

Práctica 2.b: Búsquedas Multiarranque para el Problema de la
Selección de Características

BMB
GRASP
ILS

Gustavo Rivas Gervilla DNI: 75570417F
gustavofox92@correo.ugr.es
5º Doble Grado en Ing. Informática y Matemáticas
Grupo 3

Índice

1	Descripción/formulación del problema abordado	3
2	Descripción de la aplicación de los algoritmos empleados al problema	4
3	Descripción en pseudocódigo de los algoritmos	6
3.1	Búsqueda Multiarranque Básica (BMB.py)	6
3.2	GRASP (GRASP.py)	7
3.3	ILS (ILS.py)	8
4	Breve descripción del algoritmo de comparación	9
5	Procedimiento considerado para desarrollar la práctica	10
5.1	Desarrollo	10
5.2	Manual de usuario	10
6	Experimentos y análisis de resultados	12
6.1	Descripción de los casos del problema empleados y de los valores de los parámetros	12
6.2	Resultados	13
6.3	Análisis de resultados	15
7	Bibliografía	20

1 Descripción/formulación del problema abordado

Lo que intentamos hacer con nuestros 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) como 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 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 porcentaje de aciertos
```

Nuestros algoritmos irán explorando, según su mecanismo, el espacio de soluciones empleando la función anterior para tomar decisiones sobre qué movimientos se realizan en dicho espacio de búsqueda. Para poder generar las soluciones vecinas a una dada empleamos el operador de Flip, el cual funciona del siguiente modo:

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

Cuando finalice su proceso de búsqueda los algoritmos nos devolverán la solución que ellos han elegido (junto con el score dentro de la muestra de entrenamiento calculada como hemos dicho antes). Entonces una vez tenemos la solución nos queda por evaluar cómo de bien se clasifican los datos empleando un clasificador 3NN construido sólo con aquellas características seleccionadas por nuestra solución:

```
claficador = construir clasificador con los datos de entrenamiento solo con las
caracterisíticas seleccionadas por nuestra solución
return el porcentaje de acierto de este clasificador etiquetando los datos de test de los
que consideramos solo las características seleccionadas por la solución
```

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 explicaremos cómo hemos generado las particiones). Para aquellos algoritmos que empleando algún tipo de aleatoriedad en sus decisiones inicializaremos la semilla aleatoria antes de la ejecución de dicho algoritmo.

Tanto en la BMB como en el ILS hemos de generar unas soluciones aleatorias, para cada algoritmo esta solución aleatoria se genera de un modo diferente:

Para la BMB:

```
for cada iteracion del algoritmo:
    Obtengo una muestra aleatoria de len = n_caracteristica muestreada uniforme y con
    remplazamiento.
```

Para el ILS:

```

al inicio del algoritmo y nunca mas:
    Obtengo una muestra aleatoria de len = n_caracteristica muestreada uniforme y con
    remplazamiento.

```

El proceso de búsqueda local empleado es el que implementamos en la primera práctica, como sabemos la factorización en el caso de la selección de características no es trivial por lo que en nuestro caso, al contrario que en el QAP, no empleamos ningún mecanismo de factorización para el cálculo de la función objetivo. El operador de generación de vecino ya ha sido descrito anteriormente, *flip*. El mecanismo de búsqueda de la búsqueda local es como sigue:

```

s = solucion que le pasamos por param.
n_sols_generados = 0

while n_sols_generados < max_permitido:
    for cada vecino de s explorados en orden aleatorio:
        n_sols_generados++

        if el vecino es mejor que s:
            s = el vecino
            dejamos de explorar vecinos

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

return s

```

Aquí hay un cambio fundamental con respecto a la práctica anterior y es que antes la solución de la que se partía era aleatoria, ahora nosotros le decimos con qué solución comenzar el proceso de búsqueda local, que puede ser con una solución aleatoria como en el caso de la BMB o con una mutación de la mejor solución encontrada hasta el momento como en el ILS.

3 Descripción en pseudocódigo de los algoritmos

3.1 Búsqueda Multiarranque Básica (BMB.py)

```
for 25 veces:
    s_r = sol. aleatoria
    s = sol. encontrada por las Búsqueda Local partiendo desde s_r

    if s es mejor que la mejor sol. encontrada hasta el momento:
        mejor_s = s

return mejor_s
```

3.2 GRASP (GRASP.py)

El algoritmo de generación de soluciones greedy probabilísticas funciona del siguiente modo:

```
sol = un array binario con todo False #conjunto vacio de caracteristicas

while tengamos ganancia and queden caracteristicas por aniadir:
    calcular el score de cada caracteristica por aniadir al agregarla al conjunto
    actual

    mu = mejor_score_de_los_candidatos - 0.3(mejor_score_de_los_candidatos -
    mejor_score_de_los_candidatos)

    lista_restringida_candidatos = caracteristicas de las anteriores que den ganancia
    >= mu
    elegimos una caracteristica aleatoria de la listra restringida

    if el score aniadiendola es mejor que el de el conjunto mejor al que habia:
        se agrega dicha caracteristica al conjunto
        quitamos esa caracteristica de el conjunto de caracteristicas por aniadir
    else:
        no tenemos ganancia y acabamos el bucle

return el conjunto de caracteristicas al que hemos llegado

for 25 veces:
    s_greedy = sol. encontrada por el greedy aleatorio
    s = sol. encontrada por la Busqda Local partiendo desde s_greedy

    if s es mejor que la mejor sol. encontrada hasta el momento:
        mejor_s = s

return mejor_s
```

3.3 ILS (ILS.py)

```
mutar(s):  
    t = ceil(0.1*n_caracteristicas)  
    genes_a_mutar = elegimos t genes aleatorios, sin repeticion, para ser mutados  
    cambiamos el valor de cada gen de s a mutar por su complementario  
  
s = sol. inicial aleatoria  
s = sol. encontrada por Busqueda Local partiendo de s  
  
for 24 veces:  
    if s es mejor que la mejor sol. encontrada:  
        best_s = s  
  
    s = mutar(best_s)  
    s = sol. encontrada por Busqueda Local partiendo de s  
  
return best_s
```


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

5.1 Desarrollo

El código usado en prácticas lo he implementado yo a partir de las explicaciones dadas tanto en clase de prácticas como de teoría sobre los distintos algoritmos, siendo de mucha utilidad las indicaciones del guión de prácticas.

La parte que no ha sido implementada por mí es la correspondiente a la evaluación de las soluciones, es decir, tanto el KNN como los mecanismos de evaluación de las soluciones. Para ello he usado el código desarrollado por **Alejandro García Montoro** el cuál ha implementado tanto el clasificador KNN como los mecanismos para obtener el score de las soluciones dentro (con LOO) y fuera de la muestra con PyCUDA que permite insertar código CUDA en código Python. Con lo cual los tiempos de ejecución bajan considerablemente con respecto a los tiempos de la práctica anterior ya que el cuello de botella de los algoritmos, que es la evaluación de las soluciones con LOO, se ejecuta en la tarjeta gráfica, siendo mucho más rápidos los cálculos.

5.2 Manual de usuario

Los algoritmos han sido implementados en Python 3.1 TODO para su ejecución son necesarios tener instalados los módulos compatibles con esta versión de Python siguientes:

- numpy (v 1.8.2) **sudo apt-get install python-numpy**
- scikit **sudo apt-get install python-scikits-learn**
- jinja2
- pycuda
- El resto de módulos empleados suelen venir con las distribuciones básicas de Python (random, sys y time).

Cada uno de los algoritmos están implementados en distintos ficheros de los que podemos ver su contenido en el **LEEME** que se nos pide que incluyamos con el código. Los experimentos han sido ejecutados usando el fichero main al que le pasamos por argumentos el nombre del algoritmo a usar: 3NN, SFS, BMB, GRASP e ILS. La semilla se establece dentro del fichero al inicio de la ejecución del algoritmo para cada una de las particiones. Esto lo hemos hecho así para poder cortar la obtención de datos cuando sea necesaria y poder retomar los experimento desde la partición por las que nos quedáramos, así no se desajusta la semilla con la que hacemos los experimentos al retomar.

Entonces para ejecutar por ejemplo los experimentos para el GRASP lo que haremos es *python mainCUDA.py GRASP* y se ejecutará el algoritmo sobre todas las particiones de datos que tenemos.

En el directorio **make_partitions32b** tenemos tanto los archivos arff como los códigos necesarios para elaborar las particiones que serán utilizadas por nuestros algoritmos. Esto no es necesario para ejecutar los programas puesto que en el directorio **partitions32b** tenemos almacenadas todas las particiones construidas. No obstante si se quisiera replicar el procedimiento de desarrollo de particiones lo que haríamos sería: eliminar los archivos con extensión .npy del directorio **make_partitions32b** (para tener seguridad de que no se corrompan dichos archivos), ejecutar el loader con la instrucción **python loader.py**

y a continuación elaborar las particiones con **python partition_maker.py** (dentro de este directorio que es donde se encuentran ambos códigos). Esto generará diversos archivos de extensión .npy con nombres que empiezan por arr, wdbc o libras (según la base) seguida de un número y la palabra test o training.

Estos archivos deberán reemplazar a aquellos que se encuentran en el directorio **partitions32b** y ya podremos ejecutar los algoritmos con dichas particiones desde el main.

6 Experimentos y análisis de resultados

6.1 Descripción de los casos del problema empleados y de los valores de los parámetros

Estas prácticas han sido implementadas en Python 2.7.6 y ejecutadas sobre un ordenador con S.O. Ubuntu 14.04 LTS, de 8GB de RAM y procesador Intel Core i7-4790 CPU 3.60GHz con 8 núcleos, aunque dado que el código no ha sido paralelizado estos núcleos no han sido explotados al máximo.

Hemos empleado tres bases de datos distintas, a continuación mencionamos sus tamaños ya que serán interesantes para el posterior análisis de resultados:

- **wdbc**: Tenemos **569** muestras con **30** características cada una procedentes de imágenes digitalizadas de una masa de mama. Estas muestras se clasifican en **2** clases distintas.
- **libras**: Aquí tenemos **360** muestras de **90** características cada una, éstas muestras son datos de distintos movimientos de la mano que se clasifican en **15** clases distintas.
- **arritmia**: Los datos de este conjunto son mediciones para determinar la presencia de arritmia cardiaca o no. Tenemos **386** muestras con **278** características cada una, a clasificar en **5** grupos en base al tipo de arritmia que indican los datos de la muestra.

Lo que hemos hecho con los datos ha sido un preprocesado, en primer lugar todos los datos han sido codificados como flotantes (las etiquetas de wdbc eran cadenas de texto y las hemos cambiado por los valores 0 y 1), también hemos puesto la etiqueta en la última columna de la tabla de datos. Es importante señalar que para poder emplear el KNN implementado en pyCUDA que hemos mencionado antes hemos tenido que codificar los distintos datos de las particiones en 32 bits ya que así es como estaba configurado el clasificador y CUDA no hace correctamente el casting de 64b a 32b.

También hemos normalizado cada una de las columnas de datos (sin contar la de etiquetas) de modo que los valores quedaran en el intervalo [0,1] mediante la fórmula:

$$x_j^N = \frac{x_j - Min_j}{Max_j - Min_j} \text{ (siendo } Max_j \text{ y } Min_j \text{ el máximo y mínimo valor de los datos para la característica } j\text{-ésima de las muestras).}$$

El código empleado para este formateo de los datos está en el fichero **loader.py** que se adjunta en la entrega. Como podemos ver para realizar la normalización de los datos hemos usado una utilidad del módulo scikit-learn antes mencionado que se llama MinMaxScaler. Y para futuros usos de estos datos normalizados, y con el objetivo de no tener que realizar tales operaciones cada ocasión que queramos usarlos, hemos almacenado los arrays de numpy donde hemos almacenado los datos en sendos ficheros con extensión .npy usando la función **numpy.save**. Estos ficheros son: **data_wdbc.npy**, **data_libras.npy** y **data_arrhythmia.npy**.

A continuación hemos elaborado las distintas particiones sobre estos datos que serán utilizadas en los experimentos, para ello hemos elaborado el código recogido en el fichero **partition_maker.npy** donde simplemente tomamos una muestra aleatoria de los índices de los arrays antes generados, teniendo en cuenta que la cantidad de muestras de cada una de las clases en las que se clasifican los datos sea lo más equilibrada posible entre la partición de test y su correspondiente partición de training. La semilla usada para la generación de dichas particiones, y que podemos ver en el fichero mencionado, ha sido la

12345678. Nuevamente hemos almacenado cada una de dichas particiones en ficheros .npy, los cuáles están en la carpeta **partitions32b**.

Los parámetros empleados en cada uno de los algoritmos son los que se indican en el guión de prácticas. Para cada una de las ejecuciones de lo algoritmos inicializamos la semilla aleatoria a **12345678** nuevamente, esto lo podemos ver en el fichero **mainCUDA.py**.

6.2 Resultados

Adjuntamos también las tablas en un fichero por si no se leen correctamente en el pdf.

Tabla 1: 3NN

	Wdbc				Movement_Libras				Arrhythmia			
	%_clas_in	%_clas_out	%_red	T	%_clas_in	%_clas_out	%_red	T	%_clas_in	%_clas_out	%_red	T
Partición 1-1	96,14	96,13	0,00	0,00	75,56	76,11	0,00	0,00	65,46	64,06	0,00	0,00
Partición 1-2	96,83	95,79	0,00	0,00	67,78	70,56	0,00	0,00	59,38	62,37	0,00	0,00
Partición 2-1	95,44	96,47	0,00	0,00	70,00	67,78	0,00	0,00	62,89	62,50	0,00	0,00
Partición 2-2	96,13	96,49	0,00	0,00	74,44	72,78	0,00	0,00	63,02	64,43	0,00	0,00
Partición 3-1	93,68	97,18	0,00	0,00	70,56	73,33	0,00	0,00	60,82	65,10	0,00	0,00
Partición 3-2	97,54	95,44	0,00	0,00	70,00	72,22	0,00	0,00	61,98	64,95	0,00	0,00
Partición 4-1	96,14	95,42	0,00	0,00	72,22	72,22	0,00	0,00	60,82	64,58	0,00	0,00
Partición 4-2	98,24	95,44	0,00	0,00	70,00	74,44	0,00	0,00	64,58	62,37	0,00	0,00
Partición 5-1	96,14	95,42	0,00	0,00	71,11	78,33	0,00	0,00	67,01	63,02	0,00	0,00
Partición 5-2	97,89	95,44	0,00	0,00	71,67	73,33	0,00	0,00	63,54	62,37	0,00	0,00
Media	96,42	95,92	0,00	0,00	71,33	73,11	0,00	0,00	62,95	63,58	0,00	0,00

Tabla 2: SFS

	Wdbc				Movement_Libras				Arrhythmia			
	%_clas_in	%_clas_out	%_red	T	%_clas_in	%_clas_out	%_red	T	%_clas_in	%_clas_out	%_red	T
Partición 1-1	95,44	95,42	83,33	0,24	78,89	72,22	90,00	1,04	79,90	68,23	97,12	3,00
Partición 1-2	97,54	92,98	90,00	0,15	74,44	60,00	93,33	0,69	74,48	70,10	98,56	1,51
Partición 2-1	97,19	97,54	86,67	0,20	77,22	66,11	88,89	1,16	84,54	71,35	97,12	3,00
Partición 2-2	97,54	95,79	90,00	0,15	76,11	65,56	91,11	0,92	72,92	69,59	99,28	0,85
Partición 3-1	96,49	95,77	83,33	0,24	78,33	73,89	90,00	1,04	77,32	69,79	97,48	2,62
Partición 3-2	97,89	95,09	86,67	0,20	76,11	75,00	91,11	0,92	85,94	74,23	96,04	4,27
Partición 4-1	95,09	91,55	93,33	0,11	76,11	67,22	91,11	0,92	79,90	67,19	96,04	4,24
Partición 4-2	98,24	94,04	76,67	0,33	70,56	65,56	93,33	0,69	83,33	78,87	96,76	3,41
Partición 5-1	96,84	97,18	86,67	0,20	68,33	62,78	93,33	0,70	84,02	69,79	96,04	4,26
Partición 5-2	98,24	94,04	83,33	0,24	74,44	66,11	92,22	0,81	83,85	73,71	96,04	4,25
Media	97,05	94,94	86,00	0,21	75,05	67,45	91,44	0,89	80,62	71,29	97,05	3,14

Tabla 3: Resultados BMB

	Wdbc				Movement_Libras				Arrhythmia			
	%_clas_in	%_clas_out	%_red	T	%_clas_in	%_clas_out	%_red	T	%_clas_in	%_clas_out	%_red	T
Partición 1-1	98,25	95,42	40,00	3,60	79,44	72,78	51,11	12,98	74,23	65,10	49,64	125,10
Partición 1-2	98,59	95,44	43,33	3,48	77,78	69,44	54,44	13,65	71,88	65,98	50,00	143,26
Partición 2-1	97,89	95,77	56,66	3,70	77,22	67,22	62,22	12,95	74,23	64,06	47,84	141,67
Partición 2-2	98,59	95,09	56,66	3,28	78,33	70,00	57,78	12,29	70,31	64,95	53,96	144,62
Partición 3-1	96,84	96,83	46,66	3,47	77,78	71,67	51,11	12,09	72,16	64,58	53,60	121,83
Partición 3-2	98,94	95,09	43,33	3,04	76,67	71,67	54,44	14,37	72,92	63,92	53,60	164,83
Partición 4-1	98,25	95,42	50,00	3,68	78,89	75,00	48,89	12,08	72,68	67,71	52,88	152,11
Partición 4-2	98,94	95,09	50,00	3,41	76,67	77,22	54,44	12,25	75,00	64,43	54,68	154,49
Partición 5-1	98,25	95,42	43,33	3,18	75,00	77,78	56,67	14,36	73,71	66,15	51,08	123,77
Partición 5-2	98,94	94,74	53,33	3,37	78,89	77,22	61,11	13,00	72,40	61,86	50,00	140,37
Media	98,35	95,43	48,33	3,42	77,67	73,00	55,22	13,00	72,95	64,87	51,73	141,21

Tabla 4: GRASP

	Wdbc				Movement_Libras				Arrhythmia			
	%_clas_in	%_clas_out	%_red	T	%_clas_in	%_clas_out	%_red	T	%_clas_in	%_clas_out	%_red	T
Partición 1-1	98,6	91,90	80,00	9,51	81,67	75,56	86,67	33,30	85,05	72,92	93,88	132,67
Partición 1-2	98,94	95,44	46,67	10,88	78,33	72,22	85,56	36,15	82,29	67,01	92,81	99,77
Partición 2-1	97,89	94,01	66,67	8,38	78,89	67,22	86,67	34,03	84,54	77,08	94,60	118,87
Partición 2-2	98,59	96,14	23,33	10,54	81,67	71,11	84,44	36,00	82,29	73,71	93,88	78,29
Partición 3-1	97,19	96,83	60,00	9,11	80,00	70,00	87,78	35,56	82,47	70,83	92,45	98,61
Partición 3-2	98,94	96,14	70,00	11,25	78,89	75,00	84,44	32,54	84,38	70,62	93,17	157,76
Partición 4-1	97,89	95,42	60,00	9,18	83,33	72,78	77,78	38,11	81,44	73,96	92,45	102,19
Partición 4-2	98,94	94,74	46,67	9,68	80,56	71,67	80,00	35,71	85,94	73,71	90,29	143,39
Partición 5-1	98,25	91,90	70,00	10,13	77,22	78,89	86,67	31,41	86,08	69,79	94,96	108,93
Partición 5-2	98,94	94,39	73,33	10,98	81,67	75,00	81,11	32,40	82,81	76,29	93,17	99,15
Media	98,42	94,69	59,67	9,96	80,22	72,95	84,11	34,52	83,73	72,59	93,17	113,96

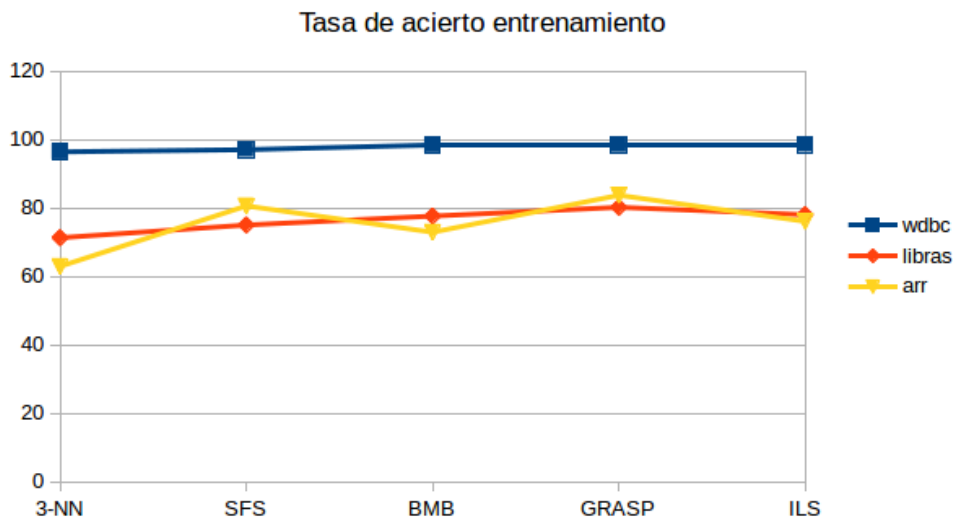
Tabla 5: ILS

	Wdbc				Movement_Libras				Arrhythmia			
	%_clas_in	%_clas_out	%_red	T	%_clas_in	%_clas_out	%_red	T	%_clas_in	%_clas_out	%_red	T
Partición 1-1	98,6	94,37	63,33	3,06	79,44	76,67	46,66	12,04	79,38	66,67	52,52	114,28
Partición 1-2	98,6	95,09	53,33	3,10	74,44	71,67	48,89	10,99	74,48	67,01	55,40	111,76
Partición 2-1	98,25	96,13	53,33	3,60	75,56	70,00	52,22	10,83	74,74	65,10	53,24	95,07
Partición 2-2	98,59	95,79	46,66	3,21	80,00	73,89	52,22	11,36	74,48	72,16	53,96	119,37
Partición 3-1	97,19	97,54	43,33	3,15	77,78	73,33	53,33	12,83	76,29	68,23	55,04	102,29
Partición 3-2	99,3	95,79	36,66	3,13	78,89	71,11	53,33	11,38	76,56	65,98	57,91	115,71
Partición 4-1	98,25	96,13	46,66	2,47	81,11	74,44	54,44	11,94	76,80	68,23	51,80	114,23
Partición 4-2	99,3	94,39	33,33	3,07	78,33	75,00	55,56	13,08	76,04	62,89	50,72	112,01
Partición 5-1	97,89	94,72	43,33	2,91	76,11	78,89	54,44	12,41	76,29	67,19	48,20	129,35
Partición 5-2	99,3	95,44	43,33	2,67	78,89	78,89	45,56	11,46	76,04	63,92	56,12	105,87
Media	98,53	95,54	46,33	3,04	78,06	74,39	51,67	11,83	76,11	66,74	53,49	111,99

Tabla 6: Resultados Totales

	Wdbc				Movement_Libras				Arrhythmia			
	%_clas_in	%_clas_out	%_red	T	%_clas_in	%_clas_out	%_red	T	%_clas_in	%_clas_out	%_red	T
3-NN	96,417	95,922	0	0	71,334	73,11	0	0	62,95	63,575	0	0
SFS	97,05	94,94	86	0,206	75,054	67,445	91,443	0,889	80,62	71,285	97,048	3,141
BMB	98,35	95,43	48,33	3,42	77,67	73,00	55,22	13,00	72,95	64,87	51,73	141,21
GRASP	98,417	94,691	59,667	9,964	80,223	72,945	84,112	34,521	83,729	72,592	93,166	113,963
ILS	98,527	95,539	46,329	3,037	78,055	74,389	51,665	11,832	76,11	66,738	53,491	111,994

6.3 Análisis de resultados



Como podemos ver la tasa de acierto sobre Wdbc es la más elevada con respecto a las otras bases, además las tasas de acierto de todos los algoritmos sobre esta base son muy parecidas, esto se debe a que, como dijo el profesor, esta base de datos es muy pequeña, es decir, el espacio de búsqueda es muy pequeño (2^{30}) y hay muy poca tasa de mejora entre una solución y otra.

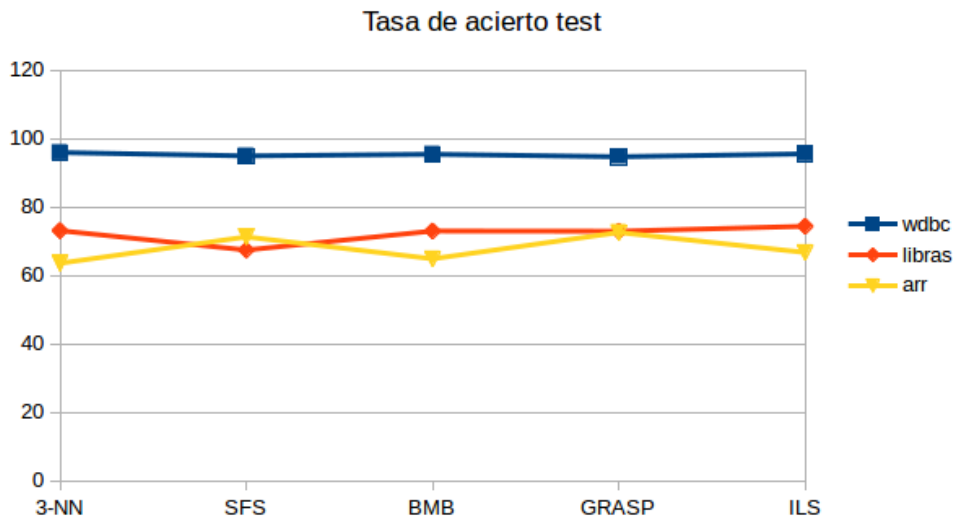
Las tasas de acierto tanto para arritmia como para libras son muy parecidas como podemos ver en la gráfica para cada algoritmo. Es importante señalar cómo la tasa de acierto de los algoritmos usados en mejor que la tasa que da el 3NN original, es decir, usando todos los datos de la muestra. Con lo cual aquí se pone de manifiesto cómo el hecho de reducir el número de características a considerar no merma la correcta clasificación de los datos, al menos dentro de la muestra ya veremos qué sucede fuera. De hecho reduciendo el número de características no sólo reducimos el cómputo necesario sino también el posible ruido que pueden tener algunos datos.

Es curioso ver cómo el algoritmo greedy produce los mejores resultados en la base de datos con un espacio de búsqueda mayor, de hecho produce mejores resultados en esa base de datos que en la de libras que tiene un espacio menor, probablemente por la naturaleza de los datos, es decir, cómo de distintas sean las muestras que componen las distintas clases entre sí. En mi opinión el hecho de que sea el SFS el que produzca mejores resultados que el resto en el espacio de búsqueda más grande se debe a que al fin y al cabo el greedy en cada momento va construyendo la mejor solución que puede mientras que en cambio el resto de métodos depende más de cómo de buena sea la exploración que realicen. Entonces en un espacio de tamaño 2^{278} el hecho de que impongamos a los algoritmos que sólo pueda hacer 5000 evaluaciones hace que su capacidad de encontrar una buena solución sea pequeña, además de depender un tanto del azar.

En cambio si podemos ver cómo para el espacio más reducido de libras sí que los algoritmos que hacen una búsqueda más extensa en el terreno, dan resultados más próximos a los que da el SFS e incluso superándolo como en las dos versiones de la Búsqueda Tabú. De hecho esta búsqueda da los mejores resultados sobre el conjunto de libras.

Vemos también cómo los algoritmos de BL, ES, BT y BText dan mejores resultados sobre la base de

libras que sobre la de arritmia por lo que hemos comentado antes probablemente.

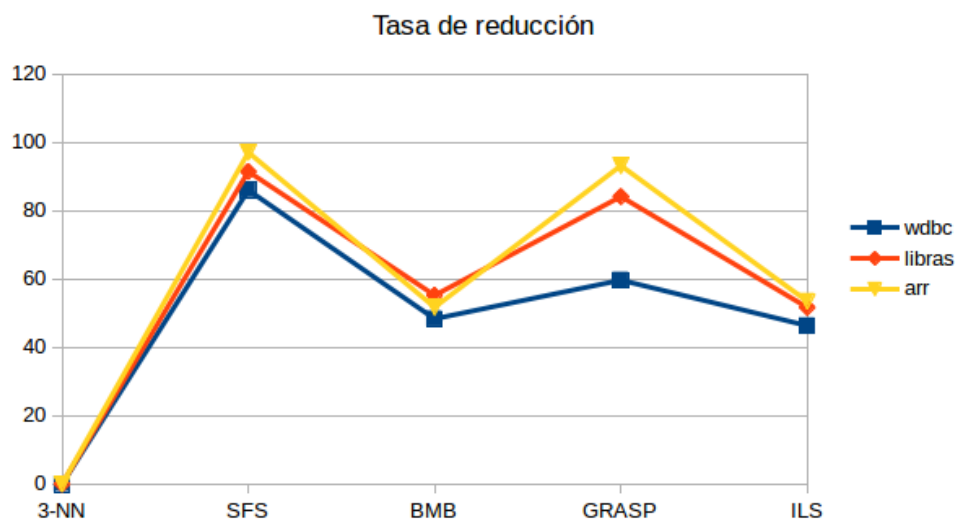


Veamos ahora cómo se comportan los algoritmos con respecto a la clasificación que producen sus características seleccionadas para los datos fuera de la muestra de entrenamiento. Lo primero que señalamos es que en la amplia mayoría de los casos la tasa de acierto es mayor dentro de la muestra que fuera, pero hay algunos casos donde es al revés. Esto se debe a que es probable que haya muestras de entrenamiento de una clase más cercanas a otra muestra pese a no pertenecer a la clase de dicha muestra. Con lo cual se puede dar el caso de que la "votación" sea mala en la muestra y en cambio la muestra fuera la distribución de puntos sea más favorable a una correcta clasificación de los datos. Por lo tanto no tiene que extrañarnos demasiado esto. Quizás si usásemos un 1NN sí que sería más probable que la muestra más cercana a otra perteneciera a la misma clase que ella (decimos la más cercana y no que en el 1NN hubiésemos obtenido una tasa de acierto del 100% dentro de la muestra puesto que usamos para evitar esto el Leave One Out).

Nuevamente el comportamiento para Wdbc es muy similar para todos los algoritmos, lo que era esperable por el razonamiento que hemos dado antes. Pero también observamos que si bien sigue habiendo diferencias entre el comportamiento de los distintos algoritmos para una determinada base de datos ahora las diferencias son menos notables entre unos y otros. Es decir, que aunque hay algunos algoritmos que son capaces de adaptarse mejor que otros para la muestra que le pasamos, luego la solución que nos dan se comporta más o menos del mismo modo fuera de la muestra. Poniendo de manifiesto que no podemos asegurar que una solución que sea buena dentro de los datos de entrenamiento vaya a serlo también fuera.

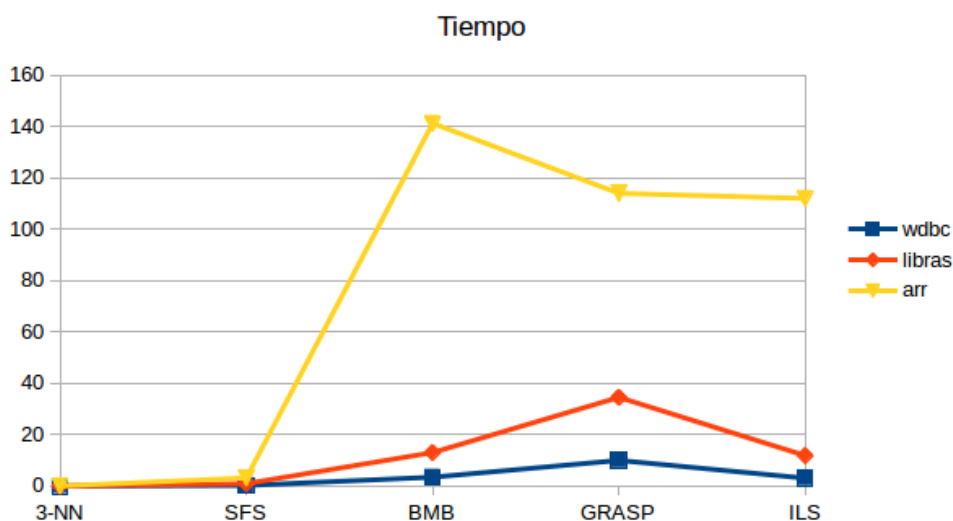
Algo que también cabe señalar es que en esta ocasión para la base de datos de libras el 3NN sí supera a dos algoritmos, la BText y el SFS, pero no obstante no mejora a todos los algoritmos con lo que no podemos decir que para obtener mejores resultados en datos desconocidos tengamos que usar toda la información de la que dispongamos. Además por ejemplo se comporta mejor que la Búsqueda Tabú Extendida pero no mejor que la Búsqueda Tabú, aquí se pone de manifiesto que depende fuertemente de los datos que se consideren porque la BT y su versión extendida no difieren mucho en comportamiento. Difieren en que la extendida hace una exploración más esparcida del espacio de búsqueda, pero esto con sólo 5000 evaluaciones puede que no se pueda ver tan reflejado en los resultados.

El hecho de que el 3NN con toda la información no mejore a todos los algoritmos puede venir de que no sólo haya características irrelevantes sino que, como hemos señalado anteriormente, haya datos cuya medición haya sido errónea y que simplemente lo que nos estén haciendo sea empeorar el resultado.



Evidentemente la tasa de reducción para el 3NN es 0 pues tomamos todas las características al no tener otro criterio. Ahora el que produce mayor reducción es el greedy lo que es lógico por la forma en la que trabaja, agregando progresivamente propiedades hasta no encontrar mejora ninguna.

Las tasas de reducción para el resto de algoritmos son muy similares entre sí, si lo pensamos dado que nuestra función de coste sólo tiene en cuenta la tasa media de acierto en la muestra de entrenamiento (no tenemos en cuenta también la reducción que consigue) es lógico que esto suceda así, o al menos que no haya un algoritmo que claramente produzca mejores resultados que otro. Partimos de una solución totalmente aleatoria y nos movemos por los vecinos solamente teniendo en cuenta su score y algún término de azar como sucede en el Enfriamiento Simulado. Por lo tanto no podemos decir nada acerca de por qué se comportan así el resto de algoritmos respecto a la reducción y sólo podemos atribuirlo al azar.



Evidentemente el 3NN tarda 0 segundo puesto que su tiempo de búsqueda de solución es 0 pues simplemente devuelve un solución con todos los valores a True, es decir, escoge todas las características sin tener ningún criterio.

Como podemos ver los tiempos de ejecución del SFS y la Búsqueda Local son "similares" puesto que en esencia funcionan del mismo modo, parten de una solución, exploran su entorno y eligen el mejor, continuando hasta que no encuentran mejora. Y si nos fijamos los tiempos de la búsqueda local son mejores incluso que los del greedy y esto se debe a lo que mayor tiempo de cómputo consume en todos los algoritmos que hemos diseñado, la evaluación de las soluciones. Y es que mientras que el greedy ha de explorar todo el entorno de la solución, la búsqueda local evalúa vecinos hasta que encuentra uno mejor que la solución actual; con lo que el número de evaluaciones es menor y por tanto el tiempo de ejecución medio también.

Observemos ahora lo que sucede con los tres algoritmos que quedan y que tienen tiempos de ejecución mayores que los anteriores. El enfriamiento simulado presenta menores tiempos de ejecución para las bases de Wdbc y Libras, esto se debe a que el enfriamiento no siempre realiza las 5000 evaluaciones que tiene como máximo ya que tiene otro criterio de salida, que es no aceptar ningún vecino en la iteración. Entonces en la mayoría de casos el enfriamiento saldrá antes de completar las 5000 evaluaciones con lo que el tiempo de cómputo, que como ya hemos dicho es la mayor parte del cálculo del score, se reduce.

Ahora bien como vemos en el caso de arritmia la Búsqueda Tabú Básica si tiene un tiempo de ejecución menor, eso probablemente se deba a que en esta base las soluciones cambian más de unas a otras y por tanto que se acepten más vecinos. Además cuanto mayor sea el tamaño del problema (número de características) más lento enfriará el algoritmo, o lo que es equivalente, más lenta bajará la probabilidad de aceptar vecinos peores, con lo que esto hace que se haga un mayor número de evaluaciones, pudiendo llegar a las 5000. Además el cálculo de la exponencial para ver si aceptamos un mal vecino o no también conlleva un cómputo.

Y finalmente si nos fijamos en las búsquedas tabú vemos cómo claramente la extendida conlleva más tiempo de cómputo. Dado que estos algoritmos tienen como único criterio de parada el realizar 5000 evaluaciones ambos realizan el mismo número de cálculo de score, ahora bien la búsqueda tabú extendida consume más

tiempo debido al mantenimiento de la lista tabú y los mecanismos de reinicialización. Además observamos otra cosa, y es que como el número de evaluaciones es el mismo sea cuál sea la base de datos actualizada ahora la ejecución en Wdbc es más lenta que en libras, ya que al haber un número mayor de muestras (569 ¿ 360) lo que más pesa es el mecanismo de Leave One Out que se realiza 569 veces en lugar de 360 (la diferencia de tamaño de las componentes de los vectores para calcular las distancias en ambos casos es de $90-30 = 60$ con lo que esto no afecta demasiado al cómputo). No sucede lo mismo para Arritmia que pese a tener un menor número de muestras tarda más esto se debe en el caso de las BT básica a que el cálculo de la distancia puede resultar muy costoso pues ya la diferencia de longitudes entre vectores es de $278-30 = 248$. En el caso de la extendida los mecanismos de reinicialización mediante memoria a largo plazo son más costosos cuanto mayor sea el número de características. Lo que acaba nuestro análisis de los datos.

7 Bibliografía

- Documentación del módulo scikit-learn: <http://scikit-learn.org/stable/documentation.html>
- Documentación de Numpy: <http://docs.scipy.org/doc/>