético Generacional-cruce aritmético	60,8183408	883,804731
10	Genético Estacionario-cruce aritmético	58,6100062	194,264066
11	Relief	45,2074873	2,02439978
12	1-NN	41,7363659	0,49346641

Aunque los mejores algoritmos en cuanto a su tasa agregada son los dos de Evolución Diferencial, los que están situadas en tercera y cuarta posición reflejan un muy buen comportamiento, ya que consiguen resultados bastante buenos pero en un tiempo mucho menor, sobre todo, el Genético Estacionario con cruce BLX. Este, en solo cinco minutos consigue soluciones que no distan mucho de la mejor solución en cuanto a clasificación, aunque está penalizado porque la tasa de reducción que consigue no es de las mejores.

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 3 de prácticas sobre los problemas QAP y APC.