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 2B
Aprendizaje de Pesos en Características:

Técnicas de Búsqueda basadas en Poblaciones

Alberto Lejárraga Rubio

DNI: 50618389N

Email: alejarragar@correo.ugr.es

Granada, 4 de mayo 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 5

Clasificador 1-NN 6

Evaluar una solución 6

Distancia euclídea 7

Generación de la población inicial 7

Métodos de selección 8

Selección para algoritmos generacionales 8

Selección para algoritmos estacionarios 8

Métodos de cruce 8

Algoritmo de cruce BLX- α 8

Algoritmo de cruce aritmético 9

Algoritmo de mutación 9

Algoritmo para obtener el mejor cromosoma 10

4. Descripción de los algoritmos 10

Algoritmos genéticos 10

Algoritmos generacionales 10

Algoritmos estacionarios 11

Algoritmos meméticos 12

Algoritmo memético 10-1 12

Algoritmo memético 10-0.1 12

Algoritmo memético 10-0.1n 12

5. Descripción del algoritmo de comparación 13

6. Breve manual de usuario 13

7. Experimentos y análisis de resultados 13

Algoritmo generacional con cruce BLX-0.3 14

Algoritmo generacional con cruce aritmético 14

Algoritmo estacionario con cruce BLX-0.3 14

Algoritmo estacionario con cruce aritmético 14

Algoritmos meméticos 15

Comparación de resultados globales 15

8. Bibliografía 16

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, que serán los genes (w_i) de algoritmos genéticos y meméticos, pueden aparecer en el vector de pesos (W), que en este caso se llamará cromosoma, 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, en esta práctica 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 algoritmos genéticos y meméticos se considerará una población como una lista de 30 (tamaño de la población) cromosomas, 30 vectores de pesos.

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 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.

Evaluar una 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
```

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.

Generación de la población inicial

He tratado de utilizar siempre la misma población inicial para todos los algoritmos y particiones 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 de todos los algoritmos, al no tener que generar para cada partición de cada set de datos en cada algoritmo 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. 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
```

Como se verá en todos los métodos implementados en el código, siempre se devuelve un valor entero correspondiente al número de veces que se ha evaluado la función objetivo, ya sea por llamar a la función evaluarSolución con un cromosoma o por consultar el valor del último elemento del cromosoma que indica el valor de la tasa agregada como se comentó anteriormente. Esto no se recoge en los pseudocódigos por ser más un aspecto de la implementación que he realizado que una parte del algoritmo en sí, como tampoco la marca booleana que se ha explicado sobre si un cromosoma es el mejor de la población.

Métodos de selección

Selección para algoritmos generacionales

En los algoritmos generacionales la población se reemplaza completamente de una generación a la siguiente, aunque manteniendo un elitismo que se comentará después. Por ello, esta selección consiste en ir obteniendo de manera aleatoria dos padres de la población actual hasta completar la nueva población. En la implementación realizada, esta aleatoriedad la consigo intercambio el orden de los cromosomas de la población para después ir generando un torneo entre padres consecutivos. Es decir, una vez mezclada la población se enfrentarán en torneo binario el último cromosoma con el primero, el primero con el segundo, el segundo con el tercero, etc.

El torneo binario consiste en comprobar la función objetivo de los dos contendientes y seleccionar al mejor. Cabe destacar que cada miembro de la población competirá en dos torneos, pudiendo estar cada cromosoma seleccionada una, dos o ninguna vez. El pseudocódigo es el siguiente:

Función selecciónGeneracional(población):
    Crear lista poblaciónResul
    Mezclar aleatoriamente la población
    Desde 0 hasta <tamaño de la población> +1, i:
        calidadPadre=obtener calidad de población[i]
        calidadMadre=obtener calidad de población[i+1]
        Si calidadPadre>calidadMadre: introducir en poblaciónResul población[i]
        En otro caso: introducir en poblaciónResul población[i+1]
    Retornar poblaciónResul

Se debe tener en cuenta que el último elemento se debe enfrentar al primero

Selección para algoritmos estacionarios

En este tipo de algoritmos, en cada generación solo se reemplazan uno o dos cromosomas en función del tipo de cruce como se explicará en los apartados que se refieren a cada algoritmo. De cualquier forma, este algoritmo de selección siempre devolverá dos padres que tendrán que cruzarse y generar los nuevos hijos. En este caso, se seleccionarán cuatro cromosomas de la población inicial aleatoriamente y después se realizará el torneo binario entre los dos primeros seleccionados y los dos últimos:

```
Función seleccionEstacionario(población):
      Crear lista vacía padres
      Crear lista de candidatos con 4 cromosomas aleatorios de la población
      Si <calidad candidato_0> > <calidad candidato_1>: introducir en padres candidato_0
      En otro caso: introducir en padres candidato_1
      Si <calidad candidato_2> > <calidad candidato_3>: introducir en padres candidato_2
      En otro caso: introducir en padres candidato_3
      Devolver padres
```

Métodos de cruce

Algoritmo de cruce BLX- α

Este algoritmo genera dos cromosomas hijos a partir de los cromosomas de dos padres mediante la introducción en cada gen de los hijos de un valor aleatorio teniendo en cuenta los valores de ese gen en los padres y el parámetro alfa que en la práctica será de 0.3. Este valor de alfa tiene una gran influencia en el algoritmo, pues determinará si este explora un rango de soluciones más amplio o bien explota más la solución actual, es decir, determinará si se analizan soluciones más o menos distantes.

Ejemplificaré este método tomando como valor del gen de los padres 0.4 y 0.7 y un α de 0.3:
CMin=0.4, CMax=0.7,I=CMax-CMin=0.3, α=0.3

gen hijo= valor aleatorio en [CMin-I* α,CMax+I* α]= [0.4-0.3*0.3,0.7-0.3*0.3]= [0.31,0.61]>

Además, el algoritmo debe recibir una probabilidad de cruce, para simular el hecho de que haya padres en la población que no puedan tener hijos. Para ello se obtiene el número de cruces antes de empezar el algoritmo en base a la probabilidad recibida y se realizan cruces entre los primeros cromosomas de la población hasta este número, teniendo en cuenta que la población ha sido aleatorizada en el método de selección:

```
Función cruceBLX (población, alfa, probCruce…):
      NúmeroDeCruces = truncar(probCruce*tamañoPoblacion/2)
      Crear lista vacía poblaciónResul
      Desde 0 hasta númeroDeCruces, i:
            Cromosoma1 = población[2*i]
            Cromosoma2 = población[2*i+1]
            Crear listas vacías hijo1, hijo2
            Desde 0 hasta númeroDeGenes, j:
                  CMax = gen j de mayor valor entre cromosoma1 y cromosoma2
                  CMin = gen j de menor valor entre cromosoma1 y cromosoma2
                  I=CMax-CMin
                  Hijo1[j]= valor aleatorio en [CMin-I* alfa,CMax+I* alfa]
                  Hijo2[j]= valor aleatorio en [CMin-I* alfa,CMax+I* alfa]
            Añadir a poblaciónResul cromosoma1 y cromosoma2
      Añadir a poblaciónResul los cromosomas no explorados de población
      Retornar poblaciónResul
```

Tamaño de cromosoma1, por ejemplo

Algoritmo de cruce aritmético

Este algoritmo genera dos hijos nuevos a partir de dos padres considerando cada uno de los genes de los hijos como la media aritmética de los dos genes de los padres, por lo que se generan dos hijos iguales. El pseudocódigo sería el mismo que el anterior cambiando el bucle "`Desde 0 hasta númeroDeGenes, j:`", que quedaría:

```
Desde 0 hasta númeroDeGenes, j:
      Hijo1[j] = (cromosoma1[j]+cromosoma2[j])/2
      Hijo2[j] = (cromosoma1[j]+cromosoma2[j])/2
```

Algoritmo de mutación

A partir de una población recibida (ya sea completa o de dos cromosomas para los algoritmos estacionarios) se mutará un número de genes en base a una probabilidad de mutación determinada, que en la práctica es de 0.001. Esta mutación consiste en intercambiar un gen aleatorio de un cromosoma aleatorio de la población utilizando la función generarVecino, que devuelve un cromosoma con el gen indicado modificado a partir de un valor aleatorio obtenido de una distribución normal de números de media 0 y varianza 0.3:

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

Para decidir cuántas mutaciones se llevan a cabo se sigue el mismo procedimiento que en los algoritmos de cruce, decidiendo este número antes de empezar aunque en este caso selecciono aleatoriamente tanto el cromosoma como el gen a mutar:

```
Función mutación(población, probMut,sigma,distNormal…):
      numeroDeMutaciones = numeroDeGenesDeCromosomas*tamanoPoblacion*probMut
      poblaciónResul=población
      Desde 0 hasta numeroDeMutaciones:
            cromosomaAMutar = número aleatorio entre 0 y tamañoPoblación-1
            genAMutar = número aleatorio entre 0 y numeroDeGenesDeCromosomas
            poblaciónResul[cromosomaAMutar]=generarVecino(población[cromosomaAMutar],
distNormal, genAMutar)
      retornar poblaciónResul
```

Algoritmo para obtener el mejor cromosoma

En la implementación esta parte está en una función llamada mejorCromosoma y la utilizaré para describir los algoritmos siguientes pero no tiene mayor complicación que ir recorriendo toda la población y evaluando cada uno de los cromosomas, devolviendo el mejor de ellos. En esta función se introduce la marca true al mejor algoritmo de la población y en realidad no se evalúan todos los cromosomas sino que se tiene en cuenta el ultimo valor de cada cromosoma en el que aparece la tasa agregada de cada uno de ellos, que se ha ido modificando cada vez que se incluía un nuevo cromosoma en la población.

4. Descripción de los algoritmos

Algoritmos genéticos

Algoritmos generacionales

Los algoritmos genéticos generacionales se basan en que en cada generación de la población cambia prácticamente toda ella. El hecho de que no cambie completamente la población se debe a que al hacer esto podrían perderse buenas soluciones. Para evitar esto se introduce el concepto de elitismo por el cual cada generación del algoritmo cambiará al peor cromosoma de la población por el mejor de la población anterior.

Para determinar si el mejor cromosoma de la generación anterior se mantiene utilizo la marca booleana que ya he comentado, de forma que aunque se reordene la población, sea muy sencillo conocer si este mejor cromosoma sigue perteneciendo simplemente buscando esta marca entre todos los cromosomas.

Además, se debe considerar como ya he explicado antes que los métodos aumentarán directamente el número de evaluaciones realizadas. Explicaré tanto el algoritmo con cruce BLX- α como el que utiliza el cruce aritmético, ya que el código de ambos es el mismo con la salvedad de la llamada al distinto tipo de cruce:

```
Función generacional(poblaciónInicial,tamañoPoblación,numEvaluaciones,probCruce,probMut):
   evaluacionesRealizadas=tamañoPoblación
   poblaciónActual=poblaciónInicial
   Mientras evaluacionesRealizadas<numEvaluaciones:
      mejorCromosomaPoblación=mejorCromosoma(poblacionActual)
      oblaciónActual=seleccionGeneracional(poblacionActual)
      poblaciónActual=cruceBLX(poblaciónActual,probCruce,…) o
                      cruceAritmetico(poblaciónActual,probCruce)
      poblaciónActual=mutación(poblaciónActual,probMut,…)
      i=0,posiciónPeor=-1,peor=<número muy grande>,encontrado=false
      Mientras i<tamañoPoblación y !encontrado:
         Si poblaciónActual[i] es el mejor de la anterior generación: encontrado=true
         En otro caso:
```

Se consideran las evaluaciones realizadas a la hora de obtener la población inicial

```
            Si calidad de poblaciónActual[i] <peor:
               Peor=calidad de poblaciónActual[i]
               posiciónPeor=i
         i++
      si !encontrado: poblaciónActual[posiciónPeor]=mejorCromosomaPoblación
   Volver
   Retornar calidad de mejorCromosoma(poblaciónActual)
```

Algoritmos estacionarios

En el caso de los algoritmos estacionarios cambia solo uno de los cromosomas de la población. En un principio introducía los dos hijos generados pero aunque pudiera mutar alguno de ellos el resultado era bastante malo con respecto a los demás algoritmos, por lo que decidí introducir solo uno, mejorando las clasificaciones.

Tras seleccionar los padres, estos cruzan siempre, no como en el modelo generacional en el que existía una probabilidad de cruce. En cuanto a la probabilidad de mutación, como la función es la misma para generacionales y estacionarios y en este caso solo se pasa una población de dos hijos, hay que determinar si van a mutar o no antes de llamar a esta función. En caso de que tengan que mutar ajusto la probabilidad de mutación para que se mute exactamente un gen. Tras la mutación se realiza el procedimiento de eliminar a los peores de la población para introducir estos si son mejores. El pseudocódigo del algoritmo, así como de la función valorarHijos() necesaria en este es el siguiente:

```
Función estacionario(poblaciónInicial,tamañoPoblación,numEvaluaciones, probMut):
   evaluacionesRealizadas=tamañoPoblación
   poblaciónActual=poblaciónInicial
   Mientras evaluacionesRealizadas<numEvaluaciones:
      cromosomasSeleccionados==seleccionEstacionario(poblacionActual)
      hijosNuevos=cruceBLX(cromosomasSeleccionados,probCruce=1,…) o
                     cruceAritmetico(cromosomasSeleccionados,probCruce=1)
      aMutar=numero aleatorio entre 0 y 1
      si aMutar<=probMut: probMutación=1/(2*numeroDeGenes)
      En otro caso: probMutación=probMut
      hijosNuevosMutados=mutación(hijosNuevos,probMutación,…)
      poblaciónActual=valorarHijos(hijosNuevosMutados,poblacionActual,…)
   Volver
   Retornar calidad de mejorCromosoma(poblaciónActual)

Función valorarHijos(hijosNuevosMutados,población,tamañoPoblación,…):
      poblaciónActual=población
      posiciónPeor=1000, peor=<numero muy grande>, i=0
      mientras i<tamañoPoblación:
            si calidad de poblaciónActual[i]<peor:
                  peor= calidad de poblaciónActual[i]
                  posiciónPeor=i
            i++
      poblaciónActual[posiciónPeor]=hijosNuevosMutados[número aleatorio 0 o 1]
      retornar poblaciónActual
```

Se consideran las evaluaciones realizadas a la hora de obtener la población inicial

Algoritmos meméticos

Los algoritmos meméticos pueden considerarse como una modificación de los algoritmos genéticos en los que se introduce en algún momento de la ejecución una búsqueda local sobre todos o algunos elementos de la población. En el caso de la práctica se ha realizado el algoritmo memético tomando como base el algoritmo genético generacional con cruce BLX- α por ser el que ha obtenido los mejores resultados frente al cruce aritmético.

En la implementación, la búsqueda local se realiza en el propio algoritmo generacional justo después de realizar la mutación y siempre cada 10 generaciones. Aunque en el código he introducido todos en una misma función, mostraré el pseudocódigo por separado para cada uno de ellos:

Algoritmo memético 10-1

Esta variante del algoritmo realiza una búsqueda local sobre toda la población actual:

```
Función memético10-1(población,tamañoPoblación,…):
	poblaciónResul=lista vacía
	Para cada cromosoma de la población:
		Añadir a poblaciónResul<-buscLocal(numeroDeEvaluaciones = 2*tamañoPoblacion,
soluciónInicial=cromosoma,…)
	Retornar poblaciónResul
```

Algoritmo memético 10-0.1

En este caso se realiza la búsqueda local sobre un diez por ciento aleatorio de la población, por lo que habrá que primero decidir el número de cromosomas que se mandarán y después aleatoriamente mandar esa cantidad de ellos. Debido a esto se debe almacenar los cromosomas que se mandan para luego añadir los demás a la población resultante.

```
Función memético10-0.1(población,tamañoPoblación,…):
	numCromosomasAMandar=truncar(0.1*tamañoPoblación)
	poblaciónResul=lista vacía
	introducidos=lista vacía
	desde 0 hasta numCromosomasAMandar:
		numAleatorio=generar aleatorio entre 0 y tamañoPoblación-1
		Añadir a introducidos numAleatorio
		Añadir a poblaciónResul<-buscLocal(numeroDeEvaluaciones = 2*tamañoPoblacion,
soluciónInicial=cromosoma,…)
	Desde 0 hasta tamañoPoblación,i:
		Si i no esta en introducidos:añadir a poblaciónResul población[i]
	Retornar poblaciónResul
```

Algoritmo memético 10-0.1n

Esta modificación es muy similar a la anterior con la única diferencia de que no se realiza la búsqueda local sobre un diez por ciento aleatorio, sino que se realiza sobre el diez por ciento mejor. Por esto, lo único que hay que hacer es reordenar la población de mejor a peor valoración y realizar la búsqueda local sobre el primer diez por ciento 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:

```
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 poblaciónInicialGeneticos<partición>.txt como sin ellos, pues estos se generan tras la primera llamada a los algoritmos.

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

Algoritmo generacional con cruce BLX-0.3

Tabla 5.1: Resultados obtenidos por el algoritmo GENERACIONAL-BLX 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	36,8378	61,3877	2372,4750	94,7368	40,6667	67,7018	244,2680	86,9565	49,0000	67,9783	1371,3240
Partición 2	82,8125	36,8378	59,8252	2320,0480	97,5000	57,3333	77,4167	204,1900	78,5714	38,1304	58,3509	1568,5350
Partición 3	87,5000	34,1351	60,8176	2401,3570	97,5000	53,1667	75,3333	214,5470	84,2857	40,3043	62,2950	1544,0340
Partición 4	81,2500	36,8378	59,0439	2321,8000	92,3077	40,6666	66,4871	248,1840	85,7143	49,0000	67,3571	1389,6790
Partición 5	87,5000	40,8919	64,1959	2255,8080	97,3684	49,0000	73,1842	218,7450	78,5714	38,1304	58,3509	1568,6550
Media	85,0000	37,1081	61,0541	2334,2976	95,8826	48,1666	72,0246	225,9868	82,8199	42,9130	62,8665	1488,4454
Desv.Tipica	2,8384	2,4174	1,9742	55,8443	2,3201	7,4536	4,7623	19,2700	3,9918	5,6270	4,6745	99,2601

Algoritmo generacional con cruce aritmético

Tabla 5.1: Resultados obtenidos por el algoritmo GENERACIONAL-ARITMÉTICO 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	31,4324	59,4662	1517,2810	97,3684	40,6666	69,0175	139,3720	81,4286	33,7826	57,6056	1004,7410
Partición 2	81,2500	34,1351	57,6926	1512,1630	94,7368	40,6666	67,7017	139,7350	82,6087	38,1304	60,3696	940,1300
Partición 3	82,8125	30,0811	56,4468	1568,5450	92,5000	40,6666	66,5833	144,8160	74,2857	38,1304	56,2081	960,4360
Partición 4	85,9375	31,4324	58,6850	1546,0010	95,0000	40,6666	67,8333	144,6810	75,7143	31,6087	53,6615	991,8870
Partición 5	79,6875	34,1351	56,9113	1520,6870	92,3077	40,6666	66,4871	141,4730	81,4286	33,7826	57,6056	985,1230
Media	83,4375	32,2432	57,8404	1532,9354	94,3826	40,6666	67,5246	142,0154	79,0932	35,0870	57,0901	976,4634
Desv.Tipica	3,2401	1,8130	1,2434	23,7977	2,0779	0,0000	1,0390	2,6187	3,8012	2,9166	2,4403	25,9302

Algoritmo estacionario con cruce BLX-0.3

Tabla 5.1: Resultados obtenidos por el algoritmo ESTACIONARIO-BLX en el problema del APC

	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	81,2500	40,0959	60,6729	549,6670	94,8718	64,2174	79,5446	57,3040	88,5714	45,6667	67,1190	353,1940
Partición 2	90,6250	33,2466	61,9358	547,6710	94,7368	59,8696	77,3032	54,7040	84,2857	45,6667	64,9762	375,7090
Partición 3	87,5000	35,9863	61,7432	527,7270	100,0000	64,2174	82,1087	52,8450	95,6522	39,0000	67,3261	378,5240
Partición 4	92,1875	38,7260	65,4568	534,6140	100,0000	59,8696	79,9348	56,6200	85,7143	34,5556	60,1349	403,3560
Partición 5	87,5000	29,1370	58,3185	559,4660	95,0000	59,8696	77,4348	56,3860	78,5714	36,7778	57,6746	407,6070
Media	87,8125	35,4384	61,6254	543,8290	96,9217	61,6087	79,2652	55,5718	86,5590	40,3333	63,4462	383,6780
Desv.Tipica	4,1926	4,3964	2,5803	12,6249	2,8116	2,3814	1,9882	1,7994	6,2534	5,1159	4,3347	22,2439

Algoritmo estacionario con cruce aritmético

Tabla 5.1: Resultados obtenidos por el algoritmo ESTACIONARIO-ARITMETICO en el problema del APC

	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	84,3750	24,6757	54,5253	324,1390	95,0000	40,6667	67,8333	38,8160	75,7143	31,6087	53,6615	229,5780
Partición 2	76,5625	27,3784	51,9704	315,3580	92,3077	40,6667	66,4872	37,1190	81,4286	31,6087	56,5186	233,5260
Partición 3	84,3750	26,0270	55,2010	311,2450	97,3684	40,6667	69,0175	37,1730	75,7143	31,6087	53,6615	234,9980
Partición 4	79,6875	27,3784	53,5329	312,2830	94,7368	40,6667	67,7018	37,0190	89,8551	25,0870	57,4710	216,5750
Partición 5	78,1250	27,3784	52,7517	309,3980	92,5000	40,6667	66,5833	38,8130	72,8571	31,6087	52,2329	237,9210
Media	80,6250	26,5676	53,5963	314,4846	94,3826	40,6667	67,5246	37,7880	79,1139	30,3043	54,7091	230,5196
Desv.Tipica	3,5971	1,2087	1,3037	5,8135	2,0779	0,0000	1,0390	0,9387	6,7637	2,9166	2,1925	8,3548

Algoritmos meméticos

Debido al elevado tiempo de ejecución (y a que yo esperé demasiado tiempo en lanzar el programa, sobre todo) no he podido completar las tablas sobre ninguno de los tres algoritmos aunque sí que he obtenido resultados sobre el set del ozono para los algoritmos 10-1 y 10-0.1n que son los siguientes:

	BLX+MEMÉTICO 10-1			
	Ozone			
	%_clas	*%red*	*Agr.*	T
Partición 1	84,3750	39,5405	61,9578	6622,6550
Partición 2	90,6250	43,5946	67,1098	6381,1320
Partición 3	85,9375	38,1892	62,0633	6715,3370
Partición 4	84,3750	42,2432	63,3091	6552,3340
Media	86,3281	40,8919	63,6100	6567,8645
Desv.Tipica	2,9578	2,4672	2,4125	141,2567

	BLX+MEMÉTICO 10-0,1N			
	Ozone			
	%_clas	*%red*	*Agr.*	T
Partición 1	87,5000	39,5405	63,5203	4223,6850
Partición 2	92,1875	40,8919	66,5397	4130,9470
Partición 3	92,1875	34,1351	63,1613	4412,9750
Partición 4	84,3750	44,9459	64,6605	3992,4330
Media	89,0625	39,8784	64,4704	4190,0100
Desv.Tipica	3,8273	4,4649	1,5204	176,4204

Con respecto al tiempo de ejecución del memético 10-0.1N debe tenerse en cuenta que se ha realizado con un ordenador bastante más lento que el resto de pruebas (3.8GHz vs 2.2GHz) por lo que no se puede estudiar en función de este valor, ya que sería mucho más bajo.

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								

Con respecto al tipo de algoritmo genético no parece haber una diferencia clara entre los generacionales y los estacionarios, pues en general consiguen bastantes buenos resultados, casi siempre por encima del 60 % o cerca excepto en algún caso concreto, lo que se debe a la diferencia entre los set de datos, pues su tamaño no es uniforme ni en número de genes ni en número de ejemplos de cada set.

En cuanto al tipo de cruce sí parece haber una diferencia muy clara a favor del cruce BLX-0.3, ya que este siempre es bastante mejor con respecto al cruce aritmético de su mismo tipo. Además, estacionario con este tipo de cruce siempre está algo por encima que el generacional, por lo que parece que este es el mejor de los algoritmos estudiados atendiendo a los datos. Otro punto muy a favor de este estacionario con BLX es su tiempo de ejecución, ya que consigue resultados muy buenos teniendo utilizando una cuarta parte del tiempo que su rival.

Esta disminución tan drástica en el tiempo que también ocurre con el estacionario con cruce aleatorio se debe a que se realizan muchas menos llamadas al clasificador 1-nn, que es la parte del programa que más tiempo consume con diferencia (algo más de medio segundo por ejecución en el caso del ozono).

En comparación con el algoritmo de búsqueda local quedan por encima los dos genéticos con cruce BLX pero sin una gran diferencia. De hecho, el la partición “Spectf-heart”, quedan muy por debajo en porcentaje de

clasificación, ya que el de búsqueda local era ya muy alto. Es probable que el memético para este set se acercara bastante o incluso superara a la búsqueda local al tener ya de por sí una buena solución inicial.

Si se tienen en cuenta los datos aportados para los algoritmos meméticos se observa que estos mejoran siempre al mejor de los algoritmos anteriores, sobre todo el 10-0.1n, que al realizar la búsqueda local sobre el diez por ciento mejor consigue generar unos resultados muy altos. El problema de estos algoritmos es el alto incremento de tiempo que se produce, pues aunque los resultados son mejores es probable que no lo sean lo suficiente como para decantarse por estos, ya que como se ha comentado, el estacionario con BLX ya consigue buenas soluciones con un tiempo relativamente bajo.

Por tanto, como conclusión y a falta de estudiar los resultados completos de los algoritmos meméticos, parece que el mejor tipo de cruce entre los estudiados es el BLX, y el mejor de los algoritmos teniendo en cuenta el tiempo de ejecución y las tasas que consigue es el estacionario con este cruce BLX.

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.