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

Anabel Gómez Ríos.
DNI: 75929914Z.
E-mail: anabelgrios@correo.ugr.es

10 de mayo de 2016

Curso 2015-2016
Problema de Selección de Características.
Grupo de prácticas: Viernes 17:30-19:30
Quinto curso del Doble Grado en Ingeniería Informática y Matemáticas.

Algoritmos considerados:

1. Greedy Sequential Forward Selection.
2. Búsqueda Multiarranque Básica
3. GRASP
4. Búsqueda Local Reiterada

Índice

1. Descripción del problema

Queremos obtener un sistema que permita clasificar un conjunto de objetos en unas determinadas clases que conocemos previamente. Para ello disponemos de una muestra de dichos objetos ya clasificados y una serie de características para cada objeto.

El problema es, por tanto, construir un clasificador que se comporte lo suficientemente bien fuera de la muestra de la que disponemos, es decir, que clasifique bien nuevos datos. Para hacer esto, lo que hacemos es particionar la muestra en dos subconjuntos, uno que utilizaremos de entrenamiento para que el clasificador aprenda y otro que utilizaremos para test, es decir, para ver cómo de bien se comporta el clasificador que hemos obtenido con el primer subconjunto fuera de los datos de entrenamiento. Además haremos distintas particiones, en concreto 5, y construiremos un clasificador para cada una de ellas. Las particiones serán además proporcionadas (habrá el mismo número de muestras de una clase en el conjunto de entrenamiento y en el de test). Podemos comprobar cómo de bien se comporta cada clasificador porque sabemos en realidad las clases de los objetos que tenemos en la muestra y podemos comparar las verdaderas clases con las que el clasificador obtiene suponiendo que no dispusiéramos de ellas.
Buscamos pues en todo momento optimizar la tasa de acierto del clasificador.
Vamos a utilizar además validación cruzada: es decir, para cada partición en dos subconjuntos primero uno será el de entrenamiento y el otro el de test y después les daremos la vuelta y volveremos a construir un clasificador. La calidad por tanto de cada algoritmo será la media de los porcentajes de clasificación (la tasa de acierto) para estas 10 particiones.

Nos queda describir cómo aprende el clasificador con los datos de entrenamiento. Ya que podemos llegar a tener muchas características de las cuales algunas podrían ser poco o nada significantes, lo que hacemos es elegir un subconjunto de características que describan bien los datos de entrenamiento, de forma que en los datos de test sólo tenemos en cuenta este subconjunto de características a la hora de deducir cuál es la clase de cada nuevo dato. Para esta "deducción"vamos a utilizar la técnica de los 3 vecinos más cercanos (3-NN): buscamos para cada dato los tres vecinos más cercanos en el conjunto de entrenamiento (si el mismo dato al que le vamos a calcular la clase pertenece al conjunto de entrenamiento tenemos que sacarlo previamente del conjunto de entrenamiento. Esto es lo que se llama *leave one out*) teniendo en cuenta las características seleccionadas hasta el momento, consultamos las clases de estos tres vecinos, nos quedamos con la clase que más veces aparezca (o, en caso de empate, con la clase del vecino más cercano) y ésta será la clase del dato. Para obtener la tasa de acierto lo que hacemos es repetir esto para todos los datos, comparar estas clases con las reales y contar el número de veces que acierta.

El cómo exploramos el espacio de búsqueda hasta encontrar la mejor solución (el subconjunto de características que da mayor tasa de acierto) es en lo que se diferencian los distintos algoritmos que vamos a ver en esta práctica.

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

El primer paso común a todos los algoritmos es normalizar los datos de los que disponemos por columnas (es decir, por características) de forma que todas se queden entre 0 y 1 y no haya así preferencias de unas sobre otras.

A continuación se genera una solución inicial, que en el caso de los algoritmos SFS y GRASP serán todas las características a *False* y se irán añadiendo en base a distintos criterios y en el caso de los algoritmos BMB e ILS serán aleatorias. Después, se irán generando vecinos de esta solución (veremos que algunos mutan, otros optimizan con búsqueda local, etc.) y nos quedaremos con la mejor solución obtenida en todos los casos.

Todos los algoritmos tienen una condición de parada común (excepto SFS), que es repetir la generación de solución inicial y posterior optimización (el bucle interno) 25 veces.

2.1. Esquema de representación de soluciones

La representación elegida para las soluciones ha sido la binaria: un vector de N posiciones, donde N es el número de características, en el que aparece *True* o *False* en la posición i según si la característica i-ésima ha sido seleccionada o no, respectivamente. Esta representación es la más sencilla y cómoda cuando no hay restricciones sobre el número de características a elegir, es decir, el número de características que se han de elegir lo aprende cada algoritmo, como es el caso de los algoritmos que vamos a utilizar.

2.2. 3-NN

Como hemos comentado, la técnica para clasificar que vamos a utilizar en todos los algoritmos será el 3NN. Consiste en considerar, para cada dato, la distancia euclídea entre el dato y todos los demás, es decir, entre las características de los datos (excluyéndose él mismo en caso de que estuviéramos preguntando por algún dato dentro del conjunto de entrenamiento) y quedarnos con las clases de los 3 con la distancia más pequeña. La clase del dato en cuestión será de estas tres la que más se repita o, en caso de empate, la clase que corresponda al vecino más cercano.
En cada momento la distancia euclídea se calcula teniendo en cuenta las características que están seleccionadas en el momento, por lo que no podemos tener una matriz fija de distancias, hay que ir calculándolas sobre la marcha.

2.3. Función de evaluación

La función de evaluación será el rendimiento promedio de un clasificador 3-NN en el conjunto de entrenamiento: calcularemos la tasa para cada dato dentro del conjunto de entrenamiento haciendo el leave one out descrito anteriormente y nos quedaremos con la media de las tasas obtenidas. El objetivo será por tanto maximizar esta función. La tasa se calcula como 100*(nº instancias bien clasificadas / nº total de instancias).

El pseudo-código es el siguiente:

```
Obtener el subconjunto de entrenamiento que se va a tener en cuenta según
las características que se estén considerando.
Para cada dato:
    Se saca del conjunto de entrenamiento (leave one out)
    Se calcula la tasa para este dato con el clasificador 3NN
    Se acumula la tasa al resto de tasas
Se divide la acumulación de tasas entre el cardinal del conjunto y se devuelve
```

La función que hace esto la llamaremos `CalcularTasa(conjunto, caracteristicas, knn)` donde `caracteristicas` es la máscara que nos indica cuáles estamos considerando (aquellas que estén a True), `conjunto` es el conjunto de entrenamiento y `knn` es el clasificador 3NN ya previamente entrenado con el conjunto `conjunto`.

En mi caso para el clasificador 3-NN he utilizado el que ha implementado mi compañero Alejandro García Montoro (de mi mismo grupo de prácticas) con python y CUDA, puesto que es mucho más eficiente.

2.4. Operadores comunes: Generación de vecinos

Se considerarán como vecinas todas aquellas soluciones que difieran en la pertenencia o no de una única característica (si se diferencian en más de una entonces no es vecina de la considerada). Por ejemplo, las soluciones (True, True, False) y (True, False, False) son vecinas porque se diferencian en una sola característica, la segunda.
`Flip` será el operador de vecino: recibe un vector que será la máscara y una posición y cambia esa posición en la máscara. El que he hecho yo además lo hace por referencia para evitar las copias e intentar mejorar en eficiencia.

```
Flip(mascara, pos) Empezar:
    # Cambiar la posición pos de la máscara por su negado
    mascara[pos] = not mascara[pos]
    Devolver mascara
Fin
```

2.5. Proceso de generación de soluciones aleatorias

En mi caso para la generación aleatoria de soluciones en los algoritmos BMB e ILS he utilizado la función `choice()` del módulo `random` de `numpy` para python, a la que se le puede pasar el vector de donde obtener las muestras (en mi caso un array de *True* y *False*) y el número de muestras a obtener (suponemos n ahora mismo, será en cada caso el número de características). El muestreo se hace con reemplazamiento:

```
generarSolAleatoria(n) Empezar:
        sol_aleatoria = random.choice(array([True, False]), n)
        Devolver sol_aleatoria
Fin
```

2.6. Búsqueda Local

El algoritmo de búsqueda local implementado ha sido el del primer mejor, es decir, exploramos los vecinos de la solución que tenemos en cada momento y en cuanto obtenemos una mejor nos quedamos con ella y empezamos a generar sus vecinos. Los vecinos se empiezan a generar por una característica aleatoria y a partir de esa característica se van cambiando las demás de forma cíclica en orden: desde la que hemos empezado hasta el final y de nuevo por el principio hasta llegar a la anterior de la primera que habíamos cambiado. Si no nos quedamos con la solución tenemos que dejar la característica que habíamos cambiado como estaba y si nos la quedamos porque tiene una mejor tasa dejamos de generar vecinos de la solución que teníamos, la cambiamos por este mejor vecino y pasamos a generar vecinos de la nueva solución. El algoritmo para cuando da una pasada entera a los vecinos y no ha encontrado una mejor solución que la que tenía.

Veamos el pseudocódigo.
`generarSecuencia(tamaño)` será la función que devuelve el orden en el que se recorrerá el vecindario: empieza desde un número aleatorio (menor que el número de características) y de ahí en orden hasta el total de características y empieza de nuevo hasta el número que se ha generado al principio. Esta función se llamará cada vez que actualicemos la solución y haya que explorar un vecindario nuevo.
Al algoritmo le pasamos como parámetros los datos de entrenamiento, las clases de los datos y el objeto knn que habremos creado previamente entrenándolo con esos datos y clases.

```
busquedaLocal(datos, clases, knn) Empezar:
    caract = solución aleatoria inicial
    tasa actual = calcularTasa(datos, caract, knn)

    while haya mejora en el vecindario y haya menos de 15000 evaluaciones
     hacer:

        posiciones = generarSecuencia(tam(caract))
        for j en posiciones y mientras vuelta_completa = False y haya menos
         de 15000 evaluaciones hacer:
            Flip(caract, j)  # Generamos vecino
            calcularTasa(datos, caract, knn)
            if la tasa del vecino es mejor que la actual:
                actualizamos la tasa actual
                vuelta_completa = False # hemos encontrado mejora antes de
                 # generar todos los vecinos, salimos del bucle y empezamos a
                 # generar vecinos de la nueva solución

            else, volvemos a la antigua solución:
                Flip(caract, j)

            if not vuelta_completa:
                vuelta_completa = True
        Fin for
```

Fin **while**

Devolver caract y tasa actual
Fin

3. Descripción de los algoritmos

3.1. Búsqueda Multiarranque Básica (BMB)

Este algoritmo consiste en generar 25 soluciones aleatorias y optimizarlas con el algoritmo de Búsqueda Local que acabamos de comentar.
El pseudocógido es el siguiente:

```
busquedaMultiBasica(clases, conjunto, knn) Empezar:
    for i entre 0 y 24:
        sol_aleatoria = generarSolAleatoria(num_caracteristicas)
        sol_actual, tasa_actual = busquedaLocal(clases, conjunto, knn)

        if tasa_actual > mejor_tasa:
            # Actualizar la mejor tasa y la mejor solución
            mejor_sol = sol_actual
            mejor_tasa = tasa_actual
        Fin
    Fin
    Devolver mejor_sol, mejor_tasa
Fin
```

3.2. GRASP

Este algoritmo es una modificación del greedy básico con el que vamos a hacer las comparaciones. Ahora, en cada paso, en lugar de elegir la mejor característica, se eligen aquellas que estén por encima de un umbral de calidad y dentro de éstas, se elige una aleatoriamente, hasta que no se obtenga mejora. Una vez tenemos construida la solución inicial greedy aleatorizada, la optimizamos con el algoritmo de Búsqueda Local. Este procedimiento lo vamos a hacer 25 veces y nos vamos a quedar con la mejor solución optimizada con Búsqueda Local de estas 25.

Vamos primero a poner el pseudocódigo de la función que nos va a elegir la característica siguiente en la solución greedy aleatorizada inicial en base a un umbral α:

```
siguienteCaracterista(clases, mascara, conjunto, alpha, knn) Empezar:
    pos = posiciones que no estén ya seleccionadas en la máscara
    tasas = vector vacío donde guardamos las tasas de los vecinos de "mascara"
    for i en pos:
        mascara[pos[i]] = True
        Calculamos con knn la tasa de la máscara con la nueva característica añadida
        tasa[i] = nueva tasa #Guardamos la tasa
        mascara[pos[i]] = False #Volvemos a la máscara original.
    Fin

    Calculamos el umbral como maximo-alpha*(maximo-minimo)
     donde maximo y minimo son el máximo y el mínimo del vector de tasas

    Seleccionamos los vecinos cuya tasa esté por encima de ese umbral
```

```
    Elegimos un vecino aleatorio de los seleccionados

    Devolver la tasa del vecino y la posición de la máscara original
     que hay que modificar para llegar a él.
Fin
```

Vamos ahora con el pseudocódigo del algoritmo GRASP, en el que vamos a hacer uso de la función anterior:

```
GRASP(clases, conjunto, knn) Empezar:
    for i entre 0 y 24:
        caract = vector de "False"
        while haya mejora:
            nueva_tasa, mejor_pos = siguienteCaracteristica(clases,
             caracteristicas, conjunto, 0.3, knn)

            if nueva_tasa > tasa_actual:
                Actualizamos tasa actual
                caract[mejor_pos] = True
            else:
                mejora = False y paramos de añadir características
            Fin
        Fin

        # Optimizamos con Búsqueda Local
        sol_opt, tasa_opt = busquedaLocal(clases, conjunto, knn)
        if tasa_opt > mejor_tasa:
            Actualizar mejor_tasa y mejor_solucion
        Fin
    Fin

    Devolver mejor_solucion, mejor_tasa
Fin
```

3.3. Búsqueda Local Reiterada (ILS)

Este algoritmo consisten en generar una solución aleatoria, optimizarla con el algoritmo de Búsqueda Local, y repetir lo siguiente 24 veces: mutar la mejor solución encontrada hasta el momento y optimizarla de nuevo con Búsqueda Local. Se devolverá la mejor solución encontrada en todo este proceso.

Veamos primero el pseudocódigo para la función con la que vamos a mutar una solución, consistente en elegir t características aleatorias de la solución y cambiarlas con el operador `Flip()`. En mi caso lo que he hecho ha sido hacer una permutación del 0 al n, donde n es el número de características en cada caso, y cambiar las t primeras posiciones del vector que he permutado en la solución.

```
mutar(solucion, n, t) Empezar:
    mutada = copia de solucion
```

```
    pos = permutación del vector [0,...,n]
    for i entre 0 y t:
        Flip(mutada, pos[i])
    Fin
    Devolver mutada
Fin
```

Veamos ahora el pseudocódigo para el algoritmo ILS. Para que se mute siempre al menos una característica cuando el número de características es bajo, utilizo la función `ceil(x)` para obtener el entero por encima del valor x.

```
ILS(clases, conjunto, knn) Empezar:
    n = número de características
    t = ceil(0.1*n)
    sol_aleatoria = generarSolAleatoria(n)
    # Inicializamos la mejor solución a la optimización de la aleatoria
    mejor_sol, mejor_tasa = busquedaLocal(clases, conjunto, sol_aleatoria, knn)
    for i entre 0 y 23:
        mutada = mutar(mejor_sol, n, t)
        # Optimizamos la mutada
        sol_actual, tasa_actual = busquedaLocal(clases, conjunto, mutada, knn)
        if tasa_actual > mejor_tasa:
            Actualizamos mejor_tasa y mejor_sol
        Fin
    Fin
    Devolver mejor_sol, mejor_tasa
Fin
```

4. Breve descripción del algoritmo de comparación

El algoritmo de comparación seleccionado ha sido el greedy Sequential Forward Selection (SFS), que parte de una solución inicial en la que no hay ninguna característica seleccionada y se va quedando en cada iteración con la característica con la que se obtiene la mejor tasa. El algoritmo no para mientras se encuentre mejora añadiendo alguna característica.
He implementado una función que me devuelve, para una máscara determinada, la característica más prometedora que se puede obtener, cuyo pseudocódigo es el siguiente:

```
caractMasPrometedora(mascara) Empezar:
    posiciones = posiciones que no estén seleccionadas de la mascara
    for i en posiciones:
        mascara[i] = True
        Se calcula la tasa con la nueva característica añadida
        if nueva tasa > mejor tasa:
            Se actualiza la mejor tasa
            Se actualiza la mejor posición
        Fin for
    Devuelve mejor tasa y mejor pos
Fin
```

Con esto, el pseudocódigo del algoritmo SFS es:

```
algoritmoSFS(datos, clases) Empezar:
    caract = solución inicial inicializada a False
    tasa actual = 0
    mejora = True
    while haya mejora:
        Se calcula la tasa y la mejor posición con caractMasPrometedora
        if nueva tasa > tasa actual:
            Se actualiza la tasa actual
            Se pone a True la característica en mejor posición
        else:
            mejora = False          #No ha habido mejora: paramos

    Fin while
    Devuelve caract y tasa
Fin
```

5. Procedimiento considerado para desarrollar la práctica

La práctica la he desarrollado en python junto con dos módulos de python: numpy y scipy (todos disponibles en Linux pero hay que instalarlos previamente), y un módulo con el 3NN con leave one out desarrollado por mi compañero Alejandro García Montoro. Numpy permite manejar arrays de forma más rápida y scipy leer ficheros en formato arff. El resto de la práctica lo he hecho yo. Para generar números aleatorios utilizo el `random` de numpy (al que le paso previamente la semilla) y para medir tiempos `time` de python.

Para hacer las particiones igualadas lo que he hecho ha sido quedarme, para cada clase, con los índices de los datos pertenecientes a esa clase, hacerles una permutación aleatoria, partir por la mitad y quedarme con la primera mitad para entrenamiento y la segunda mitad para test.

Para ejecutar la práctica es necesario que los ficheros de datos estén en el mismo directorio en el que se encuentran los ficheros .py y ejecutar desde línea de comandos `practica2.py semilla base_de_datos algoritmo` donde `base_de_datos` será 1 si se quiere ejecutar con wdbc, 2 con movement libras y 3 con arritmia y `algoritmo` será 1 si se quiere ejecutar SFS, 2 para BMB, 3 para GRASP, 4 para ILS y 5 para KNN.

6. Experimentos y análisis de resultados

6.1. Descripción de los casos del problema empleados

Los parámetros de los algoritmos, como el parámetro α para el umbral en GRASP, el parámetro t en ILS para decidir cuántas características mutar o el tope de evaluaciones a realizar, los he dejado como se recomendaba en el guión de prácticas.
Las particiones que se hacen dependen de la semilla que se le pase al generador de números aleatorios de numpy, así como las soluciones iniciales que se generan y todo lo relativo a probabilidades. La semilla, como ya he comentado, se le pasa al programa por línea de comandos y es el único parámetro que he cambiado de una base de datos a otra. Para todos los algoritmos, el primer par de particiones (que en realidad es la misma partición pero primero se utiliza una mitad para entrenamiento y la otra para test y después al revés) lo he generado con la semilla 567891234, el segundo par está generado con la semilla 123456789, el tercer par con 11235813, el cuarto par con 27182818 y el quinto par con 1414213, de forma que entre los distintos algoritmos las particiones utilizadas han sido las mismas para poder comparar resultados entre unos y otros.

6.2. Resultados

Para cada algoritmo se está midiendo el tiempo, en segundos, que tarda en encontrar el subconjunto de características óptimo más lo que tarda en evaluar esta solución. Para el caso del 3NN, como lo estamos lanzando con una máscara entera a True, el tiempo es únicamente el que tarda en hacer la evaluación, mientras que la tasa de reducción es cero por tener todas las características seleccionadas.

Cuadro 1: Resultados SFS

	Wdbc				Movement Libras				Arrhythmia			
	%_clas train	%_clas test	%_red	T	%_clas train	%_clas test	%_red	T	%_clas train	%_clas test	%_red	T
Partición 1-1	98,5915	91,5789	90,0000	0,3417	67,7778	67,2222	91,1111	1,7344	80,7292	68,5567	98,2014	3,5072
Partición 1-2	95,7895	94,0141	83,3333	0,5213	77,2222	70,5556	92,2222	1,5277	76,8041	69,2708	98,5612	2,7068
Partición 2-1	95,4225	93,3333	90,0000	0,3200	81,6667	72,7778	88,8889	2,2433	73,9583	64,9485	98,2014	3,4125
Partición 2-2	98,2456	91,9014	83,3333	0,4852	68,8889	66,6667	90,0000	1,9666	78,8660	68,2292	98,2014	3,3255
Partición 3-1	97,8873	96,4912	76,6667	0,6666	77,2222	68,8889	91,1111	1,6781	76,0417	65,9794	98,2014	3,4072
Partición 3-2	97,5439	93,6620	87,0000	0,3917	77,7778	75,0000	90,0000	1,9069	79,3814	71,3542	97,1223	5,5038
Partición 4-1	96,8310	97,8947	87,0000	0,3932	71,6667	76,1111	90,0000	2,0013	80,7292	73,7113	97,8417	4,1868
Partición 4-2	97,5439	95,4225	83,3333	0,4757	77,2222	62,2222	92,2222	1,4977	78,3505	73,4375	97,8417	4,0386
Partición 5-1	97,5352	95,7895	90,0000	0,3125	81,6667	70,0000	86,6667	2,8394	81,2500	71,6495	97,4820	4,9066
Partición 5-2	96,1404	94,7183	83,3333	0,4935	77,7778	68,8889	85,5556	3,1619	75,2577	65,6250	98,2014	3,3356
Media	97,1531	94,4806	85,4000	0,4401	75,8889	69,8333	89,7778	2,0557	78,1368	69,2762	97,9856	3,8331

Cuadro 2: Resultados BMB

	Wdbc				Movement Libras				Arrhythmia			
	%_clas train	%_clas test	%_red	T	%_clas train	%_clas test	%_red	T	%_clas train	%_clas test	%_red	T
Partición 1-1	99,2958	95,7895	46,6667	4,4865	69,4444	73,8889	37,7778	18,4451	72,9167	68,0412	47,4820	244,9250
Partición 1-2	96,8421	96,8310	46,6667	5,1123	79,4444	77,7778	52,2222	24,9961	69,0722	62,5000	46,7626	262,5724
Partición 2-1	97,1831	97,1930	36,6667	4,4819	76,1111	78,3333	47,7778	16,9308	68,2292	61,8557	46,7626	319,5488
Partición 2-2	98,9474	95,4225	60,0000	3,4035	73,3333	73,3333	47,7778	21,8960	72,1649	67,1875	51,7986	190,0089
Partición 3-1	98,2394	96,8421	40,0000	3,8526	76,6667	76,1111	45,5556	23,0397	71,3542	64,4330	48,5612	263,3452
Partición 3-2	97,8947	94,0141	56,6667	3,8266	77,2222	76,6667	48,8889	18,3171	71,1340	63,0208	53,9568	262,7427
Partición 4-1	97,5352	97,5439	56,6667	3,6765	71,6667	77,7778	48,8889	22,4129	69,2708	61,3402	50,3597	227,7444
Partición 4-2	97,8947	96,1268	33,3333	4,2255	78,8889	66,1111	50,0000	17,7298	71,1340	65,6250	53,2374	280,0143
Partición 5-1	98,2394	94,7368	46,6667	5,0359	77,7778	70,0000	48,8889	18,7127	70,8333	63,4021	51,7986	264,0515
Partición 5-2	98,5965	96,8310	50,0000	4,1433	81,6667	73,8889	56,6667	19,7715	69,5876	65,1042	51,7986	196,1780
Media	98,0668	96,1331	47,3333	4,2245	76,2222	74,3889	48,4444	20,2252	70,5697	64,2510	50,2518	251,1131

Cuadro 3: Resultados GRASP

	Wdbc				Movement Libras				Arrhythmia			
	%_clas train	%_clas test	%_red	T	%_clas train	%_clas test	%_red	T	%_clas train	%_clas test	%_red	T
Partición 1-1	99,2958	93,6842	80,0000	12,2254	73,3333	72,2222	84,4444	54,1297	83,8542	70,1031	96,0432	61,5874
Partición 1-2	96,8421	93,3099	63,3333	11,7953	81,6667	75,0000	87,7778	54,2804	78,3505	76,0417	94,6043	41,2412
Partición 2-1	97,5352	97,5439	60,0000	10,1512	80,0000	80,0000	83,3333	45,4530	77,6042	65,4639	95,6835	46,7988
Partición 2-2	98,9474	95,7746	66,6667	11,7204	76,1111	72,2222	83,3333	52,1505	84,5361	65,6250	94,9640	54,8408
Partición 3-1	98,5915	95,4386	70,0000	10,1279	77,2222	71,6667	85,5556	53,8447	78,6458	64,4330	96,4029	42,7548
Partición 3-2	97,5439	95,0704	0,9900	10,4591	80,0000	75,5556	85,5556	47,9905	77,8351	67,7083	92,4460	48,6333
Partición 4-1	97,5352	97,5439	66,6667	11,0216	73,8889	75,5556	84,4444	41,8452	78,6458	70,6186	95,3237	45,8640
Partición 4-2	97,8947	93,6620	76,6667	10,7161	82,7778	63,8889	81,1111	52,7695	81,9588	66,6667	93,5252	51,6532
Partición 5-1	98,5915	95,7895	80,0000	10,9177	80,5556	70,0000	80,0000	47,3751	83,8542	69,5876	94,2446	65,8288
Partición 5-2	97,8947	96,8310	73,3333	12,7110	82,7778	75,0000	86,6667	55,4414	77,8351	68,7500	95,6835	39,9847
Media	98,0672	95,4648	63,7657	11,1846	78,8333	73,1111	84,2222	50,5280	80,3120	68,4998	94,8921	49,9187

Cuadro 4: Resultados ILS

	Wdbc				Movement Libras				Arrhythmia			
	%_clas train	%_clas test	%_red	T	%_clas train	%_clas test	%_red	T	%_clas train	%_clas test	%_red	T
Partición 1-1	99,2958	93,6842	40,0000	4,1880	72,2222	72,7778	56,6667	15,1964	79,1667	65,9794	51,7986	296,0245
Partición 1-2	97,1930	97,5352	53,3333	8,1401	80,0000	72,7778	53,3333	32,1731	74,7423	64,0625	54,3165	500,2861
Partición 2-1	97,5352	96,4912	60,0000	3,0758	77,2222	78,8889	51,1111	12,8567	71,3542	63,9175	52,8777	199,2703
Partición 2-2	98,5965	94,3662	53,3333	7,1185	75,0000	72,7778	41,1111	32,2054	73,1959	67,1875	48,2014	447,1106
Partición 3-1	98,2394	96,1404	30,0000	4,3338	77,2222	75,0000	56,6667	18,5859	70,8333	60,8247	44,6043	253,7177
Partición 3-2	97,8947	95,7746	56,6667	7,0367	77,2222	75,0000	58,8889	33,2284	68,0412	65,1042	49,2806	471,2888
Partición 4-1	97,8873	96,8421	63,3333	3,4461	74,4444	77,2222	50,0000	15,9655	72,3958	64,4330	45,3237	256,7803
Partición 4-2	97,5439	97,1831	50,0000	6,6594	81,1111	67,7778	55,5556	34,1182	69,5876	62,5000	47,1223	485,4405
Partición 5-1	97,8873	94,7368	53,3333	3,4796	78,8889	71,6667	45,5556	15,1949	74,4792	63,9175	5,1439	308,4218
Partición 5-2	98,2456	97,1831	43,3333	8,3396	82,2222	73,3333	56,6667	30,0778	68,5567	65,1042	54,3165	544,5411
Media	98,0319	95,9937	50,3333	5,5818	77,5556	73,7222	52,5556	23,9602	72,2353	64,3030	45,2986	376,2882

Cuadro 5: Resultados KNN

	Wdbc				Movement Libras				Arrhythmia			
	%_clas train	%_clas test	%_red	T	%_clas train	%_clas test	%_red	T	%_clas train	%_clas test	%_red	T
Partición 1-1	98,5915	95,7895	0,0000	0,0131	63,8889	72,2222	0,0000	0,0339	65,1042	63,9175	0,0000	0,1146
Partición 1-2	98,5915	97,8873	0,0000	0,0123	63,8889	75,5556	0,0000	0,0333	65,1042	59,8958	0,0000	0,0853
Partición 2-1	95,0704	97,8947	0,0000	0,0131	69,4444	79,4444	0,0000	0,0340	61,4583	60,8247	0,0000	0,1145
Partición 2-2	95,0704	95,4225	0,0000	0,0123	69,4444	70,0000	0,0000	0,0333	61,4583	63,5417	0,0000	0,0854
Partición 3-1	96,1268	96,8421	0,0000	0,0131	67,2222	76,6667	0,0000	0,0343	63,0208	60,3093	0,0000	0,1155
Partición 3-2	96,1268	96,4789	0,0000	0,0122	67,2222	77,7778	0,0000	0,0330	63,0208	67,7083	0,0000	0,0853
Partición 4-1	95,7746	96,8421	0,0000	0,0131	67,7778	80,5556	0,0000	0,0339	61,9792	60,8247	0,0000	0,1146
Partición 4-2	95,7746	95,4225	0,0000	0,0122	67,7778	68,8889	0,0000	0,0332	61,9792	63,0208	0,0000	0,0853
Partición 5-1	94,7183	97,1930	0,0000	0,0165	70,5556	68,8889	0,0000	0,0339	64,5833	64,4330	0,0000	0,1145
Partición 5-2	94,7183	97,5352	0,0000	0,0134	70,5556	71,1111	0,0000	0,0332	64,5833	65,6250	0,0000	0,0853
Media	96,0563	96,7308	0,0000	0,0131	67,7778	74,1111	0,0000	0,0336	63,2292	63,0101	0,0000	0,1000

6.3. Análisis de los resultados

A continuación vemos la tabla comparativa, que tiene la última fila (las medias) de todas las tablas anteriores, junto con gráficas para cada variable a medir y para cada algoritmo:

Cuadro 6: Comparativa

	Wdbc				Movement Libras				Arrhythmia			
	%_clas train	%_clas test	%_red	T	%_clas train	%_clas test	%_red	T	%_clas train	%_clas test	%_red	T
3-NN	96,0563	96,7308	0,0000	0,0131	67,7778	74,1111	0,0000	0,0336	63,2292	63,0101	0,0000	0,1000
SFS	97,1531	94,4806	85,4000	0,4401	75,8889	69,8333	89,7778	2,0557	78,1368	69,2762	97,9856	3,8331
BMB	98,0668	96,1331	47,3333	4,2245	76,2222	74,3889	48,4444	20,2252	70,5697	64,2510	50,2518	251,1131
GRASP	98,0672	95,4648	63,7657	11,1846	78,8333	73,1111	84,2222	50,5280	80,3120	68,4998	94,8921	49,9187
ILS	98,0319	95,9937	50,3333	5,5818	77,5556	73,7222	52,5556	23,9602	72,2353	64,3030	45,2986	376,2882

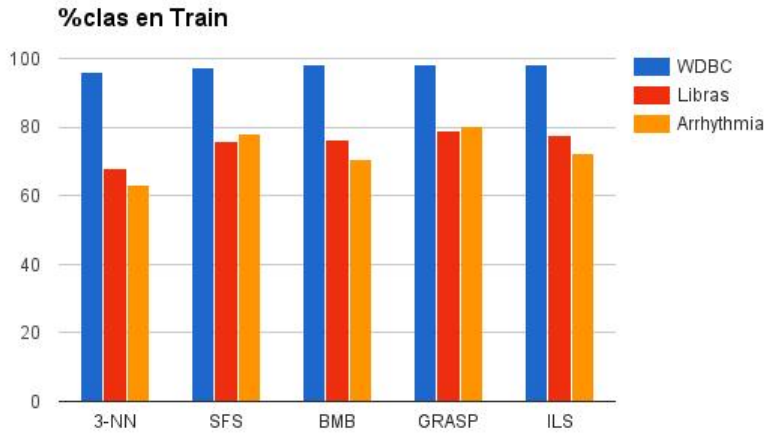

Figura 1: Tasa de acierto en el conjunto train

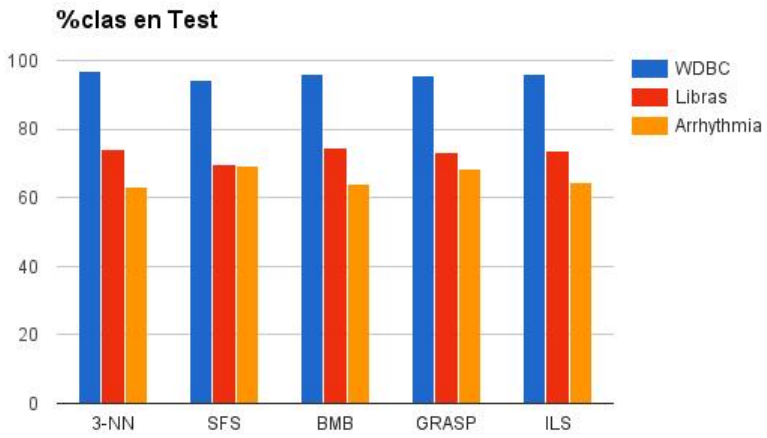

Figura 2: Tasa de acierto en el conjunto test

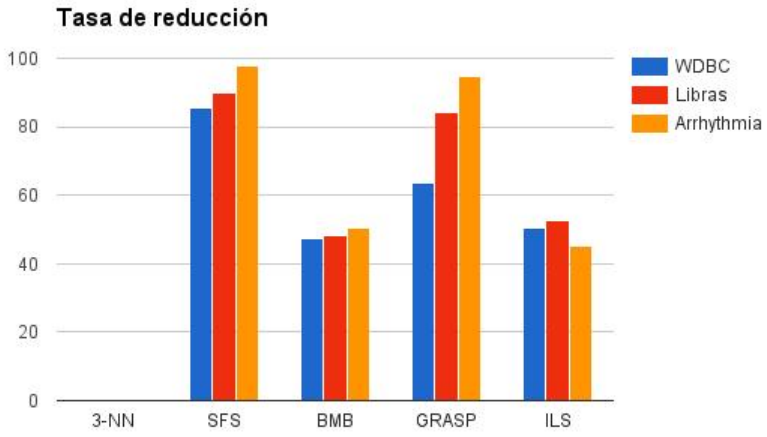

Figura 3: Tasa de reducción

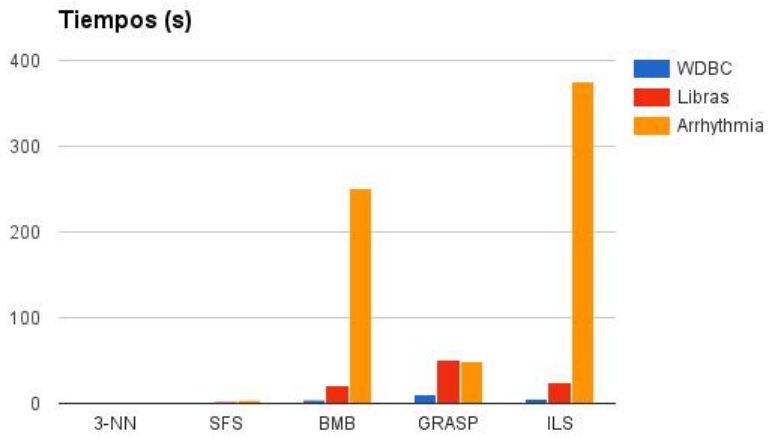

Figura 4: Tiempo en segundos

En este caso la condición de parada es para todos los algoritmos la misma (excepto para `SFS`), repetir el bucle interno 25 veces, por lo que el tiempo variará según lo que cada algoritmo haga en ese bucle interno.
En un principio podemos pensar que tiene sentido que el que menos tiempo tarde, entre `BMB`, `GRASP` e `ILS`, sea `BMB`, ya que lo único que repite 25 veces es la generación de una solución aleatoria inicial y la optimización de cada solución aleatoria con el método de `Búsqueda Local`, que de la práctica pasada, era el que menos tardaba. Después, `ILS` lo que hace es añadir a `BMB` una mutación de la mejor solución encontrada hasta el momento antes de optimizarla con `Búsqueda Local`, es decir, que cambia las últimas 24 generaciones aleatorias de `BMB` por mutaciones, por lo que los tiempos deberían ser un poco mayores pero parecidos (teniendo en cuenta que se tarda más en mutar que en generar aleatoriamente). Por último, el algoritmo `GRASP` es el más complicado en el sentido de que tiene más carga computacionalmente en principio, ya que lo que se repite 25 veces es una modificación del algoritmo `greedy` básico en el que en vez de coger la mejor característica en cada paso se elige una dentro de un cierto umbral de calidad y posteriormente se optimiza con `Búsqueda Local`, por lo que debería ser el que más tiempo tardara.
Esto es así efectivamente en las bases de datos con pocas características, como en Wdbc con 30 características o en Libras con 91 características, donde los tiempos son parecidos unos a otros. Sin embargo, para arritmia, donde tenemos 271 características, `GRASP` tarda bastante menos que `BMB` e `ILS`: `GRASP` tarda 49 segundos de media, `BMB` alrededor de 4 minutos de media e `ILS` más de 6 minutos de media por partición. De hecho, además, `GRASP` tarda un poco menos en arritmia que en Libras, como podemos ver en la gráfica de tiempos, lo que no pasa para ningún otro algoritmo.
Esto puede ser debido a que con la solución aleatoria con la que se empieza en `GRASP` ya no se encuentra mucha más mejora porque hay características que no influyen a la hora de elegir clase, de modo que la parte `greedy` modificada termina pronto dejando una solución bastante buena que la `Búsqueda Local` tampoco es capaz de mejorar mucho más (recordemos que la `Búsqueda Local` se queda en el primer óptimo local que encuentra, con lo que si ya estamos

cerca de uno, no tardará en llegar a él), de forma que las Búsquedas Locales que hace GRASP duran menos que las que hacen *BMB* e ILS, ya que en BMB hay que optimizar una solución aleatoria que no tiene por qué ser buena y en *ILS* hay que optimizar una mutación, que ha podido quedar lejos de un óptimo local, y de hecho según los tiempos, vemos que en efecto queda lejos de un óptimo local.

Obviamente, todos tardan más que el algoritmo de comparación SFS ya que este sólo se repite una vez y no tiene optimización posterior (tampoco tendría sentido ya que hemos ido eligiendo características hasta que dejamos de obtener mejora).

Entrando ahora en comparar las tasas de acierto en el conjunto de train entre todos los algoritmos, GRASP es el que mejor tasa obtiene siempre, aunque con menos diferencia en Wdbc, hay más de un punto de diferencia en libras y casi dos en arritmia. Los demás algoritmos hay unos que funcionan mejor que otros según la base de datos, como BMB, que funciona mejor en Wdbc que ILS, aunque en realidad no hay mucha diferencia, mientras que ILS sí queda más de un punto por encima en Libras casi dos en Arritmia, aunque en Arritmia tanto ILS como BMB quedan por debajo más de 5 puntos del SFS.

En cuanto a la tasa de acierto en test varía un poco con respecto a lo comentado para train, ya que ahora GRASP sólo tiene la mejor tasa en Arritmia, mientras que BMB es el que mejor tasa tiene para Wdbc y Libras, aunque las diferencias son menores que para los conjuntos de train. SFS es peor que BMB, ILS y GRASP excepto en Arritmia, donde sólo queda por debajo de GRASP, cuando en SFS estamos añadiendo características hasta que no hay mejora mientras que en BMB e ILS se comienza con soluciones aleatorias, lo que nos lleva a pensar que importa el orden en el que se elijan las características y en el SFS siempre se escoge primero la que más ganancia da, que es algo que no hace GRASP y de hecho este tiene tasas mejores.

En cuanto a la tasa de reducción, es normal que todas se queden por debajo del SFS por lo que acabamos de comentar y que el siguiente con mayor tasa de reducción sea GRASP que va añadiendo también característica a característica, mientras que el resto de algoritmos si tienen una característica que no esté aportando nada (o incluso está perjudicando) no nos estamos preocupando en eliminarla y tendrán más características al partir de soluciones aleatorias.

Con respecto al 3NN al principio parece raro que tenga menos tasa de acierto que los demás (en la mayoría de los casos) puesto que está considerando todas las características. Sin embargo como acabamos de comentar, no todas las características tienen que ser buenas definiendo una clase, además de que puede haber ruidos que hayan afectado más a unas que a otras. Es por esto que es normal que tenga un poco menos de acierto y que no haya mucha diferencia entre las tasas de acierto para entrenamiento y para test y además que la de entrenamiento no sea mejor que la de test, puesto que en este algoritmo en realidad las dos serían de test ya que no estamos utilizando ninguno de los dos conjuntos para aprender al estar utilizando todas las características.

Con todo esto, si tuviéramos que elegir uno de estos algoritmos para este problema, sería GRASP, ya que es el que mejor relación tasa/tiempo tiene, sobre todo si tenemos bastantes

características, que es donde más se nota, mientras que si hay pocas la diferencia es mucho mejor, pero éste algoritmo sigue siendo ligeramente mejor.

7. Bibliografía

1. Scipy para leer ficheros arff aquí.
2. Documentación de numpy aquí.
3. Ficheros para el 3NN aquí (aunque se encuentran también en los ficheros de mi práctica para poder utilizarlos).