

Metaheurísticas:
Práctica 5.b: Búsquedas Híbridas para el Problema de
Selección de Características.

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

11 de julio 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. Algoritmo Memético (10,1.0).
2. Algoritmo Memético (10,0.1).
3. Algoritmo Memético (10,0.1-mej).

Índice

1. Descripción del problema	3
2. Descripción de la aplicación de los algoritmos empleados al problema	4
2.1. Esquema de representación de soluciones	4
2.2. 3-NN	4
2.3. Función de evaluación	5
2.4. Proceso de generación de soluciones aleatorias	5
2.5. Mecanismo de selección considerado	6
2.6. Operadores comunes: Cruce	6
2.7. Operadores comunes: Mutación	7
2.8. Búsqueda Local	7
3. Descripción de los algoritmos	9
3.1. Algoritmo Genético Generacional	9
3.2. Algoritmo Memético (10, 1.0) (AM1)	11
3.3. Algoritmo Memético (10, 0.1) (AM2)	11
3.4. Algoritmo Memético (10, 0.1-mej) (AM3)	11
4. Breve descripción del algoritmo de comparación	12
5. Procedimiento considerado para desarrollar la práctica	13
6. Experimentos y análisis de resultados	14
6.1. Descripción de los casos del problema empleados	14
6.2. Resultados	14
6.3. Análisis de los resultados	16
7. Bibliografía	16

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 población inicial, consistente en 10 cromosomas generados de forma aleatoria. Después, en cada iteración de los algoritmos, se irá seleccionando una parte de la población que será también de 10 cromosomas según una cierta probabilidad. Una vez tenemos la nueva población se realizarán mutaciones según otra probabilidad y se irá aplicando una búsqueda local cada 10 generaciones a algunos o todos de los cromosomas resultantes después de hacer el reemplazamiento. Una vez termine todo el algoritmo nos quedaremos con la mejor solución obtenida en todos los casos, que será aquel cromosoma con la mayor tasa.

Los tres algoritmos, que difieren en a qué parte de la población se le aplica la búsqueda local, tienen una condición de parada común, que es llamar a la función de evaluación 15000 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.

En esta práctica además, vamos a ir guardando junto con cada solución en un array estructurado su tasa, para poder ordenar las poblaciones según su tasa.

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 $100 \cdot (\text{n}^\circ \text{ instancias bien clasificadas} / \text{n}^\circ \text{ total de instancias})$.

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, características, knn)` donde `características` 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. Proceso de generación de soluciones aleatorias

En mi caso para la generación aleatoria de soluciones (para generar la población inicial) 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

Como aquí la generación de soluciones aleatorias se hace para generar la población inicial, el pseudocódigo que genera la población inicial es el siguiente:

```
generarPoblacionInicial(numPob, numCar, conjunto, clases, knn) Empezar:  
    for i desde 0 hasta numCar-1:  
        poblacion[i] = generarSolAleatoria(numCar)  
    Fin  
    poblacionEval = evaluarPoblacionInicial(conjunto, clases, poblacion, knn)  
    Devolver poblacionEval
```

Fin

Donde `evaluarPoblacionInicial(...)` evalúa la población que se le pasa por parámetros y devuelve un nuevo vector donde cada cromosoma aparece junto con su tasa.

2.5. Mecanismo de selección considerado

Para el mecanismo de selección he usado el torneo binario, que consiste en elegir aleatoriamente a dos cromosomas de la población y quedarnos con el que mayor tasa de acierto tenga. En mi caso lo que devuelvo es la posición en la que el mejor de los dos cromosomas se encuentra en el vector `poblacion`. Veamos el pseudocódigo, en el que llamo `elegirAleatorio(poblacion)` a la elección aleatoria de un cromosoma de la población `poblacion` y `tasa(cromosoma)` a la función que devuelve la tasa del cromosoma `cromosoma`, que ya tenemos almacenada en `poblacion`:

```
torneo(poblacion) Empezar:
    crom1 = elegirAleatorio(poblacion)
    crom2 = elegirAleatorio(poblacion)

    if tasa(crom1) > tasa(crom2):
        Devolver crom1
    else:
        Devolver crom2
Fin
```

2.6. Operadores comunes: Cruce

El operador de cruce que he utilizado ha sido el HUX, que consiste en comparar los dos padres y si ambos tienen en el gen i a *True* o *False* ambos hijos tendrán ese gen a *True* o *False*, respectivamente, y si uno de los padres tienen el gen i y el otro no, entonces un hijo lo tendrá y el otro no, de forma aleatoria. Veamos en pseudocódigo, donde U es un número aleatorio uniforme:

```
cruce(padre1, padre2) Empezar:
    n = numero características de padre1
    for i de 0 a n-1:
        if padre1[i] == padre2[i]:
            hijo1[i] = padre1[i]
            hijo2[i] = padre1[i]
        else:
            if U < 0.5:
                hijo1[i] = padre1[i]
                hijo2[i] = padre2[i]
            else:
                hijo1[i] = padre2[i]
                hijo2[i] = padre1[i]
    Fin
    Devolver hijo1 e hijo2
Fin
```

2.7. Operadores comunes: Mutación

El operador de mutación (que se usará una vez ya se haya decidido que un gen debe mutar) será el mismo que el de vecino considerado en la búsqueda local que se va a explicar a continuación: 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 (y mutación): 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
```

2.8. Búsqueda Local

Por otro lado, vamos a explicar el algoritmo de búsqueda local que vamos a utilizar en los tres algoritmos meméticos. 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 lo paramos, ya que sólo se va a lanzar una vuelta de este algoritmo. Del mismo modo, si no se encuentra solución mejor dentro del vecindario, también se para. 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.

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.

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)  
  
    posiciones = generarSecuencia(tam(caract))  
    for j en posiciones y mientras vuelta_completa = False:  
        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

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

    Fin for

Fin while

Devolver caract y tasa actual
Fin

```


3. Descripción de los algoritmos

Todos los algoritmos meméticos son una hibridación del algoritmo genético generacional de la práctica anterior con diferentes formas de aplicar la búsqueda local, pero en todos los casos el algoritmo genético generacional no cambia, con lo que vamos a explicar en primer lugar este generacional modificado para que llame a la función `aplicarBusquedaLocal(clases, conjunto, poblacion, knn)` cada 10 generaciones y añada las evaluaciones que ha hecho en la búsqueda local al cómputo total de evaluaciones. Esta función será la que cambiará de un memético a otro y la que explicaremos en cada caso.

3.1. Algoritmo Genético Generacional

En el caso del algoritmo generacional, el esquema de evolución consiste en seleccionar por torneo 10 cromosomas de la población actual (del mismo tamaño que la población) que serán los candidatos a cruzarse para formar los hijos. El pseudocódigo de la función que hace esto es la siguiente:

```
seleccionGeneracional(poblacion) Empezar:
    for i desde 0 hasta 9:
        posicion = torneo(poblacion)
        seleccion[i] = poblacion[posicion]
    Fin
    Devolver seleccion
Fin
```

La probabilidad de que dos parejas de estos seleccionados crucen es 0.7, por lo que el número de parejas a cruzar es $0.7 \cdot 5$. Elegimos por tanto el entero más cercano a $0.7 \cdot 5$, k , y cruzamos las k primeras parejas, que podemos elegir así por ser la selección anterior aleatoria.

En cuanto al esquema de reemplazamiento, consiste en sustituir a la población actual completamente por la nueva con una excepción, que si no se conserva la mejor solución anterior (o no hay una mejor) ésta pasa a sustituir la peor de la población siguiente.

Una vez se ha hecho el reemplazamiento, se pasa a mutar la nueva población. En este algoritmo en cada iteración se mutarán $0.001 \cdot 10 \cdot \text{numero_caract}$. Lo que he hecho ha sido coger el entero por encima más cercano a este número para mutar al menos un gen cuando hay pocas características, y elegir aleatoriamente qué cromosomas y qué gen de cada uno de ellos mutarán. `ceil(x)` devuelve el entero por encima más cercano a x .

Además, añadimos que cuente el número de generaciones que lleva, ya que cada 10 generaciones es cuando tenemos que lanzar la búsqueda local.

Con todo esto, el pseudocódigo es el siguiente:

```
AGG(clases, conjunto, knn) Empezar:
    poblacion = generarPoblacionInicial(30, num_caract, conjunto, clases, knn)
    # num_caract es el número de características
    mutaciones = ceil(0.001*10*num_caract)
    num_evaluaciones = 10
    ya que la población inicial ya está evaluada
```

```

num_generaciones = 0

while num_evaluaciones < 15000:
    Ordenamos la población según su tasa de menor a mayor
    seleccion = seleccionGeneracional(poblacion)
    tope = redondeo de 0.7*5
    i = 0
    while i < 2*tope:
        hijo1, hijo2 = cruce(seleccion[i], seleccion[i+1])
        tasa1 = calcularTasa(conjunto, hijo1, knn)
        tasa2 = calcularTasa(conjuto, hijo2, knn)
        reemplazamos en seleccion[i] y seleccion[i+1] con hijo1 e hijo2
        junto con sus tasas
        num_evaluaciones = num_evaluaciones + 2
        i = i+2
    Fin while interior

    for k desde 0 hasta mutaciones-1:
        crom = aleatorio entre 0 y 9
        gen = aleatorio entre 0 y num_caract
        Flip(seleccion[crom], gen)
        Evaluamos el nuevo individuo y guardamos su tasa
        num_evaluaciones = num_evaluaciones+1
    Fin for

    # Tenemos la poblacion original ordenada de menor a mayor
    if tasa(poblacion[9]) > maximo de las tasas en la seleccion:
        modificamos el mínimo en la seleccion con poblacion[9]

    # Reemplazamos
    poblacion = seleccion
    num_generaciones = num_generaciones + 1

    if num_generaciones == 10:
        poblacion, iter_BL = aplicarBusquedaLocal(clases,
            conjunto, poblacion, knn)
        # Añadimos las evaluaciones al total de evaluaciones
        num_evaluaciones = num_evaluaciones + iter_BL

    Fin while

    pos = posición del cromosoma con máxima tasa en la población
    Devolver poblacion[pos] junto con su tasa
Fin

```

3.2. Algoritmo Memético (10, 1.0) (AM1)

En este caso la búsqueda local se aplica sobre todos los cromosomas de la población (teniendo en cuenta que son 10), luego la función `aplicarBusquedaLocal(clases, conjunto, poblacion, knn)` que se llama desde el algoritmo anterior tiene el siguiente pseudocódigo:

```
aplicarBusquedaLocal(clases , conjunto , poblacion , knn) Empezar:
    for i desde 0 hasta 9:
        poblacion[i], nueva_tasa, iter_BL =
            busquedaLocal(clases , conjunto , poblacion[i] , knn)
        tasa(poblacion[i]) = nueva_tasa
    Fin for
Devolver poblacion , iter_BL
Fin
```

3.3. Algoritmo Memético (10, 0.1) (AM2)

En este caso la búsqueda local se aplica sobre un subconjunto de cromosomas seleccionado aleatoriamente con probabilidad igual a 0.1 para cada cromosoma de ser seleccionado, por lo que por cada 10 generaciones, en media, se le aplica la BL a $0.1 \cdot 10 = 1$ cromosoma. Por tanto, lo que hacemos es aplicar la BL a un cromosoma cada 10 generaciones. Entonces, la función `aplicarBusquedaLocal(clases, conjunto, poblacion, knn)` tiene el siguiente pseudocódigo:

```
aplicarBusquedaLocal(clases , conjunto , poblacion , knn) Empezar:
    crom = num aleatorio entre 0 y 9
    poblacion[crom], nueva_tasa, iter_BL =
        busquedaLocal(clases , conjunto , poblacion[crom] , knn)
    tasa(poblacion[crom]) = nueva_tasa
Devolver poblacion , iter_BL
Fin
```

3.4. Algoritmo Memético (10, 0.1-mej) (AM3)

En este caso la búsqueda local se aplica sobre los $0,1 \cdot N$ mejores cromosomas, con N el tamaño de la población. En este caso N es 10, por lo que se aplica la BL al mejor cromosoma cada 10 generaciones. Entonces, la función `aplicarBusquedaLocal(clases, conjunto, poblacion, knn)` tiene el siguiente pseudocódigo:

```
aplicarBusquedaLocal(clases , conjunto , poblacion , knn) Empezar:
    crom = índice del cromosoma con la mejor tasa
    poblacion[crom], nueva_tasa, iter_BL =
        busquedaLocal(clases , conjunto , poblacion[crom] , knn)
    tasa(poblacion[crom]) = nueva_tasa
Devolver poblacion , iter_BL
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 `practica3.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 AM1, 3 para AM2, 4 para AM3 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 tamaño de la población o las probabilidades de cruce y mutación, 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 AM1

	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	93,6842	66,6667	73,2739	73,3333	77,7778	58,8889	156,5707	76,5625	62,8866	55,3957	776,1501
Partición 1-2	97,1930	96,4789	50,0000	109,4825	80,5556	75,0000	47,7778	193,5003	70,1031	60,9375	47,8417	734,5450
Partición 2-1	97,5352	97,1930	56,6667	102,2398	77,2222	78,8889	45,5556	188,0814	70,8333	63,9175	45,6835	917,0496
Partición 2-2	98,2456	96,1268	40,0000	119,7222	76,1111	68,8889	54,4444	155,0173	77,3196	64,0625	51,0791	759,9437
Partición 3-1	98,5916	94,7368	53,3333	87,6058	77,2222	73,3333	46,6667	169,2416	77,6042	65,9794	50,3597	844,8881
Partición 3-2	97,8947	94,3662	36,6667	109,6000	79,4444	76,6667	60,0000	134,0695	74,7423	66,6667	46,7626	761,1360
Partición 4-1	97,8873	95,7895	46,6667	95,3858	75,5556	77,2222	47,7778	157,7252	73,9583	63,9175	52,1583	848,0728
Partición 4-2	97,8947	96,1268	40,0000	101,3465	81,1111	67,7778	45,5556	164,1188	77,3196	68,7500	46,7626	695,4139
Partición 5-1	97,1831	95,4386	53,3333	83,8958	79,4444	72,2222	50,0000	162,2455	75,5208	66,4948	50,7194	839,5494
Partición 5-2	98,2456	97,8873	40,0000	94,1634	83,8889	70,0000	55,5556	158,4736	71,6495	62,5000	54,3165	713,1978
Media	98,0319	95,7828	48,3333	97,6716	78,3889	73,7778	51,2222	163,9044	74,5613	64,6113	50,1079	788,9946

Cuadro 4: Resultados AM2

	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	95,0877	56,6667	85,2651	73,8889	73,3333	58,8889	147,7682	77,0833	64,4330	0,5324	805,4652
Partición 1-2	97,1930	98,2394	46,6667	97,1011	80,0000	76,6667	62,2222	130,1093	72,6804	63,5417	0,5144	797,8680
Partición 2-1	97,5352	96,4912	63,3333	73,5152	76,6667	79,4444	47,7778	161,7477	73,4375	65,4639	0,5216	971,4121
Partición 2-2	98,9474	95,0704	40,0000	107,9068	74,4444	72,2222	57,7778	140,0962	78,8660	70,3125	0,4712	867,8862
Partición 3-1	98,9437	94,3860	50,0000	90,5736	76,1111	73,8889	46,6667	173,6694	77,0833	64,4330	0,5827	916,5146
Partición 3-2	98,2456	93,6620	53,3333	98,4654	79,4444	75,0000	53,3333	151,6055	75,2577	66,6667	0,5396	777,3816
Partición 4-1	97,5352	96,8421	36,6667	121,3363	75,0000	78,8889	58,8889	139,0470	73,9583	63,9175	0,5072	1047,3006
Partición 4-2	97,8947	95,7746	50,0000	102,9689	82,2222	66,1111	50,0000	164,7150	75,2577	68,2292	0,5108	783,6182
Partición 5-1	97,5352	95,7895	63,3333	78,3686	77,7778	70,0000	53,3333	153,8117	75,0000	65,4639	0,5360	895,1382
Partición 5-2	98,5965	97,5352	43,3333	107,9202	83,3333	72,2222	51,1111	162,3525	73,1959	65,1042	0,5108	701,9376
Media	98,2074	95,8878	50,3333	96,3421	77,8889	73,7778	54,0000	152,4923	75,1820	65,7566	0,5227	856,4522

Cuadro 5: Resultados AM3

	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	95,4386	56,6667	90,9768	73,8889	73,8889	54,4444	154,1980	76,5625	62,3711	56,1151	909,3304
Partición 1-2	97,5439	97,8873	33,3333	108,3798	81,1111	76,6667	52,2222	160,4309	77,3196	66,6667	56,4748	717,0098
Partición 2-1	97,8873	97,8947	50,0000	100,8420	76,1111	81,1111	48,8889	169,3890	70,8333	64,4330	50,0000	943,7376
Partición 2-2	98,9474	94,3662	53,3333	97,9554	76,6667	70,0000	55,5556	139,9832	77,3196	66,1458	48,9209	816,7281
Partición 3-1	98,2394	96,1404	46,6667	78,3801	77,2222	77,7778	50,0000	163,8985	77,6042	63,4021	53,9568	803,1048
Partición 3-2	97,5439	97,8873	46,6667	96,4457	80,5556	73,3333	50,0000	163,9792	73,1959	66,6667	45,3237	783,8307
Partición 4-1	97,1831	96,4912	56,6667	92,3085	76,6667	77,7778	55,5556	141,6862	73,9583	62,8866	52,1583	844,8152
Partición 4-2	97,8947	97,8873	50,0000	91,2669	81,6667	69,4444	58,8889	143,1559	77,3196	71,3542	55,7554	633,3774
Partición 5-1	97,5352	95,4386	60,0000	75,9596	77,7778	71,1111	50,0000	163,0565	75,0000	67,0103	50,0000	875,8141
Partición 5-2	97,8947	97,8873	30,0000	126,1140	81,1111	70,0000	52,2222	159,3467	71,1340	64,5833	46,4029	737,9328
Media	98,0318	96,7319	48,3333	95,8629	78,2778	74,1111	52,7778	155,9124	75,0247	65,5520	51,5108	806,5681

Cuadro 6: 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 7: 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
AM1	98,0319	95,7828	48,3333	97,6716	78,3889	73,7778	51,2222	163,9044	74,5613	64,6113	50,1079	788,9946
AM2	98,2074	95,8878	50,3333	96,3421	77,8889	73,7778	54,0000	152,4923	75,1820	65,7566	0,5227	856,4522
AM3	98,0318	96,7319	48,3333	95,8629	78,2778	74,1111	52,7778	155,9124	75,0247	65,5520	51,5108	806,5681

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).