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

Práctica 1.b: Búsquedas por Trayectorias 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

3 de abril 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

                Anihadimos 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   }
```

Otro elemento común a todos los algoritmos es la generación de la solución inicial, la cual se genera de forma aleatoria para todos los algoritmos.

```
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   }
```

2.1. Algoritmo de búsqueda local primero el mejor

El algoritmo de búsqueda local primero el mejor consiste en generar en cada iteración el entorno de la solución actual hasta que obtenemos una solución mejor que la actual o generamos el entorno por completo. Si encontramos una solución mejor, actualizamos la solución actual por la que hemos encontrado y seguimos iterando. Si tras generar el entorno completo no hemos encontrado una solución mejor que la que ya tenemos el algoritmo finaliza.

El psudocódigo 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.

2.2. Algoritmo de enfriamiento simulado

El algoritmo de enfriamiento simulado nos va a permitir evitar caer en posibles óptimos locales permitiendo movernos a soluciones peores en ciertos momentos. Este algoritmo va adaptando el criterio de aceptación de soluciones haciendo uso de una variable *temperatura*. El valor de esta variable determina si una solución peor puede ser aceptada o no. Al principio el valor de esta variable es un valor alto y se va reduciendo en cada iteración mediante un mecanismo de enfriamiento de la temperatura. Este algoritmo supone que al principio aceptará soluciones que pueden ser muy malas y conforme se enfríe la temperatura la probabilidad de aceptar soluciones peores irá disminuyendo.

El funcionamiento del algoritmo sería generamos una solución inicial de forma aleatoria y calculamos los valores de las temperaturas inicial y final. A partir de esta solución inicial generamos los vecinos de forma aleatoria y los evaluamos. A continuación, le aplicamos el criterio de aceptación y comprobamos si es una mejor solución que la que tenemos hasta el momento. Si es una mejor solución, la aceptamos. Si es peor, aún tenemos la posibilidad de aceptarla como solución actual. Para determinar esto, calculamos la diferencia entre los costes de la solución actual y la vecina y la temperatura actual. Si la temperatura es muy alta o la diferencia de los costes es muy pequeña, hay mayor probabilidad de aceptar soluciones peores. Una vez hemos generado las soluciones vecinas, enfriamos la temperatura y pasamos a la siguiente iteración hasta que la temperatura sea menor que la temperatura final.

El pseudocódigo sería el siguiente:

```
1  sa{
2
3          Generamos la mascara inicial
```

```
4           Calculamos el coste de la mascara inicial
5           Inicializamos las temperaturas inicial, actual y final
6
7           Mientras t-actual > t-final
8
9                   Desde 0 hasta el numero de iteraciones que
                        indiquemos
10                      Generamos un vecino de forma aleatoria
11
12                      Calculamos el coste del vecino
13
14                      Si la solucion vecina es mejor que la
                            actual, actualizamos
15
16                      Si no, comprobamos la probabilidad de
                            aceptacion de la solucion
17
18                      Si ya hemos generado todos los vecinos
                            posibles
19                      o hemos llegado a un determinado numero de
                             exitos, terminamos la iteracion
20
21                  Enfriamos la temperatura
22
23          return(mejor-solucion)
24  }
```

El cálculo de la temperatura inicial lo hacemos mediante la fórmula $T_0 = \frac{\mu Coste\_de\_la\_solucin\_inicial}{-ln(phi)}$ donde $\mu = \phi = 0{,}5$. Por esta razón calculamos el coste de la máscara inicial, al contrario que en la búsqueda local en la que sólo la generábamos. Para el esquema de enfriamiento utilizamos el esquema de Cauchy modificado descrito en la práctica: $T_{k+1} = \frac{T_k}{1+\beta T_k}$ siendo $\beta = \frac{T_0 - T_f}{M T_0 T_f}$.

2.3. Algoritmo de búsqueda tabú básico

La búsqueda tabú al igual que las anteriores es una búsqueda por entornos pero que además hace uso de una memoria adaptativa a corto plazo (lista tabú) para restringir el entorno de búsqueda. Al igual que el enfrimiento simulado, la búsqueda tabú permite aceptar soluciones peores con el fin de evitar caer en óptimos locales. La característica distintiva es el uso de esa lista tabú que evita la exploración de soluciones del espacio de búsqueda que ya han sido visitadas. En cada iteración se genera el entorno de la solución actual y se acepta el mejor vecino del entorno. El vecino aceptado se guarda en la lista tabú, indicando que esa solución ya ha sido seleccionada por lo que en las siguientes iteraciones, esta solución quedará fuera del espacio de búsqueda. Una solución estará en la lista tabú durante un número de iteraciones que definamos previamente. Este número

de iteraciones es lo que denominamos la tenencia tabú. Pero hay que tener en cuenta que una solución que esté en la lista tabú puede ser mejor que la solución actual, por lo que introducimos un criterio de aspiración.

El pseudocódigo del algoritmo es el siguiente:

```
1
2  bt{
3
4          Generamos una mascara aleatoria inicial
5
6          Calculamos el coste de la mascara inicial
7
8          Desde 0 hasta numero-iteraciones
9
10                 Generamos el entorno
11
12                 Para cada vecino del entorno
13
14                         Calculamos el mejor vecino del entorno
15
16                 Si la solucion no esta marcada como tabu, la
                       marcamos y actualizamos la soluci\ón
17
18                 Si estaba en la lista tabu, comprobamos el
                       criterio de aspiracion
19
20         return(mejor-solucion)
21 }
```

Nuestra lista tabú, funciona como una máscara tabú. Después de calcular el mejor vecino del entorno de la solución actual, buscamos el primer índice en el que la máscara actual y la solución vecina difieren. Esta sería nuestra característica que guardaríamos en la lista tabú. De esta manera, vamos indicando las características que han pasado a ser seleccionadas o han dejado de serlo según la mejor solución. En nuestro caso, la tenencia de la lista tabú la hemos calculado como el número de características entre 30, dado que vamos a generar 30 vecinos de cada entorno.

Para gestionar la lista tabú, si la característica que difiere no estaba en la lista tabú, ponemos el contador de dicha característica al valor de la tenencia y decrementamos en 1 el número de iteraciones del resto de características de la lista tabú.

Si por otra parte, la característica ya estaba en la lista tabú, de nuevo decrementamos le número de iteraciones del resto de características y le sumamos de nuevo la tenencia a la característica concreta. No reiniciamos su valor al valor de la tenencia dado que ya ha sido escogida dos veces, por lo que la penalizamos aumentando su valor.

El pseudocódigo sería el siguiente:

```
1
2  Si la caracteristica no esta maracada como tabu
3
4          Para cada caracteristica de la lista tabu
5                  Decrementamos su valor en 1, si su valor es mayor
                       que 0
6
7          Actualizamos el valor de la caracteristica a la tenencia
8
9          Actualizamos la solucion
10
11 Si no
12
13         Si el coste de la solucion vecina es mejor que el de la
               solucion actual (Criteriio de aspiracion)
14
15                 Para cada caracteristica de la lista tabu
16                         Decrementamos su valor en 1, si su valor
                               es mayor que 0
17
18                 Actualizamos el valor de la caracteristica sum\á
                       ndole la tenencia
19
20                 Actualizamos la solucion
```

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 los aciertos de cada clase
7
8                  Si hay ganancia, actualizamos la solucion
9
```

```
                    Si no hay ganancia
                            fin = true

        return(mejor-solucion)
}
```

4. Procedimiento del desarrollo de la práctica

b Para el desarrollo de la práctica nos juntamos varios compañeros para comprender mejor el problema y desarrollamos el código desde 0 cada uno para luego ir comparando resultados y tiempos.

Para compilar el código de la práctica se necesitan el archivo practica1.cpp random_ppio.c libarff.a libgtest.a:

g++ -o p practica1.c random_ppio.c libarff.a libgtest.a

Al ejecutar, nos muestra un menú de opciones:

- Opción 1: ejecutar SFS
- Opción 2: ejecutar LS (búsqueda local)
- Opción 3: ejectuar SA (enfriamiento simulado)
- Opción 4: ejecutar TS (búsqueda tabú)
- 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 BL en el problema de la SC

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	96,49	53,33	31,68	98,33	48,89	36,10	82,90	49,64	592,95
Partición 1-2	97,54	66,67	12,32	95,00	50,00	21,77	78,24	50,72	677,47
Partición 2-1	97,54	53,33	18,55	90,56	44,44	25,01	82,90	49,64	513,25
Partición 2-2	98,60	30,00	23,38	93,33	53,33	25,91	79,79	49,28	855,60
Partición 3-1	97,19	46,67	16,09	94,44	46,67	20,85	78,24	51,80	1190,50
Partición 3-2	98,25	36,67	15,26	95,00	51,11	24,67	82,90	41,37	556,66
Partición 4-1	96,14	43,33	15,37	92,22	52,22	27,40	81,87	51,44	599,02
Partición 4-2	98,60	46,67	14,55	93,33	40,00	29,61	81,35	45,68	1314,70
Partición 5-1	97,19	60,00	11,09	93,33	44,44	21,88	78,76	48,56	818,21
Partición 5-2	96,84	50,00	23,94	92,22	53,33	52,68	73,73	53,24	498,37
Media	97,44	48,67	18,22	93,78	48,44	28,59	80,07	49,14	761,67

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

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	97,89	50,00	73,17	97,78	50,00	40,47	81,35	48,92	155,06
Partición 1-2	97,54	50,00	73,82	94,44	48,89	41,05	83,42	48,56	154,03
Partición 2-1	97,19	60,00	73,69	90,56	46,67	42,17	80,31	46,04	153,78
Partición 2-2	98,95	53,33	73,42	94,44	62,22	43,19	80,31	50,00	153,18
Partición 3-1	94,74	46,67	73,58	94,44	48,89	43,57	80,31	51,80	153,77
Partición 3-2	98,95	53,33	73,16	95,56	48,89	43,93	78,76	50,72	154,73
Partición 4-1	98,60	56,67	73,32	91,11	43,33	44,41	82,38	47,48	154,15
Partición 4-2	96,49	53,33	73,45	92,78	56,67	43,97	80,83	51,08	153,77
Partición 5-1	95,79	50,00	73,74	91,11	48,89	44,23	80,31	47,84	153,37
Partición 5-2	97,54	50,00	73,47	92,22	55,56	44,18	79,79	45,32	153,91
Media	97,37	52,33	73,48	93,44	51,00	43,12	80,78	48,78	153,97

Tabla 5.2.4: Resultados obtenidos por el algoritmo BT 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	60,00	768,38	97,78	50,00	696,75	81,35	52,16	2625,63
Partición 1-2	97,89	40,00	864,19	95,56	45,56	778,22	79,79	46,40	2664,30
Partición 2-1	97,89	50,00	790,37	91,67	52,22	795,47	78,24	47,12	2615,37
Partición 2-2	99,30	40,00	783,85	91,11	48,89	801,45	80,83	46,76	2557,72
Partición 3-1	98,60	50,00	873,82	92,78	57,78	823,15	79,79	48,92	2508,79
Partición 3-2	98,25	36,67	902,41	93,89	50,00	864,52	81,87	50,00	2706,89
Partición 4-1	96,84	46,67	899,42	91,11	50,00	832,78	83,94	50,00	2593,14
Partición 4-2	98,60	56,67	848,55	93,89	45,56	836,95	81,35	48,92	2559,18
Partición 5-1	94,14	50,00	801,66	91,11	50,00	808,87	79,79	53,60	2600,62
Partición 5-2	94,84	40,00	897,03	92,78	50,00	825,55	77,20	47,84	2726,53
Media	97,28	47,00	842,97	93,17	50,00	806,37	80,41	49,17	2615,82

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
BL	97,44	48,67	18,22	93,78	48,44	28,59	80,07	49,14	761,67
ES	97,36843	52,333	73,48283	93,444	51,001	43,11609	80,77712	48,77697	153,974
BT básica	97,2842	47,001	842,9666	93,168	50,001	806,3705	80,4145	49,17267	2615,817

5.3. Conclusiones

Si nos fijamos en la tabla de los resultados globales podemos ver como los porcentajes de reducción del algoritmo SFS son mucho mayores que los del resto, por lo que para

la máscara coge un número muy pequeño de características. Esto puede deberse a que explora todo el espacio de búsqueda. Respecto al porcentaje de acierto de clasificación todos los algoritmos son bastantes similares, excepto SFS en el conjunto de datos Movement_Libras donde podemos ver que es peor que los demás. Por último, los tiempos de ejecución si varían bastante de un algoritmo a otro, destacando la búsqueda tabú por tener tiempos significativamente más altos que los demás.

Teniendo en cuenta estos datos, podríamos decir que la búsqueda local es el mejor algoritmo para los ejemplos Wdbc y Movement_Libras dado que es el que menos tiempo tarda y da un porcentaje de acierto bastante alto; pero si lo que queremos es un clasificador más sencillo aunque tarde un poco más, el mejor sería el SFS. Sin embargo, para el conjunto de datos de Arrhythmia el enfriamiento simulado es muy superior a los demás dado que tarda mucho menos que el resto. La búsqueda tabú, a pesar de ser un algoritmo potente, no destaca especialmente ni en el porcentaje de acierto ni en el de reducción probablemente porque tenga un sobreajuste con los datos de entrenamiento.

En general, si lo que buscamos es un clasificador con un rendimiento alto (alto porcentaje de acierto de clasificaciÃn) y que sea lo más simple posible el mejor algoritmo sería SFS. Si sólo nos interesa que tenga un rendimiento alto pero que emplee poco tiempo, para problemas con una cantidad de datos menor el mejor algoritmo es la búsqueda local. Si el número de características es muy grande, el enfriamiento simulado es el que mejor resultados da teniendo en cuenta el tiempo de ejecución.