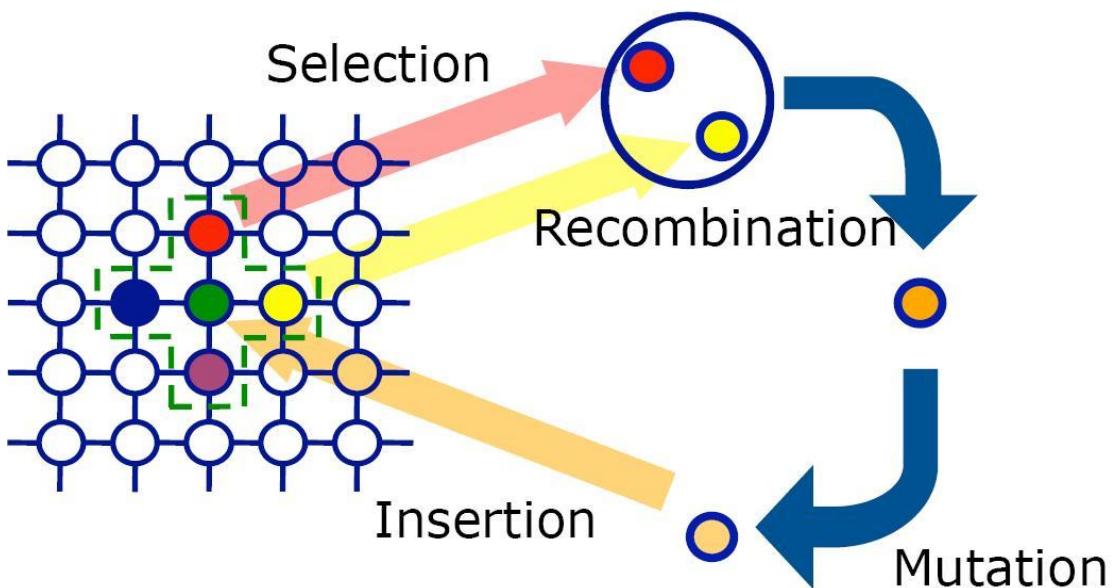


# Práctica 2.b: Técnicas de Búsqueda basadas en Poblaciones para el Problema del Aprendizaje de Pesos en Características



Problema: APC  
Algoritmos: AGG, AGE, AM

Autor: Sergio Aguilera Ramírez  
DNI: 77446084R  
Correo: [sergioaguilera@correo.ugr.es](mailto:sergioaguilera@correo.ugr.es)  
Grupo y horario: grupo 2, miércoles 17:30 – 19:30  
Curso: 2018-2019

## Índice:

c) Descripción del problema .....	2
d) Descripción general de los algoritmos .....	3
a. Representación de soluciones	
b. Representación de la función objetivo	
e) Algoritmos .....	9
a. AGG	
i. BLX	
ii. CA	
b. AGE	
i. BLX	
ii. CA	
c. AM	
i. 10 gen - Prob: 1.0	
ii. 10 gen – Prob: 0.1	
iii. 10 gen – Prob 0.1 (mejores)	
f) Algoritmo de comparación .....	18
g) Procedimiento de desarrollo de la práctica .....	19
h) Experimentos y análisis de los resultados .....	20
a. Casos del problema	
b. Parámetros utilizados en la práctica	
c. Resultados obtenidos	
d. Análisis de los resultados	
i) Bibliografía .....	36

\*\* Han sido corregidos los errores de la practica 1

### c) Descripción del problema

Este problema se basa en la clasificación de características a partir del entrenamiento de datos y su validación mediante conjuntos de datos prueba. Disponemos de diferentes tipos de datos: colposcopias (colposcopy.arff), datos de radar (ionosphere.arff) y datos sobre diferentes tipos de texturas (textura.arff). A partir de estos datos y de los algoritmos desarrollados debemos encontrar un vector de pesos ( $w$ ) que permita obtener un valor de función objetivo elevado. Además, buscamos reducir el número de características, esto nos permite seleccionar aquellas características que nos proporcionen un resultado más óptimo en la clasificación de los datos. En el desarrollo de la práctica utilizaremos el método “5-fold cross validation”, el cual consiste en dividir los datos proporcionados en 5 particiones equivalentes y manteniendo la distribución de las clases.

En los diferentes algoritmos haré uso del clasificador proporcionado por la biblioteca sklearn, K Nearest Neighbors (KNN),  $k=1$ , dicho clasificador permite entrenar y calcular la tasa de acierto en la clasificación de los datos.

Por un lado, haremos uso del clasificador KNN en los diferentes algoritmos desarrollados para encontrar un vector de pesos. Este nos permite dividir los datos según su clase de forma más óptima. Para conseguir esto, debemos dar una mayor importancia a las características con mayor relevancia (dividen mejor los datos en sus respectivas clases). Por otro lado, para la búsqueda de dicho vector utilizaremos algoritmos como búsqueda local, algoritmo genético generacional (AGG), algoritmo genético estacionario (AGE) y algoritmos meméticos(AM), los cuales se basan en la exploración, selección, cruce y mutación de una determinada población, en nuestro caso la población estará constituida por vectores de pesos.

$$tasa_{clas} = 100 * \frac{num_{aciertos} \text{ en } T}{num_{instancias}}$$

$$tasa_{red} = 100 * \frac{num_{caracteristicasnulas}}{num_{caracteristicas}}$$

$$F(w) = \alpha * tasa_{clas}(w) + (1 - \alpha) * tasa_{red}(w)$$

## d) Descripción general de los algoritmos

Para esta práctica vamos a obtener los diferentes resultados de un clasificador KNN, aplicando dos algoritmos para la búsqueda de un vector de pesos:

KNN: este clasificador no utiliza ninguna técnica de reducción de características, ni ningún peso, sino que se encarga de tomar todas las características y en función a estas encuentra el vecino más cercano de cada dato y calcula la predicción de aciertos sobre los datos test.

Relief: en este algoritmo se calcula un vector de pesos que se multiplica por los datos (características). Esto será útil para dar más importancia a aquellas características que se consideren más relevantes respecto a la clasificación de datos, también puede eliminar (reducción) una característica por no ser de utilidad.

Búsqueda Local: este algoritmo consiste en encontrar una solución “óptima”, aunque no es probable que se encuentre, mediante la modificación de cada peso  $w$  para así ir mejorando el porcentaje de aciertos en conjuntos de datos de entrenamiento. Una vez obtenida la mejor  $w$ , ahora sí clasificamos sobre datos de prueba.

El clasificador KNN usado en la práctica es el clasificador de la librería scikit-learn de Python. Así mismo, también hago uso de esta librería para la división de las particiones (StratifiedKFold), usada para dividir el conjunto de datos en “n” subconjuntos manteniendo la distribución de clases. Por otro lado, empleo la librería time para la medición de tiempo y el paquete MinMaxScaler proporcionado por la librería scikit-learn para la normalización de los datos. Finalmente en la lectura de los ficheros de datos .arff utilizo el paquete arff del módulo scipy.io.

Como he comentado en el apartado 1 los datos serán divididos siguiendo el método de k-fold cross validation (para nosotros el número de particiones será 5), para esto he utilizado el paquete StratifiedKFold. Este genera 5 particiones disjuntas al 20% manteniendo la distribución de clases.

- La representación de las soluciones de los diferentes algoritmos son: w es el vector de pesos, w multiplicado a los datos de entrenamiento y prueba, nos proporcionara una mejora en la clasificación de los datos en el KNN. Dicho vector de pesos tiene tantos valores como características presenta nuestros datos. En cada iteración de cada partición fragmentamos los datos en trainx, trainy, testx y testy que contendrán los distintos datos para entrenamiento y validación. Por último, calcularemos la tasa\_clas y tasa\_red para el posterior cálculo de la función objetivo. El objetivo de los algoritmos desarrollados es la maximización de dicha función a través de la obtención de un vector de pesos lo más eficaz posible para la partición precisa de los datos.

En esta práctica vamos a emplear 3 ficheros, colposcopy.arff, ionosphere.arff y textura.arff. Cada fichero será dividido en 5 particiones (comentado anteriormente) y serán ejecutadas por cada algoritmo implementado. Al final se va a obtener la media de los resultados obtenidos en cada ejecución (tasa\_clas\_media, tasa\_red\_media, agr\_medio y tiempo\_medio). Cada fichero presenta una cantidad de características y clases diferentes para así comprobar la eficacia de clasificación de los algoritmos.

/\*

## PRACTICA 2

\*/

En primer lugar, vamos a describir el esquema de representación de las soluciones de los algoritmos genéticos. Para esta representación he usado una lista compuesta por los diferentes individuos de la población. Un individuo está formado por un número determinado de pesos, a los cuales vamos a llamar genes del individuo, estos genes son aquellos valores que nos permitirán asignar una importancia a cada característica del dataset. Por lo que la representación de la población constará de una lista de listas (Matriz):

$$población = [ C_1[ g_{11}, g_{12} \dots g_{1n} ], C_2[ g_{21}, g_{22} \dots g_{2n} ], \dots, C_n[ g_{n1}, g_{n2} \dots g_{nn} ] ]$$

Por otro lado, cada individuo tiene asignado un valor de función objetivo, este valor es lo que distinguirá entre individuos de mejor calidad y de peor, esto lo denominamos fitness. Para la representación de dichos valores he usado una lista la cual contiene los diferentes valores función objetivo de cada cromosoma ordenados de la misma forma que la población:

$$\text{eval\_poblacion} = [ F(C_1), F(C_2) \dots F(C_n) ]$$

Para la generación de la población inicial, cuya representación he descrito en apartados anteriores, hago uso de la función genera\_poblacion(), la cual genera un número determinado de individuos de forma uniforme aleatoria en el intervalo [0,1]. El tamaño impuesto por el profesorado para la población es de 30 individuos.

**Genera\_poblacion (datos,numero\_individuos)****Inicio****Para** cada i **en** el rango [0, número\_individuos]

Añadimos a la lista población un vector aleatorio uniforme entre [0,1]

**Fin****Devolvemos** población**Fin**

El operador de selección usaremos el torneo binario, esto consiste en escoger aleatoriamente dos individuos y de estos dos devolvemos el que tenga mayor valor de función objetivo. En el algoritmo genético generacional usaremos el torneo binario tantas veces como individuos tenga la población, en cambio para el algoritmo genético estacionario lo usaremos solamente 2 veces. Puede suceder que la población de padres este formada por algunos padres repetidos, esto se debe a la aleatoriedad y posiblemente al buen valor de agr.

**Torneo\_binario (población, eval\_poblacion)****Inicio**

Obtenemos aleatoriamente un individuo (p1)

Obtenemos aleatoriamente otro individuo (p2)

**Mientras** p1 sea diferente a p2

Aleatorizamos p2

**Fin****Si** la función objetivo de p1  $\geq$  función objetivo p2 **entonces****Devolvemos** p1**Si no Devolvemos** p2**Fin**

A continuación, vamos a describir los dos operadores de cruces que se han implementado en esta práctica para la combinación de los distintos padres de la población.

Se han desarrollados dos operadores: BLX- $\alpha$  y cruce aritmético.

**BLX- $\alpha$ :** este operador se basa en distribuciones de probabilidad uniforme, BLX- $\alpha$  esta asociada a una alta exploración del espacio de los cromosomas, por lo que este introduce una alta diversidad entre los descendientes. Este operador proporciona una exploración y explotación equilibrado por lo que podemos conseguir mejores resultados. El cruce se realiza respecto a dos cromosomas, cada gen de los descendientes es generado de forma aleatoria en el siguiente rango:

- $C_{max} = \max\{C_{1i}, C_{2i}\}$
- $C_{min} = \min\{C_{1i}, C_{2i}\}$
- $I = C_{max} - C_{min}$
- $rango = [C_{min} - I * \alpha, C_{max} + I * \alpha]$

En cada cruce formado por dos padres de la población se generan dos descendientes distintos.

Cruce\_BLX (cromosoma1, cromosoma2)

**Inicio**

Para cada i en el rango [0, tam\_cromosoma1]

Calculamos cmax entre cromosoma1[i] y cromosoma2[i]

Calculamos cmin entre cromosoma1[i] y cromosoma2[i]

Obtenemos I, mediante la fórmula  $I = cmax - cmin$

Añadimos a hijo1 un valor uniforme entre ( $cmin - I * 0.3$ ,  $cmax + I * 0.3$ )

Añadimos a hijo2 un valor uniforme entre ( $cmin - I * 0.3$ ,  $cmax + I * 0.3$ )

**Fin**

Normalizamos los datos

**Devolvemos** hijo1, hijo2

**Fin**

**Cruce aritmético:** este operador está basado en realizar la media aritmética de cada par de genes de los dos cromosomas escogidos para cruzar tal y como indica la práctica, pero esto supone la generación de un solo hijo, por lo que se aplica la media ponderada, siendo Alpha = 0.3.

**Cruce\_aritmetico** (cromosoma1, cromosoma2)

**Inicio**

Hijo1 = alpha \* cromosoma1 + (1-alpha) \* cromosoma2

Hijo2 = (1-alpha) \* cromosoma1 + alpha \* cromosoma2

Normalizamos los datos

**Devolvemos** hijo1, hijo2

**Fin**

A continuación, se muestra el cálculo del valor agr asociado a cada individuo de la población. Dicho valor se obtiene mediante la clasificación de los datos de entrenamiento, mediante el entrenamiento de los datos train multiplicados por el vector de pesos correspondiente (individuo), esto nos proporciona una tasa de aciertos (%), con dicha tasa de acierto y la tasa de reducción calculada con anterioridad podremos obtener el valor de la función objetivo. En cada clasificación introducimos el leave one out, esto consiste en coger el segundo vecino más cercano, ya que como sabemos el primer vecino más cercano es él mismo, la función utilizado para esto es kneighbors(datos, n\_neighbors=2), en el pseudocodigo se muestra como realizamos el leave one out

**Evalu\_poblacion** (trainx, trainy, población, iteraciones)

**Inicio**

Para cada i en el rango (tamaño población)

Incrementamos iteraciones

Contamos el número de reducciones

Truncamos aquellos valores menores que 0.2 de la población

Multiplicamos el trainx por el vector de pesos población[i]

Creamos el clasificador y entrenamos

Obtenemos el segundo vecino más cercano (leave one out) respecto a Trainx

Guardamos las etiquetas del vecino más cercano

y calculamos la tasa de acierto y de reducción

(tasa\_clas = np.mean(trainy[ind\_near] == trainy) \* 100 )

Obtenemos la función objetivo y la añadimos al vector eval\_poblacion

**Fin**

**Devolvemos** eval\_poblacion, iteraciones

**Fin**

Por último, mostraremos el esquema seguido para la mutación de la población de hijos. Como sabemos podemos determinar el número de genes estimados para mutar, ya que teniendo una probabilidad del 0.001 y por ejemplo un número total de genes de 6000 podemos determinar mediante el siguiente cálculo `num Esperados_mutacion = 0.001 * 6000`, que el número de mutaciones es igual a 6, esto nos proporciona una ventaja ya que solo generaremos aleatoriamente una cantidad mínima de valores. En cada mutación se generará un valor aleatorio `i` para seleccionar el cromosoma de la población, y otro valor `j` para seleccionar el gen de dicho cromosoma que mutará, mediante la suma de un valor normal comprendido en el rango [0.0 , 0.3].

## AGG

### **Esquema de mutación**

Calculamos el número de mutaciones estimadas:

(round (probabilidad \* número de genes))

Establecemos mutaciones a 0

**Mientras** mutaciones < número estimado mutaciones

Generamos `i` aleatoriamente en el rango [0, tam(hijos)] para elegir el cromosoma

Generamos `k` aleatoriamente en el rango [0, tam(genes de cada cromosoma)] para elegir el gen a mutar

Creamos un valor normal comprendido entre [0.0, 0.3]

Sumamos al gen seleccionado el valor normal

Normalizamos la suma

Incrementamos mutaciones

**Fin**

## AGE

Para AGE no podemos seguir el mismo esquema debido a que con una probabilidad del 0.001 y dos hijos, no habría mutación, por lo que realizamos la mutación a nivel de cromosoma.

### **Esquema de mutación**

Calculamos la probabilidad del cromosoma (`prob_mu * len(hijos[0])`)

**Para cada `i` en rango (2):**

Generamos aleatoriamente un numero entre [0,1]

Si `n < prob_cromosoma` entonces

Generamos `k` aleatoriamente en el rango [0, tam(genes de cada cromosoma)] para elegir el gen a mutar

Creamos un valor normal comprendido entre [0.0, 0.3]

Sumamos al gen seleccionado el valor normal

Normalizamos la suma

**Fin**

## e) Descripción y pseudocódigo de los algoritmos

### Funciones Comunes a todos los Algoritmos:

En este algoritmo particionamos los conjuntos train y test de cada partición

```
Prepara_particiones (i,j,x,y)
Inicio
    Para cada k en i
        trainx añado x[k]
        trainy añado y[k]

    Para cada t en j
        testx añado x[t]
        testy añado y[t]

    Devolvemos trainx, trainy, testx, testy
Fin
```

Esta función calcula el porcentaje de aciertos obtenidos en el conjunto de datos pasado como argumento, entrenando primero con los datos train y validando con los datos test, clasificador KNN

```
calcula_funcion_objetivo (ind_near, trainy, cont_reduccion, w)
Inicio
    Calculamos tasa de acierto ((np.mean(trainy[ind_near]==trainy) * 100)
    Calculamos tasa de reducción
    Calculamos función objetivo

    Devolvemos función objetivo, tasa_clas, tasa_red
Fin
```

## BÚSQUEDA LOCAL

**busqueda\_local** (trainx, trainy, testx, testy)

**Inicio**

Establecemos w a 0

Clasificamos, entrenamos y calculamos la función objetivo mejor

**Mientras** vecinos < 20 \* tamaño de características y it < 15000

**Para** cada l en tam(w)

        Creamos valor normal z entre 0.0 y 0.3

        Sumamos z a w[l] y normalizamos

        Creamos w auxiliar y truncamos los pesos menores a 0.2

        Multiplicamos trainx por el vector de pesos truncado

        Clasificamos los datos de entrenamiento (calculamos tasa\_clas y tasa\_red)

        Calculamos la función objetivo con el vector auxiliar

**Si** funcion\_objetivo\_mejor es menor función\_objetivo\_actual

            Actualizamos la función\_objetivo\_mejor

            Actualizamos vecinos = 0

            Break

**Si** no

        Recuperamos w[l] anterior

        Actualizamos vecinos += 1

**Fin**

**Fin**

Clasificamos, entrenamos y predecimos (con datos test)

Calculamos la función objetivo final

**Devolvemos** tasa\_red, función\_objetivo, tasa\_clas, tiempo

**Fin**

-Este algoritmo consiste en la búsqueda de un vector de pesos, el cual sea superior a todos los calculados anteriormente, hasta llegar al límite de superar un número de vecinos determinado o un número máximo de iteraciones, en cada iteración modificamos una componente del vector de pesos sumándole un valor normal entre [0.0,0.3] y comprobamos su clasificación mediante los datos de entrenamiento, si dicha función objetivo es mejor que la global, guardamos su valor y establecemos el contador de vecinos a 0 y volvemos a iterar sobre el nuevo conjunto de pesos, en caso contrario, si este valor no supera al global recuperamos el valor de la componente y pasamos a la siguiente componente, hasta llegar a un número máximo de exploraciones de vecinos, en este caso no habrá ninguna modificación del vector de pesos la cual consiga un resultado de la función objetivo mejor que la actual, o de iteraciones, cuando esta exploración ha finalizado obtendremos el mejor vector de pesos calculado en la búsqueda. Para finalizar clasificamos

y entrenamos de nuevo los datos y calculamos la tasa de acierto, ahora sí, respecto a los datos de prueba.

/\*

## PRACTICA 2

\*/

### a) Algoritmo Genético Generacional (AGG)

El primer algoritmo desarrollado es el AGG elitista. Este algoritmo consta de diferentes partes en su implementación, las cuales son selección, cruce, mutación y reemplazamiento.

$$poblacion_{inicial} = \{C_1, C_2 \dots C_n\}$$

- Selección: la selección se lleva a cabo mediante torneo binario, el cual selecciona aleatoriamente dos individuos de la población y se queda con el mejor de los dos, formando una lista de padres de igual tamaño a la población actual.

$$P_{padres} = \{C'_1, C'_2 \dots C'_n\}$$

- Cruce: utilizamos dos operadores BIX- $\alpha$  y cruce aritmético, los cuales combinan un determinado número de parejas de padres, generando descendientes. Los padres son emparejados de manera fija, es decir, el primero con el segundo, el tercero con el cuarto etc.

$$P_{descendientes} = \{C''_1, C''_2 \dots C''_n\}$$

- Mutación: como ya hemos explicado en apartados anteriores, la mutación se realiza sobre un número determinado de genes de manera aleatoria.

$$P_{descendientes} = \{C''_1, C''_2 \dots C''_n\}$$

- Reemplazamiento: este algoritmo es elitista, es decir, cuando genera la siguiente población, que sería la lista de hijos después de la mutación, debemos reemplazar el peor cromosoma de los hijos por el mejor individuo de la población actual.

$$P_{descendientes} = \{C''_1, C''_2 \dots C''_{n-1}, C_{mejor}\}$$

El pseudocódigo del algoritmo genético generacional elitista es el siguiente:

```
AGG (trainx, trainy, testx, testy, tipo_cruce)
Inicio
    Generamos la población inicial (función: genera_poblacion())
    Calculamos el valor correspondiente a cada individuo (evalua_poblacion())
    Mientras num_iteraciones < 15000

        # selección
        Para cada i en el rango (0, tam (población))
            Insertamos en padres el resultado obtenido mediante el torneo binario
        Fin

        # cruce
        Calculamos el número de cruces a realizar (0.7 * tam(población)/2)
        Establecemos numero_cruces a 0
        Si tipo_cruce == 'BLX' entonces
            Mientras numero_cruces < máximo_cruces
                Calculamos descendientes mediante el operador BLX
                Insertamos los descendientes en la lista de hijos
                Incrementamos en dos el número_cruces
            Fin
        Else
            Mientras numero_cruces < máximo_cruces
                Calculamos descendientes mediante el operador cruce aritmético
                Insertamos los descendientes en la lista de hijos
                Incrementamos en dos el número_cruces
            Fin
        Fin
        Añadimos a los hijos aquellos padres que no han cruzado

        # mutación
        Realizamos mutación del AGG
        Aumentamos una iteración por cada cálculo de agr
        Evaluamos lista de mutaciones (hijos mutados y no mutados)

        # Elitismo
        Sustituimos en la lista de hijos el peor individuo por el mejor individuo de la población actual

        Actualizamos la población y calculamos su evaluación
    Fin
    Calculamos la tasa_clas, tasa_red y función objetivo mediante la clasificación de los datos test
        (multiplicamos los datos por el mejor individuo de la población final)
    Devolvemos tasa_red, función_objetivo, tasa_clas , tiempo
Fin
```

AGG hace uso de las funciones torneo\_binario(), genera\_poblacion(), cruce\_BLX() y cruce\_aritmetico() descrita en el apartado anterior.

### b) Algoritmo Genético Estacionario (AGE)

El segundo algoritmo desarrollado es el AGE. Este algoritmo consta de diferentes partes en su implementación, las cuales son selección, cruce, mutación y reemplazamiento, las mismas que el algoritmo AGG.

$$poblacion_{inicial} = \{C_1, C_2 \dots C_n\}$$

- Selección: la selección se lleva a cabo mediante torneo binario, el cual selecciona aleatoriamente dos individuos de la población y se queda con el mejor de los dos, formando una lista de padres de tamaño 2.

$$P_{padres} = \{C'_1, C'_2\}$$

- Cruce: utilizamos dos operadores BIX- $\alpha$  y cruce aritmético, los cuales combinan los padres generados, dando lugar a los correspondientes descendientes.

$$P_{descendientes} = \{C''_1, C''_2\}$$

- Mutación: como ya hemos explicado en apartados anteriores, la mutación se realiza sobre un número determinado de genes de manera aleatoria.

$$P_{descendientes} = \{C''_1, C''^{m_1}_2\}$$

- *Reemplazamiento*: en el reemplazamiento los individuos obtenidos después de la mutación compiten por acceder a la población, es decir, si el peor de los individuos de la población actual tiene una función objetivo menor que la de algunos de los dos descendientes, este será reemplazado por el de mayor agr.

$$P = \{C''_1, C''^{m_2}_2 \dots C''_{n-1}, C_{mejor}\}$$

El pseudocódigo del algoritmo genético generacional elitista es el siguiente:

```
AGE (trainx, trainy, testx, testy, tipo_cruce)
Inicio
    Generamos la población inicial (función: genera_poblacion())
    Calculamos el valor correspondiente a cada individuo (evalua_poblacion())
    Mientras num iteraciones < 15000
        # selección
        Para cada i en el rango (0,2)
            Insertamos en padres el resultado obtenido mediante el torneo binario
        Fin

        # cruce
        Si tipo_cruce == 'BLX' entonces
            Calculamos descendientes mediante el operador BLX
            Insertamos los descendientes en la lista de hijos
        Else
            Calculamos un descendiente mediante el operador cruce aritmético
            Sumamos al hijo calculado anteriormente un vector uniforme Alpha entre 0.0 y 0.3 de igual
            tamaño y normalizamos los valores
            Insertamos los descendientes en la lista de hijos
        Fin

        # mutación
        Realizamos mutación del AGE

        Aumentamos una iteración por cada cálculo de agr
        Evaluamos la lista de hijos

        # Elitismo
        Para cada i en rango (0,2)
            Cogemos el peor de la población (indi_peor)
            Si indi_peor.agr < hijo[i].agr entonces intercambiamos
        Fin
        Actualizamos la población y calculamos su lista de evaluación
    Fin

    Calculamos la tasa_clas, tasa_red y función objetivo mediante la clasificación de los datos test
    (multiplicamos los datos por el mejor individuo de la población final)
    Devolvemos tasa_red, función_objetivo, tasa_clas, tiempo
Fin
```

En el AGE solo se seleccionan y cruzan dos padres, por lo que solo se obtendrán dos descendientes. Estos descendientes podrán ser mutados de forma aleatoria. Para implementar el elitismo, los descendientes mutados deben competir para entrar en la población, esto consiste en coger el peor individuo de la población actual y compararlo con alguno de estos descendientes si alguno supera a dicho individuo se sustituye en la población, si no estos se rechazan.

### c) Algoritmo Memético (AM)

Los algoritmos meméticos se basan en el refinamiento de los cromosomas aplicando una búsqueda local sobre ellos. Los algoritmos meméticos son una hibridación entre los AGG y la búsqueda local. Se va a llevar a cabo tres métodos para aplicar la búsqueda local a los diferentes individuos de la población en dicho algoritmo.

- 1- AM-(10,1.0): en el que se aplicara la búsqueda local a todos los individuos de la población cada 10 generaciones.
- 2- AM-(10,0.1): en él se aplicará la búsqueda local a los individuos de forma aleatoria con probabilidad 0.1, es decir, el número de cromosomas a los que le aplicaremos la BL será  $0.1 * \text{tam}(\text{población})$ .
- 3- AM-(10,0.1, mejores): se aplicará la búsqueda local a los mejores individuos de la población, de modo que el número de individuos a los que se le aplicaran la BL es  $0.1 * \text{tam}(\text{población})$ .

El esquema general para los diferentes AM son equivalentes al algoritmo AGG, la diferencia es la aplicación de la búsqueda local después de realizar el elitismo sobre los descendientes.

```
AM (trainx, trainy, testx, testy, tipo_cruce)
Inicio
    Generamos la población inicial (función: genera_poblacion())
    Calculamos el valor correspondiente a cada individuo (evalua_poblacion())
    Mientras num_iteraciones < 15000
        # selección
        Para cada i en el rango (0, tam (población))
            Insertamos en padres el resultado obtenido mediante el torneo binario
        Fin
        # cruce
        Calculamos el número de cruces a realizar ( $0.7 * \text{tam}(\text{población})/2$ )
        Establecemos numero_cruces a 0
        Si tipo_cruce == 'BLX' entonces
            Mientras numero_cruces < máximo_cruces
                Calculamos descendientes mediante el operador BLX
                Insertamos los descendientes en la lista de hijos
                Incrementamos en dos el número_cruces
            Fin
        Else
            Mientras numero_cruces < máximo_cruces
                Calculamos descendientes mediante el operador cruce aritmético
                Insertamos los descendientes en la lista de hijos
                Incrementamos en dos el número_cruces
            Fin
        Fin
        Añadimos a la lista de hijos aquellos padres que no han cruzado
        Realizamos mutación del AGG
        Evaluamos lista de hijos
        Sustituimos en la lista de hijos el peor individuo por el mejor individuo de la población actual
        Aplicamos BL (3 hibridaciones diferentes)
        Actualizamos la población
    Fin
    Calculamos la tasa_clas, tasa_red y función objetivo mediante la clasificación de los datos test (multiplicamos los datos por el mejor individuo de la población final)
    Devolvemos tasa_red, función_objetivo, tasa_clas, tiempo
Fin
```

A continuación, se mostrará el esquema seguido para cada método de aplicación de la búsqueda local:

i) AM-(10,1.0)

En este método se aplica la búsqueda local cada 10 generaciones a todos los cromosomas actualizando cada uno de estos por su w y agr obtenidos mediante este algoritmo.

**Esquema de búsqueda**

```
Si generación % 10 == 0 and generación != 0
  Para cada i en tam(población)
    BL(población[i])
    Añado al vector pobla_ls la w resultante
    Añado al vector eval_ls el valor agr respectivo
  Fin
  Actualizo el vector mutaciones = pobla_ls
  Actualizo el vector de eval_mutacion = eval_ls
Fin
```

ii) AM-(10,0.1)

En este método se aplica la búsqueda local cada 10 generaciones a un número determinado de cromosomas de forma aleatoria, siendo este valor el resultado de  $0.1 * \text{len}(\text{población})$ , actualizando cada uno de estos por su w y agr obtenidos mediante este algoritmo.

**Esquema de búsqueda**

```
Si generación % 10 == 0 and generación != 0
  Num_busqueda = 0,1 * tam (hijos)
  Para cada i en tam(0, num_busqueda)
    Obtengo un individuo aleatoriamente (guardamos índice en j)
    BL(población[j])
    Actualizo el individuo con la w y agr obtenida
  Fin
Fin
```

### iii) AM-(10,1.0, mejores)

En este método se aplica la búsqueda local cada 10 generaciones a un número determinado de cromosomas, la selección de los cromosomas se realiza mediante la estrategia primero el mejor, siendo este valor el resultado de  $0.1 * \text{len}(\text{población})$ , actualizando cada uno de estos por su w y agr obtenidos mediante este algoritmo.

#### Esquema de búsqueda

```
Si generación % 10 == 0 and generación != 0
    Num_busqueda = 0,1 * tam(hijos)
    Para cada i en tam(0,num_busqueda)
        Obtengo el mejor individuo entre los hijos (obtengo el índice en j)
        BL(población[j])
        Actualizo el individuo con la w y agr obtenida
    Fin
Fin
```

### Búsqueda local para los AM

#### busqueda\_local\_memeticos (trainx, trainy, testx, testy, w, it1)

##### Inicio

Clasificamos, entrenamos y calculamos la función objetivo mejor

Mientras vecinos <  $2 * \text{tamaño de características}$

Para cada l en tam(w)

Creamos valor normal z entre 0.0 y 0.3

Sumamos z a w[l] y normalizamos

Creamos w auxiliar y truncamos los pesos menores a 0.2

Multiplicamos trainx por el vector de pesos truncado

Clasificamos los datos de entrenamiento (calculamos tasa\_clas y tasa\_red)

Calculamos la función objetivo con el vector auxiliar

Si función\_objetivo\_mejor es menor función\_objetivo\_actual

Actualizamos la función\_objetivo\_mejor

Actualizamos vecinos = 0

Break

Si no

Recuperamos w[l] anterior

Actualizamos vecinos += 1

Fin

Fin

Devolvemos tasa\_red, función\_objetivo, tasa\_clas, tiempo

Fin

## f) Algoritmo de comparación

### KNN

**KNN** (trainx, trainy, testx, testy)

**Inicio**

Preparamos las particiones trainx, trainy, testx, testy  
Clasificamos, entrenamos y calculamos la tasa de acierto  
Calculamos la función objetivo

Devolvemos tiempo, tasa\_clas, función\_objetivo

**Fin**

-Este clasificador consiste en dividir los datos en entrenamiento y prueba, calcular la tasa\_clas mediante el clasificador KNN de scikit-learn y por último calcular la función objetivo, la tasa\_red es 0.0 ya que no se utiliza ningún truncamiento ni eliminación de características, este clasificador tampoco consta de ningún tipo de pesos de características por lo que su predicción puede resultar menos acertada que otros algoritmos que utilizan la obtención de un vector de pesos. La Alpha para todos los algoritmos será de 0.5.

### RELIEF

**Relief** (trainx, trainy, testx, testy)

**Inicio**

Establecemos w a 0

Preparamos las particiones trainx, trainy, testx, testy

**Para** cada i **en** tam(trainx)

Calculamos vecino amigo y vecino enemigo

Actualizamos w ( $w = w + |\text{trainx}[i] - \text{enemigo}| - |\text{trainx}[i]-\text{amigo}|$ )

**Fin**

**Para** cada j **en** tam(w)

**Si**  $w[j] < 0.0$ :

$w[j] = 0.0$

**Else:**

$w[j] = w[j] / \max(w)$

Truncamos a 0.0 aquellos pesos menores que 0.2

Multiplicamos trainx y testx por w

Clasificamos, entrenamos y calculamos la tasa de acierto

Calculamos la función objetivo

**Devolvemos** tasa\_red, función\_objetivo, tasa\_clas, tiempo

**Fin**

-Este algoritmo consiste en buscar el vecino más cercano de la misma clase y el vecino más cercano de diferente clase de cada conjunto de características de los datos trainx, es decir, calcular la distancia “euclídea” a cada vecino y elegir el más cercano, y actualizar w en cada iteración mediante la fórmula expresada anteriormente. También recalcar que en la función buscar\_vecino\_amigo\_enemigo () se tiene en cuenta el leave one out, lo cual consiste en no considerar como vecino a el mismo.

Método que obtiene amigo y enemigo

```
buscar_vecino_amigo_enemigo (trainx,trainy,d)
Inicio
    Establecemos id_amigo y id_enemigo a 0
    Establecemos val_amigo y val_enemigo a 9999.0
    Eliminamos a el mismo (leave one out) q = np.delete(trainx,d,0)
    Para cada j en tam(q)
        Calculamos distancia al vecino (sumatoria_d – sum(q[j])**2)
        Si trainy[j] == trainy[d] y dis < val_amigo
            Actualizamos id_amigo = j
            Actualizamos val_amigo = dis
        Si trainy[j] != trainy[d] y dis < val_enemigo
            Actualizamos id_enemigo = j
            Actualizamos val_enemigo = dis
    Fin
    Devolvemos trainx[id_amigo], trainx[id_enemigo]
Fin
```

## g) Procedimiento de desarrollo de la práctica

Esta práctica esta realizada en Python, este lenguaje proporciona módulos para la normalización, partición, lectura y clasificación de los datos. Para la implementación del clasificador KNN, como he comentado en los apartados anteriores, he utilizado el clasificador del módulo scikit-learn. Los distintos módulos utilizados para esta práctica son scipy para la lectura de los ficheros con extensión .arff, sklearn.neighbors para el clasificador KNN, sklearn.model\_selection para las particiones, sklearn.preprocessing para la normalización de los datos, time para el cálculo de los tiempos y numpy para el control de los arrays.

## **h) Experimentos y análisis**

### **a- Casos del problema:**

Se han utilizado 3 conjuntos de datos para la ejecución de los distintos algoritmos, estos datos son:

1. Colposcopy (colposcopia): procedimiento ginecológico basado en la exploración del cuello del útero. Este dataset consta de 287 ejemplos, los cuales están compuestos por 62 características y existen 2 clases.
2. Ionosphere: datos de radar, los cuales se basan en la medición de electrones libres en la ionosfera. Este dataset está compuesto por 352 ejemplos, cada ejemplo consta de 34 características y como en el dataset anterior existen dos clases.
3. Texture: Este dataset recoge características de los diferentes tipos de texturas, por ejemplo: papel, césped, piel etc. Este conjunto dispone de 550 ejemplos compuestos por 40 características cada uno y este a diferencia de los demás dataset consta de 11 clases.

### **b- Parámetros utilizados en la práctica:**

En el desarrollo de la práctica, se han utilizado diferentes parámetros aplicados a los algoritmos. El valor de Alpha para el cálculo de la función objetivo de 0.5 tal y como indica el guion de prácticas, el número de vecinos utilizado en el clasificador es 1.

En el algoritmo de búsqueda se han empleado dos condiciones de parada, tanto un número máximo de iteraciones = 15000 como un número máximo de exploración de vecinos =  $20 * \text{len}(w)$ . La semilla escogida es 2, se podría utilizar cualquier semilla mientras que en todos los algoritmos se utilice la misma. La semilla es fijada al principio del código “main”. Al utilizar la opción shuffle=True en la generación de las particiones (StratifiedKFold), debemos especificar el número de la semilla (random\_state=2).

Para el desarrollo de la práctica he utilizado diferentes valores de probabilidades y tamaño de poblaciones (como se especifica en el guion de prácticas). En los algoritmos AGG y AGE el tamaño de la población es de 30 individuos (cromosomas). La probabilidad de cruce para el AGG es del 70%, en cambio la del AGE es del 100%. La probabilidad de mutación para estos dos algoritmos es la misma, esta probabilidad es del 1%. La condición de finalización del AGG y AGE es evaluar 150000 veces la función objetivo.

Para los algoritmos meméticos el tamaño de la población es de 10 individuos, el tamaño es menor que el del resto de algoritmos ya que al aplicar la BL el tiempo de ejecución del algoritmo podría extenderse demasiado. Las probabilidades de cruce y mutación son el 70% y 1% respectivamente. El número de iteraciones a realizar en la búsqueda local es de  $2 * \text{len}(\text{poblacion}[0])$ , por último la condición de parada de los AM son 150000 evaluaciones de la función objetivo.

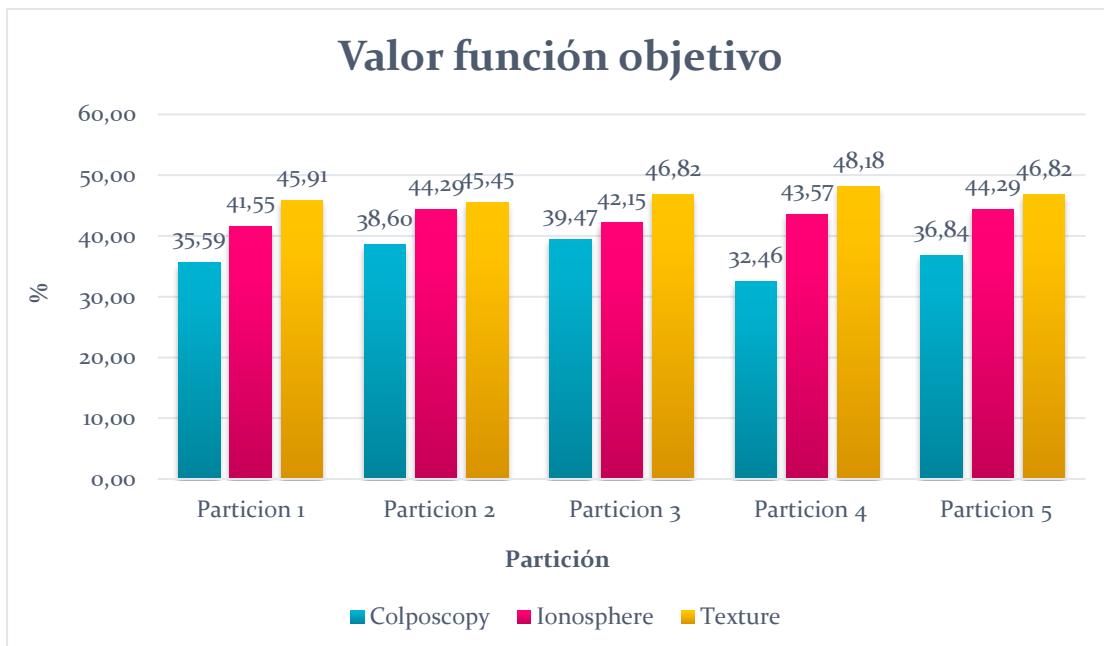
### c- Resultados obtenidos:

En este apartado mostraremos las tablas de resultados obtenidos por cada algoritmo, un gráfico de comparación entre las diferentes particiones y dataset de los experimentos y un breve análisis sobre los resultados obtenidos. Al finalizar dichas tablas y gráficas se realizará el análisis de manera más específica sobre los datos obtenidos.

## KNN

Tabla 5.1: Resultados obtenidos por el algoritmo KNN en el problema del APC

Colposcopy				Ionosphere				Texture				
%_clas	%red	Agr.	T	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T	
Partición 1	71,19	0,00	35,59	0,01	83,10	0,00	41,55	0,04	91,82	0,00	45,91	0,04
Partición 2	77,19	0,00	38,60	0,01	88,57	0,00	44,29	0,03	90,91	0,00	45,45	0,02
Partición 3	78,95	0,00	39,47	0,01	84,29	0,00	42,15	0,03	93,64	0,00	46,82	0,04
Partición 4	64,91	0,00	32,46	0,01	87,14	0,00	43,57	0,03	96,36	0,00	48,18	0,04
Partición 5	73,68	0,00	36,84	0,01	88,57	0,00	44,29	0,03	93,64	0,00	46,82	0,04
Media	73,18	0,00	36,59	0,01	86,33	0,00	43,17	0,03	93,27	0,00	46,64	0,04

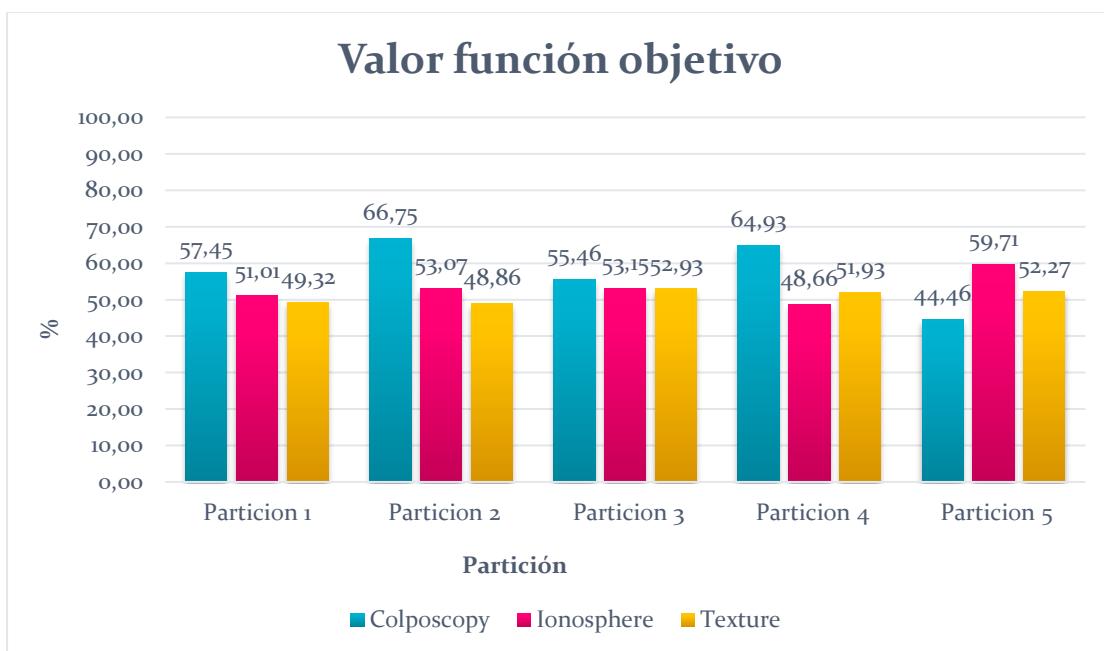


Observando los resultados, vemos como obtenemos mejores resultados de tasa de acierto en el dataset Texture, esto es debido a la mayor diversidad entre sus clases respecto a los demás conjuntos (y no se utiliza ponderación), en consecuencia, en este conjunto obtenemos mejores valores de agr. La reducción en este algoritmo es 0 en todos los casos.

## RELIEF

Tabla 5.1: Resultados obtenidos por el algoritmo RELIEF en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T
Partición 1	74,58	40,32	57,45	0,72	87,32	14,71	51,01	0,70	93,64	5,00	49,32	1,97
Partición 2	75,44	58,06	66,75	0,67	91,43	14,71	53,07	0,72	92,73	5,00	48,86	1,98
Partición 3	75,44	35,48	55,46	0,65	85,71	20,59	53,15	0,67	93,36	12,50	52,93	1,95
Partición 4	70,18	59,68	64,93	0,66	91,43	5,88	48,66	0,61	96,36	7,50	51,93	2,16
Partición 5	6,67	82,26	44,46	0,66	90,00	29,41	59,71	0,66	94,55	10,00	52,27	2,15
Media	60,46	55,16	57,81	0,67	89,18	17,06	53,12	0,67	94,13	8,00	51,06	2,04

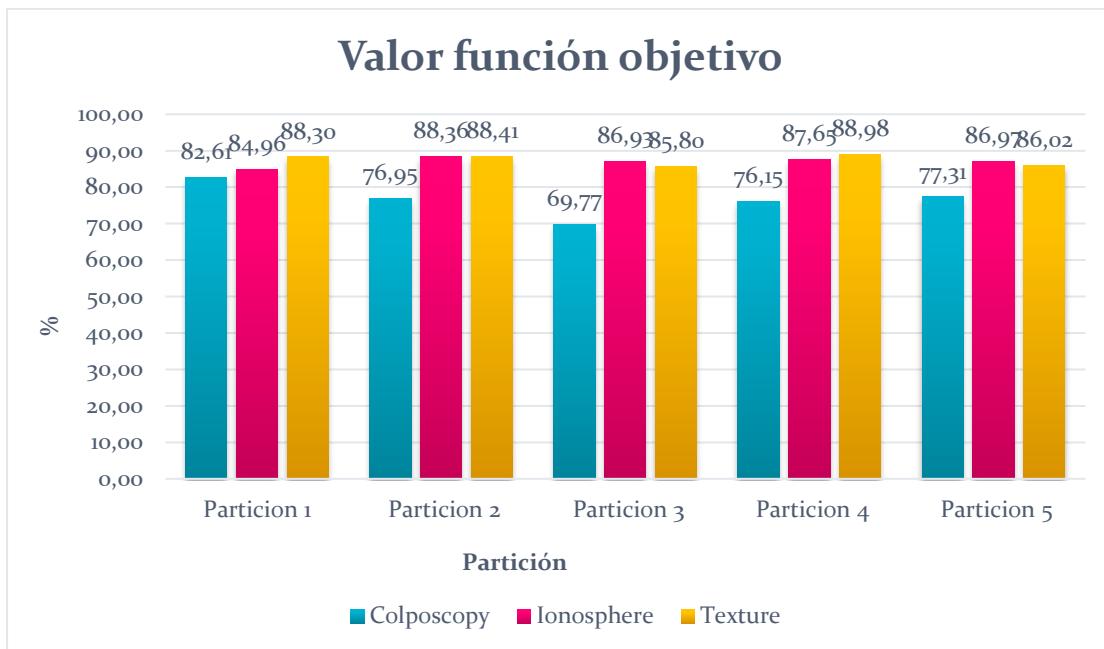


En el algoritmo Relief, podemos observar como en el dataset Colposcopy obtenemos mejores valores de reducción con respecto a los demás conjuntos, en cambio la tasa de acierto es menor. En comparación sobre la función objetivo en el dataset Colposcopy se obtiene de media un mayor valor de agr.

## BL

Tabla 5.1: Resultados obtenidos por el algoritmo BL en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T
Partición 1	81,36	83,87	82,61	31,26	81,69	88,24	84,96	4,73	89,09	87,50	88,30	16,86
Partición 2	68,42	85,48	76,95	14,39	91,43	85,29	88,36	8,97	91,82	85,00	88,41	13,16
Partición 3	70,18	69,35	69,77	20,45	88,57	85,29	86,93	5,41	89,09	82,50	85,80	12,06
Partición 4	68,42	83,87	76,15	15,29	90,00	85,29	87,65	6,36	95,45	82,50	88,98	18,74
Partición 5	77,19	77,42	77,31	17,73	85,71	88,24	86,97	5,45	84,55	87,50	86,02	15,05
Media	73,11	80,00	76,56	19,82	87,48	86,47	86,98	6,19	90,00	85,00	87,50	15,17

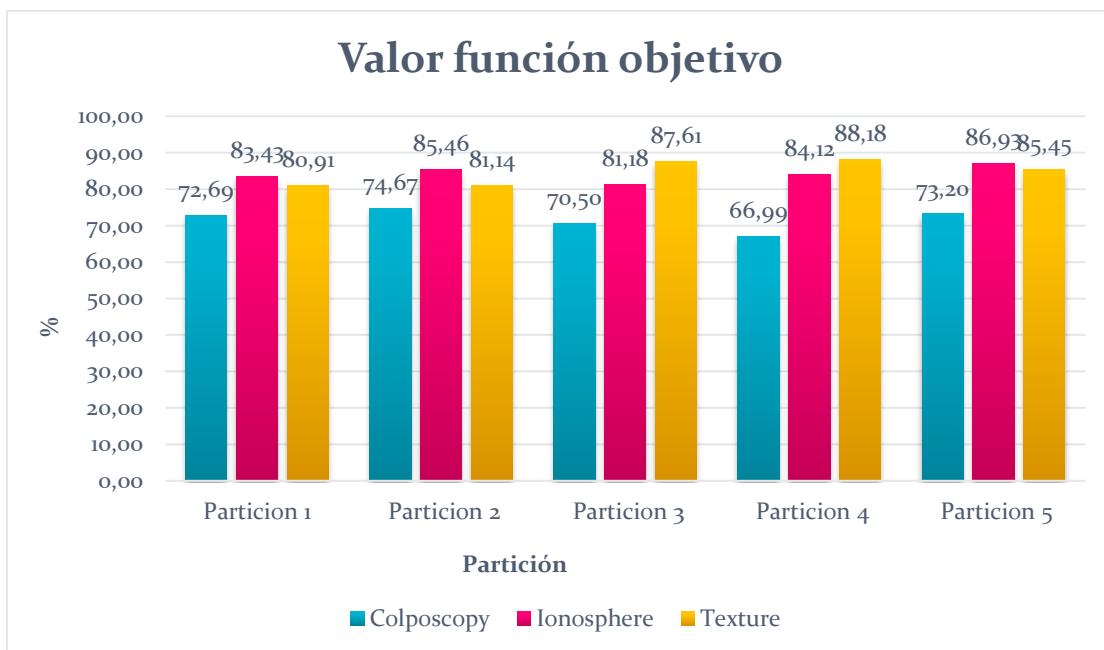


Los resultados obtenidos por la BL en los conjuntos Texture e Ionosphere son similares en cuanto a tasa de acierto, tasa de reducción y función objetivo, en cambio el dataset Colposcopy no obtiene tan buenos resultados como estos. La diferencia de ejecución es significante en cuanto respecta al conjunto Ionosphere siendo menor que en los demás dataset.

## **AGG-BLX**

Tabla 5.3: Resultados obtenidos por el algoritmo AGG (BLX-alpha) en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T
Partición 1	71,19	74,19	72,69	64,12	84,51	82,35	83,43	45,10	81,82	80,00	80,91	74,05
Partición 2	71,93	77,42	74,67	60,41	88,57	82,35	85,46	44,02	87,27	75,00	81,14	88,44
Partición 3	68,42	72,58	70,50	62,03	80,00	82,35	81,18	39,34	92,73	82,50	87,61	78,00
Partición 4	61,40	72,58	66,99	58,32	80,00	88,24	84,12	38,85	96,36	80,00	88,18	69,36
Partición 5	75,44	70,97	73,20	61,76	88,57	85,29	86,93	44,76	90,91	80,00	85,45	76,95
<b>Media</b>	69,68	73,55	71,61	61,33	84,33	84,12	84,22	42,41	89,82	79,50	84,66	77,36

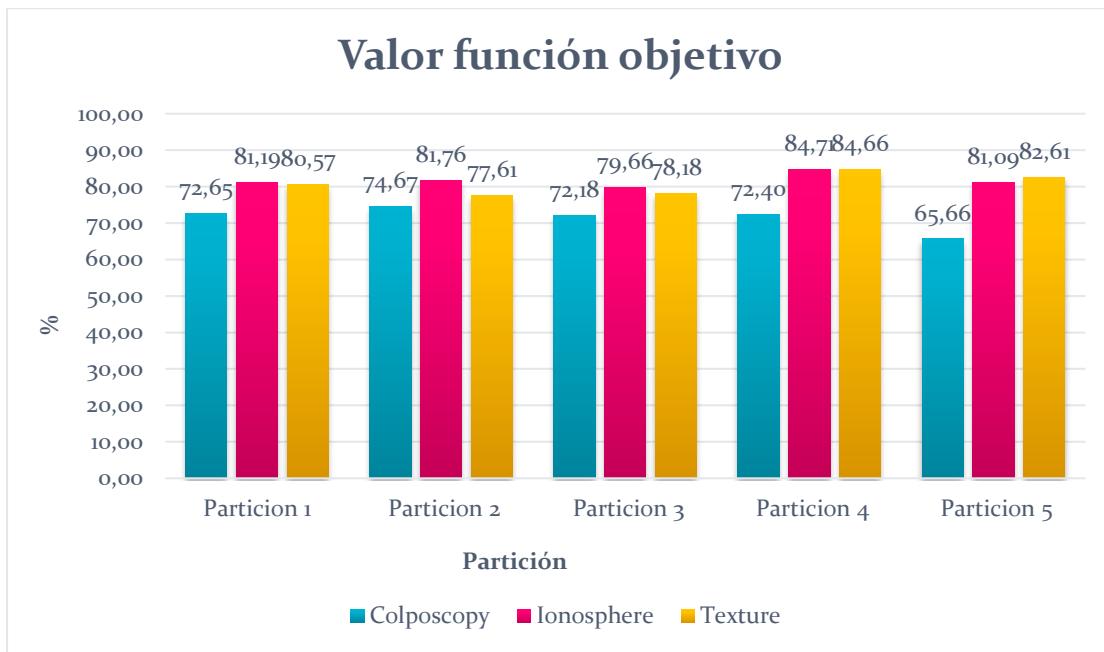


En este algoritmo vemos como hay una diferencia notable entre los valores obtenidos, en lo que respecta a la función objetivo, por las diferentes particiones sobre los conjuntos Texture y Ionosphere, siendo estos mayores que los obtenidos en el conjunto Colposcopy. Podemos ver como en algunas particiones el conjunto Ionosphere obtiene un mayor agr que el conjunto Texture.

## **AGG-CA**

Tabla 5.3: Resultados obtenidos por el algoritmo AGG (CA) en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T
Partición 1	69,49	75,81	72,65	55,54	85,92	76,47	81,19	48,82	83,64	77,50	80,57	73,21
Partición 2	71,93	77,42	74,67	63,67	90,00	73,53	81,76	50,20	92,73	62,50	77,61	78,28
Partición 3	70,18	74,19	72,18	64,87	82,86	76,47	79,66	45,14	86,36	70,00	78,18	82,07
Partición 4	75,44	69,35	72,40	61,88	90,00	79,41	84,71	53,11	91,82	77,50	84,66	73,35
Partición 5	68,42	62,90	65,66	63,48	85,71	76,47	81,09	45,34	92,73	72,50	82,61	77,99
Media	71,09	71,94	71,51	61,89	86,90	76,47	81,68	48,52	89,45	72,00	80,73	76,98

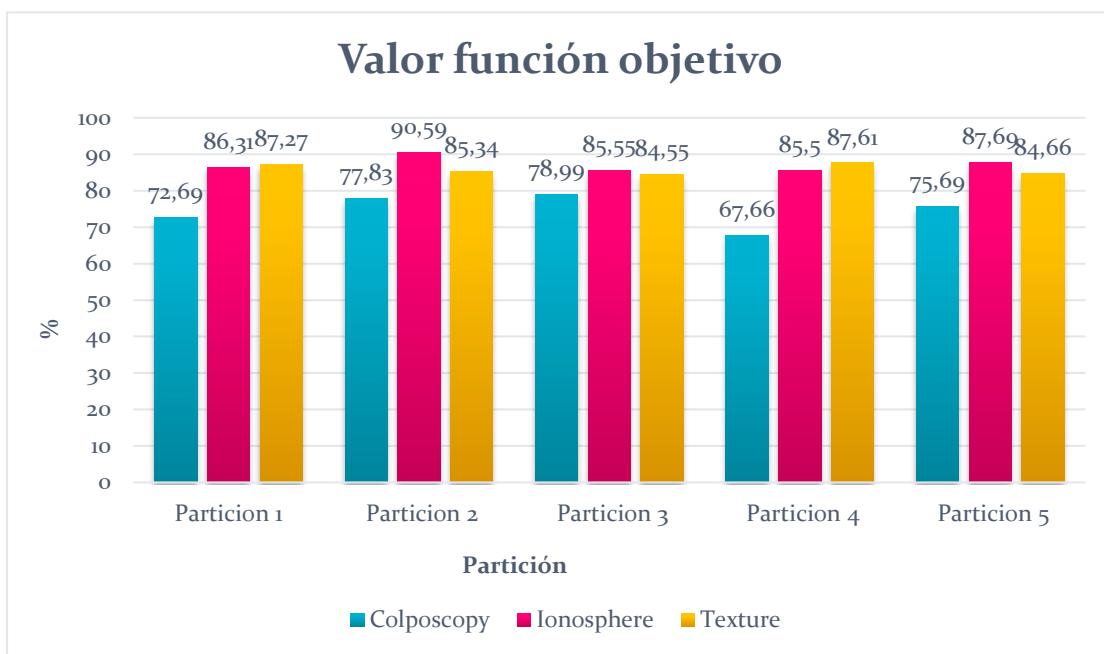


Los resultados obtenidos en este algoritmo, comparando los dataset Texture y Ionosphere vemos como los resultados son similares, siendo el conjunto de Ionosphere el de menor tiempo de ejecución. Respecto al conjunto Colposcopy como hemos visto en los anteriores algoritmos obtenemos peores resultados que en los demás conjuntos.

## AGE-BLX

Tabla 5.4: Resultados obtenidos por el algoritmo AGE (BLX-alpha) en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T
Partición 1	71,19	74,19	72,69	65,99	87,32	85,29	86,31	46,23	94,55	80,00	87,27	73,22
Partición 2	70,18	85,48	77,83	63,61	90,00	91,18	90,59	42,46	88,18	82,50	85,34	72,94
Partición 3	78,95	79,03	78,99	63,18	82,86	88,24	85,55	40,38	89,09	80,00	84,55	68,95
Partición 4	57,89	77,42	67,66	69,30	85,71	85,29	85,50	47,14	92,73	82,50	87,61	77,11
Partición 5	77,19	74,19	75,69	63,37	87,14	88,24	87,69	42,90	91,82	77,50	84,66	80,31
<b>Media</b>	71,08	78,06	74,57	65,09	86,61	87,65	87,13	43,82	91,27	80,50	85,89	74,50

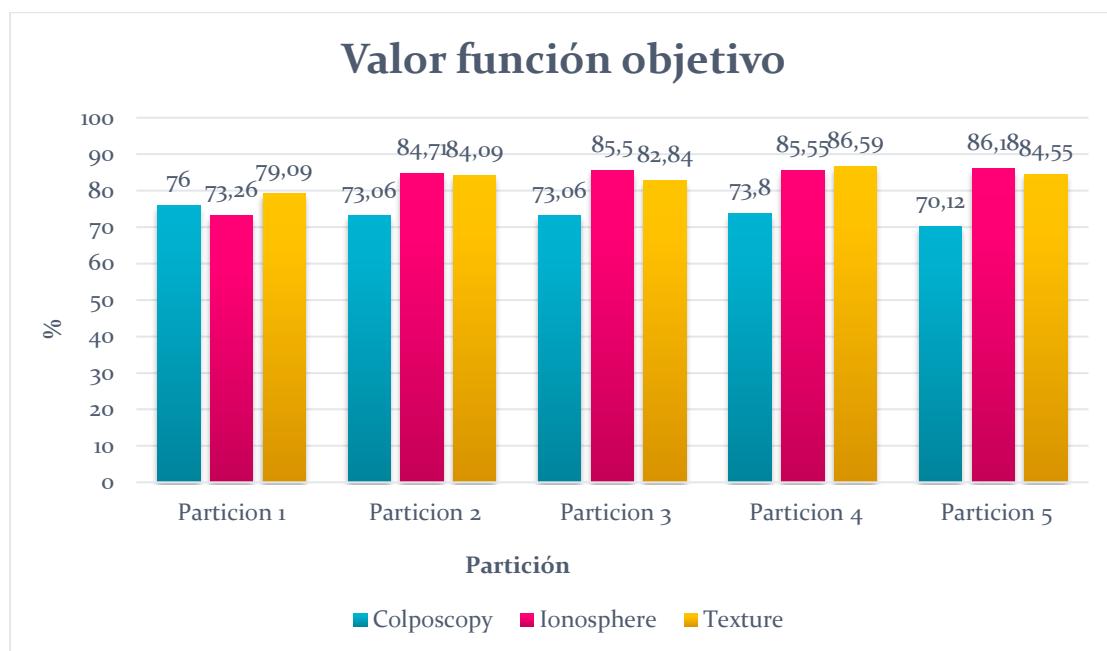


En AGE-BLX, como indican los resultados, obtenemos mejores resultados de tasa de acierto en el dataset Texture, en cambio la tasa de reducción media es mayor en Ionosphere. Si comparamos en medida al valor de agr, Ionosphere obtiene mejores resultados que los demás dataset.

## **AGG-CA**

Tabla 5.3: Resultados obtenidos por el algoritmo AGG (CA) en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T
Partición 1	74,58	77,42	76,00	55,56	78,87	67,65	73,26	46,93	88,18	70,00	79,09	72,30
Partición 2	71,93	74,19	73,06	57,29	90,00	79,41	84,71	45,04	88,18	80,00	84,09	74,56
Partición 3	71,93	74,19	73,06	57,17	85,71	85,29	85,50	36,86	88,18	77,50	82,84	72,38
Partición 4	70,18	77,42	73,80	54,00	82,86	88,24	85,55	39,13	88,18	85,00	86,59	76,85
Partición 5	78,95	61,29	70,12	55,38	90,00	82,35	86,18	44,76	89,09	80,00	84,55	71,25
<b>Media</b>	73,51	72,90	73,21	55,88	85,49	80,59	83,04	42,55	88,36	78,50	83,43	73,47

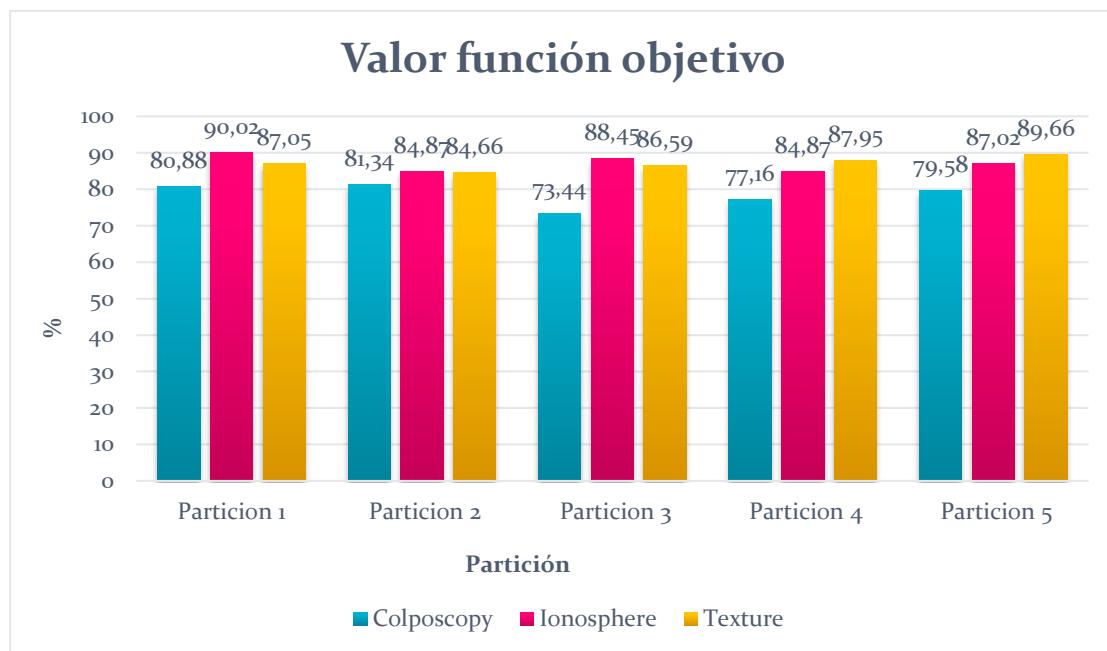


En los resultados obtenidos por este algoritmo vemos como el mejor valor medio de función objetivo es en el dataset Texture, pero no supera por una gran diferencia a Ionosphere, el dataset Colposcopy no obtiene tan buenos resultados como estos dos conjuntos

## **AM-(10,1.0)**

Tabla 5.5: Resultados obtenidos por el algoritmo AM(10,1.0) en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T
Partición 1	76,27	85,48	80,88	62,74	85,92	94,12	90,02	33,60	89,09	85,00	87,05	68,60
Partición 2	77,19	85,48	81,34	59,56	78,57	91,18	84,87	52,47	81,82	87,50	84,66	69,83
Partición 3	61,40	85,48	73,44	56,38	85,71	91,18	88,45	39,62	88,18	85,00	86,59	68,75
Partición 4	73,68	80,65	77,16	63,92	78,57	91,18	84,87	38,97	90,91	85,00	87,95	72,84
Partición 5	73,68	85,48	79,58	59,90	82,86	91,18	87,02	50,04	91,82	87,50	89,66	69,31
<b>Media</b>	<b>72,45</b>	<b>84,52</b>	<b>78,48</b>	<b>60,50</b>	<b>82,33</b>	<b>91,76</b>	<b>87,05</b>	<b>42,94</b>	<b>88,36</b>	<b>86,00</b>	<b>87,18</b>	<b>78,54</b>

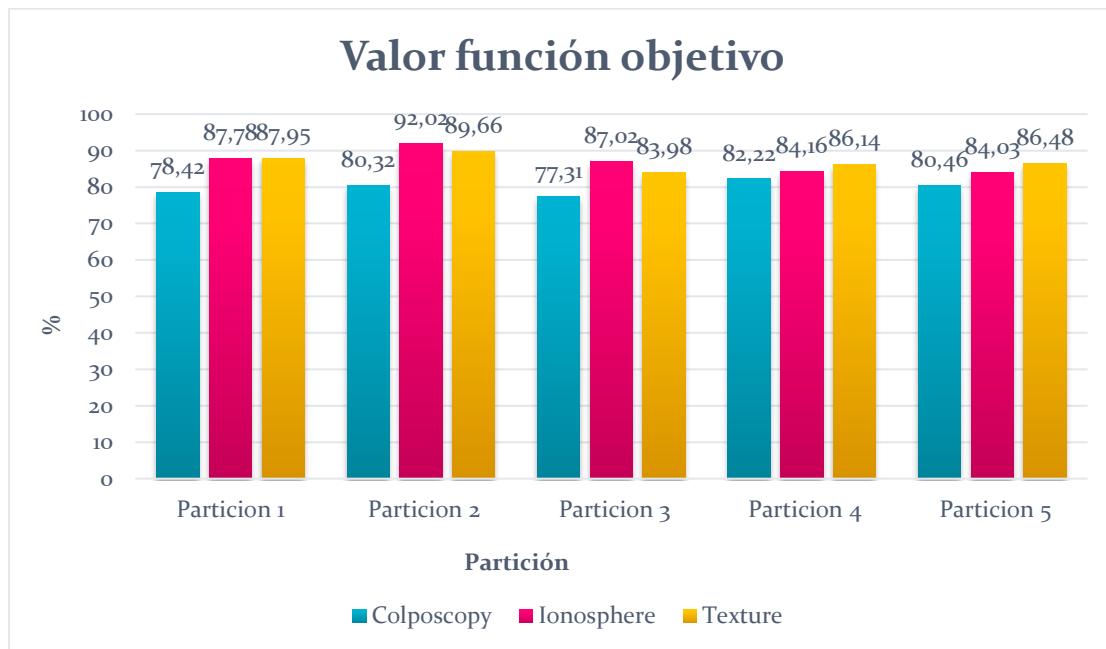


Mediante la gráfica y con ayuda de la tabla de datos vemos como los el valor referente a tasa de acierto y reducción son similares en los conjuntos Texture e Ionosphere, aunque la diferencia de tiempo es menor sobre el conjunto Ionosphere.

## **AM-(10,0.1)**

Tabla 5.6: Resultados obtenidos por el algoritmo AM(10,0.1) en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T
Partición 1	74,58	82,26	78,42	54,61	87,32	88,24	87,78	36,43	90,91	85,00	87,95	70,93
Partición 2	71,93	88,71	80,32	54,00	92,86	91,18	92,02	39,71	91,82	87,50	89,66	74,92
Partición 3	77,19	77,42	77,31	57,65	82,86	91,18	87,02	46,13	85,45	82,50	83,98	67,15
Partición 4	78,95	85,48	82,22	57,68	77,14	91,18	84,16	35,82	87,27	85,00	86,14	68,98
Partición 5	75,44	85,48	80,46	55,16	85,71	82,35	84,03	54,53	85,45	87,50	86,48	61,53
Media	75,62	83,87	79,74	55,82	85,18	88,82	87,00	42,52	88,18	85,50	86,84	68,70

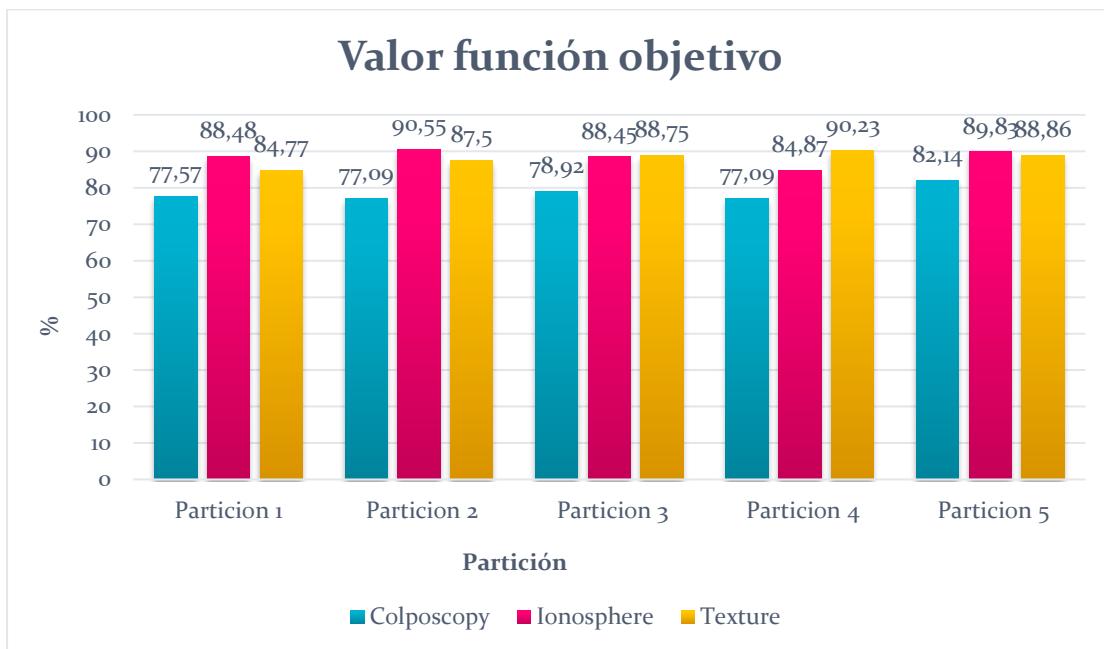


En el algoritmo memético (10,01), volvemos a obtener valores similares en los conjuntos Texture e Ionosphere, aunque la tasa de reducción es algo mayor en Ionosphere, en lo que respecta al dataset Colposcopy los resultados obtenidos siguen siendo menores que en el resto de conjuntos.

## **AM-(10,0.1, mejores)**

Tabla 5.7: Resultados obtenidos por el algoritmo AM(10,0.1,mejores) en el problema del APC

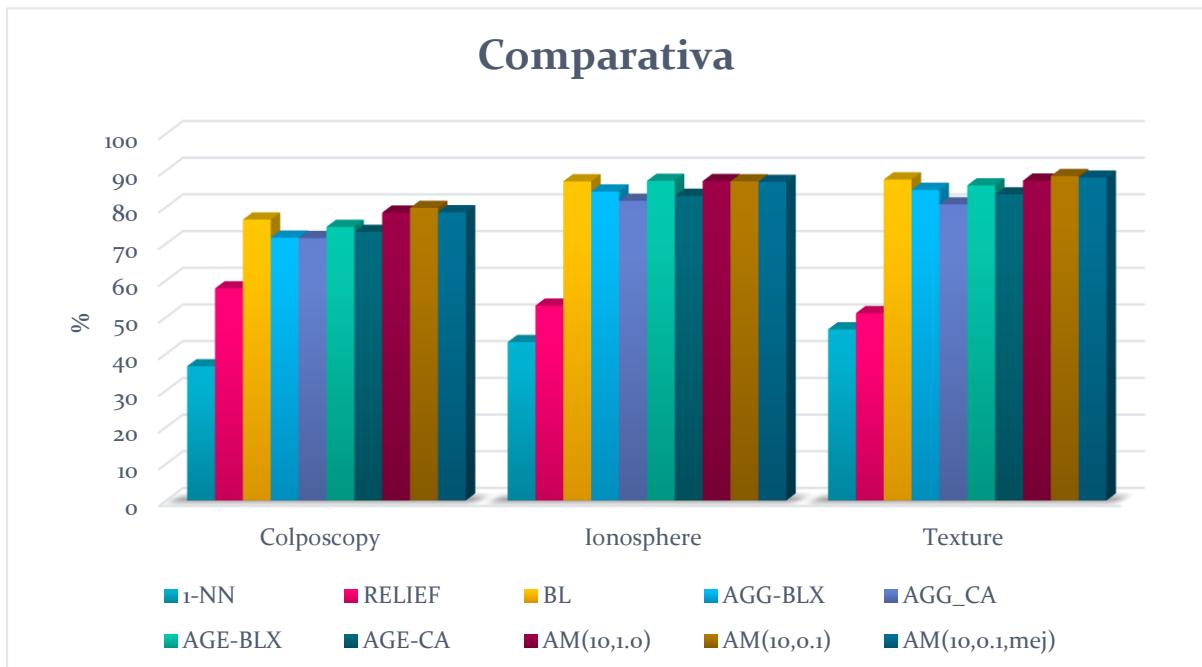
	Colposcopy				Ionosphere				Texture			
	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T
Partición 1	72,88	82,26	77,57	57,98	88,73	88,24	88,48	35,45	84,55	85,00	84,77	75,30
Partición 2	71,93	82,26	77,09	60,85	92,86	88,24	90,55	36,15	90,00	85,00	87,50	76,93
Partición 3	77,19	80,65	78,92	62,41	85,71	91,18	88,45	36,27	90,00	87,50	88,75	61,67
Partición 4	71,93	82,26	77,09	57,83	78,57	91,18	84,87	36,09	95,45	85,00	90,23	71,09
Partición 5	77,19	87,10	82,14	51,89	91,43	88,24	89,83	39,52	92,73	85,00	88,86	67,10
Media	74,23	82,90	78,56	58,19	87,46	89,41	88,44	36,70	90,55	85,50	88,02	70,42



Los resultados obtenidos en este algoritmo, como podemos ver el valor de agr en el dataset Colposcopy crece en algunas particiones, aunque no llegue a alcanzar los valores obtenidos sobre los demás conjuntos. La tasa de acierto respecto el conjunto Texture es mayor que la obtenida en el conjunto Ionosphere, en cambio no pasa lo mismo con la tasa de reducción, siendo esta es mejor en Ionosphere.

## **COMPARATIVA**

	Colposcopy				Ionosphere				Texture			
	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T
<b>1-NN</b>	73,18	0,00	36,59	0,01	86,33	0,00	43,17	0,03	93,27	0,00	46,64	0,04
<b>RELIEF</b>	60,46	55,16	57,81	0,67	89,18	17,06	53,12	0,67	94,13	8,00	51,06	2,04
<b>BL</b>	73,11	80,00	76,56	19,82	87,48	86,47	86,98	6,19	90,00	85,00	87,50	15,17
<b>AGG-BLX</b>	69,68	73,55	71,61	61,33	84,33	84,12	84,22	42,41	89,82	79,50	84,66	77,36
<b>AGG-CA</b>	71,09	71,94	71,51	61,89	86,90	76,47	81,68	48,52	89,45	72,00	80,73	76,98
<b>AGE-BLX</b>	71,08	78,06	74,57	65,09	86,61	87,65	87,13	43,82	91,27	80,50	85,89	74,50
<b>AGE-CA</b>	73,51	72,90	73,21	55,88	85,49	80,59	83,04	42,55	88,36	78,50	83,43	73,47
<b>AM(10,1.0)</b>	72,45	84,52	78,48	60,50	82,33	91,76	87,05	42,94	88,36	86,00	87,18	69,87
<b>AM(10,0,1)</b>	75,62	83,87	79,74	55,82	85,18	88,82	87,00	42,52	88,18	85,50	86,84	68,70
<b>AM(10,0,1mej)</b>	74,23	82,90	78,56	58,19	87,46	89,41	88,44	36,70	90,55	85,50	88,02	70,42



En la comparación entre todos los algoritmos implementados hasta el momento, observando la tabla de datos, ya que por medio de la gráfica no podemos distinguir que algoritmo es mejor respecto a la obtención de un mejor valor de función objetivo, vemos como el algoritmo AM(10,0,1,mejores) obtiene ligeramente mejores resultados que los demás AM, los algoritmos genéticos obtienen buenos resultados pero no llegan a igualar a los meméticos, en cambio la búsqueda local obtiene valores relativamente similares a los AM en función a valor de agr.

## d- Análisis de los resultados:

/\*

PRACTICA 1

\*/

En primer lugar, vamos analizar los resultados obtenidos por el clasificador KNN, observamos que el máximo valor de la función objetivo es obtenido en el dataset textura, uno de los factores para la obtención de buenos resultados sobre un dataset es la separabilidad de las clases del conjunto. El clasificador KNN no hace uso de un vector de pesos para la clasificación de características. En este clasificador la tasa de reducción es 0.0 ya que no desprecia ninguna característica. En valores de tiempo es el que mejor resultados obtiene de todos los algoritmos, debido a que no implementa ninguna heurística que requiera un elevado tiempo de ejecución.

Respecto al algoritmo Relief podemos observar una mejora tanto en la tasa\_clas como en la función objetivo, ya que en este algoritmo se usa un vector de pesos el cual se obtiene mediante el cálculo de los vecinos amigos y enemigos. En este algoritmo aparece la tasa\_red debido a que los valores menores de 0.0 los trunca a 0.0 y los valores mayores que 1.0 los trunca a 1.0, esto se ve reflejado en la función objetivo que como podemos observar es mucho mayor que la obtenida en el KNN sin pesos.

Para terminar, el algoritmo de búsqueda local incrementa la función objetivo respecto a los anteriores descritos, ya que este algoritmo busca la solución más “buena” (que no tiene porqué ser la óptima) de entre todas las exploradas, por lo que también se ve reflejado en la tasa de aciertos (tasa\_clas) y en la tasa de reducción (tasa\_red). Ya que esta explora una cantidad definida de vecino u iteraciones en busca de una solución “óptima”, esto se verá reflejado en el tiempo de ejecución del algoritmo, ya que es con diferencia el que más tiempo requiere de ejecución.

/\*

## PRACTICA 2

\*/

En primer lugar, vamos a comparar los resultados obtenidos entre los diferentes operadores de cruce de un mismo algoritmo:

AGG-BLX vs AGG-CA y AGE-BLX vs AGG-CA:

Como podemos observar los valores obtenidos por el operador de cruce BLX-alpha respecto a la función objetivo, son mejores a los obtenidos por el operador de cruce aritmético, uno de los factores es la diferencia de exploración del espacio, ya que en el cruce aritmético los descendientes quedaran en algún lugar en el espacio comprendido por los padres, en cambio el cruce BLX-alpha permite una mayor exploración del espacio. La diferencia entre los resultados del mismo algoritmo no es muy elevada.

Una observación relevante de los algoritmos AGG y AGE es que no logran superar a la búsqueda local, en lo que nos referimos a valores de función objetivo, esto puede ser debido a la baja diversidad entre los cromosomas, lo que hace que todos los individuos de la población sean parecidos o al bajo tamaño de la población en dichos algoritmos.

Los resultados obtenidos por los AG's están influenciados por varios factores:

- Convergencia: se debe ajustar las búsquedas en regiones prometedores mediante la presión selectiva.
- Diversidad: se debe evitar la convergencia prematura, ya que si convergemos rápidamente hacia zonas donde no se encuentre el óptimo global, no podremos obtener el mejor resultado.

A continuación, vamos a comparar los resultados obtenidos por las diferentes hibridaciones de los Algoritmos meméticos implementados:

En los resultados obtenidos por estos algoritmos (AM(10,1.0) , AM(10,0.1) , AM(10,0.1,mejores)) podemos observar como el AM(10,0.1) obtiene mejores resultados en el dataset colposcopy, en cambio en los dos dataset restantes (ionosphere y texture) el AM(10,0.1,mejores) obtiene resultados algo mejores que los otros algoritmos meméticos, esto puede suponerse debido a que dicha hibridación siempre refina sobre el mejor de la población, es decir, se le aplica la BL al mejor de la población permitiendo aumentar su valor de agr.

En valores de tiempo vemos como estos algoritmos, aun aplicando la BL cada 10 generaciones, tardan el mismo tiempo de ejecución que los demás algoritmos genéticos, esto se debe a que el número de iteraciones de parada,

es decir, el número máximo de cálculo de agr es el mismo, la diferencia es que en los AM se realizan la mayoría de estas iteraciones a la hora de aplicar la BL.

A continuación, vamos a comparar los algoritmos genéticos con los algoritmos meméticos: la gran diferencia entre estos dos algoritmos se debe a la aplicación de la búsqueda local en los AM sobre la población después del elitismo, lo que puede suponer una mejoría en la obtención de individuos de mayor calidad, tal y como indica la gráfica comparativa los AM obtienen mejores resultados que los algoritmos genéticos, tanto el generacional como el estacionario.

En comparación con los algoritmos de la practica 1, podemos observar como el clasificador 1-NN y el algoritmo Relief obtienen valores muy bajos de agr con respecto a los algoritmos de la practica 2, esto se debe principalmente a que no contienen ninguna heurística, en cambio el tiempo de ejecución es menor.

Respecto a la búsqueda local implementada en la practica 1, observamos como este obtiene mejores resultados los AG's de la practica 2, las razones de ello las he comentado anteriormente. Esto no sucede con los AM, ya que estos superan, no por una diferencia muy grande, a la BL.

## Bibliografía

- Información sobre las particiones: [https://scikit-learn.org/stable/modules/generated/sklearn.model\\_selection.StratifiedKFold.html](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html)
- Información sobre el clasificador scikit-learn: <https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html>
- Información sobre la lectura de ficheros arff:  
<https://discuss.analyticsvidhya.com/t/loading-arff-type-files-in-python/27419>
- Información sobre generación de gráficos