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

Práctica 2.b: Búsquedas Multiarranque 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

27 de abril de 2016

Índice

1. Introducción al problema de Selección de Características 3

2. Aplicación de los algoritmos 3

3. Algoritmo de comparación: SFS 9

4. Procedimiento del desarrollo de la práctica 9

5. Experimentos y análisis de los resultados 10

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 llamar al clasificador 3-NN y hacer el recuento de aciertos (coste) con la máscara que le pasamos. En pseudocódigo sería:

```
1 funcion_objetivo{
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)
}
```

En los algoritmos de enfriamiento simulado y búsqueda tabú el movimiento $Flip(s, i)$ no se hace para cada posición sino que se escoge la posición i de forma aleatoria y se hace 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)
}
```

En el caso de la búsqueda tabú tenemos que generar el vecindario, para ello generamos tantos vecinos como indiquemos y los generamos utilizando la función $generarVecinoAleatorio$. Esta función se llama $generarVecindario$ y el pseudocódigo sería el siguiente:

```
generarVecindario{
        Desde 0 hasta tamanio
```

```
3
4                   generarVecinoAleatorio()
5
6                   Guardamos el vecino en la matriz de vecinos
7
8           return(vecinos)
9   }
```

La generación de la solución aleatoria inicial de los algoritmos BMB e ILS se crea simplementen generando valores aleatorios 0 o 1.

```
1   generarMascaraInicial{
2           Desde 0 hasta el numero de caracteristicas
3                   Generamos valores aleatorios 0 o 1 para cada
                        elemento de la mascara
4
5           return(mascara)
6   }
```

El algoritmo de búsqueda local empleado es el implementado en la práctica 1. El pesudocódigo del algoritmo es el siguiente:

```
1   bl{
2
3           Generamos la mascara inicial
4
5           Mientras fin = false
6
7                   Generamos los vecinos
8
9                   Para cada vecino generado
10
11                          Calculamos el coste de cada vecino
12
13                          Si el coste del nuevo vecino es mayor
14                                  Actualizamos la solucion
15                                  Criterio de parada del bucle
                                        interno
16
17                  Si hemos generado todos los vecinos
18                          fin = true
19
20          return(mejor-solucion)
21  }
```

La máscara inicial, la generación de los vecinos y el cálculo del coste de cada uno lo hacemos utilizando las funciones descritas anteriormente.

El método de exploració del entorno consiste en los dos bucles anidados del algoritmo. Dentro del primer bucle generamos el entorno de nuestra solució actual, el cual evaluamos vecino a vecino en un segundo bucle. Si en algún momento uno de los vecinos es mejor que la solución actual, actualizamos nuestra solución y salimos del bucle interno para de nuevo generar el nuevo entorno de nuestra nueva solución. La exploración del entorno continúa hasta que no encontremos ningún vecino que mejore nuestra solución actual.

2.1. Algoritmo de búsqueda multiarranque básica

El algoritmo de búsqueda multiarranque básica (BMB) consiste en generar soluciones aleatorias, optimizarlas con el algoritmo de búsqueda local y escoger la mejor solución obtenida.

El pseudocódigo sería el siguiente:

```
bmb{

        Desde 0 hasta numero iteraciones

                Generamos mascara aleatoria

                Optimizamos con la busqueda local

                Si la mascara optimizada es mejor que nuestra
                   mejor solucion, actualizamos

        return (mejor solucion)
}
```

2.2. Algoritmo GRASP

El algoritmo GRASP es un poco más elaborado que el algoritmo BMB. Al igual que en el algoritmo BMB, consiste en generar soluciones aleatorias, optimizarlas con la búsqueda local y es coger la mejor solución. La diferencia radica en la forma de generar la solución aleatoria. En el caso de GRASP, la solución aleatoria se genera con un algoritmo greedy probabilístico. El algoritmo greedy genera el entorno de una solución inicial con todas las posiciones a 0 y evalúa cada máscara del entorno. A continuación, selecciona las mejores soluciones. El número de soluciones que selecciona es el 10 % del total de característica. De entre esas mejores soluciones, se seleccina una al azar. La solución seleccionada al azar por el algoritmo greedy será la solución inicial que optimizaremos con la búsqueda local.

El pseudocódigo del algoritmo es el siguiente:

```
1
2  grasp{
3
4          Desde 0 hasta numero iteraciones
5
6                  Generamos mascara aleatoria con el algoritmo
                       greedy
7
8                  Optimizamos con la busqueda local
9
10                 Si la mascara optimizada es mejor que nuestra
                       mejor solucion, actualizamos
11
12         return (mejor solucion)
13 }
```

El pseudocódigo de la generación de soluciones aleatorias del greedy es el siguiente:

```
1
2  generarSolucionGrasp{
3
4          Desde 0 hasta el numero de caracteristicas:
5
6                  Hacemos el cambio Flip
7
8                  Evaluamos la solucion
9
10                 Deshacemos el cambio Flip
11
12         Seleccionamos las n mejores (n = 0.1*numCaracteristicas)
13
14         Escogemos una de ellas al azar
15
16         return(solucion-aleatoria)
17
18 }
```

2.3. Algoritmo de búsqueda local reiterada (ILS)

El algoritmo ILS consiste en generar una solución inicial aleatoria, optimizarla con el algoritmo de búsqueda local y escoger la mejor solución entre la mejor solución actual y la solución optimizada. La solución inicial aleatoria se genera de la misma forma que en el algoritmo BMB. La principal diferencia con los anteriores algoritmo es que una vez escogemos la mejor solución, realizamos una mutación de esta y volvemos a hacer el

proceso de optimizarla con la búsqueda local y escoger la mejor solución. Este proceso, se repite durante el número de iteraciones que indiquemos.

El pseudocódigo sería:

```
1
2  ils{
3
4
5          Generamos solucion aleatoria inicial
6          mejor-solucion = solucion-inicial
7
8          Desde 0 hasta numero de iteraciones
9
10                 Optimizamos con la busqueda local la solucion
                       actual
11
12                 Si la solucion optimizada es mejor que la mejor
                       solucion
13
14                       Actualizamos la solucion
15                       Mutamos la solucion optimizada
16
17                 Si no
18
19                       Mutamos la mejor solucion actual
20
21         return (mejor-solucion)
22 }
```

La función de mutación de la solución consiste en cambiar el valor de un número de características de la solución que queramos mutar. En nuestro caso, mutarán el 10 % de las características. Para implementar esta función hacemos uso de la función generarVecinoAleatorio descrita al principio de la sección.

El pseudocódigo de la función de mutación sería el siguiente:

```
1
2  mutacion{
3
4          Desde 0 hasta n= 0.1*num_caracteristicas
5
6                  generarVecinoAleatorio(mascara)
7
8          return (mascara)
9  }
```

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
2  sfs{
3
4          Mientras fin = false
5
6                  Calculamos efectividad
7                          Para cada mascara en la que cambiemos una
                               caracteristica calculamos su coste
8
9                  Si hay ganancia, actualizamos la solucion
10
11                 Si no hay ganancia
12                         fin = true
13
14         return(mejor-solucion)
15 }
```

4. Procedimiento del desarrollo de la práctica

b 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 practica2.cpp random_ppio.c libarff.a libgtest.a:

g++ -o p practica2.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 SFS
- Opción 2: ejecutar BMB

- Opción 3: ejectuar GRASP
- Opción 4: ejecutar ILS
- Opción 5: 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.

Para el enfriamiento simulado ajusté el número de evaluaciones a 15000. En el caso de la búsqueda tabú, realiza 100 iteraciones por lo que en total se hacen 3000 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 BMB en el problema de la SC

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	97,54	43,33	66,81	97,22	51,11	47,16	81,35	52,87	259,47
Partición 1-2	97,89	43,33	69,87	93,89	46,67	56,07	81,87	48,56	209,05
Partición 2-1	97,89	36,67	73,01	91,67	51,11	70,11	78,76	48,92	224,73
Partición 2-2	98,60	46,67	78,41	93,89	43,33	58,42	83,94	50,72	275,41
Partición 3-1	97,19	43,33	73,06	95,56	57,78	47,46	78,76	48,92	264,68
Partición 3-2	97,89	36,67	64,38	95,00	45,56	44,31	79,27	49,64	268,71
Partición 4-1	97,19	53,33	74,74	90,56	57,78	48,25	82,38	45,68	245,75
Partición 4-2	97,19	43,33	73,72	93,33	57,78	47,59	82,38	46,76	274,15
Partición 5-1	97,19	36,67	78,58	92,22	52,22	56,44	92,90	48,20	237,95
Partición 5-2	96,14	46,67	82,21	91,11	54,44	52,33	77,72	50,72	255,39
Media	97,47	43,00	73,48	93,45	51,78	52,81	81,93	49,10	251,53

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

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	96,14	66,67	139,89	97,78	77,78	461,64	77,72	91,73	5520,07
Partición 1-2	98,25	70,00	152,99	92,78	85,56	474,31	80,31	98,56	5397,04
Partición 2-1	97,19	36,67	140,51	88,33	84,44	504,51	77,72	94,60	5426,94
Partición 2-2	97,54	50,00	176,12	91,11	77,78	491,11	94,46	91,37	5705,92
Partición 3-1	98,60	33,33	155,63	91,11	87,78	513,37	81,35	90,65	5616,67
Partición 3-2	96,84	66,67	142,33	93,33	91,11	492,50	82,90	89,93	5397,44
Partición 4-1	96,84	63,33	139,17	88,89	86,67	529,77	83,42	92,45	5446,43
Partición 4-2	97,89	43,33	149,50	90,56	90,00	528,66	82,90	91,37	5333,47
Partición 5-1	98,25	73,33	137,93	91,11	91,11	507,85	80,31	93,53	5295,59
Partición 5-2	95,44	93,33	143,81	95,56	93,33	525,45	81,87	96,04	5400,77
Media	97,30	59,67	147,79	92,06	86,56	502,92	82,30	93,02	5454,03

Tabla 5.2.4: Resultados obtenidos por el algoritmo ILS básica en el problema de la SC

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	96,49	70,00	46,01	97,22	53,33	46,58	81,35	48,20	201,86
Partición 1-2	97,89	43,33	41,80	95,56	62,22	53,40	79,79	50,72	196,84
Partición 2-1	97,54	40,00	40,53	90,00	45,56	65,86	79,79	50,00	183,69
Partición 2-2	99,30	50,00	42,51	92,22	93,38	42,53	78,77	51,44	226,81
Partición 3-1	97,19	43,33	32,78	94,44	41,11	40,25	76,17	48,92	232,07
Partición 3-2	96,14	66,67	31,84	94,44	38,89	42,33	79,27	48,56	226,73
Partición 4-1	97,54	40,00	39,89	91,11	44,44	48,46	80,83	48,56	208,41
Partición 4-2	97,85	46,67	35,75	92,78	48,89	53,94	79,79	50,72	236,61
Partición 5-1	97,89	50,00	39,21	92,78	48,89	52,18	78,76	51,80	206,76
Partición 5-2	96,14	53,33	45,78	92,22	41,11	51,03	80,31	48,20	188,73
Media	97,40	50,33	39,61	93,28	51,78	49,66	79,48	49,71	210,85

Tabla 5.2.5: Resultados globales en el problema de la SC

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
SFS	96,9033	86,0013	40,12368	85,335	96,113	97,86969	83,10882	97,33808	1909,167
BMB	97,47	43,00	73,48	93,45	51,78	52,81	81,93	49,10	251,53
GRASP	97,29825	59,666	147,7882	92,056	86,556	502,9158	82,29536	93,02158	5454,034
ILS	97,39901	50,333	39,61063	93,27678	51,78178	49,65563	79,48277	49,71228	210,8504

5.3. Conclusiones

A partir de la tabla de los datos globales podemos ver que los algoritmos BMB e ILS dan resultados bastantes similares a SFS en cuanto a tiempo y tasa de acierto, excepto en el conjunto de datos Arrhytmia en el que SFS tarda significativamente más que BMB e ILS. Aún así, los porcentajes de reducción de SFS son muchos mayores que los de BMB e ILS. Respecto a GRASP podemos ver que es el que mejor porcentaje de reducción tiene respecto a los algoritmo de búsqueda multiarranque, aunque tarda mucho más que SFS. La tasa de acierto es bastante similar en los cuatro algoritmos.

Aunque BMB e ILS tengan unos resultados muy similares podríamos decir que ILS es mejor que BMB dado que en todos los casos tarda algo menos que BMB, dando porcentajes de acierto y de reducción muy similares. Sin embargo, los porcentajes de reducción del algorimot GRASP son mucho mejores que los de BMB e ILS y obtiene resultados similares en cuanto a rendimiento. A pesar de todo esto, el agoritmo SFS sigue siendo mejor en el conjunto de datos Wdbc da porcentajes de acierto muy similares a los demás de algoritmos, porcentajes de reducción muy altos y un tiempo muy similar a ILS. En el conjunto de datos Movement_Libras, aunque SFS da mejor tasa de reducción que ILS tarda casi el doble de tiempo y tiene una menor tasa de acierto. Respecto al conjunto Arrhytmia, SFS y GRASP dan resultados similares en cuanto a tasa de acierto y tasa de reducción aunque GRASP emplea mucho más tiempo para los mismos resultados. En este caso, BMB e ILS dan buenos resultados de clasificación , y aunque tengan una menor tasa de reducción ésta se compensa al tardar considerablemente menos que SFS y GRASP.

Como conclusión podemos decir que el mejor algoritmo sería ILS teniendo en cuenta el rendimiento del clasificador (alto porcentaje de acierto) y la tasa de reducción respecto al tiempo que tarda, siendo el que menos tiempo emplea, aunque BMB da resultados muy similares tardando un poco más. El algoritmo GRASP aunque es el que mejor resultados da teniendo en cuenta la relación entre el porcentaje de acierto y el de reducción tiene tiempos demasiado elevados respecto a los demás algoritmos.