Metaheurísticas (2015-2016)
Grado en Ingeniería Informática
Universidad de Granada

Práctica 3.b: Algoritmos genéticos para el Problema de Selección de Características

---

Laura Tirado López
DNI: 77145787S
email: muscraziest@correo.ugr.es
Grupo de prácticas 1
Horario: Jueves 17:30-19:30

5 de mayo de 2016

Índice

1. Introducción al problema de Selección de Características

El problema de selección de características consiste en determinar la clase a la que pertenecen un conjunto de datos o instancias caracterizadas por una serie de atributos. En el aprendizaje supervisado utilizamos un conjunto de instancias con sus correspondientes atributos y un atributo adicional que indica la clase a la que pertenece cada instancia. La idea es crear y generalizar una regla o un conjunto de reglas a partir de este conjunto que nos permita clasificar de la mejor manera posible un conjunto de instancias. Para clasificar los datos existen muchos métodos o reglas. En nuestro caso, vamos a implementar un clasificador k-NN que sigue el criterio de los k vecinos más cercanos. Els número de vecinos que utilizaremos será 3.

A pesar de esto, en muchos casos se observa que no todas las variables o características aportan la misma calidad de información, pudiendo incluso complicar la creación de las reglas de clasificación y hacer que clasifiquen de forma errónea. Por lo tanto, el objetivo del problema de selección de variables no es sólo clasificar las instancias sino también encontrar un subconjunto de variables con las que se pueda clasificar de forma óptima, clasificando la mayor cantidad posible de instancias de forma correcta.

La búsqueda de este subconjunto de variables es un problema NP-duro, por lo cual el uso de metaheurísticas en este problema nos permite obtener soluciones razonablemente buenas en un tiempo razonable sin llegar a explorar todo el espacio de soluciones.

Para resolver este problema, vamos a utilizar metaheurísticas de búsquedas por trayetoria: búsqueda local primero el mejor, enfriamiento simulado y búsqueda tabú. Además, haremos una comparación entre estas heurísticas y un algoritmo de comparación greedy.

2. Aplicación de los algoritmos

En este apartado explicaremos las consideraciones comunes a todos los algoritmos como el esquema de representación de soluciones y las consideraciones de algunas funciones comunes a todos los algoritmos como le función objetivo y la generación de vecinos.

Primero tenemos que definir la representación de las soluciones. En este caso, hemos utilizado una representación de un vector binario, en el que si en la posición i hay un 1 significa que la característica i es seleccionada y si hay un 0 no es seleccionada. Esta representación es lo que llamamos la máscara.

La función objetivo consiste en una llamada al clasificador 3-NN en la que hacemos el recuento de aciertos (coste) con la máscara que le pasamos. En pseudocódigo sería:

```
1 knn{
2         Para cada caracteristica
3                 Si knn(datos,mascara) = datos[caracteristica][
                      total caracteristicas-1] \\
```

```
                        coste++
        return (coste)
}
```

A la hora de evaluar un vecino, si el coste con el nuevo vecino es mayor que el coste de la mejor solución hasta el momento, la mejor solución pasa a ser el nuevo vecino y actualizamos el valor del coste.

Para la generación de vecinos, los generamos haciendo el cambio de pertenencia $Flip(s, i)$, en el que modificamos el valor de la posición i de la máscara s a su valor contrario. En el caso de la búsqueda local generamos tantos vecinos como características tengamos. Para cada vecino se aplica un movimiento $Flip(s, i)$ empezando la i en 0 para el primer vecino, 1 para el segundo y así hasta generarlos todos. La función se llama $generarVecinos$. El pseudocódigo es el siguiente:

```
generarVecinos{
        Desde 0 hasta el numero de caracteristicas:

                Hacemos el cambio Flip

                Aniadimos la mascara a la matriz de vecinos

                Deshacemos el cambio Flip

        return (vecinos)
}
```

También podemos utilizar el movimiento $Flip(s, i)$ escogiendo la posición i de forma aleatoria y haciendo el cambio de valor de esa posición. La función se llama $generarVecinoAleatorio$.

```
generarVecinoAleatorio{

        Generamos un indice aleatorio
        Cambiamos el valor de la mascara en dicho indice

        return (mascara)
}
```

La generación de la solución aleatoria inicial se crea simplementen generando valores aleatorios 0 o 1.

```
generarMascaraInicial{
        Desde 0 hasta el numero de caracteristicas
                Generamos valores aleatorios 0 o 1 para cada
                   elemento de la mascara

```

```
        return(mascara)
}
```

El mecanismo de selección considerado es un mecanismo de selección por torneo en el cual se escogen dos soluciones distintas al azar y se compara su valor dado por la funcón objetivo. El que tenga mayor valor será el seleccionado. En el caso del algoritmo AGG, se hacen 30 torneos para seleccionar una nueva población del mismo tamaño y en el algoritmo AGE sólo se hace un sólo torneo. El pseudocódigo es el siguiente:

```
torneo{

        Desde 0 hasta el numero de torneos especificado

                Elegimos dos soluciones al azar, primero y segundo

                Si coste(primero) < coste(segundo)
                        Seleccionamos la segunda solucion

                Si no
                        Seleccionamos la primera solucion

        Devolvemos las soluciones seleccionadas
}
```

El operador de cruce empleado es un operador de cruce de dos puntos, en el que seleccionamos dos puntos diferentes al azar, es decir, dos posiciones e intercambiamos el contenido comprendido entre esos dos puntos de dos soluciones. El criterio para que se crucen dos soluciones será distinto para cada algoritmo. El pseudocódigo es el siguiente:

```
cruce{

        Desde 0 hasta el numero total de cruces

                Seleccionamos dos puntos al azar, p1 y p2

                Intercambiamos el contenido comprendido entre p1 y
                    p2 de dos soluciones

        Devolvemos las soluciones tras el cruce
}
```

Por último, el operador de mutación será el mismo que empleamos en la práctica 2:

```
mutacion{

        Desde 0 hasta numero de caracteristica a mutar
```

```
                    generarVecinoAleatorio(mascara)

        Devolvemos la mascara
}
```

2.1. Algoritmo AGG

El algoritmo AGG es un algoritmo genético con un esquema generacional con elitismo. Este algoritmo genera una población nueva del mismo tamaño que la anterior en cada iteración pero sustituye el peor individuo de la población nueva por el mejor de la población anterior. El esquema del algoritmo es:

- Generamos una población de forma aleatoria y evaluamos el coste de cada individuo
- A continuación entramos el proceso generacional, en el que mientras no se alcance el número tope de evaluaciones, seleccionamos una nueva población con el operador de selección descrito en el apartado anterior.
- Una vez tenemos la nueva población, procedemos al cruce de los individuos de ésta. En este caso, la selección de las parejas se hace en orden, es decir, se cruza el primero con el segundo, el tercero con el cuarto y así sucesivamente.
- Tras realizar los cruces en la población nueva, mutamos algunos de los individuos escogidos de forma aleatoria.
- Por último, pasamos al esquema de reemplazamiento. En este caso, la nueva población generada a partir del operador de selección y a la que le hemos realizados los cruces y mutaciones necesarios, sustituye a la antigua población e intercambiamos el peor individuo de la nueva población por el mejor de la población anterior.
- Por último, devolvemos la mejor solución obtenida tras realizar 15000 evaluaciones

El pseudocódigo del algoritmo es el siguiente:

```
agg{

        Generamos una poblacion de 30 individuos de forma
           aleatoria
        Evaluamos el coste de cada individuo
        num_evaluaciones = 0

        Mientras num_evaluaciones < 15000

                Seleccionamos una nueva poblacion

                Realizamos los cruces en la nueva poblacion

```

```
                    Realizamos las mutaciones en la nueva poblacion

                    Intercambiamos el peor elemento de la nueva
                       poblacion por el mejor de la antigua poblacion

                    Sustituimos la antigua poblacion por la nueva

            Devolvemos la mejor solucion

}
```

2.2. Algoritmo AGE

El algoritmo AGE es un algoritmo genético con un esquema estacionario. En este algoritmo únicamente se seleccionan dos padres, los cuáles se cruzan y se mutan y se sustituye el peor individuo de la población anterior por el mejor de los padres generados. El esquema del algoritmo es:

- Generamos una población de forma aleatoria y evaluamos el coste de cada individuo
- A continuación entramos el proceso generacional, en el que mientras no se alcance el número tope de evaluaciones, seleccionamos dos individuos utilizando el operador de selección.
- Procedemos al cruce de los dos individuos seleccionados.
- Tras realizar los cruces, mutamos o no los individuos dependiendo de la probabilidad de mutación.
- Por último, pasamos al esquema de reemplazamiento. En este caso, se sustituye el mejor de los individuos generados por el peor individuo de la generación anterior.
- Por último, devolvemos la mejor solución obtenida tras realizar 15000 evaluaciones.

El pseudocódigo del algoritmo es el siguiente:

```
agg{

        Generamos una poblacion de 30 individuos de forma
           aleatoria
        Evaluamos el coste de cada individuo
        num_evaluaciones = 0

        Mientras num_evaluaciones < 15000

                Seleccionamos dos individuos

                Cruzamos los dos individuos
```

```
12
13                  Realizamos las mutaciones en los dos individuos
14
15                  Intercambiamos el peor elemento de la antigua
                        poblacion por el mejor de los individuos
                        generados
16
17          Devolvemos la mejor solucion
18
19  }
```

3. Algoritmo de comparación: SFS

El algoritmo de comparación es un algoritmo greedy SFS (Sequential Forward Selection). Este algoritmo parte de una máscara con todos los valores a 0 y calcula el coste de cada característica al ser añadida a la solución actual de forma individual, es decir, genera todas las posibles soluciones con cada característica. La característica que añadimos a la máscara es la que proporcione mayor ganancia. Si en algún momento no se encuentra ninguna característica que mejore la solución actual, el algoritmo finaliza. El pseudocódigo sería el siguiente:

```
1   sfs{
2
3           Mientras fin = false
4
5                   Calculamos efectividad
6                           Para cada mascara en la que cambiemos una
                                caracteristica calculamos su coste
7
8                   Si hay ganancia, actualizamos la solucion
9
10                  Si no hay ganancia
11                          fin = true
12
13          return(mejor-solucion)
14  }
```

4. Procedimiento del desarrollo de la práctica

Para el desarrollo de la práctica desarrolle el código desde 0 a partir de lo explicado en clase y en las diapositivas,

Para compilar el código de la práctica se necesitan el archivo practica3.cpp random_ppio.c libarff.a libgtest.a:

g++ -o p practica3.c random_ppio.c libarff.a libgtest.a

En la carpeta FUENTES se incluye un script para compilar el código y crear el ejecutable en la carpeta BIN.

Al ejecutar, nos muestra un menú de opciones:

- Opción 1: ejecutar AGG
- Opción 2: ejecutar AGE
- Opción 3: salir

Y a continuación nos pide el número de simulación, es decir, el archivo con los datos que le vamos a pasar. Las opciones van desde 1 a 30. De cada conjunto de datos hay 10 archivos distintos:

- Opciones 1-10: archivos de datos de Movement_Libras
- Opciones 11-20: archivos de datos de Arrhytmia
- Opciones 21-30: archivos de datos Wdbc

5. Experimentos y análisis de los resultados

5.1. Casos del problema

Para los datos utilizados para probar los algoritmos, creé diez archivos de cada conjunto de datos mezclándolos para tener las 10 simulaciones.

El tamaño de las poblaciones es de 30 individuos y el criterio de parada de los algoritmos es de 15000 evaluaciones. La probabilidad de cruce para el algoritmo AGG es de 0,7 y 1 en el caso del AGE. La probabilidad de mutación para ambos casos es 0,001. El criterio de parada para ambos algoritmos es de 15000 evaluaciones.

El valor de la semilla que he utilizado es 5 para todos los algoritmos.

5.2. Resultados

Los resultados obtenidos se muestran en las siguientes tablas:

Tabla 5.2.1: Resultados obtenidos por el algoritmo SFS en el problema de la SC

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	94,39	83,33	45,93	81,67	97,78	59,75	84,46	97,84	1639,31
Partición 1-2	97,54	86,67	38,79	85,56	94,44	118,25	81,87	96,76	2344,13
Partición 2-1	96,49	90,00	31,22	86,67	95,56	102,38	80,83	97,84	1603,89
Partición 2-2	98,95	86,67	38,69	90,00	93,33	150,93	81,87	97,48	1824,78
Partición 3-1	96,14	86,67	38,67	77,78	96,67	89,21	85,49	97,48	1814,64
Partición 3-2	95,44	86,67	38,48	90,56	95,56	112,43	84,46	97,84	1579,54
Partición 4-1	96,49	83,33	45,79	91,67	95,56	112,47	84,97	97,48	1794,46
Partición 4-2	97,45	83,33	46,46	79,44	97,78	68,92	82,38	96,76	2245,55
Partición 5-1	98,60	86,67	38,55	85,00	96,67	92,94	78,76	97,48	1800,08
Partición 5-2	97,54	86,67	38,66	85,00	97,78	71,42	86,01	96,40	2445,29
Media	96,90	86,00	40,12	85,34	96,11	97,87	83,11	97,34	1909,17

Tabla 5.2.2: Resultados obtenidos por el algoritmo AGG en el problema de la SC

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	81,75	36,67	20,21	90,56	47,78	25,71	83,42	47,48	81,65
Partición 1-2	95,09	63,33	20,59	93,89	52,22	25,32	79,79	51,80	82,40
Partición 2-1	86,32	63,33	20,34	91,11	43,33	25,34	78,24	45,68	81,33
Partición 2-2	96,84	53,33	20,32	86,67	46,67	25,31	80,83	50,00	85,97
Partición 3-1	93,33	53,33	20,32	91,11	46,67	25,37	78,24	49,64	83,07
Partición 3-2	87,37	36,67	20,45	95,00	48,89	25,58	79,79	46,78	84,24
Partición 4-1	84,91	53,33	20,69	91,67	46,67	25,42	81,87	50,72	83,73
Partición 4-2	98,25	63,33	20,69	85,00	52,22	25,24	77,20	48,20	84,86
Partición 5-1	83,16	43,33	20,88	92,22	46,67	25,25	78,24	48,56	84,42
Partición 5-2	80,00	50,00	21,53	90,00	53,33	25,25	81,35	51,08	85,57
Media	88,70	51,67	20,60	90,72	48,45	25,38	79,90	48,99	83,72

Tabla 5.2.3: Resultados obtenidos por el algoritmo AGE en el problema de la SC

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	82,11	53,33	21,04	93,33	53,33	27,40	79,27	51,44	85,82
Partición 1-2	96,84	46,67	21,02	84,44	50,00	27,58	84,46	51,44	87,33
Partición 2-1	96,49	60,00	21,63	88,33	51,11	27,05	80,31	50,72	86,24
Partición 2-2	98,25	66,67	20,94	86,67	51,11	27,49	81,87	51,80	86,22
Partición 3-1	95,09	40,00	21,19	94,44	37,78	27,15	78,76	52,52	89,52
Partición 3-2	97,89	40,00	21,47	90,56	42,22	27,99	79,79	39,57	87,15
Partición 4-1	96,48	66,67	21,26	93,33	45,56	28,05	82,90	52,88	85,71
Partición 4-2	97,19	50,00	21,19	90,00	53,33	27,38	84,46	52,87	86,86
Partición 5-1	96,84	40,00	21,15	93,33	57,78	27,85	76,68	51,44	87,66
Partición 5-2	95,09	36,66	21,30	90,00	42,22	27,66	78,76	48,56	86,38
Media	95,23	50,00	21,22	90,44	48,44	27,56	80,73	50,32	86,89

Tabla 5.2.4: Resultados globales en el problema de la SC

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
SFS	96,90	86,00	40,12	85,34	96,11	97,87	83,11	97,34	1909,17
AGG	88,70	51,67	20,60	90,72	48,45	25,38	79,90	48,99	83,72
AGE	95,23	50,00	21,22	90,44	48,44	27,56	80,73	50,32	86,89

5.3. Conclusiones

Como podemos ver en la tabla de resultados globales, los algoritmos AGG y AGE dan resultados muy similares, tanto en porcentaje de acierto, reducción y tiempo de ejecución. La única diferencia más significativa es con el conjunto de datos $Wdbc$ con los que el algoritmo AGE consigue un mejor porcentaje de acierto. De nuevo, el mayor porcentaje de reducción lo consigue el algoritmo SFS pero tarda mucho más que los algoritmo genéticos y a pesar de todo tienen un porcentaje de acierto similar.

Al igual que en las prácticas anteriores si lo que buscamos es un clasificador con un buen rendimiento (porcentaje de acierto alto) y que sea sencillo (porcentaje de reducción alto) sin tener en cuenta el tiempo necesario, el algoritmo SFS sería el más indicado. Sin embargo, los algoritmos genéticos tienen un rendimiento similar a SFS y reducen mucho el tiempo de ejecución aunque el porcentaje de reducción no sea tan alto, por lo que son clasificadores con un buen rendimiento en un tiempo bastante razonable.

Como conclusión, aunque ambos algoritmos genéticos dan resultados prácticamente iguales, considero el algoritmo AGE algo mejor que AGG por los resultados obtenidos en el conjunto $Wdbc$, donde AGE tiene un mejor porcentaje de acierto que AGG.