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

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

9 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. Algoritmo Genético Generacional.
3. Algoritmo Genético Estacionario.

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 óptimo, aquel 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 obtiene el clasificador 3NN
    Se calcula la tasa para este dato
    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).

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 = np.random.choice(np.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 AGG

	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	99,6479	94,3860	60,0000	78,5619	72,2222	73,8889	45,5556	178,7664	80,7292	68,0412	52,1583	748,3829
Partición 1-2	97,5439	96,8310	40,0000	115,9536	81,1111	72,7778	52,2222	154,2362	72,6804	59,8958	51,0791	739,2541
Partición 2-1	97,1831	98,5965	40,0000	111,5717	78,8889	78,8889	48,8889	166,4796	72,9167	61,3402	52,8777	830,0614
Partición 2-2	98,9474	94,3662	50,0000	85,1722	78,3333	70,5556	47,7778	168,8781	79,8969	67,1875	50,7194	693,1010
Partición 3-1	98,5916	94,3860	56,6667	90,4621	78,3333	76,1111	54,4444	150,3366	77,0833	66,4948	51,4388	855,1478
Partición 3-2	97,8947	95,7746	50,0000	95,9497	83,3333	75,0000	58,8889	140,4027	77,3196	69,2708	47,1223	786,8225
Partición 4-1	97,5352	95,7895	40,0000	97,1104	77,2222	81,6667	61,1111	134,1479	76,0417	63,4021	52,1583	792,3615
Partición 4-2	97,8947	95,4225	43,3333	102,4731	82,7778	66,6667	58,8889	138,8140	76,2887	66,6667	55,3957	661,6351
Partición 5-1	98,5916	96,4912	70,0000	57,9619	80,5556	70,5556	51,1111	161,2022	78,6458	63,9175	49,6403	826,4084
Partición 5-2	98,5965	97,8873	40,0000	108,1109	83,3333	70,0000	53,3333	153,5616	73,7113	67,1875	54,3165	650,4075
Media	98,2427	95,9931	49,0000	94,3327	79,6111	73,6111	53,2222	154,6825	76,5314	65,3404	51,6906	758,3582

Cuadro 3: Resultados AGE

	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	95,0877	43,3333	111,9277	71,6667	76,1111	45,5556	182,9612	77,0833	67,5258	45,3237	929,8706
Partición 1-2	97,5439	97,5352	40,0000	110,6645	81,1111	74,4444	54,4444	150,1978	70,1031	61,4583	50,0000	737,2140
Partición 2-1	97,8873	96,1404	23,3333	142,1192	78,3333	78,3333	51,1111	159,6853	72,9167	63,4021	25,5396	1146,5426
Partición 2-2	99,2982	95,0704	56,6667	101,8055	77,2222	72,2222	45,5556	176,3094	77,3196	64,5833	25,5396	998,2010
Partición 3-1	98,9437	97,1930	43,3333	100,1677	78,8889	77,2222	56,6667	143,8691	73,9583	65,9794	45,6835	919,9947
Partición 3-2	98,2456	96,1268	33,3333	111,6887	81,1111	72,7778	45,5556	176,5872	73,7113	65,6250	24,4604	1036,6571
Partición 4-1	97,8873	94,3860	23,3333	133,0441	76,6667	80,5556	38,8889	212,0066	77,6042	62,3711	22,3022	1154,9914
Partición 4-2	98,2456	95,7746	26,6667	130,8645	81,6667	66,6667	55,5556	147,1362	76,2887	67,1875	34,8921	880,4354
Partición 5-1	97,8873	96,8421	36,6667	116,6638	80,0000	68,8889	58,8889	142,2323	78,6458	64,4330	43,5252	894,1407
Partición 5-2	98,5965	97,8873	36,6667	111,9023	83,8889	71,1111	58,8889	137,9519	71,1340	63,5417	19,4245	933,6901
Media	98,3831	96,2043	36,3333	117,0848	79,0556	73,8333	51,1111	162,8937	74,8765	64,6107	33,6691	963,1738

Cuadro 4: 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 5: 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
AGG	98,2427	95,9931	49,0000	94,3327	79,6111	73,6111	53,2222	154,6825	76,5314	65,3404	51,6906	758,3582
AGE	98,3831	96,2043	36,3333	117,0848	79,0556	73,8333	51,1111	162,8937	74,8765	64,6107	33,6691	963,1738

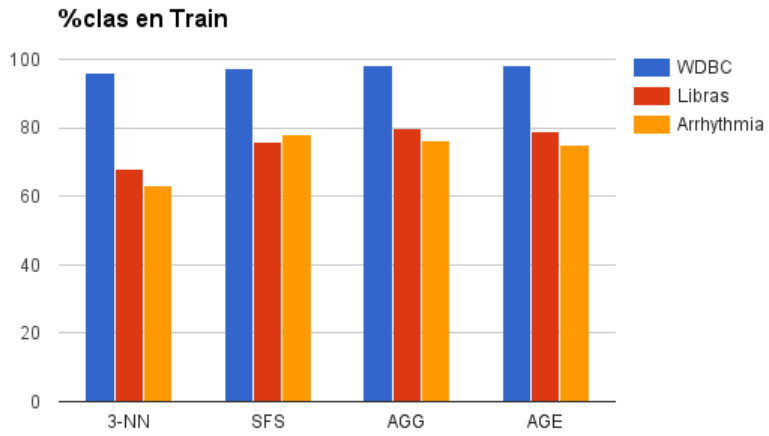

Figura 1: Tasa de acierto en el conjunto train

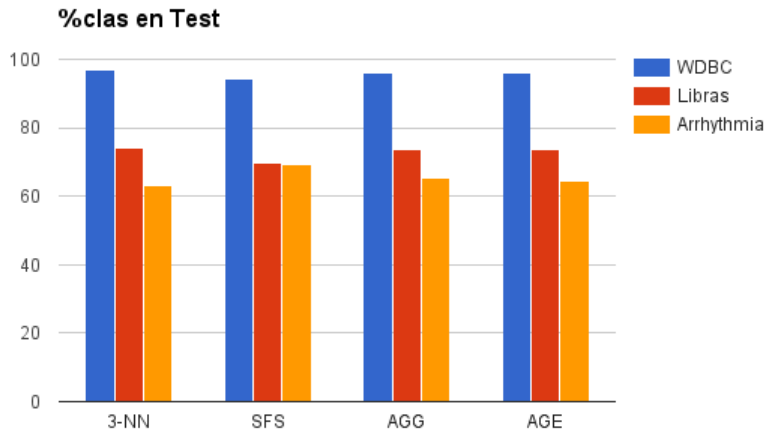

Figura 2: Tasa de acierto en el conjunto test

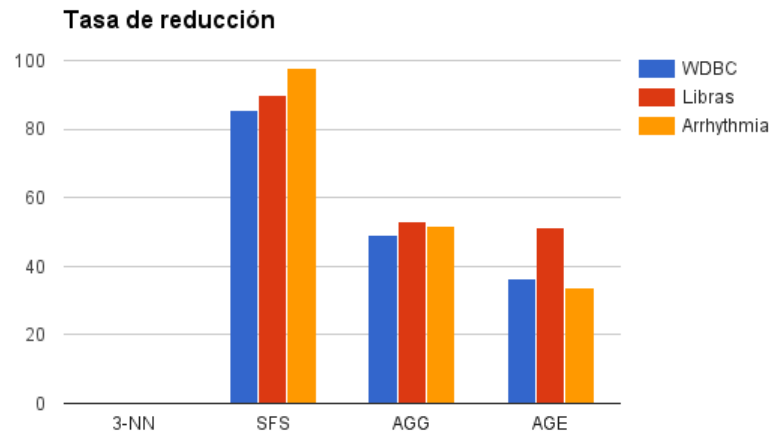

Figura 3: Tasa de reducción

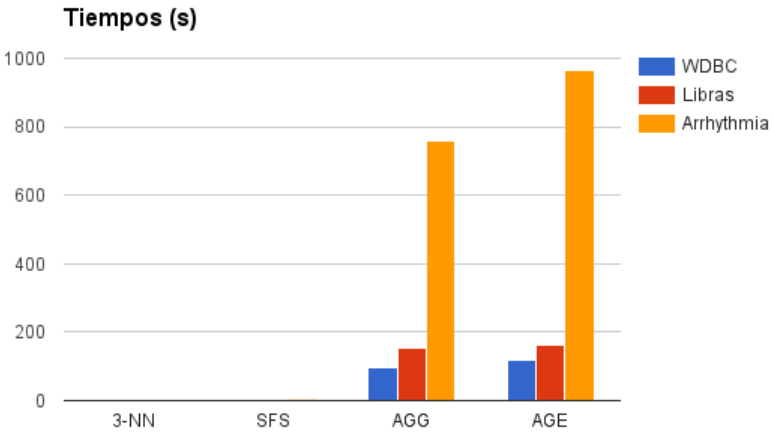

Figura 4: Tiempo en segundos

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).