Técnicas basadas en trayectorias múltiples para el problema de la selección de características

Mª Cristina Heredia Gómez, *Metaheurísticas, Universidad de Granada*

En esta práctica se estudia la aplicación de tres técnicas Multi-arranque: Búsqueda Multiarranque básica, GRASP e ILS, para optimizar las soluciones obtenidas al problema de la selección de caracteríticas, en tres bases de datos distintas.

Descripción del problema

El problema de la selección de características, consiste en seleccionar aquellas características de una base de datos dada que resulten las más representativas,

de forma que dicha selección nos permita maximizar la tasa de acierto test de un clasificador.

En estas prácticas tomamos un clasificador Knn con k=3 como función objetivo, por lo que buscamos maximizar la tasa de acierto de clasificación, aplicando metaheurísticas que nos ayuden a elegir correctamente que características seleccionar en cada base de datos.
KNN es un clasificador sencillo basado en distancias, concretamente , con k=3 tenemos en cuenta el voto de los 3 vecinos más cercanos para decidir la clase de un dato dado.

Por lo tanto, nuestra función a maximizar es:

$$tasaclass = 100 \cdot \frac{n^o instancias\_bien\_clasificadas}{n^o total\_instancias}$$

donde las características se codifican de forma binaria para ser representadas en el problema(1 representa seleccionada/ 0 representa no seleccionada). Para realizar los experimentos se han trabajado con tres bases de datos:

- Wdbc (Wisconsin Database Breast Cancer)
- Movement_Libras
- Arrhythmia

de diferentes tamaños.

Representación del problema

En este problema las soluciones se representan como un vector de ceros y unos, que tendrán la longitud del número de columnas de la base de datos considerada en ese momento, menos uno (menos la clase).
Así pues una posible solución sería s=(0110...00) donde 0 significa que la característica i-ésima de la base de datos no se selecciona y 1 lo contrario.

La **función objetivo** consiste en maximizar la tasa de clasificación mostrada anteriormente. Para ello, **se parte de una solución inicial** que será generada aleatoriamente en todos los casos (excepto en el Greedy que parte de una solución vacía, con todas las características a 0) y se tratará, a lo largo de la ejecución del algoritmo, de mejorar esa solución.

Para esto es fundamental a lo largo de la práctica, el uso de tres funciones: la que genera una solución vecina a partir de una solución dada, siguiendo un **esquema de inversión a nivel de bits**

Algorithm 1 flip

```
procedure FLIP(SOLUCION,INDEX)
    if solucion[index] == 1 then
        solucion[index] ← 0
    else
        solucion[index] ← 1
     return solucion
```

Esta función llamada **flip** recibe como argumento una solución de 0 y 1, y un índice, que es la posición del vector en la que deseamos cambiar el bit, y realiza la inversión del bit presente en dicha posición.

Otra función clave es **getFeaturesForm** que recibe como argumentos un vector de ceros y unos y una base de datos, y devuelve una fómula compuesta por los nombres de las características de la base de datos especificada que están a 1 en el vector. Devuelve una fórmula con forma:

$$class \sim feature_1 + feature_2 + ... + feature_n$$

que posteriormente se la pasaremos al clasificador.

Algorithm 2 getFeaturesForm

```
procedure GETFEATURESFORM(SELECTED,DATASET)
    names ← 0
    featuresList ← 0
    i ← 0
    formula ← 0
    names ← obtener nombres de las columnas del dataset

loop: for i in selected (itera sobre selected)
    if selected[i] == 1 then
        featuresList ← names[i] (mete nombre en la lista)
end loop
        formula ← paste(class,∼,featuresList,separador='+')(componer formula)
return formula
```

La última función, **Adjust3nn** es la que se invoca para ajustar el modelo usando el clasificador 3NN. Recibe como argumentos una fórmula como la mencionada anteriormente y los datos de training de donde tomará las características indicadas en la fórmula, y devuelve el modelo ajustado con los datos de train.
Para poder hacerlo más modular puse la clase de todos los datasets al final de estos y las nombré como **class** en los tres conjuntos de datos.

Algorithm 3 Adjust3nn

procedure ADJUST3NN(FORMULA,TRAINING DATA)
    modelo ← 0 (inicialmente modelo no contiene nada)
    modelo ← ajusta modelo(formula,training data)
return modelo

Para ajustar el modelo en el código uso el método **train** de la librería **caret**. Para calcular las predicciones y el acierto de test uso **predict** y **postResample**, también de esta librería.

Técnicas multiarranque estudiadas en esta práctica

BMB

En la búsqueda multiarranque básica se generan 25 soluciones iniciales de forma aleatoria que posteriormente se optimizan aplicando una búsqueda local sobre cada una de ellas. Una vez que están todas optimizadas, el algormitmo devuelve aquella que da mejor resultado para el acierto de test.

La BMB implementada recibe como parámetros el conjunto de train y de test, ya que son necesarios para ajustar el modelo y calcular el acierto de test, respectivamente.
Para generar las soluciones iniciales aleatorias, he usado un **sample** de 0 y 1, con reemplazamiento. Para asegurarme de que no genera siempre la misma, antes de hacer el sample pongo una semilla distinta cada vez, haciendo que la semilla dependa del número de la iteración actual.

La optimización de las soluciones mediante búsqueda local se hacen de forma paralela con **parLapply** en el código, aunque en el pseudocódigo se muestra como un bucle convencional.

Algorithm 4 BMB

```
procedure BMB(TRAINING,TEST)
    nfeatures ← num_columnas(training)-1
    modelosBL ← 0 (vector con los 25 modelos.Inicialmente vacio)
    vecina ← 0
    modelo ← 0
    BestAccuracyGlobal ← 0
    i ← 0
    bestIndex ← 0 indice del mejor modelo de la lista ModelosBL

loop: repetir 25 veces
    vecina ←genera solución aleatoria con tam=nfeatures
    modelo ←LocalSearchModified(training,test,vecina)
    modelosBL ← modelo (meter modelo en el array)
end loop

loop: for each i in modelosBL (obtener el mejor modelo de los 25)
    if Accuracy Test de modelosBL[i] > BestAccuracyGlobal then
        BestAccuracyGlobal ← Accuracy Test de modelosBL[i]
        bestIndex ← i
end loop

return modeloBL[bestIndex]
```

LocalSearchModified es la Local Search de la práctica anterior pero modificada para que acepte una solución inicial como parámetro. **ModelosBL** es una lista con los resultados de los 25 modelos tras aplicarles BL.

GRASP

Este algoritmo se compone de dos partes: en la primera, se hace uso de una función, de nombre **greedyRndm** que construye soluciones greedy probabilísticas. Esta función es llamada tantas veces como soluciones greedy se quieran obtener. Para ello, el algoritmo GRASP recibe como parámetro, además del training y el test, un numero de soluciones a generar.
En la segunda fase de este algoritmo se trata de optimizar las soluciones greedy aleatorizadas obtenidas anteriormente, aplicando búsqueda local sobre ellas.

Finalmente, se devuelve la mejor solución, que será la de más acierto en el test.

La diferencia con el algoritmo anterior es clara. Esta vez no se parte de soluciones aleatorias, sino que se parte de una solución mejor, casi siempre, pues ha sido cuidadosamente calculada previamente (con el cómputo extra que el greedy conlleva).

El algoritmo de búsqueda local empleado es el mismo que en el caso anterior. El procedimiento de greedy aleatorizado sí es nuevo. Recibe como argumentos los conjuntos de train, test y una semilla que uso para meter más aleatoriedad al tomar características aleatorias de la LRC.

Algorithm 5 greedyRndm

procedure GREEDYRNDM(TRAINING,TEST,SEED)
featuresList ← lista de caracteristicas del dataset
final ← FALSE
selectedAndCandidate ← 0 (inicialmente no hay seleccionadas)
ganancias ← 0 (inicialmente está vacía)
selected ← 0 (inicialmente está vacía)
cmejor ← 0
cpeor ← 0
umbral ← 0
alpha ← 0.3
randomFeature ← 0
evalua ← 0
bestAccu ← 0 (mejor acierto test hasta ahora)
LRC ← 0 (inicialmente está vacía)

loop: mientras(featuresList!=0 AND !final)
ganancias ←calcular ganancia test de cada caracteristica por separado
cmejor ←max(ganancias)
cpeor ←min(ganancias)
umbral ←cmejor-(alpha(cmjor-cpeor))
LRC ←caracteristicas cuyas (ganancias>=umbral)
randomFeature ← caracteristica alatoria de LRC
selectedAndCandidate[randomFeature] ← 1
evalua ← ajustar de nuevo usando las caracteristicas de selectedAndCandidate

if evalua > bestAccu **then**
bestAccu ← evalua
selected ← selectedAndCandidate
featuresList[randomFeature] ← no se puede volver a seleccionar

else final ← TRUE

selectedAndCandidate ← selected

end loop
return list(selected,bestAccu)

Además de la solucción binaria seleccionada y el mejor acierto de clasificación

para test, el código devuelve información acerca del mejor modelo asociado, aunque se obvie en el pseudocódigo por simplicidad.
El algoritmo Grasp también se encuentra paralelizado con **parLapply** y **parSapply** usando clusters typo FORK, aunque en el pseudocódigo se muestre como bucles convencionales.

Algorithm 6 GRASP

procedure GRASP(TRAINING,TEST,NUMSOL)
GreedySolutions ← 0 (vector con las 25 soluciones greedy.Inicialmente vacio)
modelosBL ← 0 (vector con los 25 modelos.Inicialmente vacio)
BestAccuracyGlobal ← 0
bestIndex ← 0 indice del mejor modelo de la lista ModelosBl

loop: repetir numSol veces (en este caso 25)
GreedySolutions ←greedyRndm(training,test,semilla) (genera y guarda las soluciones greedy en el vector)
end loop

loop: for i=1 to 25 aplicar BL a cada solucion greedy
ModelosBL ←LocalSearchModified(training,test,GreedySolutions[i])
end loop

loop: for i in 1:25 (obtener el mejor modelo de los 25)
if Accuracy Test de modelosBL[i] > BestAccuracyGlobal **then**
BestAccuracyGlobal ← Accuracy Test de modelosBL[i]
bestIndex ← i
end loop

return modeloBL[bestIndex]

ILS

Es una búsqueda local reiterada, en el que se parte de una solución generada aleatoriamente y sobre la que se aplica el algoritmo de búsqueda local. Posteriormente se comprueba si la solución optimizada obtenida es mejor que la mejor hasta el momento " **bestAtMoment**", en cuyo caso se actualiza la mejor solución hasta el momento y se aplica una mutación sobre ella de $t = 0{,}1 \cdot n$, es decir, un 10 % de sus características.

Para mutarlas, se generan t índices aleatorios dentro de la solución y se invierte su contenido (llamando a flip).Por ejemplo, si t=2 y se toman aletoriamente los índices 2 y 4 que contienen 0 y 1 respectivamente, ahora en la posición 2 habrá un 1 y en la 4 un 0.
Sobre la solución mutada se vuelve a aplicar una búsqueda local, y sobre la solución obtenida se comprueba si es mejor que la mejor obtenida hasta el momento, y se actualiza si procede. Este proceso se repite un total de 25 veces.

El algoritmo recibe como parámetros los conjunto de taining, test y una semilla que varía en cada ejecución y que se usa para evitar que siempre tome la misma solución inicial.

Algorithm 7 Búsqueda Local Reiterada

```
procedure ILS(TRAINING,TEST,SEED)
    set.seed ← seed
    sIni ← generar solucion inicial aleatoria
    sLSearch ← LocalsearchModified(training,test,Ini)
    bestAtMoment ← 0 (Inialmente vacio)
    bestAccu ← 0 (Inialmente el mejor porcentaje de class es 0)
    iter ← 1
    i ← 0
    t ← 0.1*numero de caracteristicas
    mutated ← 0 (Inialmente vacia)
    randomIndex ← 0 (Lista de indices aleatoriamente seleccionados.Inialmente vacía)
    AccuIni ← Accuracy inicial de(sLSearch)

    if Accuracy test de sLSearch > bestAccu then
        bestAtMoment ← sLEarch
        bestAccu ← Accuracy test de sLSearch

loop: mientras iter ≠ 25
    randomIndex ← obtener t caracteristicas aleatoriamente,sin reemplazamiento
    mutated ← bestAtMoment

    loop: para cada i←indice en randomIndex
    mutated ← flip(mtated,i) (invierte elementos en las posiciones dadas)
    end loop

    sLSearch ← LocalsearchModified(training,test,mutated)

    if Accuracy test de sLSearch > bestAccu then
        bestAtMoment ← sLSearch
        bestAccu ← Accuracy test de sLSearch

    iter ← iter+1

end loop
     return list(bestAtMoment,bestAccu,Accuini)
```

Tratando de ser lo más fiel posible a mi código, en el pseudódigo se devuelve al final del algoritmo una lista con la mejor solución encontrada, así como su

acierto de test y el acierto de test inicial de la solución incial aleatoria con el que empezó el algoritmo. Esto no es necesario, lo hago porque R permite hacerlo fácilmente y luego uso estos datos para visualizar el comportamiento de los algoritmos. Simplemente bastaría con devolver la mejor solución encontrada en la ejecución del algoritmo.

Procedimiento considerado

En la realización de la práctica he usado lenguaje R y el IDE RStudio, beneficiándome de las facilidades que R aporta por no ser un lenguaje orientado a objetos ni fuertemente tipado, así como de las potentes librerías de las que dispone. Por otra parte el empleo de R ha supuesto una gran pérdida en eficiencia, llegando a tener en algunos casos ejecuciones de hasta 15h. (aún paralelizando con 4 cores en muchos casos).

El procedimiento a seguir ha sido el siguiente: en primer lugar se realiza la carga de los datos especificando la ruta de los mismos, y luego he realizado un pequeño "preprocesamiento"que consiste en:

- si hay columnas en los datos con todos los valores iguales, eliminarlas.
- normalizar los datos, sin normalizar la clase.
- tomar la columna de la clase y colocarla al final de todas las bases de datos.
- nombrar igual a las columnas con la clase de los 3 datasets."**class**", para hacerlo más modular en las funciones.

Una vez listos los datos, he usado, como mencioné arriba, el knn de la librería caret de R, fijando el k a 3, como clasificador y su acierto test como resultado a maximizar.
También he usado el método **createDataPartition** de esta librería, para hacer el particionamiento estratificado de los datos. Parto los datos cada vez con una semilla, que es i·9876543, donde i va de 1 a 5(cada interación de CV), y que se replica para cada ejecución de cada algoritmo distinto lanzada con cada base de datos. Luego se intercambian las particiones.
El objetivo es que los resultados obtenidos para cada algoritmo sean realmente comparables.

Además de usar **caret** he usado otras librerías como **foreign** para leer datos en arff y **parallel** para paralelizar.
Tanto las metaheurísticas como funciones auxiliares necesarias son de implementación propia. En algunos casos se emplean estructuras de control como breaks ya que R no acepta dobles condiciones en bucles for (por lo que hay que simularlas con if y breaks) y se sustituyen bucles por versiones apply.

El 5x2 está hecho de forma funcional y se encuentra en la sección de ejecución del algoritmo correspondiente, en el Rscript. Se separan ejecuciones de 5 en 5, en train versus test y test versus train. Las tasas de reducción correspondientes de calculan justo debajo.

Análisis de los resultados

Se observa en las tablas de resultados(ver resultados_tablas.ods) que aplicando cualquiera de los tres algoritmos estudiados en esta práctica: BMB,GRASP e ILS, llegamos a mejorar el acierto de test que obtenemos con el 3NN tomando todas las características (menos la clase), por lo que realmente se están seleccionando características relevantes de los conjuntos que nos ayudan a optimizar la solución.

En cuanto a los tiempos de ejecución vemos en las tablas que el algoritmo que más tarda, para las tres bases de datos, es el **GRASP**, seguramente debido al Greedy aleatorizado que incorpora en su primera fase, lo cual ralentiza el proceso. Luego el BMB y por último, el ILS. Aunque en las tablas los números dicen que el ILS tarda más que GRASP y BMB, esto es debido a que la función **sys.time()** que uso para medir tiempos calcula el tiempo secuencial y es **falso**, ya que BMB y GRASP los he lanzado e implementado de forma paralela y con 4 núcleos, mientras que el ILS se implementa y ejecuta de forma secuencial. Teniendo esto en cuenta, pienso que el ILS tarda menos que el BMB y GRASP.

En cuanto a la calidad de los resultados Inicialmente, con 3nn y todas las características obtenemos, en media un acierto test de 0.96, 0.70 y 0.63 para Wdbc, Movement Libras y Aritmia, respectivamente. Estos resultados se mejoran hasta un máximo de 0.98629108 de acierto para test en el caso de wbdc, con ILS, mientras que para Movement Libras la mejora máxima en acierto test es de 0.7932098667 obtenida mediante la ejecución de GRASP, y para Arritmia es de 0.813802075 y se obtiene también mediante GRASP.

A pesar de que los tres algoritmos no están muy alejados en los resultados

que obtienen, sí que difieren notoriamente en el número de caracteristicas que seleccionan cada uno. Por ejemplo, para **Wdbc** la tasa de reducción obtenida es de 50.40, en media, para BMB frente a la 70.64 obtenida para GRASP o la 96.81 obtenida para ILS. Lo cual nos dice que, a pesar de que dan resultados muy parecidos los tres, BMB está seleccionando menos características que GRASP (lo cual era de esperar, pues GRASP primero aplica un Greedy que va construyendo desde 0 una solución inicial) y GRASP está seleccionando menos características que ILS, debido seguramente al operador de mutación de ILS.

De aquí sacamos una conclusión clara: **Wdbc** es una base de datos simple en la que la combinanción de unas pocas características basta para obtener buenos resultados. Por eso no varían mucho las mejoras alcanzadas entre algoritmos, a pesar de que sí lo hacen sus tasas de reducción.

Un caso similar se da con **Movement Libras** donde vemos que las tasas de mejora test obtenidas por los algoritmos van desde 0.77 a 0.79, mientras que las tasas de reducción van desde 49.17 con BMB y 83.02 con GRASP a 99.15 con ILS. Por lo que podemos concluir que a partir de un número de caracteríticas seleccionadas, (creo que más o menos, casi la mitad) es difícil encontrar mejora para **Movement Libras** en nuestro caso de estudio.

Para **Arritmia** las tasas clasificación de test obtenidas por los algoritmos varían entre 0.70 y 0.84 (sin tener en cuenta 3nn), sin embargo las tasas de reducción varían entre 51.47 para BMB, 94.68 para GRASP y 99.7 para ILS. A pesar de que ILS parece que selecciona más características que GRASP, es GRASP el que obtiene el mejor resultado para esta base de datos, por lo que se podría decir que en Arritmia hay un alto porcentaje de características relevantes, pero no todas lo son, por lo que parece que si nos pasamos selecciconando características metemos más ruido que otra cosa.

Conlusión: no hay algoritmo entre estos tres que funcione mejor que los otros. Con unos hemos obtenido mejores resultados para unas bases de datos y peores para otras, pero ninguno de los tres ha resultado ser el infalible, aunque en todo momento han obtenido resultados parecidos de calidad entre sí. Aunque si tuviéramos que elegir un algoritmo de los tres por el tiempo de ejecución, podríamos elegir ILS, pero esto es un poco personal, ya que sabemos que los tiempos de ejecución pueden variar haciendo una implementación más eficiente.

Ilustración de resultados

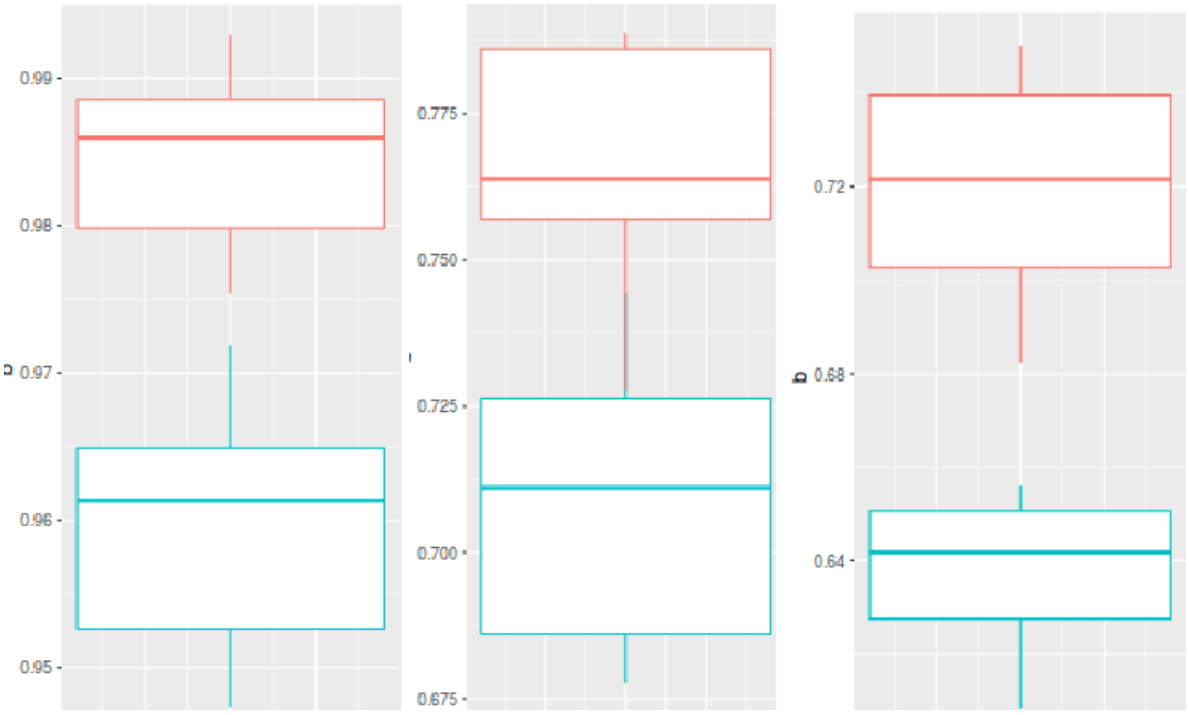

Figura 1: *BMB: Soluciones iniciales y optimizadas para WDBC,Mov.Libras y Arritmia*

En las tres imágenes se representan los resultados de la ejecución de BMB sobre cada dataset. De izquierda a derecha, las imágenes se corresponen con los datasets: WDBC, Movement Libras y Arritmia.

Los **boxplot** azules representan el conjunto de **soluciones iniciales** en cada caso considerado, que vemos que se distribuyen:

- soluciones con tasa class de test entre $0.94-0.97$ en WDBC.
- soluciones con tasa class de test entre $0.68-0.73$ en Movement Libras.
- soluciones con tasa class de test entre $0.60-0.65$ en Arritmia.

Donde la línea en medio de la caja representa la mediana de esos datos, y las líneas que la atraviesan verticalmente representan datos del conjunto que se dan con menor frecuencia. Los boxplot **rojos** representan como se distribuyen

las soluciones obtenidas cuando finaliza el algoritmo, esto es, las iniciales ya optimizadas. Se observa que:

- se alcanzan soluciones con tasa class de test entre $0.98.5-0.99$ (casi) en WDBC.
- se alcanzan soluciones con tasa test entre $0.76-0.78$ en Movement Libras.
- se alcanzan soluciones con tasa test entre $0.68-0.75$ en Arritmia.

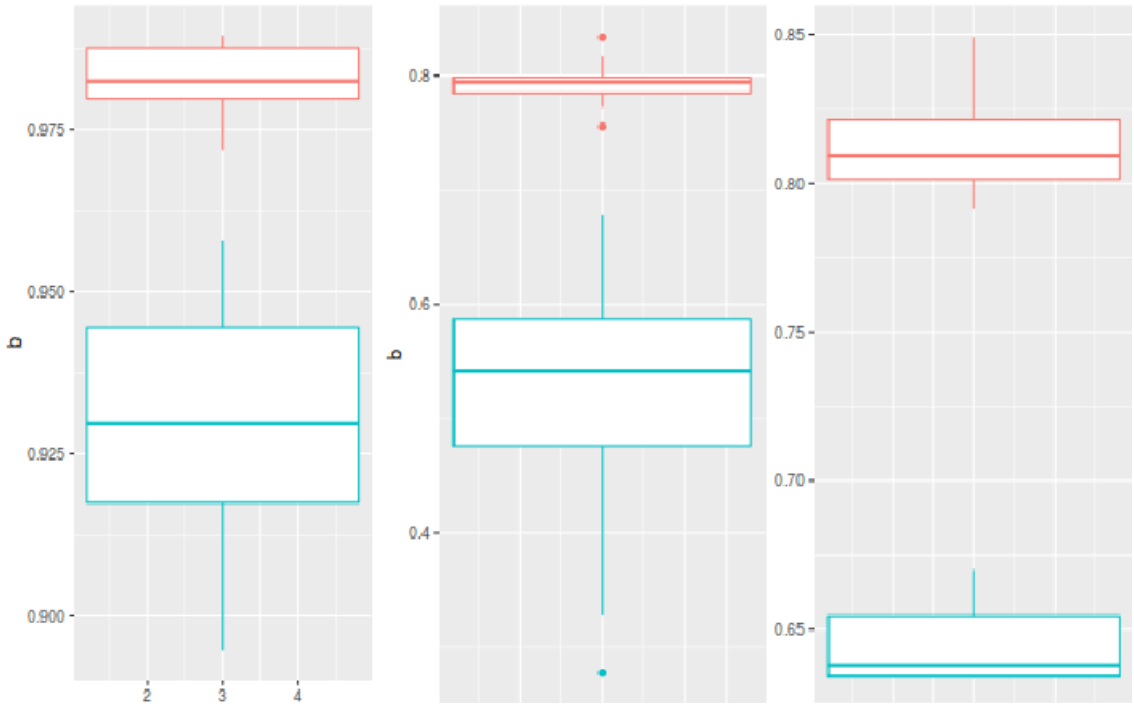

Figura 2: *GRASP: Soluciones iniciales y optimizadas para WDBC,Mov.Libras y Arritmia*

Ahora se representan los resultados de la ejecución del Algoritmo GRASP sobre los tres datasets de estudio. De izquierda a derecha, las imágenes se corresponen con los datasets: WDBC, Movement Libras y Arritmia.
de nuevo los boxplots azules representan los conjuntos de soluciones inciales mientras que los rojos representan las soluciones finales obtenidas. En este caso ambas distribuciones son diferentes a la anterior, especialmente para Arritmia. Con repecto a las soluciones iniciales observamos que:

- presentan una tasa class de test entre $0.90-0.95$ en WDBC.
- soluciones con tasa class de test entre $0.45-0.7$ en Movement Libras.
- soluciones con tasa class de test entre $0.60-0.67$ en Arritmia.

Los puntos rojos y azules que aparecen en el plot de Movement Libras representan **outliers**, esto son, datos que rara vez se dan en el conjunto. Concretamente aquí tenemos un outlier que se corresponde con una solución inicial de 0.32 de acierto en test, y otros dos en las soluciones finales obtenidas, una de 0.83 y otra de 0.75 de acierto test. (una inusualmente alta y otra inusualmente baja).

Sobre las soluciones finales obtenidas mediante GRASP podemos decir que:

- presentan una tasa class de test entre $0.9-0.99$ en WDBC.
- soluciones con tasa class de test entre $0.75-0.8$ en Movement Libras.
- soluciones con tasa class de test entre $0.75-0.85$ en Arritmia.

A veces los boxplots se pueden presentar muy achatados, como en el caso de las soluciones finales en movement Libras, lo significa que todos los datos se concentran en un intervalo pequeño de valores. En este caso, entre 0.75 y 0.80.

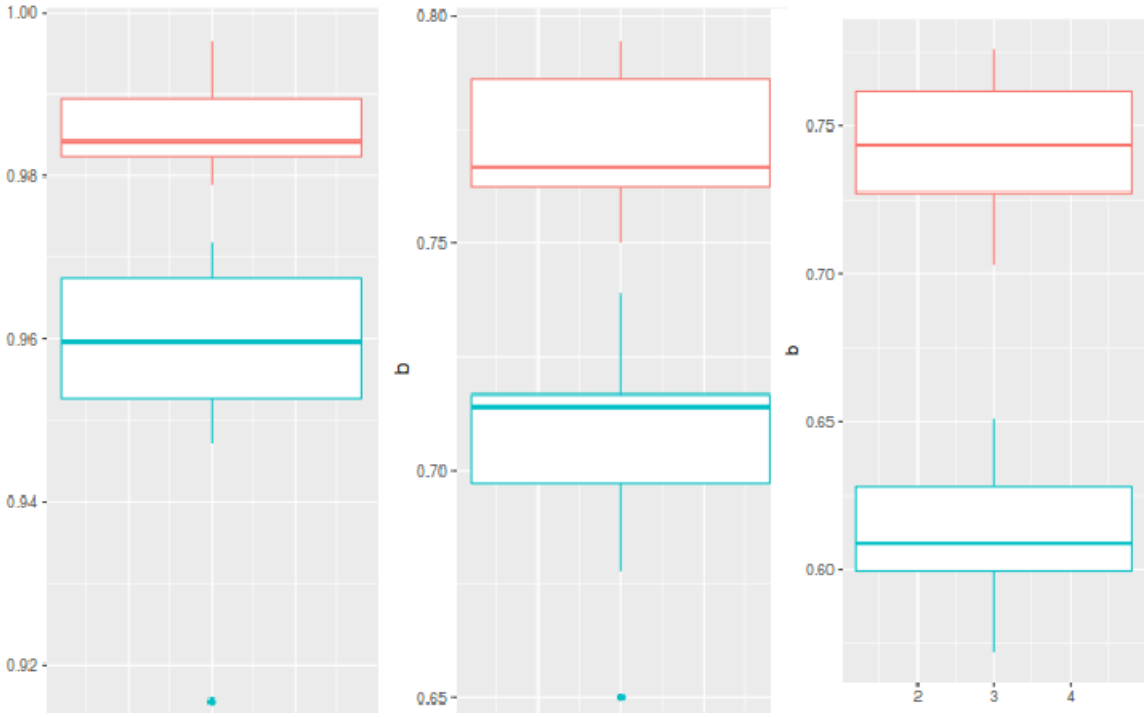

Figura 3: *ILS: Soluciones iniciales y optimizadas para WDBC,Mov.Libras y Arritmia*

Con respecto a las ejecuciones de ILS en las distintas bases de datos, podemos decir, con respecto a las soluciones iniciales, que:

- presentan una tasa class de test entre $0.95-0.97$ en WDBC.
- soluciones con tasa class de test entre $0.69-0.73$ en Movement Libras.
- soluciones con tasa class de test entre $0.55-0.65$ en Arritmia.

Entre las soluciones iniciales se dan dos outliers, uno en Wdbc y otro en Movement Libras. El primero representa una solución que se da de 0.91 de acierto de clasificación en test y el segundo na solución incial de 0.65 acierto test.
Resultados finales obtenidos:

- soluciones con tasa class de test entre $0.98-0.99$ en WDBC.
- soluciones con tasa class de test entre $0.75-0.79$ en Movement Libras.

- soluciones con tasa class de test entre $0.71-0.8$ en Arritmia.

Por último, se ilustra una gráfica comparativa de los resultados globales obtenidos por los algoritmos:

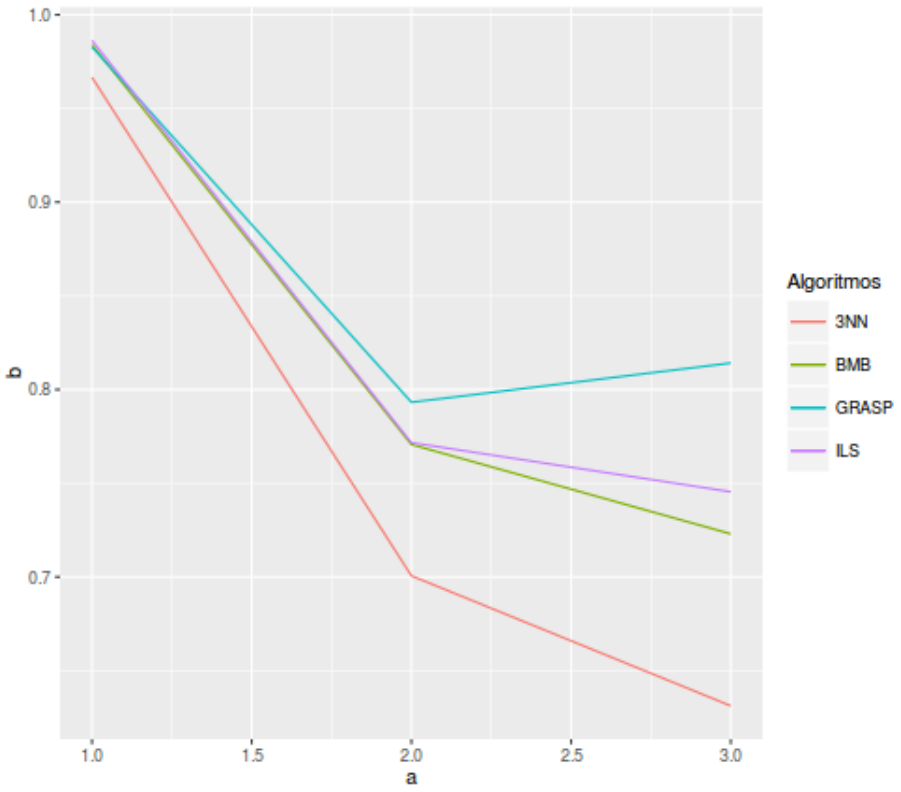

Figura 4: *Comparativa de los algoritmos en términos de calidad media obtenida*

En el gráfico se representan, en el **eje x** valores de 1 al 3 ,donde en el 1 se representan los valores obtenidos para Wdbc, en el 2 los obtenidos para Movement Libras y en el 3 los de Arritmia, para cada algoritmo.
Cada línea de color representa el comportamiento global de un algoritmo. En el **eje y** se representa la tasa de clasificación test alcanzada, que va desde 0 a 1, como máximo.

A la vista del gráfico podemos reincidir en que no hay ningún algoritmo que siempre se comporte mejor que otro. Es evidente que todos superan a 3NN, ya que todos quedan por encima de la línea roja que lo representa, pero no hay ninguna línea que siempre quede por encima de todas las demás.

Aparentemente GRASP queda por encima de los otros para Movement Libras(y=2) y Arritmia(y=3), sin embargo vemos que para Wdbc(y=1) tenemos que es ILS el que queda por encima de él.