Metaheurísticas (2015-16)
Grado en Ingeniería Informática
Universidad de Granada

PRÁCTICA 1: BÚSQUEDA POR TRAYECTORIAS PARA EL PROBLEMA DE LA SELECCIÓN DE CARACTERÍSTICAS

Miguel López Campos
54120359W
miguelberja@correo.ugr.es
Grupo Viernes 18:30

7 de abril de 2016

Índice

1. **Descripción del problema** 3
2. **Descripción de los aspectos comunes de los algoritmos** 4
3. **Algoritmo de búsqueda local** 6
4. **Algoritmo de enfriamiento simulado** 8
5. **Búsqueda tabú básica** 10
6. **Descripción algoritmo de comparación** 11
7. **Aspectos técnicos de la práctica** 12
8. **Experimentos y análisis** 12

Índice de figuras

1. Descripción del problema

El problema que estamos abordando es la selección de características. Este problema es muy útil en el campo de "machine learning".

Tenemos un conjunto de datos de entrenamiento y otro de validación, ambos etiquetados o clasificados. Lo que queremos hacer es 'aprender' una función que a partir de las características del conjunto de datos de entrenamiento, nos permita estimar el etiquetado de otros vectores de características. Lo que nosotros queremos hacer es eliminar las características que no son relevantes en el problema, eliminando de esta manera ruido en el conjunto de datos y mejorando la eficiencia de nuestro clasificador. Es decir, no sólo mejoraremos el tiempo, si no muy probablemente la calidad de nuestras soluciones también (en cuanto al error se refiere).

La gran dificultad de este problema radica en el gran número de soluciones posibles, llevándonos al punto de que un algoritmo Greedy que nos garantice la solución óptima podría llevarnos días de ejecución para determinados problemas. Es por esto por lo que tenemos que usar Metaheurísticas. Necesitamos soluciones buenas (aunque no sea la mejor) en un tiempo menor.

Nosotros usaremos para clasificar el algoritmo 3NN. Lo que hace este algoritmo es calcular la distancia euclídea entre el vector de características al cual queremos estimar una clase y el resto de vectores de características del conjunto de entrenamiento. Lo que hace el 3NN es coger los 3 elementos menos distantes y la clase mayoritaria entre esos 3 será la estimación que haremos.

Validaremos con la técnica 5x2 Cross Validation. Usaremos 5 particiones de los datos distintas al 50 % (y aleatorias) y aprenderemos el clasificador con una submuestra y validaremos con la otra y después al contrario. Con esta técnica tendremos el porcentaje de acierto, que nos servirá para ver la calidad de nuestro algoritmo.

Otros datos con los que valoraremos la calidad de nuestros algoritmos serán los tiempos de ejeución y los porcentajes de reducción, es decir, el porcentaje de características que hemos reducido.

Con nuestras metaheurísticas querremos optimizar la función de acierto. Es decir, queremos maximizar el acierto, siendo la función:
$tasaclass = 100 * \frac{n^{o}instanciasbienclasificadas}{n^{o}instanciasTotal}$

2. Descripción de los aspectos comunes de los algoritmos

La práctica ha sido desarrollada en C++.

1. Representación de las soluciones. Para representar las soluciones utilizaremos un array de booleanos. Será común a todos los algoritmos. Si la componente i es true, esto indicará que la característica i se tendrá en cuenta (no ha sido eliminada).

2. Función objetivo. La función que queremos optimizar se trata del porcentaje de acierto de estimaciones de clases, descrita en el apartado anterior.

   En pseudocódigo es la siguiente:

```
funcion_objetivo(conjunto_training, conjunto_test,
    caracteristicas_activas)
begin
  Para todo elemento i del conjunto_test
  begin
    elemento <- elemento i del conjunto_test
          clase <- 3NN(conjunto_training, elemento,
              caracteristicas_activas)

          Si la clase estimada por 3NN se corresponde a la clase
              real -> aciertos++
  end

  promedio <- aciertos/tamanio conjunto_test

  devolver promedio
end
```

3. Función clasificadora. Como función clasificadora usaremos el algoritmo 3NN, descrito anteriormente.

   El pseudocódigo es el siguiente:

```
3NN(conjunto_training, vector_caracteristicas,
    caracteristicas_activas)
begin
  Para cada vector i de caracteristicas de training
  begin

    array_distancias.aniadir(distanciaeuclidea(i,
        vector_caracteristicas, caracteristicas_activas))

   end
```

```
9
10    minimo1 <- minimo(array_distancias)
11    minimo2 <- minimo(array_distancias-minimo1)
12    minimo3 <- minimo(array_distancias-minimo1-minimo2)
13
14    Si la clase de vector_caracteristicas[minimo2]==clase de
          vector_caracteristicas[minimo3] entonces
15      La clase del vector de caracteristicas es esa
16    Si no
17      La clase del vector de caracteristicas es la clase de
            vector_caracteristicas[minimo1]
18
19    devolver clase del vector de caracteristicas
20
21  end
```

4. Antes de trabajar con cualquier algoritmo hay que normalizar los conjuntos de datos.

5. Todos los algoritmos (menos SFS) tendrán como criterio de parada que se hayan explorado como mucho 15000 soluciones distintas. En la búsqueda tabú he reducido a 250 este número por el mucho tiempo que tarda. En la búsqueda local se añade como condición que cuando no se mejore la solución, pare. En SFS la condición es mientras la solución mejore.

6. El operador de generación de vecinos en búsqueda local, búsqueda tabú y enfriamiento simulado se trata de la inversión de una componente aleatoria de una solución. Es decir, $flip(s, i)$ cambia la componente i del vector solución s a true si era false y a false si era true. i es aleatorio.

7. La solución inicial en SFS será el vector solución puesto entero a false. En cambio en los demás algoritmos se generará una solución aleatoria.

8. Para cada algoritmo he plantado el mismo valor de semilla para una correspondiente iteración.

9. He usado para tomar tiempos y para crear números aleatorios las funciones dadas en decsai.

3. Algoritmo de búsqueda local

La descripción en pseudocódigo del algoritmo es la siguiente:

```
busqueda_local(training, test)
begin
  Solucion <- Generar_solucion_aleatoria
  coste_solucion <- funcion_objetivo(training, test, Solucion)

  Mientras que se encuentre una solucion mejor y no se superen 15000
     soluciones exploradas
  begin
    S <- Solucion

    Mientras que no se mejore y mientras que no se haya generado todo
       el entorno de S
    begin
      S <- flip(S, repetidos) //Con repetidos evitamos repetir dos
                              //soluciones de un mismo entorno

      coste_S <- funcion_objetivo(training, test, S)

      Si coste_\S es mejor que coste_solucion entonces
        //S mejora a Solucion
        Solucion <- S
        coste_solucion <- coste_S

     end

   end

   devolver Solucion y coste_solucion

 end
```

La exploración del entorno la realizo cambiando aleatoriamente una componente del vector solución (con la función flip). Para asegurarme de que esta solución nueva del entorno no se repite dentro de este entorno, tengo un vector de booleanos (repetidos) que la función flip se encarga de comprobar. Si es true para el índice generado aleatoriamente, vuelvo a generar otro número aleatorio. Así hasta que llegue a una solución no explorada del entorno. Cuando una solución del entorno es mejor que la solución cuyo entorno es el que estamos explorando, pasaremos a explorar ahora el entorno de esta nueva solución. Describiéndolo en pseudocódigo obtendríamos lo siguiente:

La función flip:

```
Flip(Solucion, Repetidos)
begin

  Mientras no sea valido
    a <- Aleatorio entre 0 y (tamanio de Solucion)-1
    Si Repetidos[a] es falso entonces
      Cambio Solucion[a] a falso si es verdadero o a verdadero si es
         falso
      pongo Repetidos[a] a verdadero
          pongo valido a verdadero
  end

  devolver Solucion
end
```

Para la exploración del entorno en general realizo lo siguiente:

```
Mientras que no genere todos los vecinos y no se mejore la solucion
  S <- Mejor Solucion
  S <- flip(S, repetidos)

  Si la componente cambiada por flip no es la que devolveria a S a
     ser la solucion anteriormente explorada entonces
    Coste_S <- funcion_objetivo(training, test, S)

    Si Coste_\S es mejor que el coste de la mejor solucion
    entonces
      actualizo la mejor solucion
      paso a explorar entorno de la nueva mejor solucion


  Si el contador de soluciones es igual al tamanio de S-1
  entonces
  He explorado todo el entorno de S
end
```

4. Algoritmo de enfriamiento simulado

La descripción del algoritmo en pseudocódigo es la siguiente:

```
enfriamiento_simulado(training, test)
begin
  Solucion <- Generar_solucion_aleatoria
  coste_solucion <- funcion_objetivo(training, test, Solucion)
  S <- Solucion
  coste_S <- coste_solucion

  T0 <- Inicializacion T0
  T <- T0
  TF <- 0.003
  max_vecinos <- 2*tamanio de solucion
  max_exitos <- 0.1*max_vecinos

  Mientras haya exitos o mientras que no se exploren mas de 15000
     soluciones
  begin

    vecinos generados <- 0
    contador exitos <- 0

    Mientras vecinos generados < max_vecinos y contador exitos <
       max_exitos
    begin
      s' <- S
      s' <- flip(s')
      s'_coste <- funcion_objetivo(training, test, s')

      incremento vecinos generados
      incremento soluciones totales

      incremento_coste <- coste_s' - coste_S

      Si incremento_coste > 0 o Unif(0,1) <= exp(-incremento_coste/T)
      entonces
        S <- S'
        coste_S <- S'_coste

        Si S'_coste > coste_solucion
        entonces
          Solucion <- S'
          coste_solucion <- S'_coste
          incremento exitos
    end

    Actualizo T //Enfriamiento
```

```
44    end
45
46    devolver(Solucion, coste_solucion)
47  end
```

Unif(0,1) es un número aleatorio entre 0 y 1. max vecinos probé a iniciarlo con 10*tamañosolucion, pero los tiempos eran extremadamente malos y lentos. Por lo tanto lo puse como 2*tamañosolucion.

La temperatura para inicializarla sigo el procedimiento dado por el guión de prácticas, donde $T_0 = \frac{\mu C(S_0)}{-ln(\phi)}$ donde $\mu = \phi = 0{,}3$. Para el enfriamiento, he seguido el esquema también dado en el guión donde:

$$T_{k+1} = \frac{T_k}{1 + \beta T_k}$$

siendo $\beta = \frac{T_0 - T_f}{M * T_0 * T_f}$, donde $M = 15000/maxvecinos$

Para el enfriamiento simulado he realizado una modificación sobre la función flip. Ahora en lugar de meter un vector de 'repetidos' meto el último índice que ha sido modificado, para no volver atrás sobre nuestros pasos.

5. Búsqueda tabú básica

El algoritmo en pseudocódigo es el siguiente:

```
busqueda_tabu(training, test)
begin
  Solucion <- Generar_solucion_aleatoria
  coste_solucion <- funcion_objetivo(training, test, Solucion)

  lista_Tabu <- array de enteros

  S <- Solucion
  coste_S <- coste_Solucion

  Mientras no hayamos superado el limite de soluciones generadas
  begin
    coste_mejor_vecino <- 0
    mejor_vecino <- vacio

    Para i desde 1 hasta 30
    begin
    vecino <- S
    vecino <- flip(vecino)
    coste_vecino <- funcion_objetivo(training, test, vecino)

    Si el movimiento con el que hemos generado el vecino es valido
    entonces
      si coste_vecino mejor que coste_mejor_vecino
      entonces
        mejor_vecino <- vecino
        coste_mejor_vecino <- coste_vecino

      Si la lista tabu esta llena
      entonces
        extraigo el movimiento mas antiguo e introduzco el nuevo
      si no
      entonces
        introduzco el nuevo movimiento


      Si coste_vecino es mejor que coste_solucion
        Solucion <- vecino
        coste_solucion <- coste_vecino

    Si el movimiento no es valido
    entonces
      Si coste_vecino es mejor que coste_solucion
        Solucion <- vecino
        coste_solucion <- coste_vecino
```

```
46          mejor_vecino <- vecino
47          coste_mejor_vecino <- coste_vecino
48
49      end
50
51      S <- mejor_vecino
52      coste_S <- coste_mejor_vecino
53
54    end
55
56    devolver Solucion y coste_solucion
57  end
```

En el número de soluciones máximo generadas probé con 15000 pero los tiempos eran muy lentos. Después fui probando y puse 250, que nos da unos tiempos razonables así como unas soluciones medio razonables.

La lista tabú se trata de un vector de enteros en los que guardo el índice de la componente del vector solución que hemos modificado con flip. El tamaño será de n/3 (tamaño del vector solución entre 3). Para manejarla realizo lo siguiente:

```
1  movimiento <- flip(S) // S es modificada por referencia y flip
                         //devuelve un entero
2
3  Para i desde 0 hasta (tamanio de lista tabu)-1
4  begin
5    Si lista_tabu[i] es igual que movimiento
6    entonces es no valido y salgo del bucle
7  end
8
9  Si es valido
10 entonces
11   .... //Acciones de actualizacion de vecino, etc.
12   Si la lista esta llena
13   entonces
14     Elimino el primer elemento
15     Introduzco al final el nuevo movimiento
16   si no
17   entonces
18     Introduzco al final el nuevo movimiento
```

6. Descripción algoritmo de comparación

El algoritmo de comparación SFS es muy simple. Primero se genera una solución con todo a falso. A partir de aquí, exploramos todo el vector solución y cogemos la característica con la que vayamos a obtener mayor ganancia. Una vez la escojamos, volvemos a realizar otra iteración

cogiendo la siguiente característica que nos de más ganancia y así sucesivamente. El algoritmo acaba cuando ya no haya mejora en una búsqueda completa sobre el vector solución.

7. Aspectos técnicos de la práctica

La práctica ha sido desarrollada en C++. El código ha sido implementado basándome en los pseudocódigos de las transparencias de clase (adaptándolos al problema). Cada uno de los algoritmos está implementado en un cpp diferente. Dentro de estos cpp tenemos las funciones de evaluación, así como de lectura de los ficheros de datos. Cada algoritmo por lo tanto se evaluará en un ejecutable distinto. Para compilar el código simplemente hay que usar make (hay un makefile implementado) y en la carpeta bin se crearán los ejecutables. Cada ejecutable tendrá como salida los datos de las ejecuciones de los algoritmos.

Los ficheros de datos que he usado son los que hay subidos en la plataforma de la asignatura, a excepción de movement_libras, cuyo fichero de datos he tenido que descargarlo de la web dada en las transparencias ya que para leer el que había en la plataforma tuve problemas (pero el contenido de los datos es el mismo).

Para la toma de tiempos he usado las funciones dadas en decsai. También para generar números aleatorios he usado las funciones dadas por los profesores.

8. Experimentos y análisis

Datos de 3NN.

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	85,96	x	0,12	34,44	x	0,13	66,84	x	0,43
Partición 1-2	83,45	x	0,13	77,22	x	0,13	67,88	x	0,43
Partición 2-1	87,72	x	0,13	35,00	x	0,13	68,91	x	0,44
Partición 2-2	78,17	x	0,12	73,33	x	0,13	68,39	x	0,44
Partición 3-1	84,91	x	0,13	35,56	x	0,13	65,29	x	0,43
Partición 3-2	79,58	x	0,13	73,89	x	0,14	70,98	x	0,45
Partición 4-1	83,86	x	0,13	40,00	x	0,13	68,39	x	0,44
Partición 4-2	82,04	x	0,12	77,22	x	0,13	68,91	x	0,44
Partición 5-1	85,97	x	0,12	37,78	x	0,13	69,43	x	0,44
Partición 5-2	81,34	x	0,12	77,22	x	0,13	72,02	x	0,43
Media	83,30	#DIV/0!	0,13	56,17	#DIV/0!	0,13	68,70	#DIV/0!	0,44

Estos son los datos para el algoritmo de comparación SFS.

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	89,12	93,33	8,90	36,11	97,78	29,52	81,87	97,12	805,07
Partición 1-2	88,03	90,00	11,49	83,89	88,89	95,33	80,83	97,12	820,05
Partición 2-1	87,02	93,33	8,80	36,67	96,67	35,37	79,79	97,12	816,49
Partición 2-2	87,68	90,00	11,55	76,67	85,56	120,95	82,90	96,76	912,17
Partición 3-1	90,52	83,33	17,17	37,78	95,56	44,00	77,70	97,12	812,85
Partición 3-2	86,97	96,67	5,94	82,22	88,89	96,65	75,13	97,48	728,18
Partición 4-1	87,72	93,33	8,81	39,44	97,78	26,42	79,79	96,40	972,91
Partición 4-2	87,32	93,33	8,77	83,89	91,11	77,82	79,27	98,56	428,34
Partición 5-1	88,07	93,33	8,79	40,00	94,44	52,42	74,61	97,84	669,26
Partición 5-2	88,73	93,33	8,70	86,11	90,00	88,77	78,76	97,12	864,51
Media	88,12	92,00	9,89	60,28	92,67	66,73	79,07	97,26	782,98

Datos de la búsqueda local.

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	91,93	86,67	19,23	36,67	71,11	10,25	72,54	73,38	242,65
Partición 1-2	86,97	13,33	6,99	83,89	37,78	16,82	70,98	46,76	224,12
Partición 2-1	88,42	63,33	5,11	35,00	33,33	13,13	73,58	74,82	325,86
Partición 2-2	89,79	33,33	11,50	78,33	60,00	17,55	73,58	95,68	171,53
Partición 3-1	90,53	33,33	13,39	36,11	8,89	15,76	73,58	78,78	306,06
Partición 3-2	85,56	73,33	5,07	80,56	68,89	19,77	77,20	8,63	424,24
Partición 4-1	88,07	30,00	13,00	40,00	61,11	14,67	78,24	63,67	332,83
Partición 4-2	85,56	43,33	15,20	83,33	75,56	26,63	72,54	3,59	251,78
Partición 5-1	88,42	23,33	7,07	39,44	78,89	10,37	72,02	17,63	293,17
Partición 5-2	88,03	83,33	5,76	83,33	25,56	25,50	46,27	8,63	309,92
Media	88,33	48,33	10,23	59,67	52,11	17,05	71,05	47,16	288,22

Datos para el enfriamiento simulado.

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	89,47	33,33	28,19	36,67	70,00	24,80	78,24	47,84	364,13
Partición 1-2	86,62	43,33	20,71	77,78	15,56	22,76	71,50	49,28	355,85
Partición 2-1	88,42	50,00	13,92	36,11	40,00	22,51	72,02	64,03	235,37
Partición 2-2	88,38	60,00	31,70	74,44	52,22	28,44	72,02	48,20	358,58
Partición 3-1	89,12	56,67	21,91	36,67	48,89	23,12	68,39	65,11	232,61
Partición 3-2	83,80	66,67	14,53	80,56	51,11	34,50	75,65	67,99	235,54
Partición 4-1	86,32	50,00	12,49	40,00	55,56	33,64	74,09	44,60	356,73
Partición 4-2	83,80	43,33	14,62	80,00	48,89	27,52	74,09	12,65	240,68
Partición 5-1	88,07	20,00	14,78	39,44	70,00	21,51	74,09	46,40	491,29
Partición 5-2	87,32	50,00	29,36	80,56	52,22	54,76	75,65	53,96	481,90
Media	87,13	47,33	20,22	58,22	50,45	29,36	73,57	50,01	335,27

Datos para la búsqueda tabú

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	89,82	76,67	26,70	36,67	71,11	26,99	69,95	74,46	85,25
Partición 1-2	85,56	56,67	28,38	82,78	35,56	30,07	69,95	2,52	107,47
Partición 2-1	88,07	63,33	28,37	35,00	32,22	33,98	70,98	74,82	84,63
Partición 2-2	89,44	53,33	29,11	77,22	60,00	27,83	74,61	39,93	95,73
Partición 3-1	89,12	43,33	30,34	36,67	11,11	31,53	70,47	78,42	83,62
Partición 3-2	86,62	76,67	27,62	79,44	43,33	28,88	72,54	31,30	97,97
Partición 4-1	88,07	30,00	30,43	40,56	60,00	29,82	73,58	59,35	89,74
Partición 4-2	84,56	63,33	27,91	80,56	34,44	29,69	75,65	58,63	89,80
Partición 5-1	88,42	26,67	30,99	40,56	78,89	25,70	74,09	16,55	102,42
Partición 5-2	88,03	60,00	29,07	76,11	70,00	26,72	76,17	42,44	94,38
Media	87,77	55,00	28,89	58,56	49,67	29,12	72,80	47,84	93,10

Datos de todos los algoritmos

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
3-NN	83,3	0	0,13	56,17	0	0,13	68,7	0	0,44
SFS	88,12	92	9,89	60,28	92,67	66,73	79,07	97,26	782,98
BL	88,33	48,33	10,23	59,67	52,11	17,05	71,05	47,16	288,22
ES	87,13	47,33	20,22	58,22	50,45	29,36	73,57	50,01	335,27
BT básica	87,77	55	28,89	58,56	49,67	29,12	72,8	47,84	93,1

Como vemos, en general SFS y BL tienen los mejores resultados en cuanto a tasa de clasificación y SFS sobre todo en tasa de reducción. SFS, como también es lógico, tiene unos tiempos mucho mayores sobre todo para los conjuntos de datos de Movement_libras y Arrhythmia (que son los de mayor tamaño). Esto se debe al gran número de soluciones que tiene que evaluar de forma exhaustiva.

Todos los algoritmos implementados mejoran la solución dada por el 3NN para las mismas submuestras. Esto confirma el hecho de que reduciendo el número de características también mejoramos la calidad de la predicción.

La búsqueda local es ligeramente mejor que enfriamiento simulado y la búsqueda tabú en los conjuntos de datos de menor espacio de búsqueda. En cambio, en arrhythmia si se nota una leve mejoría de estos algoritmos respecto al de búsqueda local. Esto puede deberse a que la búsqueda local es buena para espacios de búsqueda menores, en cambio cuando el espacio de búsqueda es mucho mayor (como en el caso de la base de datos de Arrhythmia donde teníamos 279 características por vector de características), búsqueda local pierde algo de calidad, ya que enfriamiento simulado y búsqueda tabú permiten explorar nuevos entornos (más variedad de entornos que en la búsqueda local) evitando caer en óptimos locales, a consta de explorar soluciones peores.

En búsqueda tabú probablemente podría haber tenido mejores resultados permitiendo un mayor número límite de soluciones exploradas, pero los tiempos me salían muy altos. También en enfriamiento simulado podría haber hecho un enfriamiento más lento, dando así posibilidad a estudiar nuevos entornos.

Otro aspecto a comentar es el hecho de que para movement libras los resultados de clasificación sean tan malos. En mi opinión puede deberse a que movement libras cuenta con un número alto de clases distintas y al crear aleatoriamente las submuestras puede haber desequilibrios entre un subconjunto y otro.