

Práctica 1.b: Búsquedas por Trayectorias para el  
Problema de la Selección de Características  
SFS  
Búsqueda Local de Primero el Mejor  
Enfriamiento Simulado  
Búsqueda Tabú

Gustavo Rivas Gervilla DNI: 75570417F  
gustavofox92@correo.ugr.es  
5º Doble Grado en Ing. Informática y Matemáticas  
Grupo de los Viernes de 17:30 a 19:30

# Índice

<b>1</b>	<b>Descripción/formulación del problema abordado</b>	<b>3</b>
<b>2</b>	<b>Descripción de la aplicación de los algoritmos empleados al problema</b>	<b>4</b>
<b>3</b>	<b>Descripción en pseudocódigo de los algoritmos</b>	<b>5</b>
3.1	Búsqueda Local (BL.py) . . . . .	5
3.2	Enfriamiento simulado (ES.py) . . . . .	6
3.3	Búsqueda Tabú (BT.py) . . . . .	8
3.4	Búsqueda Tabú Extendida (BText.py) . . . . .	9
<b>4</b>	<b>Breve descripción del algoritmo de comparación</b>	<b>10</b>
<b>5</b>	<b>Procedimiento considerado para desarrollar la práctica</b>	<b>11</b>

# 1 Descripción/formulación del problema abordado

Lo que intentamos hacer con nuestro algoritmos es encontrar un conjunto de características de unos datos que nos permitan realizar una clasificación suficientemente buena de nuevos datos que nos lleguen con las mismas características.

Hay un problema muy habitual en la vida real y es la de clasificar una serie de elementos en distintas categorías en función de información sobre ellos, esta tarea puede ser realizada por personas o, lo que es más eficiente, por un ordenador. Para realizar tal clasificación es habitual que se recojan multitud de datos sobre los distintos elementos que se quieren clasificar, de modo que en base a esta información podamos decidir si el elemento es de una categoría o de otra. Pensemos por ejemplo en clasificar fruta en base a si se desecha o no. Podemos pensar en recoger datos sobre el tamaño de esa fruta, su color, su textura o su dureza y en base a estas mediciones una máquina debería clasificar la fruta en buena o mala.

El problema está en que normalmente no se conoce tan bien el campo de estudio como para saber a ciencia cierta qué datos recoger, qué datos serán más relevantes a la hora de clasificar elementos de una determinada población. Entonces lo que se hace es recoger gran cantidad de información sobre los elementos para al menos intentar que no haya carencias en la información, esto por supuesto conlleva tanto el coste de adquirir esa información (no sabemos cómo de caro es realizar una determinada medición) y el coste computacional de procesar toda esa información. Entonces lo que nos gustaría es averiguar qué información, de entre toda la que hemos obtenido, es la verdaderamente relevante para la clasificación que queremos realizar.

Entonces partiendo de un conjunto de datos de aprendizaje, valores de características de distintos elementos, queremos ver con qué subconjunto de características podemos hacer una buena clasificación de esos elementos, así si tenemos que cada dato viene dado por una lista de  $n$  características  $[f_1, f_2, \dots, f_n]$  queremos obtener un subconjunto de esas características, de modo que teniendo sólo la información  $[f_{s1}, \dots, f_{sm}]$ , se haga una buena clasificación del conjunto de datos de aprendizaje, del que conocemos por supuesto la clasificación perfecta de dichos datos. Y esperamos que con esa misma información se clasifiquen lo mejor posible nuevos elementos de fuera de la muestra de aprendizaje.

## 2 Descripción de la aplicación de los algoritmos empleados al problema

Dado que estamos ante un problema de selección las soluciones se representarán como vectores binarios de booleanos de tamaño el número de características a elegir, indicando si una característica se considera o no, así tendremos claramente un espacio de búsqueda de  $2^n$ , siendo el  $n$  el número de características a elegir que también lo podemos ver como el tamaño del problema abordado.

Entonces para evaluar como de buena es una determinada solución hacemos lo siguiente:

```
tomamos de cada dato de entrenamiento las características
seleccionada por la solución.

for cada dato de entrenamiento:
    clasificador_dato = construir clasificador 3NN con el resto
    de datos
    ver si clasificador_dato clasifica correctamente a ese dato

return media del numero de aciertos
```

Para generar las soluciones vecinas a una dada usamos el operador flip que funciona del siguiente modo:

```
def flip(sol, idx):
    cambiar el valor de la pos. idx de la sol por su negado
```

Cuando queramos ver cómo de buena es la solución final que ha obtenido nuestro algoritmo de búsqueda lo que hacemos es:

```
clasificador = construir clasificador con los datos de entrenamiento
solo con las caracterisiticas seleccionadas por nuestra
solucion
ver el porcentaje de acierto de este clasificador etiquetando los
datos de test
```

A cada algoritmo le daremos solamente los datos de entrenamiento separados en características y etiquetas para que con la estrategia de búsqueda que implemente nos devuelva la mejor solución posible para él, luego la evaluación de la solución final la haremos fuera del algoritmo con los datos de test (en la sección 8 ya diremos cómo hemos generado las particiones TODO).

## 3 Descripción en pseudocódigo de los algoritmos

### 3.1 Búsqueda Local (BL.py)

```
s = solucion inicial aleatoria
while True:
    for cada vecino de s explorados en orden aleatorio:
        if el vecino es mejor que s:
            s = el vecino

        if se ha generado el numero max total de sols:
            return s

        if se ha encontrado vecino mejor a la sol actual:
            break

    if no hemos encontrado ninguna solucion mejor en todo el
    vecindario:
        return s
```

## 3.2 Enfriamiento simulado (ES.py)

El esquema de enfriamiento que seguimos es el de Cauchy modificado para el que calculamos la nueva temperatura a partir de la anterior del siguiente modo:

```
nueva_T = T_actual/(1+beta*T_actual) #beta es una param. del
    problema
```

y la temperatura inicial se calcula en base al "costo" de la solución inicial, que en este caso es su poder de clasificación para las muestras de entrenamiento de las que partamos, y dos parámetros  $\mu$  y  $\phi$  que fijamos a 0.3 con lo que el cálculo de la temperatura inicial es tan sencillo como el siguiente:

```
T0 = -mu*coste_sol_inicial/log(fi)
```

hay algo que me sorprende de este hecho y es que dado que el logaritmo neperiano de 0.3 es negativo, en nuestro cálculo, cuanto mejor sea la solución inicial de la que partamos, más elevada será dicha temperatura, es decir, más tardaremos en llegar a la temperatura final.

Pasamos ya a describir el esquema de búsqueda del algoritmo para el enfriamiento simulado:

```
generamos una sol. inicial aleatoria y calculamos los params. de el
    problema

while se acepte algun vecino en el vecindario explorado de la sol
    actual and no se hayan generado 15000 vecinos en total:

    n_exitos = 0 #vecinos generados y aceptados hasta el
        momento
    for hasta el max_vecinos permitidos:
        generamos un vecino aleatorio, cambiando una
            característica de la sol actual al azar
        n_vecinos_generados++

        delta = diferencia entre coste sol. actual y el
            vecino

        if (el vecino es mejor que la sol actual or U(0,1)
            <= exp(-delta/temperatura_actual) #U(0,1) valor
                aleatorio uniforme en el [0,1]) and delta !=
                0:
            sol actual = vecino
            n_exitos++

            if la nueva sol es mejor que la mejor sol
                encontrada hasta el momento:
                    mejor_sol = sol_actual

        if n_exitos == max_exitos or hemos generado el
            numero total de vecinos posibles:
                break

    actualizar temperatura

return la mejor solucion encontrada
```

Hemos añadido la condición de que delta no sea cero ya que de otro modo la exponencial que usamos para aceptar soluciones peores valdría uno con lo que estaríamos tomando siempre aquellos vecinos con el mismo coste que la solución actual, esto hacía que el algoritmo generase siempre los 15000 vecinos que tiene como tope, tardando media hora para una partición de wdbc.

### 3.3 Búsqueda Tabú (BT.py)

Aquí tenemos el pseudocódigo para la búsqueda tabú básica, en el también se ve cómo manejamos la lista tabú como una lista cíclica que consultamos para ver si un vecino ha sido generado mediante un movimiento tabú:

```
while no se hayan generado el maximo de vecinos permitido:
    elegimos n vecinos aleatorios a revisar, siendo n el maximo
    permitido por iteracion

    for cada vecino elegido:

        if el movimiento que lo genera esta en la lista
        tabu(LT):
            if el vecino es mejor que la mejor sol
            encontrada and mejor que el mejor
            vecino explorado en esta iteracion:
                mejor vecino = nuevo vecino
        else:
            if mejor que el mejor vecino explorado en
            esta iteracion:
                mejor vecino = nuevo vecino

        if hemos generado el maximo de vecinos totales
        permitidos:
            break

    sol_actual = mejor vecino encontrado en la iteracion

    if sol_actual mejor que la mejor encontrada hasta el
    momento:
        mejor_sol_encontrada = sol_actual

    LT[posicion_correspondiente] = movimiento que genero el
    mejor vecino de esta iteracion
    posicion_correspondiente = a la siguiente ciclicamente

return mejor_sol_encontrada
```



### 3.4 Búsqueda Tabú Extendida (BText.py)

## 4 Breve descripción del algoritmo de comparación

Para el algoritmo de comparación, SFS, lo que hacemos es lo que vamos a reflejar en el siguiente pseudocódigo, la principal diferencia es que no tenemos la función usual flip sino que hemos hecho una función especial para poder realizar una función vectorizada en Python (aunque no produce ninguna mejora en tiempo a la versión que tendríamos si simplemente tuviésemos un for que recorriese las características que quedan por añadir hasta encontrar la de mayor ganancia), esta función crea una nueva solución poniendo a True una componente de la solución que le pasamos y nos da su porcentaje de clasificación, como hacemos en el resto de algoritmos. Dicho esto pasamos al pseudocódigo:

```
sol = un array binario con todo False #conjunto vacío de  
      características  
  
while tengamos ganancia and queden características por añadir:  
    calcular el score de cada característica por añadir al  
      agregarla al conjunto actual  
    tomar la característica que de mejor score en el cálculo  
      anterior  
  
    if el score añadiéndola es mejor que el de el conjunto  
      mejor al que había:  
        se agrega dicha característica al conjunto  
        quitamos esa característica de el conjunto de  
          características por añadir  
    else:  
        no tenemos ganancia y acabamos el bucle  
  
return el conjunto de características al que hemos llegado
```

## **5 Procedimiento considerado para desarrollar la práctica**