Metaheurísticas:
Práctica 1.b: Búsquedas por Trayectorias para el Problema de la Selección de Características

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

7 de abril 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 Local.
3. Enfriamiento Simulado.
4. Búsqueda Tabú Básica.
   a) Búsqueda Tabú Básica Modificada
5. Búsqueda Tabú Extendida.

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. 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 estará en todos los casos (menos en el algoritmo greedy, que se empieza desde cero) generada aleatoriamente y se irán generando vecinos de esta solución (o se darán saltos, como ya veremos) y nos quedaremos con la mejor solución obtenida en todos los casos.

Todos los algoritmos tienen una condición de parada común, que es llegar a un número máximo de evaluaciones: 15000.

2.1. 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 utilzar.

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 (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)` donde caracteristicas es la máscara que nos indica cuáles estamos considerando (aquellas que estén a True).

En mi caso el clasificador 3-NN lo he utilizado de un módulo de python, como está descrito en la sección 7, porque el que había hecho yo era mucho menos eficiente (tardaba 6-7 minutos para una partición con el algoritmo greedy para la base de datos wdbc).

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
```

3. Descripción de los algoritmos

3.1. 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 y las clases de los datos.

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

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

        posiciones = generarSecuencia(tam(caract))
        for j en posiciones:
            Flip(caract, j)  # Generamos vecino
            calcularTasa(caract)
            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
                break, salimos del bucle interior y empezamos a
                 generar vecinos de la nueva solución

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

            if vuelta_completa:
                no ha habido mejora en el vecindario, break
            else:
```

```
            vuelta_completa = True

        if hemos superado el numero de evaluaciones
            break y salimos del bucle interior
    Fin for

Fin while

Devuelve caract y tasa actual
Fin
```

3.2. Enfriamiento Simulado

El esquema de enfriamiento que he utilizado ha sido el esquema de Cauchy modificado, en el que la temperatura inicial y la modificación en cada iteración del algoritmo se realiza de la siguiente forma:

```
Tf = 0.001, fi = 0.3, mu = 0.3
Tini = (mu*tasa_inicial)/(-np.log(fi))
beta = (Tini - Tf)/(M*Tini*Tf) #M depende del número de vecinos a generar
                               #y del número de evaluaciones
Tk = Tk/(1+beta*Tk)      #Al final del bucle de enfriamiento
max vecinos = 10n (n num características
max exitos = n
M = 15000/max vecinos
```

El algoritmo de enfriamiento simulado intenta mejorar el problema de la búsqueda local de quedarse en el primer óptimo local que encuentra, aceptando para ello soluciones que pueden ser peores que la actual bajo algunas circunstancias, en concreto, la temperatura. Al iniciar el algoritmo la temperatura es más alta y se va reduciendo con las iteraciones: la idea es aceptar más soluciones peores que la actual cuanto mayor es la temperatura, es decir, al inicio del algoritmo, e ir aceptando menos progresivamente según la temperatura va bajando. Devolvemos la mejor solución encontrada en general a lo largo del algoritmo. Necesitamos para ello guardar la mejor solución junto con su tasa y la solución actual junto con su tasa.
En cada enfriamiento generamos vecinos hasta un máximo de forma aleatoria y nos la quedamos si es mejor que la actual o supera la probabilidad de aceptar una solución peor.

Con esto, el pseudocódigo es el siguiente:

```
enfriamientoSimulado(datos, clases) Empezar:
    caract = solución aleatoria inicial
    mejor solucion = caract
    mejor tasa = calcularTasa(datos, caract)
    Se calculan los parámetros para la temperatura y los vecinos totales

    while haya éxitos en el enfriamiento actual y Tk>Tf y haya menos de 15000 evals:
        while no se haya generado el máximo de vecinos ni se sobrepasen los éxitos:
            pos = num aleatorio entre 0 y n #n num de características
            caract = Flip(caract, pos)
```

```
        nueva tasa = calcularTasa(datos, caract)
        Se aumenta num vecinos y num evaluaciones
        delta = nueva tasa - tasa actual
        if delta!=0 y (delta>0 o U(0,1)<=exp(delta/Tk)):
            #Se ha aceptado el vecino
            Se actualiza la tasa actual
            Se aumenta el número de éxitos
            if tasa actual > mejor tasa:
                Se actualizan la mejor solución y la mejor tasa

        else:
            #No se ha aceptado el vecino, volvemos a la solución anterior
            Flip(caract, pos)

        Se comprueba si hemos pasado el número de evaluaciones
    Fin while

    Se comprueba si ha habido éxitos en este enfriamiento
    Ponemos el número de éxitos a 0
    Ponemos el número de vecinos a 0

    Tk = Tk/(1+beta*Tk) #Actualizamos temperatura
Fin while

Devolvemos mejor solución y mejor tasa
Fin
```

$U(0,1)$ es un número aleatorio extraído de la distribución uniforme en el intervalo $[0,1]$.

He tenido que poner como condición adicional en el `if` para aceptar una solución que `delta!=0` porque si las tasas son iguales y la diferencia es cero entonces la exponencial de $delta/Tk$ es siempre 1 al ser delta 0 y un número entre 0 y 1 es siempre menor o igual que 1, con lo que siempre cogía como éxito una solución con la misma tasa, haciendo que el algoritmo tardara mucho más al haber siempre éxitos en el enfriamiento actual, cuando en realidad no se estaba moviendo ni hacia arriba ni hacia abajo.

3.3. Búsqueda Tabú Básica

En la búsqueda tabú también buscamos salir de óptimos locales con unos mecanismos distintos a los del enfriamiento simulado. En este caso vamos a tener almacenada una lista tabú que nos guarda las posiciones de las características que hemos cambiado en los $n/3$ movimientos anteriores, donde n es el número de características. La lista tiene por tanto $n/3$ posiciones y está implementada como una lista circular, de forma que tenemos un índice marcando la última posición modificada y el siguiente a ese índice módulo $n/3$ es el próximo que tenemos que modificar. Se guardan estas posiciones para que no se pueda volver a cambiar una característica en esas posiciones durante el tiempo que permanecen en la lista tabú para no volver hacia soluciones que habíamos considerado peores, a no ser que modificar una característica en la lista tabú mejore a la mejor solución que tenemos en ese momento (criterio de aspiración).

El método de búsqueda en este caso es la generación de 30 vecinos aleatorios de la solución actual (sin repetición) y quedarnos con el mejor de ellos, sea mejor o no que el que teníamos actualmente. Guardamos siempre la mejor solución por la que hemos pasado, que será la que se devolverá.
Con esto, el pseudocódigo es el siguiente:

```
busquedaTabu(datos, clases) Empezar:
    caract = solución aleatoria inicial
    mejor tasa = calcularTasa(datos, caract)
    mejor solución = copiar(caract)
    lista tabú = vector de longitud n/3 inicializado a -1
    plista = -1          #indica la última posición modificada en la lista

    while num evaluaciones < 15000:
        tasa actual = 0
        mejor pos = -1
        posiciones = 30 vecinos generados aleatoriamente sin repetición

        for j en posiciones:
            caract = Flip(caract, j)        #Generamos vecino
            nueva tasa = calcularTasa(datos, caract)
            Se aumenta el número de evaluaciones

            if j esta en la lista tabú:
                if nueva tasa > mejor tasa y nueva tasa > tasa actual:
                    Se actualiza la tasa actual con la nueva
                    Se cambia j como mejor posición
            elif nueva tasa > tasa actual:
                Se actualiza la tasa actual con la nueva
                Se cambia j como mejor posición

            # Volvemos a la solución que teníamos para seguir generando vecinos
            caract = Flip(datos, caract)
        Fin for

        # Nos quedamos con el mejor vecino
        caract = Flip(caract, mejor pos)
        plista = (plista+1)mod(n/3)
        lista tabu[plista] = mejor pos

        if tasa actual > mejor tasa:
            Se actualiza la mejor tasa y la mejor solución.

    Fin while
    Devuelve mejor solución y mejor pos
Fin
```

3.3.1. Búsqueda Tabú Básica Modificada

Haciendo pruebas me di cuenta de que la búsqueda tabú avanza muy lentamente hacia la mejor solución, y como tarda tanto porque siempre llega las 15000 evaluaciones aunque no haya mejora en las soluciones aceptadas, he pensado en una modificación que podría bajar los tiempos sin bajar mucho la tasa de acierto. Consiste en si no hay una diferencia (en valor absoluto) entre dos tasas de 3 (es decir, del 3 %) en 30 iteraciones (que sería que no hubiera diferencia en 600 evaluaciones) entonces paro el algoritmo. Es decir, cogemos una tasa, y si la diferencia entre esa tasa y la tasa de los 30 siguientes mejores vecinos no supera 3, entonces paro, y si en algún momento sí se supera esa diferencia entonces actualizo la tasa con la que voy a hacer la comparación con las siguientes tasas.
El pseudocódigo sería el siguiente (es todo igual excepto una comprobación al final del bucle):

```
busquedaTabu(datos, clases) Empezar:
    ... (igual que antes)

    while num evaluaciones < 15000 y haya mejora:
        ... (igual que antes)

        for j en posiciones:
            ... (igual que antes)
        Fin for

        # Nos quedamos con el mejor vecino
        caract = Flip(caract, mejor pos)
        plista = (plista+1)mod(n/3)
        lista tabu[plista] = mejor pos

        if tasa actual > mejor tasa:
            Se actualiza la mejor tasa y la mejor solución.

        if valorAbs(tasa mejora - mejor tasa) < 3:
            num no mejora++ #Aumentamos el número de iteraciones sin cambio
            if num no mejora > 30):
                mejora = False
            else:
                tasa mejora = mejor tasa
                num no mejora = 0

    Fin while
    Devuelve mejor solución y mejor pos
Fin
```

Este cambio está hecho en el mismo fichero que la búsqueda tabú simple sin modificar y se encuentra comentado por defecto: habría que descomentarlo para probarlo.

3.4. Búsqueda Tabú Extendida

La búsqueda tabú extendida utiliza el mismo mecanismo de memoria a corto plazo que la tabú sencilla, la lista tabú, aunque le iremos cambiando el tamaño. Incorpora además una memoria a largo plazo en la que se guarda el número de veces que se selecciona una característica para obtener la frecuencia de cada característica y si en algún momento se llega a 10 iteraciones sin mejora en la mejor solución, se reinicializa la búsqueda desde una nueva solución que tendrá probabilidad 0.25 de ser aleatoria, probabilidad 0.25 de ser la mejor solución y probabilidad 0.5 de ser generada de forma aleatoria utilizando el vector de frecuencias.

El pseudocódigo es el mismo que el de la búsqueda tabú sencilla con una modificación al final:

```
busquedaTabuExtendida(datos, clases) Empezar:
    ... (igual que antes)
    frec = vector de frecuencias de longitud n (num características)
     inicializado a cero

    while num evaluaciones < 15000:
        ... (igual que antes)
        for j en posiciones:
            ... (igual que antes)
        Fin for

        # Nos quedamos con el mejor vecino
        caract = Flip(caract, mejor pos)
        plista = (plista+1)mod(n/3)
        lista tabu[plista] = mejor pos
        Se aumenta frec[mejor pos] en 1

        if tasa actual > mejor tasa:
            Se actualiza la mejor tasa y la mejor solución.
            no_mejora = 0
        else:
            no_mejora += 1

        if no_mejora llega a 10:
            no_mejora = 0
            Se elige un numero entre 3 en el que los dos primeros tienen
             probabilidad 0.25 de aparecer y el tercero tiene 0.5
                if sale el primero:
                    caract = solución aleatoria
                elif sale el segundo:
                    caract = mejor solución
                else:
                    u = número uniforme entre 0 y 1
                    for i de 0 hasta n:
                        if u < 1 - frec[i]:
                            se pone la característica i a True
```

```
                else
                    se pone a False
        u = número uniforme entre 0 y 1
        if u < 0.5
            Se aumenta al doble la lista tabú
        else:
            Se disminuye a la mitad la lista tabú

    Fin while

    Devuelve mejor solución y mejor pos
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 tres módulos de python: numpy, scipy y sklearn (de scikit) (todos disponibles en Linux pero hay que instalarlos previamente). Numpy permite manejar arrays de forma más rápida, scipy leer ficheros en formato arff y sklearn contiene el knn y cross-validation para hacer leave one out. 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 `practica1.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 BL, 3 para ES, 4 para BT, 5 para BTE y 6 para KNN.

6. Experimentos y análisis de resultados

6.1. Descripción de los casos del problema empleados

Los parámetros de los algoritmos, como los del enfriamiento simulado o el tope de evaluaciones a realizar, lo 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. Para el caso del 3NN, como lo estamos lanzando con una máscara entera a True, este tiempo el 0, al igual que la tasa de reducción.

Cuadro 1: Resultados SFS

	Wdbc				Movement Libras				Arrhythmia			
	%_clas train	%_clas test	%_red	T (s)	%_clas train	%_clas test	%_red	T (s)	%_clas train	%_clas test	%_red	T (s)
Partición 1-1	98,9437	93,3333	83,3333	47,7583	65,5556	64,4444	92,2222	128,1156	81,7708	70,1031	97,1223	474,7185
Partición 1-2	95,7895	94,0141	83,3333	47,5713	73,8889	70,0000	91,1111	141,5071	80,9278	78,6458	97,1223	474,0171
Partición 2-1	95,4225	93,3333	90,0000	32,6485	70,0000	70,5556	91,1111	141,5607	78,6458	70,1031	97,4820	417,5319
Partición 2-2	97,5439	94,3662	90,0000	32,8499	67,2222	66,1111	93,3333	110,7466	85,0515	70,8333	96,4029	582,5531
Partición 3-1	97,8873	96,4912	76,6667	61,5079	71,1111	70,0000	90,0000	159,5794	75,0000	69,0722	98,9209	209,9304
Partición 3-2	97,5439	93,6620	86,6667	40,9398	66,6667	70,0000	92,2222	127,1383	78,8660	71,3542	97,8417	365,9823
Partición 4-1	96,8310	97,8947	86,6667	39,8884	67,7778	70,5556	92,2222	128,9092	82,8125	73,7113	96,7626	526,7995
Partición 4-2	97,5439	95,4225	83,3333	46,9100	78,3333	62,2222	90,0000	157,2638	85,0515	73,4375	96,0432	632,3626
Partición 5-1	97,5352	95,7895	90,0000	32,3780	80,0000	62,2222	86,6667	204,0013	82,2917	68,0412	97,4820	424,4544
Partición 5-2	96,1404	94,7183	83,3333	47,9195	77,7778	68,8889	90,0000	159,4742	78,8660	72,3958	97,1223	485,9460
Media	97,1181	94,9025	85,3333	43,0372	71,8333	67,5000	90,8889	145,8296	80,9284	71,7698	97,2302	459,4296

Cuadro 2: Resultados BL

	Wdbc				Movement Libras				Arrhythmia			
	%_clas train	%_clas test	%_red	T (s)	%_clas train	%_clas test	%_red	T (s)	%_clas train	%_clas test	%_red	T (s)
Partición 1-1	97,5352	93,6842	46,6667	3,7190	62,2222	69,4444	52,2222	8,8476	68,2292	63,4021	50,3597	9,8135
Partición 1-2	94,3860	95,4225	46,6667	1,2674	69,4444	76,1111	46,6667	12,3067	67,0103	63,5417	45,3237	32,3330
Partición 2-1	95,7746	96,8421	63,3333	3,3681	67,7778	74,4444	57,7778	1,5256	65,6250	61,3402	56,1151	65,7732
Partición 2-2	96,4912	92,9577	46,6667	7,1604	65,5556	65,0000	56,6667	17,8602	68,0412	64,5833	46,4029	51,7505
Partición 3-1	96,1268	96,1404	33,3333	11,9843	67,7778	70,0000	56,6667	10,5577	66,6667	63,9175	46,4029	32,6144
Partición 3-2	96,4912	97,1831	43,3333	11,0846	66,6667	70,5556	50,0000	7,8249	67,5258	65,6250	52,5180	36,3594
Partición 4-1	96,1268	95,4386	40,0000	5,8840	59,4444	70,5556	53,3333	7,8953	67,1875	62,3711	43,8849	16,5154
Partición 4-2	95,7895	94,3662	63,3333	10,4619	72,2222	61,6667	44,4444	7,9087	65,9794	66,6667	52,8777	4,9554
Partición 5-1	96,1268	96,8421	46,6667	9,4036	71,1111	67,7778	53,3333	8,5307	65,1042	64,9485	48,9209	4,5913
Partición 5-2	97,5439	95,7746	50,0000	8,3311	71,1111	68,3333	51,1111	11,8468	63,9175	63,0208	46,0432	42,4668
Media	96,2392	95,4652	48,0000	7,2665	67,3333	69,3889	52,2222	9,5104	66,5287	63,9417	48,8849	29,7173

Cuadro 3: Resultados ES

	Wdbc				Movement Libras				Arrhythmia			
	%_clas train	%_clas test	%_red	T (s)	%_clas train	%_clas test	%_red	T (s)	%_clas train	%_clas test	%_red	T (s)
Partición 1-1	98,9437	95,0877	43,3333	190,6666	61,1111	68,3333	51,1111	383,7357	73,9583	62,3711	49,6403	1641,5006
Partición 1-2	95,7895	96,1268	46,6667	186,5556	70,0000	73,3333	55,5556	381,7689	70,1031	62,5000	50,0000	1663,5517
Partición 2-1	97,5352	95,7895	50,0000	185,7704	70,5556	75,5556	62,2222	383,1403	66,1458	61,8557	49,2806	1571,5591
Partición 2-2	98,2456	94,0141	53,3333	188,5999	66,6667	62,7778	44,4444	390,1831	70,1031	63,5417	51,0791	1562,8782
Partición 3-1	97,1831	95,0877	60,0000	185,9046	66,6667	72,2222	55,5556	384,5502	69,2708	65,4639	47,4820	1593,5539
Partición 3-2	97,8947	94,7183	46,6667	188,8094	68,8889	73,3333	53,3333	374,4300	70,6186	66,6667	53,5971	1536,6501
Partición 4-1	97,1831	96,8421	53,3333	197,0699	62,7778	76,1111	52,2222	379,0105	70,8333	64,9485	50,0000	1532,1212
Partición 4-2	96,8421	94,0141	36,6667	210,1286	73,3333	59,4444	55,5556	384,5273	69,5876	66,1458	47,1223	1599,9626
Partición 5-1	96,8310	96,8421	50,0000	195,4052	71,1111	66,6667	41,1111	391,5431	71,8750	64,9485	48,5612	1600,2462
Partición 5-2	97,5439	96,4789	53,3333	189,5962	72,2222	70,0000	48,8889	387,5911	68,0412	68,7500	50,3597	1555,9471
Media	97,3992	95,5001	49,3333	191,8507	68,3333	69,7778	52,0000	384,0480	70,0537	64,7192	49,7122	1585,7971

Cuadro 4: Resultados BT

	Wdbc				Movement Libras				Arrhythmia			
	%_clas train	%_clas test	%_red	T (s)	%_clas train	%_clas test	%_red	T (s)	%_clas train	%_clas test	%_red	T (s)
Partición 1-1	99,6479	94,0351	53,3333	4482,1553	71,1111	67,7778	67,7778	2792,4891	77,6042	64,9485	50,3597	3547,2839
Partición 1-2	97,8947	96,4789	40,0000	4525,3261	77,7778	71,1111	50,0000	2811,5240	72,1649	61,4583	54,3165	3689,6839
Partición 2-1	98,2394	96,8421	40,0000	4414,8685	77,7778	75,5556	51,1111	2816,0023	69,7917	62,8866	56,8345	3547,7961
Partición 2-2	99,6491	94,3662	50,0000	4387,6807	72,7778	65,5556	57,7778	2793,7549	78,8660	65,1042	56,4748	3540,1120
Partición 3-1	98,9437	94,3860	56,6667	4435,5301	75,5556	72,7778	58,8889	2804,0871	75,0000	65,9794	60,0719	3567,9060
Partición 3-2	98,5965	95,7746	63,3333	4402,7661	76,6667	70,5556	63,3333	2784,8322	75,7732	67,7083	56,4748	3516,0451
Partición 4-1	98,2394	96,1404	50,0000	4398,5370	72,2222	73,8889	56,6667	2817,5327	76,5625	65,9794	53,2374	3498,7134
Partición 4-2	98,5965	96,4789	63,3333	4400,6981	81,1111	62,2222	61,1111	2776,8963	72,1649	67,1875	51,0791	3538,5967
Partición 5-1	98,9437	96,4912	50,0000	4389,9218	79,4444	68,8889	62,2222	2802,4383	74,4792	65,4639	48,9209	3524,8939
Partición 5-2	98,9474	97,5352	43,3333	4422,8620	80,0000	68,3333	62,2222	2812,4940	74,2268	67,1875	56,8345	3471,6671
Media	98,7698	95,8529	51,0000	4426,0345	76,4444	69,6667	59,1111	2801,2051	74,6633	65,3904	54,4604	3544,2698

Cuadro 5: Resultados BT Modificada

	Wdbc			Movement Libras			Arrhythmia		
	%_clas train	%_clas test	T (s)	%_clas train	%_clas test	T (s)	%_clas train	%_clas test	T (s)
Partición 1-1	99,2958	94,3860	299,6978	68,3333	71,6667	365,3628	75,0000	65,4639	477,8076
Partición 1-2	97,5439	97,8873	297,9895	75,0000	73,3333	245,4732	70,1031	62,5000	248,7286
Partición 2-1	97,5352	97,1930	290,0433	76,1111	76,1111	375,3009	67,1875	63,9175	239,4865
Partición 2-2	98,9474	95,4225	287,7853	68,8889	66,6667	235,9328	70,6186	64,0625	304,3770
Partición 3-1	98,5915	94,7368	291,0515	71,6667	69,4444	232,1882	69,2708	63,4021	343,2357
Partición 3-2	98,2456	96,1268	288,4757	72,2222	73,8889	237,2195	70,6186	67,1875	274,5305
Partición 4-1	97,8873	97,5439	306,1516	67,7778	70,0000	303,4044	72,3958	65,4639	315,8250
Partición 4-2	98,2456	95,4225	331,3072	78,3333	63,3333	349,9913	68,5567	65,1042	361,7439
Partición 5-1	97,8873	97,5439	337,8707	75,5556	70,0000	358,2944	70,8333	64,4330	346,6857
Partición 5-2	97,8947	96,4789	312,0586	78,8889	70,5556	397,2670	65,4639	65,6250	251,6011
Media	98,2074	96,2742	304,2431	73,2778	70,5000	310,0434	70,0048	64,7160	316,4021

Cuadro 6: Resultados BTE

	Wdbc				Movement Libras				Arrhythmia			
	%_clas train	%_clas test	%_red	T (s)	%_clas train	%_clas test	%_red	T (s)	%_clas train	%_clas test	%_red	T (s)
Partición 1-1	99,2958	94,3860	53,3333	4891,4500	68,8889	67,2222	58,8888	3159,3639	75,0000	65,4639	52,1583	4398,2495
Partición 1-2	97,5439	96,8310	20,0000	4855,4829	75,5556	72,2222	46,6666	3183,8460	69,0722	63,0208	51,7986	4597,1095
Partición 2-1	97,8873	96,8421	30,0000	4932,8456	76,6667	76,6667	46,6666	3251,4145	70,8333	60,8247	44,2446	4412,4157
Partición 2-2	99,2982	93,6620	40,0000	4889,1054	70,0000	67,2222	50,0000	3174,5812	74,2268	67,7083	48,9209	4466,9630
Partición 3-1	98,9437	96,1404	33,3333	4874,8818	72,7778	70,5556	50,0000	3163,0693	75,0000	65,4639	52,5180	4320,1314
Partición 3-2	98,2456	96,1268	43,3333	4837,6802	73,3333	73,3333	45,5555	3183,1151	74,7423	67,7083	55,3957	4414,9784
Partición 4-1	97,8873	96,8421	40,0000	4682,4851	71,6667	77,7778	60,0000	3161,6695	73,4375	64,9485	44,2446	4255,8287
Partición 4-2	98,2456	97,8873	56,6667	4468,9579	81,1111	62,7778	62,2222	3134,6418	72,6804	66,6667	51,0791	4119,7669
Partición 5-1	98,2394	94,7368	53,3333	4478,3304	76,6667	67,7778	50,0000	3182,7247	70,8333	64,4330	48,5612	4185,7275
Partición 5-2	98,5965	97,1831	43,3333	4373,2153	80,0000	70,0000	58,8889	3164,9642	69,0722	65,1042	49,2806	4117,4000
Media	98,4183	96,0638	41,3333	4728,4435	74,6667	70,5556	52,8889	3175,9390	72,4898	65,1342	49,8201	4328,8571

Cuadro 7: Resultados 3NN

	Wdbc				Movement Libras				Arrhythmia			
	%_clas train	%_clas test	%_red	T (s)	%_clas train	%_clas test	%_red	T (s)	%_clas train	%_clas test	%_red	T (s)
Partición 1-1	98,5915	95,7895	0,0000	0,0000	60,0000	70,0000	0,0000	0,0000	65,6250	65,4639	0,0000	0,0000
Partición 1-2	95,4386	97,8873	0,0000	0,0000	68,3333	76,1111	0,0000	0,0000	62,8866	60,9375	0,0000	0,0000
Partición 2-1	95,0704	97,8947	0,0000	0,0000	66,6667	76,1111	0,0000	0,0000	65,1042	62,3711	0,0000	0,0000
Partición 2-2	95,7895	95,4225	0,0000	0,0000	62,2222	65,0000	0,0000	0,0000	64,4330	64,5833	0,0000	0,0000
Partición 3-1	96,1268	96,8421	0,0000	0,0000	64,4444	71,1111	0,0000	0,0000	65,1042	63,4021	0,0000	0,0000
Partición 3-2	95,4386	96,4789	0,0000	0,0000	65,0000	76,1111	0,0000	0,0000	65,4639	68,2292	0,0000	0,0000
Partición 4-1	95,7746	96,8421	0,0000	0,0000	60,0000	75,0000	0,0000	0,0000	64,5833	64,4330	0,0000	0,0000
Partición 4-2	96,1404	95,4225	0,0000	0,0000	71,6667	64,4444	0,0000	0,0000	63,4021	65,6250	0,0000	0,0000
Partición 5-1	94,7183	97,1930	0,0000	0,0000	67,7778	65,5556	0,0000	0,0000	64,5833	67,0103	0,0000	0,0000
Partición 5-2	95,7895	97,5352	0,0000	0,0000	66,6667	70,0000	0,0000	0,0000	64,4330	66,1458	0,0000	0,0000
Media	95,8878	96,7308	0,0000	0,0000	65,2778	70,9444	0,0000	0,0000	64,5619	64,8201	0,0000	0,0000

6.3. Análisis de los resultados

A continuación vemos la tabla comparativa, que tiene la última fila (las medias) de todas las tablas anteriores:

Cuadro 8: 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	95,8878	96,7308	0,0000	0,0000	65,2778	70,9444	0,0000	0,0000	64,5619	64,8201	0,0000	0,0000
SFS	97,1181	94,9025	85,3333	43,0372	71,8333	67,5000	90,8889	145,8296	80,9284	71,7698	97,2302	459,4296
BL	96,2392	95,4652	48,0000	7,2665	67,3333	69,3889	52,2222	9,5104	66,5287	63,9417	48,8849	29,7173
ES	97,3992	95,5001	49,3333	191,8507	68,3333	69,7778	52,0000	384,0480	70,0537	64,7192	49,7122	1585,7971
BT basica	98,7698	95,8529	51,0000	4426,0345	76,4444	69,6667	59,1111	2801,2051	74,6633	65,3904	54,4604	3544,2698
BT extendida	98,4183	96,0638	41,3333	4728,4435	74,6667	70,5556	52,8889	3175,9390	72,4898	65,1342	49,8201	4328,8571

Como vemos, el algoritmo que más tarda es la búsqueda tabú extendida, puesto que es el que más carga computacional tiene. Entre ambas búsquedas tabú se diferencian relativamente poco en tiempo (teniendo en cuenta que tardan mucho, sin embargo entre una y otra hay en arritmia por ejemplo 800 segundos de diferencia, que sí es una diferencia considerable) porque en lo que se diferencian es en que la simple sólo maneja la lista tabú y la extendida además de manejarla y cambiarla bastantes veces maneja también la memoria a largo plazo (la lista de frecuencias). Sin embargo si miramos las tasas esta diferencia en tiempo podría no ser asumible ya que prácticamente no hay diferencia considerable entre las tasas (aunque también es cierto que habrá problemas en los que esta diferencia, por poca que sea, sí será considerable y será mejor esperar más tiempo). De hecho para el conjunto de entrenamiento sólo hay mejora en la base de datos Wdbc, mientras que en las demás hay incluso una peora y en los conjuntos de test sólo hay como mucho un punto de mejora.

El problema de las dos búsquedas tabú es que se van acercando a los óptimos muy lentamente, que es algo fácil de ver al ir mostrando las tasas que va obteniendo en todas las iteraciones, y además no tienen condición de parada adicional, de forma que sólo paran cuando se llega exactamente a 15000 evaluaciones, no como por ejemplo el enfriamiento simulado, que tiene como condición de parada adicional no haber obtenidos éxitos en un enfriamiento completo, lo que hace que la mayoría de las veces no se completen las 15000 evaluaciones, pare antes y por lo tanto tarde bastante menos (alrededor de 3 minutos frente a 74 en wdbc y alrededor de 26 minutos frente a 59 en arritmia) dando unas tasas de acierto bastante buenas si comparamos con la búsqueda tabú simple, por ejemplo: la búsqueda tabú mejora como mucho 1 punto en el conjunto de test y 6 en el de entrenamiento si miramos todas las bases de datos. Es decir, a mi parecer merece más la pena utilizar un buen enfriamiento simulado quizás ya entrando también en modificarle lo parámetros para intentar obtener mejores tasas que utilizar la búsqueda tabú, ya que tarda bastante menos y los resultados son casi igual de buenos que en la búsqueda tabú.

Para intentar arreglar que no llegue al número de evaluaciones la búsqueda tabú básica si no hay suficiente cambio (en concreto un cambio del 3 % en 600 evaluaciones) es para lo que he hecho la modificación de la búsqueda tabú básica. Como vemos en la tabla en el apartado anterior conseguimos tiempos que se acercan al enfriamiento simulado (aunque siguen siendo un poco mayores) y por tanto son mucho menores que los de las tabú. Aunque en la tasa de entrenamiento se queda por debajo esta modificación (lo que significa que la búsqueda tabú tiene cambios muy lentos) en las dos primeras bases de datos la tasa en el conjunto de test es ligeramente mayor.

Si entramos ahora en comparar tanto las búsquedas tabú como los enfriamientos simulados

con el algoritmo greedy SFS, en las bases de datos wdbc y movement libras SFS se queda un poco peor (aunque no mucho, en torno a un 1-3 %) mientras que en la base de datos de arritmia el algoritmo SFS es el mejor de todos con más de un 5 % de diferencia. En cuanto a tasas de reducción es normal que todos sean peores con respecto al greedy SFS puesto que los demás empiezan todos con soluciones iniciales aleatorias y si una característica no aporta nada y no da peor resultado tenerla escogida frente a no tenerla el resto de algoritmos no la eliminan, mientras que el SFS va escogiendo en orden aquellas características que más tasa de acierto aportan, con lo que no escoge ninguna de sobra. Es por esto que también podemos pensar que si en wdbc y en movement libras las tasas son mejores con tabú y enfriamiento que con SFS es porque importa en qué orden se elijan las características en el greedy.

Queda sólo por comentar la búsqueda local. Es el más rápido de todos pero a costa de quedarse en el primer óptimo local que encuentra y por tanto tiene menos tasa de acierto que el enfriamiento y la búsqueda tabú, y peor tasa en entrenamiento pero algo mejor en test si comparamos con el SFS (excepto en arritmia). Éste es el algoritmo que escogeríamos si necesitáramos soluciones rápidas y buenas, pero no necesitáramos las óptimas.

Con respecto al 3NN al principio parece raro que tenga menos tasa de acierto que los demás puesto que está considerando todas las características. Sin embargo, 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 la de entrenamiento no sea mejor que la de test, como suele ocurrir (ya que lo que optimizamos es la tasa de acierto justo en el entrenamiento), puesto que en este algoritmo en realidad las dos serían de test ya que no estamos utilizando ninguna de las dos para aprender.

7. Bibliografía

1. Sklearn (scikit) para 3NN aquí
2. Sklearn (scikit) para leave one out aquí
3. Scipy para leer ficheros aquí
4. Documentación de numpy aquí