UNIVERSIDAD DE GRANADA

PRÁCTICA 1.b: Algoritmos Genéticos para el Problema del Aprendizaje de Pesos en Características (APC)

Metaheurísticas || Curso 2018-2019

Alumno: José María Sánchez Guerrero
DNI: 76067801Q
Correo: jose26398@correo.ugr.es
Grupo: A3 – Jueves 17:30

Contenido

DESCRIPCIÓN DEL PROBLEMA

He elegido el problema del **Aprendizaje de Pesos en Características (APC)**, en el cual tendremos que realizar un clasificador de características partiendo de unos datos iniciales sin procesar. Este clasificador va a ser del tipo **k-NN** (donde k=1), en el que la clase que asociaremos a cada dato, será la clase del vecino más cercano.

El **vecino más cercano** se obtiene a partir de la distancia euclídea entre el propio dato y el vecino. Por tanto, la distancia entre dos ejemplos e_1 y e_2, utilizando por ejemplo de para las características y w para los pesos (que ponderan su importancia dentro del contexto), se calcularía de la siguiente forma:

$$d_e(e_1, e_2) = \sqrt{\sum_i w_i \left(e_1^i - e_2^i\right)^2 + \sum_j w_j \cdot d_h(e_1^j - e_2^j)}$$

Los datos iniciales utilizados en el problema tendrán que ser divididos en datos de entrenamiento y datos para testear. Esta división se realizará de tal forma que 80% sea de entrenamiento y el 20% para test, manteniendo la distribución de clases.

Al final de cada algoritmo vamos a devolver cuatro valores que nos van a servir para evaluar tanto la precisión como la complejidad del clasificador. Estos valores son:

- **Tasa_clas** (precisión). Se calculará dividiendo las instancias bien clasificadas en T entre el número de instancias totales de T.
- **Tasa_red** (reducción). Se calculará dividiendo el número de valores de w menores que 0.2 entre el número total de características.
- **Agr** (agregado). Valor de la función objetivo y que calcularemos sumando los dos valores anteriores multiplicados por α y 1-α respectivamente. En nuestro caso, éste va a ser de 0.5.
- **T** (tiempo). Tiempo que tardará en ejecutarse el algoritmo.

Los algoritmos que vamos a implementar van a ser el algoritmo de comparación greedy RELIEF, el algoritmo de Búsqueda Local, Algoritmo Genético Generacional con cruce BLX, Algoritmo Genético Generacional con cruce aritmético y Algoritmos Meméticos. Realizaremos 5 ejecuciones para cada uno de ellos utilizando los tres ficheros de datos proporcionados y mostraremos los 4 estadísticos anteriores en tablas para poder comparar el rendimiento de los algoritmos.

DESCRIPCIÓN DE LA APLICACIÓN DE LOS ALGORITMOS

Lectura de datos

Esta parte en muy simple. Sólo son 3 líneas que servirán para extraer los datos de los archivos (guardados en la carpeta doc/).

```
df = read_csv('doc/colposcopy.csv')
data = df.extraer_valores
datos = quitarPrimeraColumna
```

Quitamos la primera columna porque en los archivos tenemos guardado ahí el número de los datos (o la id de los datos), la cual no vamos a necesitar.

División de los datos

Gracias a esto vamos a poder dividir los datos en 5 partes iguales conservando la proporción de 80% entrenamiento y 20% test. La división está realizada principalmente por la función ***StratifiedKFold*** perteneciente a la librería de python *scikit learn*.

Ésta proporciona automáticamente índices de entrenamiento y test para después poder dividir los datos. Vamos a verlo mejor con el pseudocódigo:

```
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=73)

x = extraerAtributos(datos)
y = extraerClases(datos)

for indiceTrain, indiceTest in skf.sacarIndicesDe(x,y) do
   x_train, y_train = x[indiceTrain], y[indiceTrain]
   x_test, y_test = x[indiceTest], y[indiceTest]

   train.añadirFinal(x_train, y_train)
   test.añadirFinal(x_test, y_test)
end for
```

La función ***StratifiedKFold*** la he dejado tal cual es, para que se vea cómo selecciono el número de divisiones (n_split=5), shuffle es para que salga aleatorio y random_state es la semilla, por si desea cambiarla a otra.

Después, dividimos los datos anteriores en atributos y las clases; y gracias a los índices que nos proporciona la función anterior, añadimos a los vectores ***train*** y ***test*** un dato aleatorio al final de ellos.

Función objetivo (función evaluación)

La función objetivo evalúa los pesos calculados para cada algoritmo. Como la he implementado de forma muy similar a la función para obtener las ganancias de cada dato, la explicaré posteriormente, en la sección del *Algoritmo de comparación* (o el apartado f) en el guion de prácticas).

Esquema de representación de los datos

El esquema de representación de soluciones se obtendrá gracias a la función evaluar (que veremos posteriormente). Al final de cada una de las ejecuciones, esta función guardará en una matriz los resultados obtenidos, correspondiendo cada fila de la matriz a una de ellas. Las columnas de la matriz estarán formadas por los cuatro valores estadísticos mencionados al principio.

A continuación, muestro una tabla en representación de la matriz para verlo mejor visualmente:

Matrix	**%tasa_clas**	**%tasa_red**	**Agr**	**T**
Ejecución 1				
Ejecución 2				
Ejecución 3				
Ejecución 4				
Ejecución 5				

Las matrices (o tablas) se guardarán al final en una variable del tipo DataFrame, para poder imprimirlas por pantalla más fácilmente y también para poder sacarlas a un .csv externo y tener un acceso más sencillo a los resultados.

Generación de soluciones aleatorias

Los algoritmos partirán de una población generada aleatoriamente. Para ello utilizaré la función *np.random.uniform* que permite generar aleatorios en un rango y con un tamaño específico. Posteriormente, calcularé las ganancias de cada dato, las guardaré en otro array junto con el índice del dato, y lo ordenaré creciente o decrecientemente (esto nos será útil posteriormente).

El pseudocódigo es el siguiente:

```
filas = tamPoblacion
columnas = numAtributos
poblacion = generarAleatorios(min=0, max=1, tamaño=(filas, columnas))

for indice in tamPoblacion do
   ganancias.añadirFinal( indice, obtenerGanancia(poblacion[indice]) )
end for

ganancias.ordenar(X)          # X = ascendente o descendente
```

Gracias a la función de *numpy*, no necesitaremos truncar los valores ni hacer bucles para generar la matriz de datos, por lo que nos ahorraremos algo de tiempo en la ejecución.

Función de selección o torneo binario

Elige aleatoriamente dos individuos de la población y selecciona al mejor de ellos. Para los algoritmos genéticos generacionales, se haremos tantos como individuos haya en la población; mientras que en los estacionarios sólo tendremos que sacar dos padres.

Por otra parte, como en el cruce aritmético generamos un hijo utilizando dos padres, necesitaremos el doble de padres, tendremos que hacer el doble de torneos binarios.

Para comprender mejor las iteraciones que da en el pseudocódigo, vamos a fijar antes cuántos padres vamos a sacar:

- **Generacionales.** Con cruce BLX el $numTorneos = tamPoblacion$ y con cruce aritmético será $numTorneos = 2 * tamPoblacion$.
- **Estacionario.** Con cruce BLX el $numTorneos = 2$ y con cruce aritmético será $numTorneos = 2 * 2$
- **Meméticos.** En mi caso, como los he hecho partiendo del generacional con cruce BLX, tendrán la misma estructura que estos.

Ahora sí, vamos a ver el pseudocódigo:

```
for i in numTorneos do
   indice1, indice2 = generarAleatorios(tamPoblacion)
   cromo1, cromo2 = poblacion[indice1, indice2]
   ganancia1 = obtenerGanancia(cromo1)
   ganancia2 = obtenerGanancia(cromo2)

   if ganancia1 > ganancia2 then
      padres.insertar(cromo1)
      gananciasPadres.insertar(i, ganancia1)
   else
      padres.insertar(cromo2)
      gananciasPadres.insertar(i, ganancia2)
   end if
end for
```

Ya tenemos los padres que van a ser cruzados, así que ahora vamos a ver los distintos operadores de cruce que tenemos.

Operador de cruce BLX

Este operador nos va a generar dos descendientes. Para cada uno de ellos va a asignarles componente a componente un valor entre el siguiente rango:

$$r = [\min(padre1[i], padre2[i]) - I * \propto, \max(padre1[i], padre2[i]) + I * \propto]$$

Donde el valor de I será el intervalo entre el máximo y el mínimo de cada padre, y el valor $\propto$ tendrá un valor de 0.03 (según nos indica el guion). Para generar un valor aleatorio en ese rango, he utilizado la misma función *np.random.uniform* que en la generación de soluciones iniciales, pero esta vez con tamaño 1.

El pseudocódigo del operador es el siguiente:

```
function cruceBLX(padre1, padre2, alpha)
   for i in numAtributos do
      cMax = max(padre1[i], padre2[i])
      cMin = min(padre1[i], padre2[i])
      I = cMax – cMin     # Intervalo

      padre1[i] = generarAleatorio( min=(cMin-I*alpha), max=(cMax+I*alpha) )
      padre2[i] = generarAleatorio( min=(cMin-I*alpha), max=(cMax+I*alpha) )

      padre1[i] = acotarEntre(0, 1)
      padre2[i] = acotarEntre(0, 1)
   end for
end function
```

Al final tendremos que acotar entre 0 y 1, ya que cabe la posibilidad de que en los nuevos valores, el intervalo donde se generan sea o menor que 0 o mayor que 1.

Operador de cruce aritmético

Este operador sólo genera un descendiente cuyas características son la media de las características de sus dos padres, es decir, que el valor de la característica i es:

$$hijo[i] = (padre1[i] + padre2[i])/2$$

El pseudocódigo es:

```
function cruceAritmetico(padre1, padre2)
   for i in padre1.size do
      hijo[i].insertar( (padre1[i] + padre2[i])/2 )
   end for
end function
```

Operador de mutación

Esta función se ejecutará cada vez que insertemos un nuevo hijo en la población. Esto se debe a que la probabilidad de que se mute la calculo dentro de esta función, y no fuera de ella (en el algoritmo utilizado).

Para ello, he utilizado los números aleatorios. Como la probabilidad de que salga es del 0.001, cuando obtenga un valor inferior a este generando números aleatorios entre 0 y 1, realizaré una mutación a ese cromosoma.

La mutación consiste simplemente en generar dos números aleatorios: un entero para obtener el índice del gen a modificar; y un flotante que será el valor que sumemos al gen. Este último valor será un aleatorio que seguirá una distribución normal N = (0, 0.3), según nos indica el guion.

La implementación en pseudocódigo es la siguiente:

```
function mutate(cromosoma)
   if generarAleatorio(min=0, max=1) <= 0.001 then
      indice = generarAleatorioEntero(min=0, max=cromosoma.size)
      cromosoma[indice] += generarAleatorioNormal(0, 0.3)
   end if

   return cromosoma
end function
```

DESCRIPCION DE LOS ALGORITMOS

Algoritmo Genético Generacional con cruce BLX

La población inicial de este algoritmo será de 30 datos.

Para poder trabajar mejor con los datos, aparte de generar los datos únicamente, también genero una matriz de ganancias. Esta matriz tiene un número de filas igual al número de datos, y dos columnas: la primera guarda los índices de los individuos de la población y la segunda el valor de la ganancia.

En los generacionales, esta matriz estará ordenada de mayor ganancia a menor cuando generamos los datos.

El número máximo de evaluaciones con las que trabajaremos será de 15000.

Como indica en el guion, el cruce se realizará para el 70% de la población, por lo que para el resto tendremos que volver a coger los padres que habíamos cruzado. Posteriormente, ordenaremos el vector de ganancias de los hijos de menor a mayor para facilitar la parte del elitismo.

Para el elitismo simplemente tendremos que comparar la ganancia del peor hijo con la mejor de la población. En el pseudocódigo vemos que los índices son los mismos, y esto es por cómo hemos ordenado los vectores (para los hijos el más malo primero y para la población el más bueno primero).

La mutación la realizamos cuando cruzamos los dos hijos. Como hemos visto antes, la probabilidad de que mute o no se determinará dentro de la propia función. En caso de que lo haga, se insertará ya mutado dentro del vector de hijos.

Por último, el vector solución que sacamos lo hacemos a partir de las últimas ganancias calculadas. Como éstas están ordenadas descendentemente, sólo tendremos que utilizar el índice de la mejor ganancia (el primero) y extraer el individuo de la población al cual pertenece éste índice. Cuando lo tengamos ya simplemente evaluamos y guardamos el resultado de la ejecución en la matriz.

A continuación, veamos la implementación del pseudocódigo:

```
function AGG-BLX()
   población, ganancias = generarSolucionesAleatorias()

   while nEval < maxEval do
      # Seleccionar P(t) torneo binario
      padres, gananciasPadres = torneoBinario()
         nEval += 1

      # Cruce P(t) con BLX
      for p in (tamPoblacion*0.7) and p+=2 do
         hijo1, hijo2 = cruceBLX(padres[p], padres[p+1], 0.3)
         hijos.insertar( mutate(hijo1) )
         hijos.insertar( mutate(hijo2) )
         gananciasHijos.insertar( p, obtenerGanancia(hijo1) )
         gananciasHijos.insertar( p+1, obtenerGanancia(hijo2) )
      end for

       # Completar los que faltan con los padres
      for p = (tamPoblacion*0.7) in tamPoblacion do
         hijos.insertar( padres[p] )
         gananciasHijos.insertar( gananciasPadres[p] )
      end for

      gananciasHijos.ordenar(ascendente)
      # Elitismo
      if ganancias[0][1] > gananciasHijos[0][1] do
         # Inserta en hijo a partir del indice
         hijos[ gananciasHijos[0][0] ] = poblacion[ ganancias[0][0] ]
         # Inserta la nueva ganancia
         gananciasHijos[0][1] = ganancias[0][1]
      end if

      poblacion = hijos
      ganancias = gananciasHijos
      ganancias.ordenar(descendente)

   end while

   w = poblacion[ ganancias[0][0] ]
   resultados.insertar(evaluar(w))
```

Algoritmo Genético Generacional con cruce aritmético

La estructura de este algoritmo es exactamente igual que el anterior, lo único que cambiamos es el operador de cruce. Como éste genera sólo un hijo, tendremos que tener el doble de padres generados. En caso contrario, la población se reduciría en cada una de las evaluaciones.

Esto lo hicimos en el torneo binario explicado anteriormente.

A continuación, la implementación en pseudocódigo del algoritmo:

```
function AGG-aritmetico()
   poblacion = generarSolucionesAleatorias()

   while nEval < maxEval do
      # Seleccionar P(t) torneo binario
      padres, gananciasPadres = torneoBinario()
         nEval += 1

      # Cruce P(t) con aritmetico
      aux = 0
      for p in (tamPoblacion*0.7) do
         hijo1 = cruceAritmetico(padres[aux], padres[aux+1])
         hijos.insertar( mutate(hijo1) )
         gananciasHijos.insertar( p, obtenerGanancia(hijo1) )
         aux += 2
      end for

       # Completar los que faltan con los padres
      for p = (tamPoblacion*0.7) in tamPoblacion do
         hijos.insertar( padres[p] )
         gananciasHijos.insertar( gananciasPadres[p] )
      end for

      gananciasHijos.ordenar(ascendente)

      # Elitismo
      if ganancias[0][1] > gananciasHijos[0][1] do
         # Inserta en hijo a partir del indice
         hijos[ gananciasHijos[0][0] ] = poblacion[ ganancias[0][0] ]
         # Inserta la nueva ganancia
         gananciasHijos[0][1] = ganancias[0][1]
      end if

      poblacion = hijos
      ganancias = gananciasHijos
      ganancias.ordenar(descendente)

   end while

   w = poblacion[ ganancias[0][0] ]
   resultados.insertar(evaluar(w))

end function
```

Algoritmo Genético Estacionario con cruce BLX

La población inicial de este algoritmo será de 30 datos.

Para poder trabajar mejor con los datos, aparte de generar los datos únicamente, también genero una matriz de ganancias. Esta matriz tiene un número de filas igual al número de datos, y dos columnas: la primera guarda los índices de los individuos de la población y la segunda el valor de la ganancia.

En los estacionarios, esta matriz estará ordenada de menor ganancia a mayor cuando generamos los datos.

El número máximo de evaluaciones con las que trabajaremos será de 15000.

Como indica en el guion, el cruce se realizará para el 100%, es decir, siempre vamos a cruzar los padres para generar dos hijos. Por esto, tampoco necesitaremos completar el vector de hijos con los padres anteriores.

El vector de ganancias de los hijos no será necesario ordenarlo, puesto que sólo hemos generado dos de ellos. Una vez comparados estos dos con los de la población (y reemplazado o no) ya si tendremos que ordenarlo. Esto lo hacemos para asignar a los nuevos hijos un estatus dentro de la población, dependiendo de sus ganancias.

Aquí no habrá elitismo, los dos hijos que hemos generado competirán con los peores de la población. En caso de que estos dos nuevos hijos sean mejores, sustituirán a los de la anterior población.

La mutación la realizamos cuando cruzamos los dos hijos. Como hemos visto antes, la probabilidad de que mute o no se determinará dentro de la propia función. En caso de que lo haga, se insertará ya mutado dentro del vector de hijos.

Por último, el vector solución que sacamos lo hacemos a partir de las últimas ganancias calculadas. Como éstas están ordenadas ascendentemente, tendremos que utilizar el índice de la mejor ganancia (el último) y extraer el individuo de la población al cual pertenece éste índice. Cuando lo tengamos ya simplemente evaluamos y guardamos el resultado de la ejecución en la matriz.

Veamos el pseudocódigo:

```
function AGE-BLX()
   poblacion = generarSolucionesAleatorias()

   while nEval < maxEval do
      # Seleccionar P(t) torneo binario
      padres, gananciasPadres = torneoBinario()

      # Cruce P(t) con BLX
      hijo1, hijo2 = cruceBLX(padres[0], padres[1], 0.3)
      hijos.insertar( mutate(hijo1) )
      hijos.insertar( mutate(hijo2) )
      gananciasHijos.insertar( 0, obtenerGanancia(hijo1) )
      gananciasHijos.insertar( 1, obtenerGanancia(hijo2) )

      if gananciasHijos[0][1] >= gananciasHijos[1][1] then
         mejor = gananciasHijos[0]
         peor = gananciasHijos[1]
      else
         mejor = gananciasHijos[1]
         peor = gananciasHijos[0]
      end if

      if mejor[1] > ganancias[1][1] then
         poblacion[ int(ganancias[1][0]) ] = hijos[ int(mejor[0]) ]
         ganancias[1] = mejor

         if peor[1] > ganancias[0][1] then
            poblacion[ ganancias[0][0] ] = hijos[ peor[0] ]
            ganancias[0] = peor
         elif mejor[1] > ganancias[0][1]
            poblacion[ ganancias[0][0] ] = hijos[ mejor[0] ]
            ganancias[0] = peor
         end if

      end if

      ganancias.ordenar(ascendente)

   end while

   w = poblacion[ ganancias[ganancias.size-1][0] ]
   resultados.insertar(evaluar(w))

end function
```

Algoritmo Genético Estacionario con cruce aritmético

La estructura de este algoritmo es exactamente igual que el anterior, lo único que cambiamos es el operador de cruce. Como éste genera sólo un hijo, tendremos que tener el doble de padres generados. En caso contrario, la población se reduciría en cada una de las evaluaciones.

Esto lo hicimos en el torneo binario explicado anteriormente.

A continuación, veamos el pseudocódigo:

```
function AGE-aritmetico()
   poblacion = generarSolucionesAleatorias()

   while nEval < maxEval do
      # Seleccionar P(t) torneo binario
      padres, gananciasPadres = torneoBinario()

      # Cruce P(t) aritmetico
      hijo1 = cruceAritmetico(padres[0], padres[1])
      hijo2 = cruceAritmetico(padres[2], padres[3])
      hijos.insertar( mutate(hijo1) )
      hijos.insertar( mutate(hijo2) )
      gananciasHijos.insertar( 0, obtenerGanancia(hijo1) )
      gananciasHijos.insertar( 1, obtenerGanancia(hijo2) )

      if gananciasHijos[0][1] >= gananciasHijos[1][1] then
         mejor = gananciasHijos[0]
         peor = gananciasHijos[1]
      else
         mejor = gananciasHijos[1]
         peor = gananciasHijos[0]
      end if

      if mejor[1] > ganancias[1][1] then
         poblacion[ int(ganancias[1][0]) ] = hijos[ int(mejor[0]) ]
         ganancias[1] = mejor

         if peor[1] > ganancias[0][1] then
            poblacion[ ganancias[0][0] ] = hijos[ peor[0] ]
            ganancias[0] = peor
         elif mejor[1] > ganancias[0][1]
            poblacion[ ganancias[0][0] ] = hijos[ mejor[0] ]
            ganancias[0] = peor
         end if

      end if

      ganancias.ordenar(ascendente)

   end while

   w = poblacion[ ganancias[ganancias.size-1][0] ]
   resultados.insertar(evaluar(w))
```

Algoritmo de Búsqueda Local

Vamos a ver primero la implementación en pseudocódigo del algoritmo y posteriormente lo explicaremos paso a paso:

```
function busquedaLocal()

   while atributosExplorados < 2*n and not fin do
      hijosBL = hijos

      for posNuevo in permutacionAleatoria(tamañoAtributos) and not fin do
         hijosBL[i][posNuevo] += generarAleatorioNormal(0, 0.3**2)
         hijosBL[i][posNuevo] = acotarEntre(0, 1)
         evaluacionesRealizadas += 1
         gananciaNueva = obtenerGanancia(hijosBL[i])

         if gananciaNueva > gananciaAntigua then    # gananciaAntigua es la
            gananciaAntigua = gananciaNueva         # de los hijos antes de
            atributosExplorados = 0                 # la búsqueda local
            fin = True
         else
            atributosExplorados += 1
            hijos[i][posNuevo] = hijosBL[i][posNuevo]
         end if

         if atributosExplorados > 20*n:
            fin = True
         end if

      end for
   end while

end function
```

Antes de empezar hay que aclarar que esto será ejecutado en un bucle dentro de los algoritmos meméticos. Por eso, las variables *i* o *gananciaAntigua* no se declaran en ninguna parte, vienen del bucle y de la población de hijos antigua, respectivamente.

Tras esto ya si nos metemos en el algoritmo. Comenzamos con un while que pare de ejecutarse cuando explore **2*n atributos** (siendo n el número de atributos que tiene cada hijo).

Dentro del bucle, lo primero que hacemos es copiar a una variable auxiliar nuestros hijos para modificarla y no perder los datos si esta no mejora la ganancia. Después, generaremos un **vector aleatorio permutando posiciones** (obviamente con el tamaño de los atributos) y lo recorreremos de la siguiente forma. Generamos un vecino en la propia posición a través de una **distribución normal** de media 0 y varianza 0.3^2, y se lo sumaremos al vector hijosBL. Una vez hecho esto, calculamos la ganancia que tiene este array modificado.

Si esta **ganancia** nueva **es mejor** que la ganancia calculada al principio, la guardaremos como nueva mejor ganancia y terminaremos el bucle, ya que hemos encontrado una w mejor.

Al terminar el bucle, también tendremos que volver a poner el número de atributos explorados a 0 (volvemos al principio del while con nuevo array aleatorio de atributos).

En caso de que esta nueva **ganancia no sea mejor**, simplemente aumentaremos el número de atributos explorados en 1 y copiaremos el valor del hijos guardado anteriormente en el hijosBL que estamos modificando, ya que es peor.

Después de esta comprobación, también realizaré otra para comprobar lo mismo que se comprueba en el while, es decir, que haya explorado 2*n atributos. Esto es para que, si ha cumplido alguna de estas condiciones, corte el bucle for y no lo haga completamente.

Por último, cuando ya ha terminado, podremos seguir utilizando nuestro array de hijos ya modificado.

Algoritmo Memético (10, 1.0)

Los algoritmos meméticos implementado están basados en el algoritmo genético generacional con cruce BLX. Lo que cambia en estos es que, tras realizar todos los cruces y generar los nuevos hijos con sus ganancias, se realizará una búsqueda local como la que hicimos en la práctica anterior.

Lo primero que tenemos que saber es que los meméticos se van a ejecutar con una población de tamaño 10, pero con el mismo número de evaluaciones.

También tenemos que saber que la búsqueda local no se va a realizar en cada generación, si no que cuando el algoritmo generacional llegue a 10 evaluaciones, la realizará. En mi caso he utilizado una variable auxiliar *nGeneraciones* para no interferir en las evaluaciones. No obstante, dentro de la búsqueda local, también tendremos que aumentar el número de evaluaciones.

Por otra parte, vamos a tener tres tipos de algoritmos meméticos. En este caso, aplicaremos la búsqueda local sobre todos los cromosomas de la población, es decir, que vamos a recorrer todos los individuos y les vamos a hacer la búsqueda local a cada uno de ellos. El resto de casos se explicarán a continuación.

Por último, el vector solución que sacamos lo hacemos a partir de las últimas ganancias calculadas. Como éstas están ordenadas descendentemente, sólo tendremos que utilizar el índice de la mejor ganancia y extraer el individuo de la población al cual pertenece éste índice. Cuando lo tengamos ya simplemente evaluamos y guardamos el resultado de la ejecución en la matriz.

Veamos la implementación en pseudocódigo:

```
function AM()
   poblacion = generarSolucionesAleatorias()

   while nEval < maxEval do
      # Seleccionar P(t) torneo binario
      padres, gananciasPadres = torneoBinario()
         nEval += 1

      # Cruce P(t) con BLX
      for p in (tamPoblacion*0.7) and p+=2 do
         hijo1, hijo2 = cruceBLX(padres[p], padres[p+1], 0.3)
         hijos.insertar( mutate(hijo1) )
         hijos.insertar( mutate(hijo2) )
         gananciasHijos.insertar( p, obtenerGanancia(hijo1) )
         gananciasHijos.insertar( p+1, obtenerGanancia(hijo2) )
      end for

       # Completar los que faltan con los padres
      for p = (tamPoblacion*0.7) in tamPoblacion do
         hijos.insertar( padres[p] )
         gananciasHijos.insertar( gananciasPadres[p] )
      end for

      gananciasHijos.ordenar(ascendente)


      # Busqueda Local dentro del generacional
      if nGeneraciones % 10 == 0 do

         for i in tamPoblacion do
            gananciaAntigua = obtenerGanancia(hijos[i])
            busquedaLocal(i, gananciaAntigua)
         end for

      end if


      # Elitismo
      if ganancias[0][1] > gananciasHijos[0][1] do
         # Inserta en hijo a partir del indice
         hijos[ gananciasHijos[0][0] ] = poblacion[ ganancias[0][0] ]
         # Inserta la nueva ganancia
         gananciasHijos[0][1] = ganancias[0][1]
      end if

      poblacion = hijos
      ganancias = gananciasHijos
      ganancias.ordenar(descendente)

   end while

   w = poblacion[ ganancias[0][0] ]
   resultados.insertar(evaluar(w))
```

Algoritmo Memético (10, 0.1)

Este algoritmo memético sigue la misma estructura que el anterior, sólo que aquí tenemos que seleccionar un subconjunto de cromosomas de tamaño *tamPoblacion*0.1* y seleccionar aleatoriamente uno de ellos.

Esto es lo que tendremos que cambiar:

```
# Busqueda Local dentro del generacional
      if nGeneraciones % 10 == 0 do

         for j in tamPoblacion*0.1 do
            i = generarEnteroAleatorio(tamPoblacion)
            gananciaAntigua = obtenerGanancia(hijos[i])
            busquedaLocal(i, gananciaAntigua)
         end for

      end if
```

Algoritmo Memético (10, 0.1-mejor)

Este algoritmo memético sigue la misma estructura que los anteriores, sólo que aquí tenemos que seleccionar un subconjunto de cromosomas de tamaño *tamPoblacion*0.1* y en vez de seleccionar uno de ellos aleatoriamente, seleccionamos los mejores de la población. En mi caso, gracias a que tengo los vectores de ganancias ordenados no tengo que cambiar mucho el código.

Esto es lo que modifico:

```
# Busqueda Local dentro del generacional
      if nGeneraciones % 10 == 0 do

         for i in tamPoblacion*0.1 do
            gananciaAntigua = obtenerGanancia(hijos[i])
            busquedaLocal(i, gananciaAntigua)
         end for

      end if
```

Algoritmo de comparación

Esta función la he "dividido" en dos funciones:

- **evaluar(w, atributos, clases, tiempo).** Es mi clasificador 1NN, que compara los pesos de la función objetivo *w* con el *test* y evalúa como de buena es. Esta función devuelve los 4 valores estadísticos a partir de los parámetros anteriores.

- **obtenerGanancia(w, atributos, clases).** Clasificador 1NN, que compara los pesos de los propios datos con el *train*, ya que será utilizada en el entrenamiento de la función y no al comprobarlos como en el caso anterior. Esta función es la que utilizo en los distintos algoritmos para obtener sólo las nuevas ganancias (sin los otros valores estadísticos) y poder compararlas después.

Aclarado esto, vamos a ver la implementación en pseudocódigo de la función:

```
function evaluar(w, atributos, clases, antes)
   wNormalizados = quitarCaracterísticas(0.2, w)
   prod = (atributos * wNormalizados)

   arbol = cKDTree(prod)
   vecinos = arbol.obtenerVecinos(prod, k=2)        # leave-one-out

   _clas = clases[vecinos] / totalClases            # Hacer la medias
   _red = wMenoresQue(0.2) / totalCaracteristicas
   despues = obtenerTiempo()

   return _clas*100, _red*100, (_clas*0.5 + _red*0.5)*100, despues – antes
end function
```

Esta función está hecha **vectorizando**, así nos evitamos poner bucles. Primero selecciona los datos que sean menores que 0.2 (como indica en el guion) y después multiplicamos los atributos por sus pesos y mandamos el producto a un **KDTree**. Conseguimos una mejora de rendimiento porque al enviar todos los puntos al árbol directamente, podemos obtener los vecinos más cercanos sin tener que ir calculando las distancias euclídeas uno a uno en un bucle.

Después obtenemos los dos vecinos más cercanos y aplicamos la técnica del ***leave-one-out*** (k=2). Esta técnica consiste en elegir el segundo vecino más cercano y no tener en cuenta el primero, ya que este es el propio dato que estamos procesando. Si no aplicásemos esta técnica, la distancia euclídea sería 0.

Es cierto que un mismo dato no va a estar en el *train* y en el *test* simultáneamente, por lo que podemos pensar que no hace falta aplicar el *leave-one-out*. Sin embargo, cuando entrenamos nuestro algoritmo, ejecutamos nuestro clasificador 1NN ("obtenerGanancia(w, atributos,clases)") con el conjunto de datos *train*, por lo que este será nuestro conjunto tanto de entrenamiento como de prueba.

Por último, obtenemos los valores estadísticos como explicamos en la "Descripción del Problema". También volveremos a obtener el tiempo actual, ya que en este momento se acabará de ejecutar el algoritmo. Cuando los devolvemos, los valores están multiplicados

por 100 para obtener el porcentaje (menos el tiempo que simplemente hacemos una resta entre el inicial y el final).

La función de obtenerGanancia() es exactamente igual, exceptuando que no tendremos que pasarle el tiempo como parámetro y los valores que devolveremos no tienen que ser los 4 estadísticos, sino sólo el agregado. Éste valor es el único que nos va a hacer falta al comparar ganancias en los distintos algoritmos.

Desarrollo de la práctica y manual de usuario

La práctica está realizada en su totalidad en **python**, sin utilizar los códigos proporcionados en la web, otros códigos u otros frameworks externos.

La aplicación está desarrollada por completo en un script de python(.py), y también se incluye junto a este una carpeta doc, en la cual podremos encontrar los .**csv** utilizados para obtener los datos. Estos documentos no están exactamente igual que los .arff proporcionados originalmente, es decir, ya están parseados para no tener que complicar el código innecesariamente (al decir que no están exactamente igual, me refiero a la forma de mostrar los datos; obviamente los valores de estos siguen siendo exactamente los mismos).

En este script se hacen uso de las siguientes **librerías**, las cuales serán necesarias tener descargadas para poder ejecutar el script:

```
-   import pandas as pd
-   import numpy as np
-   import time
-   from scipy.spatial import KDTree
-   from scipy.spatial import cKDTree
-   from sklearn.model_selection import StratifiedKFold
```

En la línea 14 tendremos el **documento** que analizaremos y que podremos cambiar dependiendo de cuál queramos. Las opciones son: *'doc/colposcopy.csv'*, *'doc/ionosphere.csv'* y *'doc/texture.csv'*.

La **semilla** está declarada tanto en la línea 8 como en la línea 23. Las pruebas que yo he realizado utilizan la semilla 76067801 para generar los aleatorios.

Cuando ejecutemos el archivo nos saldrá un pequeño menú para poder decidir que algoritmo ejecutar. Esto lo he hecho así para no tener que estar mucho tiempo esperando a que todos terminen de ejecutarse, en caso de que queramos ver uno sólo.

Este **menú** tendrá principalmente 9 opciones:

- 1 - Búsqueda Local

- 2 - Greedy RELIEF
- 3 - AGG Cruce BLX
- 4 - AGG Cruce Aritmético
- 5 - AGE Cruce BLX
- 6 - AGE Cruce Aritmético
- 7 - AM (10,1.0)
- 8 - AM (10,0.1)
- 9 - AM (10,0.1mej)

Cada número representa una de ellas. En la consola insertamos el que queramos ejecutar y le damos a enter.

Las cuatro últimas líneas del código sirven para **visualizar** los datos obtenidos. Por defecto, yo lo dejo para que muestre por consola los resultados de la búsqueda local. Opcionalmente, si quitamos el comentario a las siguientes, podremos mostrar el resultado del algoritmo RELIEF y también podremos exportar el resultado a un .csv externo con el nombre *'result.csv'*.

Análisis de los resultados

Se han ejecutado los 3 conjuntos de datos proporcionados:

- **Colposcopy**. Conjunto de datos de colposcopias que dispone de 287 ejemplos con 62 características y deben ser clasificados en 2 clases.
- **Ionosphere**. Conjunto de datos de radar que dispone de 352 ejemplos con 34 características y deben ser clasificados en 2 clases.
- **Texture**. Conjunto de datos de imágenes y hay que diferenciar a que pertenece cada una. Dispone de 550 ejemplos con 40 características y deben ser clasificados en 11 clases.

Otros parámetros a tener en cuenta pueden ser: la semilla, para la cual yo he utilizado el valor 14; el número máximo de evaluaciones y el número máximo de vecinos, que valdrá 15000 y 2*n respectivamente; la media y la varianza para generar números aleatorios a partir de una distribución normal (0 y 0.3 respectivamente); y el tamaño de la población, que vale 30 excepto para los algoritmos meméticos que vale 10.

Para cada experimento se han realizado 5 particiones y, a continuación, vamos a mostrar las tablas de los resultados obtenidos:

Resultados obtenidos por el algoritmo AGG-BLX en el problema del APC												
	Colposcopy				Ionosphere				Texture			
	%_class	*%red*	*Agr.*	T	*%_class*	*%red*	*Agr.*	T	*%_class*	*%red*	*Agr.*	T
Partición 1	61,01	41,93	51,47	38,35	76,06	52,94	64,50	47,78	88,18	47,50	67,84	73,56
Partición 2	68,42	35,48	51,95	37,53	82,85	44,11	63,49	55,21	80,90	57,50	69,20	66,98
Partición 3	70,17	35,48	52,82	40,22	78,57	50,00	64,28	51,08	78,18	55,00	66,59	64,44
Partición 4	64,91	48,38	56,64	39,08	85,71	52,94	69,33	46,66	86,36	42,50	64,43	73,21
Partición 5	71,92	30,64	51,28	46,61	82,88	52,94	67,90	43,88	75,45	55,00	65,22	72,55
Media	**67,29**	**38,38**	**52,83**	**40,36**	**81,21**	**50,59**	**65,90**	**48,92**	**81,81**	**51,50**	**66,66**	**70,15**

Resultados obtenidos por el algoritmo AGG-Aritmetico en el problema del APC												
	Colposcopy				Ionosphere				Texture			
	%_class	*%red*	*Agr.*	T	*%_class*	*%red*	*Agr.*	T	*%_class*	*%red*	*Agr.*	T
Partición 1	64,40	35,48	49,94	37,86	84,50	29,41	56,96	48,62	87,27	30,00	58,63	67,52
Partición 2	73,68	25,80	49,74	36,00	87,14	41,17	64,16	46,98	85,45	35,00	60,22	66,35
Partición 3	75,43	32,25	53,84	37,13	74,28	38,23	56,26	47,45	82,72	42,50	62,61	64,07
Partición 4	64,91	37,09	51,00	38,58	74,28	32,35	53,31	45,96	87,27	35,00	61,13	72,07
Partición 5	70,17	33,87	52,02	36,29	77,14	38,23	57,69	42,23	76,36	45,00	60,68	61,69
Media	**69,72**	**32,90**	**51,31**	**37,17**	**79,47**	**35,88**	**57,68**	**46,25**	**83,81**	**37,50**	**60,65**	**66,34**

Resultados obtenidos por el algoritmo AGE-BLX en el problema del APC												
	Colposcopy				Ionosphere				Texture			
	%_class	*%red*	*Agr.*	T	*%_class*	*%red*	*Agr.*	T	*%_class*	*%red*	*Agr.*	T
Partición 1	62,71	40,32	51,51	40,62	83,09	44,11	63,61	65,30	87,27	32,50	59,88	82,17
Partición 2	64,91	37,09	51,00	41,23	85,71	35,29	60,50	67,93	86,36	47,50	66,93	76,25
Partición 3	71,92	35,48	53,70	37,54	80,00	52,94	66,47	67,22	78,18	52,50	65,34	74,98
Partición 4	68,48	37,09	52,75	39,65	82,85	55,88	69,36	64,58	87,27	42,50	64,88	67,66
Partición 5	70,17	37,09	53,63	40,88	81,43	41,17	61,30	64,32	75,45	50,00	62,72	73,36
Media	**67,64**	**37,41**	**52,52**	**39,98**	**82,62**	**45,88**	**64,25**	**65,87**	**82,91**	**45,00**	**63,95**	**74,88**

Resultados obtenidos por el algoritmo AGE-Aritmetico en el problema del APC												
	Colposcopy				Ionosphere				Texture			
	%_class	*%red*	*Agr.*	T	*%_class*	*%red*	*Agr.*	T	*%_class*	*%red*	*Agr.*	T
Partición 1	72,88	43,55	58,21	43,39	84,50	29,41	56,95	62,12	81,82	45,00	63,41	4,12
Partición 2	73,68	19,35	46,52	41,22	82,85	35,29	59,07	62,84	80,91	50,00	65,45	2,93
Partición 3	77,19	24,19	50,69	38,21	74,28	41,14	57,73	63,56	82,73	30,00	56,36	5,01
Partición 4	66,67	32,26	49,46	41,29	80,00	17,64	48,82	63,49	87,27	52,50	69,89	3,79
Partición 5	73,68	40,32	57,00	40,66	78,57	41,17	59,87	61,70	79,09	47,50	63,30	5,08
Media	**72,82**	**31,94**	**52,38**	**40,95**	**80,04**	**32,93**	**56,49**	**62,74**	**82,36**	**45,00**	**63,68**	**4,18**

Resultados obtenidos por el algoritmo AM-(10,0.1) en el problema del APC												
	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	64,40	37,09	50,75	32,07	83,09	41,17	62,13	60,39	88,18	40,00	64,04	79,25
Partición 2	68,42	24,19	46,30	40,32	84,28	44,11	64,20	64,57	83,63	50,00	66,81	68,58
Partición 3	77,19	19,35	48,27	39,25	82,86	44,11	63,48	71,61	81,81	40,00	60,90	81,74
Partición 4	57,89	38,70	48,30	31,14	78,57	41,17	59,87	65,43	85,45	22,50	53,97	79,78
Partición 5	66,66	41,93	54,30	36,25	84,28	35,29	59,78	72,22	74,54	47,50	61,02	77,45
Media	**66,91**	**32,25**	**49,58**	**35,81**	**82,62**	**41,17**	**61,89**	**66,84**	**82,72**	**40,00**	**61,35**	**77,36**

Resultados obtenidos por el algoritmo AM-(10,0.1-mejor) en el problema del APC												
	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	64,40	33,87	49,13	29,48	83,09	38,23	60,66	36,33	89,09	32,50	60,79	63,62
Partición 2	68,42	30,64	49,53	28,22	87,14	47,05	67,10	37,17	86,36	42,50	64,43	52,65
Partición 3	77,11	29,03	53,11	32,48	82,85	38,23	60,54	42,53	81,81	42,50	62,15	52,82
Partición 4	63,15	33,87	48,51	30,62	77,14	38,23	57,68	39,06	86,36	30,00	58,18	63,71
Partición 5	71,92	32,25	52,09	32,09	81,42	35,29	58,36	45,89	71,81	42,50	57,15	51,72
Media	**69,00**	**31,93**	**50,47**	**30,58**	**82,33**	**39,41**	**60,87**	**40,20**	**83,09**	**38,00**	**60,54**	**56,90**

Antes de analizarlos, comentar que como en la práctica anterior los algoritmos tenían errores, no voy a compararlos con estos nuevos por falta de tiempo. No obstante, el algoritmo de búsqueda local creo que lo he arreglado correctamente.

Analizando los resultados obtenidos vemos que todos, o casi todos, los resultados son bastante similares. Aun así, los mejores resultados generalmente los tienen los algoritmos meméticos.

No obstante, el AM-(10,1.0) es el que más tarda con diferencia. Esto se debe a que la búsqueda local la realiza para toda la población (un bucle for más en cada evaluación de esta). Aún así no ofrece resultados malos, pero teniendo en cuenta los otros dos algoritmos meméticos, no merece la pena.

En general la tasa de acierto son bastante parecidas las unas a las otras, y da que pensar si estos algoritmos tan costosos merecen la pena para una ganancia tan baja. Cierto es que los conjuntos de datos utilizados son bastante pequeños y los algoritmos evolutivos no demuestran del todo su potencial.

Por otra parte, analizando los datos dentro de la aplicación he visto que el sobreaprendizaje es un problema. El sobreaprendizaje puede provocar que el algoritmo sea muy bueno a la hora de entrenar los datos, pero después al hacer el test, no funciona tan bien. Esto es porque se centra tanto en estos datos que pierde eficacia al valorar los datos que hay fuera del conjunto.

Otro aspecto a analizar el la convergencia y las 15000 iteraciones. Pienso que tantas iteraciones son innecesarias, ya que analizando desde la aplicación estos datos, se ve que el algoritmo llega a la solución bastante rápido (no necesita 15000 iteraciones). Al llegar a la solución tan rápido, lo que hace es ajustar muchísimo los decimales y esto nos lo podríamos ahorrar, ya que el resultado apenas se vería afectado.

Esto toma más valor si tenemos en cuenta que estamos entrenando los datos y ajustarlos tanto puede ser insignificante luego para los test, ya que los datos que tenemos aquí son totalmente diferentes a los anteriores.

Como conclusión, decir que estos algoritmos genéticos con una buena optimización probablemente sí que sean mucho mejores que los de prácticas anteriores, ya que por poco que sea, es preferible mejorar el ajuste cuanto más mejor. En mi caso, con los tiempos que he obtenido, quizás si me es de utilidad estos algoritmo, pero probablemente por falta de optimización mía tardan demasiado como para la ganancia que suponen.

Bibliografía

- Seminaro 2. Problemas de optimización con técnicas basadas en búsqueda local.
- Tema 2. Modelos de Búsqueda: Entornos y Trayectorias vs Poblaciones.
- https://en.wikipedia.org/wiki/Feature_learning
- https://machinelearningmastery.com/feature-selection-machine-learning-python
- https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.cKDTree.html
- https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.KDTree.html#scipy.spatial.KDTree
- https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html
- https://en.wikipedia.org/wiki/K-means_clustering
- https://docs.scipy.org/doc/numpy/reference/routines.random.html
- https://docs.python.org/2/library/time.html
- https://docs.scipy.org/doc/numpy/reference/generated/numpy.clip.html