Práctica Algoritmos Genéticos y Meméticos APC Metaheurísticas

Ignacio Aguilera Martos
DNI: 77448262V e-mail: nacheteam@correo.ugr.es
Grupo de prácticas 1 Lunes 17:30-19:30

Curso 2017-2018

Índice

1. Introducción del problema

Para el problema de clasificación partimos de un conjunto de datos dado por una serie de tuplas que contienen los valores de atributos para cada instancia. Esto es una n-tupla de valores reales en nuestro caso.
El objetivo del problema es obtener un vector de pesos que asocia un valor en el intervalo $[0, 1]$ indicativo de la relevancia de ese atributo. Esta relevancia va referida a lo importante que es en nuestro algoritmo clasificador ese atributo a la hora de computar la distancia entre elementos.
Resumiendo lo que tenemos es un algoritmo clasificador que utiliza el vector de pesos calculado para predecir la clase a la que pertenece una instancia dada. Este algoritmo clasificador es el KNN con k=1. Lo que hace es calcular según la distancia euclídea (o cualquier otra) la tupla más cercana a la que queremos clasificar ponderando cada atributo con el correspondiente peso del vector, es decir, la distancia entre dos elementos sería:

$$d(e, f) = \sqrt{\sum_{i=0}^{n} w_i * (e_i - f_i)}$$

Donde e y f son instancias del conjunto de datos, w el vector de pesos y n la longitud de e y f que es la misma.
La calificación que se le asigna al vector w depende de dos cosas: la tasa de aciertos y la simplicidad.
La tasa de aciertos se mide contando el número de aciertos al emplear el clasificador descrito y la simplicidad se mide como el número de elementos del vector de pesos que son menores que 0.2, ya que estos pesos no son empleados por el clasificador, o lo que es lo mismo, son sustituidos por cero. Por lo tanto las calificaciones siguen las fórmulas:

$$Tasa\_acierto = 100 \cdot \frac{n^o\ aciertos}{n^o\ datos}\ ,\ Tasa\_simplicidad = 100 \cdot \frac{n^o\ valores\ de\ w\ <\ 0{,}2}{n^o\ de\ atributos}$$

$$Tasa\_agregada = \frac{1}{2} \cdot Tasa\_acierto + \frac{1}{2} \cdot Tasa\_simplicidad$$

Cabe destacar que todas las tasas están expresadas en porcentajes, por lo tanto cuanto más cercano sea el valor a 100 mejor es la calificación.
De esta forma a través del algoritmo que obtiene el vector de pesos para el conjunto de datos dado y el clasificador obtenemos un programa que clasifica de forma automática las nuevas instancias de datos que se introduzcan.

2. Introducción de la práctica

En esta práctica he desarrollado algoritmos genéticos tanto estacionarios como generacionales con los dos operadores de cruce propuestos (aritmético y BLX) así como un algoritmo memético basado en el genético generacional.
Al igual que en la práctica anterior el objetivo es ejecutar estos algoritmos sobre los conjuntos de datos dados para observar su comportamiento y realizar una comparativa entre los mismos.
A los algoritmos mencionados anteriormente se les suman el 1NN con todos los pesos a uno, el greedy Relief y la búsqueda local (algoritmos implementados en la primera práctica).
Igual que en la práctica anterior he realizado un procesamiento de los datos para eliminar tuplas repetidas, de forma que ya tenemos implementado el leave one out para conjuntos distintos en el KNN y además, he implementado una versión más rápida de KNN usando la librería NumPy con la intención de reducir tiempos en la búsqueda local, en los algoritmos genéticos y en los meméticos.

En la práctica se desarrollará cómo he implementado los algoritmos genéticos (en sus dos variantes) incluyendo los operadores de cruce y mutación así como su adaptación a algoritmo memético.

3. Descripción común a todos los algoritmos

Los algoritmos empleados han sido el KNN, el algoritmo greedy Relief, la metaheurística de búsqueda local, un algoritmo genético estacionario, un algoritmo genético generacional y un memético basado en el algoritmo genético generacional.
Estos algoritmos comparten ciertos métodos y operadores que pasaré a explicar en esta sección.
Para empezar se debe destacar que la representación escogida para las soluciones es un vector de números reales, es decir, si n es el número de características:

$$w \in \mathbb{R}^n \ t.q. \ \forall i \ con \ 0 \leq i < n \ se \ tiene \ w_i \in [0,1]$$

O lo que es lo mismo, un vector de tamaño n con todas las posiciones rellenas con números del intervalo [0,1].
A estos números me referiré como pesos asociados a las características, ya que lo que nos indican es el grado de importancia de dicha característica a la hora de clasificar los datos, siendo 1 el máximo de relevancia y 0 el mínimo.
Así mismo cabe destacar que nuestra intención en este problema es obtener una buena calificación de dicho vector de pesos. Esto lo medimos mediante las tasas de acierto y simplicidad que se definen como:

$$Tasa\_acierto = 100 \cdot \frac{n^o \ aciertos}{n^o \ datos} \ , \ Tasa\_simplicidad = 100 \cdot \frac{n^o \ valores \ de \ w \ < \ 0{,}2}{n^o \ de \ atributos}$$

$$Tasa\_agregada = \frac{1}{2} \cdot Tasa\_acierto + \frac{1}{2} \cdot Tasa\_simplicidad$$

La tasa de aciertos lo que nos mide es en un porcentaje cuántas instancias hemos clasificado correctamente mediante el algoritmo KNN usando el vector de pesos w.
La tasa de simplicidad nos mide cuántos de los valores que tiene el vector de pesos son menores que 0.2. Esto se hace ya que, como imposición del problema, tenemos que si alguno de los pesos es menor que 0.2 no debemos usarlo, o lo que es lo mismo, debemos sustituirlo por un 0 en la función de la distancia que luego describiré. Midiendo esto obtenemos un dato de cuanto sobreajuste ha tenido nuestro algoritmo a la hora de obtener el vector de pesos. Cuantas menos características necesitemos para discernir la clase a la que pertenece una instancia de los datos, más simple será clasificar dicha instancia. Se expresa en porcentaje indicando 0 como ninguna simplicidad y 100 como la máxima simplicidad.
De esta forma combinando ambas tasas obtenemos la tasa agregada que nos hace la media entre ambas tasas, de forma que le asignamos la misma importancia a acertar en la clasificación de las instancias y a la simplicidad en la solución. Cabe destacar que es imposible obtener una tasa de un 100 % a no ser que los datos se compongan únicamente de un punto ya que ello implicaría que la simplicidad ha de ser un 100 % (todos las posiciones del vector menores que 0.2) y por tanto la distancia sería 0 en todos los casos. De esta forma aspiraremos a una calificación lo mas alta posible pero teniendo en cuenta las restricciones de la función objetivo construida.
Las funciones y operadores de uso común los he agrupado en un fichero llamado auxiliar.py. Este fichero contiene las funciones de lectura de datos, distancias, una función que devuelve el elemento más común de una lista, la norma euclídea, una función para dividir los datos en el número de particiones que queramos manteniendo el porcentaje de elementos de cada clase que había en el conjunto de datos original y el operador de mutación.

3.1. Función de lectura de datos

La función de lectura de datos recibe la ruta del fichero arff y lee el contenido del mismo dando como resultado una lista de listas en la que cada una de ellas es una tupla o instancia de los datos.

El pseudocódigo de la función es:

Algorithm 1 lecturaDatos(nombre_fich)

```
data ← [ ]
for linea de nombre_fich do
  if se ha leído @data then
    data ← [data,linea]
  end if
end for
for tupla en data do
  for atributo en tupla do
    Normalizar.
  end for
end for
for i=0,...,longitud(data)-1 do
  for j=i,...,longitud(data)-1 do
    if data[i]==data[j] then
      Eliminar data[i].
    end if
  end for
end for
return data
```

Para esta implementación en concreto nos apoyamos en que Python tiene polimorfismo para todos los tipos de datos sin necesidad de declarar las variables, de forma que no nos importa que los datos sean numéricos o de tipo string.
Nótese así mismo que hemos eliminado los duplicados en el conjunto de datos gracias a los dos últimos bucles anidados.

3.2. Funcion de distancia

Se debe tener en cuenta que e1 y e2 son ambos dos tuplas del conjunto de datos de las que vamos a obtener la distancia y w es el vector de pesos que toma parte en el cómputo.

Algorithm 2 distanciaEuclidea(e1,e2,w)

```
if longitud(e1)!=longitud(e2) then
  No se puede hallar la distancia.
else
  distancia = ∑_{i=1}^{longitud(e1)} w_i · (e1_i − e2_i)^2
end if
```

Esta fórmula está integrada en el algoritmo KNN para realizarla de forma matricial, de forma que hacemos el cálculo mucho más rápido.

3.3. Función MasComun

Esta función lo que hace es devolvernos el elemento que más se repite dentro de un vector, esto es utilizado en el algoritmo KNN para obtener la clase más común entre los k elementos con distancia mínima al dado.

Algorithm 3 masComun(lista)

```
vector_repeticiones ← [ ]
for elemento en lista do
  Contar el número de veces que aparece e introducirlo en el vector de repeticiones.
end for
Obtener el elemento con mayor número de apariciones.
```

3.4. Función de norma euclídea

Esta función toma una lista de elementos y devuelve la norma euclídea asociada al vector que representa la lista.

Algorithm 4 normaEuclidea(e)

```
norma ← 0
for ei en e do
  norma ← norma+ei^2
end for
norma ← sqrt(norma)
return norma
```

3.5. Función de división de datos

Esta función divide el conjunto de datos inicial en n particiones todas ellas respetando el porcentaje de ocurrencias de cada clase que tenía el conjunto de datos inicial.

Algorithm 5 divideDatosFCV(data,n)

```
particiones ← [ ]
tam_particion ← longitud(data) / n
clases ← Posibles clases del conjunto data.
proporciones_clases ← [ ]
for clase en clases do
  proporciones_clases ← [proporciones_clases, nueva_proporcion]
end for
for i en 0..n do
  particion ← [ ]
  for j en 0..numero_clases do
    numero_elementos_clase = proporciones_clases[j]*tam_particion
    for k en 0..longitud(data) do
      Introducir en particion el número de elementos de clase calculado para cada clase.
    end for
  end for
end for
Si han sobrado datos sin colocar los ponemos en la última partición.
return particiones
```

3.6. Función de mutación

Esta función recibe como entrada un vector de pesos y una posición que es la que se desea mutar, devolviendo como resultado el vector de pesos ya mutado y la posición aumentada en

una unidad (se usa para el algoritmo de búsqueda local aunque puede ignorarse).

Algorithm 6 mutacion(w,pos)

```
incremento = gauss(mu=0,sigma=0.3)
posicion_nueva = pos+1
w[pos]+=incremento
Truncar el vector w (0 si es negativo y 1 si es mayor que 1).
return w,pos_nueva
```

Esta función es usada en búsqueda local y en todos los genéticos y meméticos a la hora de realizar la mutación de los cromosomas.

4. KNN

El algoritmo KNN ha sido utilizado en estas prácticas tanto como clasificador (y por tanto con el objetivo de obtener una valoración de un vector de pesos) como algoritmo de comparación. El guión de prácticas nos pedía implementar el algoritmo 1NN, pero yo he decidido implementarlo de forma genérica para poder comprobar cómo cambiaban los resultados cuando incrementamos el número de vecinos más cercanos.
Este algoritmo toma como entrada un vector de pesos, un conjunto de datos de entrenamiento con sus etiquetas de clase, otro de test con el conjunto de etiquetas también, un valor k que nos indica cuántos vecinos más cercanos queremos tomar para clasificar cada dato y un booleano que nos indica si los dos conjuntos pasados son iguales (cosa que ocurre por ejemplo en la búsqueda local, genéticos y el memético) con la intención de aplicar el leave-one-out. El procedimiento consiste en tomar cada dato del conjunto de test y hallar los k datos del conjunto de entrenamiento que tienen distancia mínima con el dato tomado del conjunto de test. De ahí lo que hacemos es clasificar el dato con la clase más frecuente de entre los k vecinos más cercanos calculados. De esta forma, intuitivamente, asignamos la misma clase a datos que están próximos entre sí. Una vez obtenida la clase de estos elementos lo que hacemos es comprobar cuántas clases hemos acertado al clasificar de esta forma. Con esto obtenemos tras la ejecución del algoritmo un número entre 0 y 100 que nos indica cual ha sido nuestra tasa de acierto (100 todo correcto, 0 ninguna correcta).
Para emplear este algoritmo como comparación para los otros lo que hacemos es introducir como vector de pesos un vector con todas las posiciones a 1. De esta forma obtenemos la calificación que obtendríamos si no hubiéramos incluido en la fórmula de la distancia una ponderación con pesos.
Nótese que para esta implementación el vector de pesos pasado tiene, en las posiciones con valor inicial menor que 0.29, ceros. Esto se hace con la intención de mejorar la eficiencia del algoritmo.
El pseudocódigo del algoritmo es:

Algorithm 7 KNN(w,datos_test,datos_entrenamiento, etiquetas_entrenamiento, etiquetas_test, k, mismos_conjuntos)

```
tam_datos_entrenamiento ← longitud(datos_entrenamiento)
clases ← []
for i=0,...,longitud(datos_test) do
    p ← datos_test[i]
    w_m ← Repetir el vector w tantas veces como datos haya en datos_entrenamiento.
    p_m ← Repetir el vector p tantas veces como datos haya en datos_entrenamiento.
    dist ← w_m · (p_m − datos_entrenamiento)^2
    if mismos_conjuntos then
        dist[i] ← ∞
    end if
    mins ← Los k índices correspondientes a las distancias más pequeñas.
    clases ← [clases, masComun(etiquetas_entrenamiento[mins])]
end for
return Numero de elementos de clases que han acertado con respecto a etiquetas_test / longitud(etiquetas_test)
```

Cabe notar que el número que devolvemos está entre 0 y 1, por lo que en los algoritmos de valoración debemos tener esto en cuenta para multiplicarlo por 100 y convertirlo en un porcentaje.

5. Relief

El algoritmo Relief es un algoritmo greedy que busca rapidez en detrimento de mejores resultados de clasificación.
El algoritmo toma como entrada únicamente el conjunto de datos. Para cada elemento del conjunto buscamos el elemento de la misma clase más cercano y el elemento de la clase contraria más cercano de forma que tenemos al amigo y enemigos más cercanos. En la literatura es común encontrar estos elementos como near hit y near miss.
Partiendo de un vector de pesos con todas las posiciones a 0 tomamos el amigo y enemigo más cercano y restamos a nuestro elemento del conjunto de datos tanto el amigo como el enemigo. Después de esto lo que hacemos es sumar la resta del enemigo y restar la resta del amigo, de forma que aumentamos la distancia que dicho vector de pesos produce para hacer que el amigo esté más cercano del dato que el enemigo.
Esto ocurre ya que sumamos la distancia del elemento al enemigo haciendo que los pesos sean mayores para el mismo y restamos la distancia del elemento al amigo haciendo que nos acerque en distancia a dicha tupla.
A continuación se describe el funcionamiento en pseudocódigo:

Algorithm 8 elementoMinimaDistancia(e,lista)

```
distancias ← [ ]
for l en lista do
  if l!=e then
    distancias ← [distancias, distancia(e,l,[1..1])]
  else
    distancias ← [distancias, max(distancias)]
  end if
end for
indice_menor_distancia ← índice del elemento de menor valor del vector distancias.
return lista[indice_menor_distancia]
```

Tenemos en primer lugar la función elementoMinimaDistancia que nos devuelve el elemento de la lista más cercano al elemento e. De esta forma podemos hallar el amigo y enemigo más cercano para el algoritmo Relief. Cabe destacar que para que el vector de pesos esté con números entre 0 y 1 tenemos que poner al final del algoritmo los elementos del vector negativos a 0 y el resto los normalizamos dividiendo por el máximo elemento del vector de forma que todas las posiciones nos queden entre 0 y 1.

Algorithm 9 Relief(data)

```
w ← vector de pesos a 0
for elemento en data do
  clase ← clase de elemento
  amigos ← [ ]
  enemigos ← [ ]
  for e en data do
    if e!=elemento AND e[longitud(e)-1]==clase then
      amigos ← [amigos, e]
    else
      enemigos ← [enemigos, e]
    end if
  end for
  amigo_cercano ← elementoMinimaDistancia(elemento, amigos)
  enemigo_cercano ← elementoMinimaDistancia(elemento, enemigos)
  resta_enemigo ← element-enemigo_cercano
  resta_amigo ← element-amigo_cercano
  w ← w + resta_enemigo - resta_amigo
  w_max ← máximo de w
end for
for i en [0..longitud(w)-1] do
  if w[i]<0 then
    w[i] ← 0
  else
    w[i] ← w[i] / w_max
  end if
end for
return w
```

6. Búsqueda Local

El algoritmo de búsqueda local tiene como intención, al igual que el anterior, proporcionarnos un vector de pesos que maximice la tasa agregada que obtenemos como ya he mencionado anteriormente.
Este algoritmo comienza con un vector de pesos aleatorio generado con una distribución uniforme en el intervalo [0,1], obteniendo a partir de ahí mejores vectores a cada iteración. La generación de vecinos viene dada por un operador que toma posiciones aleatorias del vector de pesos (sin repetición) y les suma un valor aleatorio generado con una distribución normal de media 0 y desviación 0.3. Tras esto truncamos el vector de pesos para que cumpla las restricciones del problema.
Dado el vector de pesos inicial obtenemos su valoración de la tasa agregada (incluye la simplicidad y la tasa de aciertos). Generamos un vecino mediante la forma mencionada anteriormente y comprobamos si su tasa es mayor que la del vector de pesos actual. Si esto es así descartamos nuestro vector de pesos y tomamos al vecino, en caso contrario continuaremos explorando el vecindario hasta encontrar uno que mejore el resultado, agotar las posibilidades de vecinos o llegar a 15.000 evaluaciones de la función objetivo o $20 \cdot n$ vecinos explorados donde n es el número de atributos de cada tupla de datos.
De esta manera otorgamos una forma de parar a nuestro algoritmo y no excederse en la búsqueda.
Las funciones que intervienen, en pseudocódigo, son las siguientes:

Algorithm 10 primerVector(n)

```
w ← [ ]
for i en [0..n-1] do
    w ← [w, random.uniforme(0,1)]
end for
return w
```

Debe tomarse en cuenta que la función Valoracion(data,data,k,w) devuelve la tasa agregada teniendo en cuenta la simplicidad y la tasa de acierto como se ha descrito en el algoritmo KNN.
Este algoritmo, como ya discutiré en la parte de resultados, puede quedarse atrapado fácilmente en un máximo local y va a consumir mucho más tiempo, ya que cada llamada de la función valoración ejecuta el algoritmo KNN sobre dicho conjunto de datos sumando así mayor tiempo de ejecución que el de sus competidores.

7. Genético Estacionario

8. Procedimiento de desarrollo de la práctica

9. Resultados

Algorithm 11 busquedaLocal(data,k)

```
MAX_EVALUACIONES ← 15000
MAX_VECINOS ← 20 · longitud(data[0])
vecinos ← 0
evaluaciones ← 0
posicion_mutacion ← 0
w ← primerVector(longitud(data[0]))
valoracion_actual ← Valoracion(data,data,k,w)
while evaluaciones<MAX_EVALUACIONES AND vecinos<MAX_VECINOS do
  evaluaciones ← evaluaciones+1
  vecinos ← vecinos+1
  vecino, posicion_mutacion ← mutacion(w,posicion_mutacion)
  valoracion_vecino ← Valoracion(data,data,k,vecino)
  if valoracion_vecino>valoracion_actual then
    vecinos ← 0
    w ← vecino
    valoracion_actual ← valoracion_vecino
    posicion_mutacion ← 0
  else if posicion_mutacion==longitud(w) then
    posicion_mutacion ← 0
  end if
end while
return w
```

	Ozone				Parkinsons				Spectf-Heart			
	%_clas	%_red	Agr.	T (seg)	%_clas	%_red	Agr.	T (seg)	%_clas	%_red	Agr.	T (seg)
Partición 1	71.8750	0.0000	35.9375	0.4403	76.3158	0.0000	38.1579	0.0546	70.5882	0.0000	35.2941	0.2561
Partición 2	84.3750	0.0000	42.1875	0.4341	81.5789	0.0000	40.7895	0.0527	77.9412	0.0000	38.9706	0.2959
Partición 3	71.8750	0.0000	35.9375	0.4329	94.7368	0.0000	47.3684	0.0543	67.6471	0.0000	33.8235	0.3161
Partición 4	81.2500	0.0000	40.6250	0.4340	73.6842	0.0000	36.8421	0.0530	60.2941	0.0000	30.1471	0.2984
Partición 5	85.9375	0.0000	42.9688	0.4484	76.7442	0.0000	38.3721	0.0586	66.2338	0.0000	33.1169	0.2715
Media	79.0625	0.0000	39.5313	0.4380	80.6120	0.0000	40.3060	0.0546	68.5409	0.0000	34.2704	0.2876

Cuadro 1: Resultados 1NN

	Ozone				Parkinsons				Spectf-Heart			
	%_clas	%_red	Agr.	T (seg)	%_clas	%_red	Agr.	T (seg)	%_clas	%_red	Agr.	T (seg)
Partición 1	71.8750	98.6301	85.2526	0.9703	78.9474	47.8261	63.3867	0.1254	75.0000	46.6667	60.8333	0.4568
Partición 2	87.5000	97.2603	92.3801	1.0143	81.5789	26.0869	53.8330	0.1267	73.5294	53.3333	63.4314	0.6407
Partición 3	71.8750	97.2603	84.5676	1.0385	86.8421	47.8261	67.3341	0.1263	67.6471	60.0000	63.8235	0.7070
Partición 4	81.2500	95.8904	88.5702	1.0530	78.9474	52.1739	65.5606	0.1254	72.0588	31.1111	51.5850	0.6594
Partición 5	85.9375	98.6301	92.2838	0.9963	72.0930	43.4783	57.7856	0.1180	72.7273	48.8889	60.8081	0.3986
Media	79.6875	97.5342	88.6109	1.0145	79.6818	43.4783	61.5800	0.1244	72.1925	48.0000	60.0963	0.5725

Cuadro 2: Resultados Relief con K=1

	Ozone				Parkinsons				Spectf-Heart			
	%_clas	%_red	Agr.	T (seg)	%_clas	%_red	Agr.	T (seg)	%_clas	%_red	Agr.	T (seg)
Partición 1	71.8750	34.2466	53.0608	186.6400	78.9474	26.0870	52.5172	9.0305	70.5882	35.5556	53.0719	52.5939
Partición 2	76.5625	54.7945	65.6785	436.2914	81.5789	39.1304	60.3547	6.8897	82.3529	24.4444	53.3987	92.8766
Partición 3	70.3125	30.1370	50.2247	185.3929	86.8421	17.3913	52.1167	4.3802	70.5882	57.7778	64.1830	120.7886
Partición 4	82.8125	54.7945	68.8035	354.7868	78.9474	52.1739	65.5606	21.2622	63.2353	31.1111	47.1732	70.7596
Partición 5	84.3750	39.7260	62.0505	224.3229	74.4186	30.4348	52.4267	9.3806	64.9351	44.4444	54.6898	90.2908
Media	77.1875	42.7397	59.9636	277.4868	80.1469	33.0435	56.5952	10.1886	70.3400	38.6667	54.5033	85.4619

Cuadro 3: Resultados Búsqueda Local con K=1

	Ozone				Parkinsons				Spectf-Heart			
	%_clas	%_red	Agr.	T (seg)	%_clas	%_red	Agr.	T (seg)	%_clas	%_red	Agr.	T (seg)
1-NN	79.0625	0.0000	39.5313	0.4380	80.6120	0.0000	40.3060	0.0546	68.5409	0.0000	34.2704	0.2876
Relief	79.6875	97.5342	88.6109	1.0145	79.6818	43.4783	61.5800	0.1244	72.1925	48.0000	60.0963	0.5725
BL	77.1875	42.7397	59.9636	277.4868	80.1469	33.0435	56.5952	10.1886	70.3400	38.6667	54.5033	85.4619

Cuadro 4: Resultados globales con K=1