Practica 2: Algoritmos Genéticos y Meméticos

Metahurística

José Manuel Pérez Lendínez 26051613-L

jmplz14@correo.ugr.es
Grupo 1 curso 18/19

Contents

1 Descripción del problema

El problema se basa en optimizar la clasificación de nuevos elementos a partir de dos particiones. La particiones de entrenamiento y la partición de test.

Tendremos un conjunto de datos que dividiremos en 5 particiones, 4 para entrenamiento y una para test. La partición de test ira rotando hasta que todas pasen por test una vez.

Los conjuntos de datos vendrán dados por una clase y un conjunto de características. Se representarán como un vector con la siguiente estructura.

$$(x_1, x_2, ..., x_n, y)$$

Donde las x corresponde a las características y la y a la clase que pertenece ese elemento.

El objetivo sera ser capaces de acertar con el vector de características a la clase que pertenece. Para ello entrenaremos nuestro modelo con los datos de entrenamiento y usaremos los datos de test para ver si el modelo da resultados mas o menos acertados.

El clasificador tendrá que ser capaz de adivinar la clase utilizando la distancia a los datos de entrenamiento mas cercanos a la muestra elegida de los datos de test. Esta técnica se conoce como k-vecinos mas cercanos y sera la utilizada en todos nuestros algoritmos. Nosotros usaremos una k=1 por lo que solo buscaremos el vecino mas cercano para clasificar sin tener en cuenta a el mismo como vecino.

Al trabajar con datos reales sera muy difícil llegar a la solución exacta por lo que se buscaran aproximaciones.

Para esto utilizaremos tres metahurísticas que explicaremos a continuación y que son:

1. Algoritmo genético generacional
2. Algoritmo genético estacionario
3. Algoritmo memético

2 Aplicación de los algoritmos

En esta parte se explicaran la representación del problema de una forma mas completa y las características comunes que tienen todos los algoritmos.

2.1 Descripción de la representación y calculo de distancias

Para representar los datos usaremos matrices de numpy para los datos, que tendrán la siguiente estructura después de cargarlos desde los fichero.

$$datos = \begin{bmatrix} x_{1,1} & x_{1,2} & ... & x_{1,j} & y_1 \\ . & . & . & . & . \\ . & . & . & . & . \\ x_{i,1} & x_{i,2} & ... & x_{i,j} & y_i \end{bmatrix}$$

Esta matriz tendrá que ser normalizada la parte de los datos (las x) entre [0,1] para realizar con ella los cálculos de distancias para ver cual es el vecino mas cercano.

Para los algoritmos RELIEF y para la búsqueda local necesitaremos un vector de pesos que se representara de la siguiente manera:

$$(w_1, w_2, ..., w_j)$$

Este vector sera utilizado para ponderar la importancia de las distintas características de nuestros problemas. El vector tendrá unos valores acotados entre [0,1] y tendremos en cuenta que los valores menores que 0.2 no se utilizara para calcular las distancias en la clasificación. Esto restara importancia a las variables que tengan una ponderación muy baja y que podrían introducir ruido.

En esta practica añadimos también una nueva representación a tener en cuenta para los nuevos algoritmos. Los algoritmos genéticos y meméticos costan de unas poblaciones que tendremos que almacenar para ir trabajando con estas. Cada individuo de la población sera un vector de pesos como los mencionados anteriormente. La representación de la población necesita dos características, los vectores de pesos y el valor de la función objetivo para cada individuo.

Para esto usaremos una matriz para almacenar cada individuo de la población y un vector para el valor de la función objetivo para cada individuo de la población.

$$Poblacion = \begin{bmatrix} w_{1,1} & w_{1,2} & ... & w_{1,j} \\ . & . & . & . \\ . & . & . & . \\ w_{i,1} & w_{i,2} & ... & w_{i,j} \end{bmatrix}$$

$$eval\_poblacion = (valor_1, valo_2, ..., valor_i)$$

2.2 Calculo de la tasa de reducción

La tasa de reducción vendrá dada por la cantidad de pesos de nuestro vector w que sean inferior a 0.2 de forma que usaremos el siguiente pseudónimo para calcularla. La formula de la tasa de reducción es la siguiente:

$$tasa_reduccion = 100\frac{n^o\ de\ w_j < 0.2}{w.size}$$

El pseudocódigo seria el siguiente:

```
num_reducciones = 0
for i in pesos
        if pesos < 0.2
                num_reducciones++
end
tasa_reduccion = 100*(num_reducciones / pesos.size)
```

La implementación con python es muy sencilla y se relizal en una linea:

2.3 Calculo de la tasa de clase

La tasa de clase nos dará el porcentaje de acierto a la hora de entrenar nuestro modelo. Para ello utilizamos la siguiente formula:

$$tasa_clase = 100\frac{n^o\ de\ istancias\ bien\ clasificadas}{n^o\ instancias\ totales\ en\ test}$$

Para saber si una instancia esta bien clasificada se usara el algoritmos de 1-nn vecino mas cercano que explicaremos mas adelante. El pseudocodigo para calcular la tasa de clase seria:

```
num_aciertos = 0
for i in test
        clase_clasificado = clasificador_1NN(i,pesos)
        if clase(i) == clase_casificado
                num_aciertos += 0
end
tasa_clasificacion = 100*(num_aciertos / Y_test.size)
```

2.4 Función de evaluación

La función de evaluación es la encargada dar un valor numérico que representara lo bueno que es nuestro clasificador. Para ello utiliza tanto la tasa de reducción como la tasa de clase y un valor α.

La formula es la siguiente:

$$F(pesos) = (tasa\_clase(pesos) + tasa\_reduccion(pesos))/2$$

Esto hará que le demos la misma importancia la tasa de reducción y a la tasa de clase a la hora de ver como ajustan los datos. No voy a poner el pseudocódigo porque simplemente es implementar es función sin ninguna complicación mas.

2.5 Función evaluación para la búsqueda local

La implementación utilizada en es la siguiente:

```
        def evaluate(weights, X, y):
                X_transformed = (X * weights)[:, weights > 0.2]
                kdtree = KDTree(X_transformed)
                neighbours = kdtree.query(X_transformed, k=2)[1][:, 1]
                accuracy = np.mean(y[neighbours] == y)
                reduction = np.mean(weights < 0.2)
                return 100*accuracy,reduction*100,100*(accuracy + reduction)
                     / 2
```

Tengo que darle las gracias a Antonio Molner que compartió esta función para que pudiéramos utilizarla. Gracias a KDTree se puede buscar rápidamente a los vecinos más cercanos de cualquier punto. Se consigue el leave-one-out gracias a que en el kdtree utilizamos k=2 para quedarnos con los dos vecinos mas cercanos y a continuación con las funciones para array de python nos quedamos con la ultima columna que sera la que no contenga a si mismo.

2.6 Generación de la población inicial

Para esto se ha usado una distribución aleatoria uniforme con la que inicializaremos los pesos de cada individuo de la población. Esto lo hacemos con una simple orden en python.

$$Poblacion = np.random.uniform(0, 1, (num\_individuos, num\_genes))$$

Num_individuos representa el tamaño de población que tendremos y el num_genes el número de características que tienen nuestros datos.

Cuando se tiene generada los cromosomas de una poblacion necesitamos evaluarlos para saber su función objetivo. Esto lo hacemos mediante la siguiente función.

```
function evaluarPobalcion(X, y, poblacion[][])
        //array donde almacenaremos los valores de la funcion objetivo.
        valores = [0,0,..,0]

        //tamano_poblacion representa el numero de individuos que contiene
             esta.
        for i in [0,tamano_poblacion]
                valores[i] = evaluate(poblacion[i], X, y)
        end

        return valores

end
```

2.7 Torneo binario

El torneo binario sera el método mediante el que elegiremos dentro de una poblacion los padres que pasaran a la siguiente generación. Se basa en enfrentar los individuos de la poblacion actual por parejas y quedarnos con el que mayor función objetivo tenga.

En nuestro caso pasaremos a la función de torneo binario la poblacion actual, sus valores de función objetivo y el número de enfrentamientos que tendremos. Por cada enfrentamiento que se de saldrá un vencedor que permanecerá en la poblacion durante la siguiente generación y el perdedor sera eliminado.

La selección de los padres a enfrentar se hará con dos enteros aleatorios que representaran los padres de la poblacion a enfrentar.

```
function torneoBinario(poblacion,valores,num_torneos)
        nueva_poblacion = matriz[num_torneos][num_genes]
        nuevos_valores = array[num_torneos]

        for i in [0,num_torneos]
                padre0 = random_entero(0,numero_cromosomas_poblacion - 1)
                padre1 = random_entero(0,numero_cromosomas_poblacion - 1)

                if valores[padre0] > valores[padre0]
                        nueva_poblacion[i] = poblacion[padre0]
                        nuevos_valores[i] = valores[padre0]
                else
                        nueva_poblacion[i] = poblacion[padre1]
                        nuevos_valores[i] = valores[padre1]
                end


        end
        return nueva_poblacion, nuevos_valores
end
```

2.8 Cruce BLX

Se basa en mediante dos cromosomas de la poblacion $\mathrm{C}_1 = (\mathrm{c}_{11}, \ldots, \mathrm{c}_{1n})\,\mathrm{y}\mathrm{C}_2 = (\mathrm{c}_{21}, \ldots, \mathrm{c}_{2n})$ y se crean dos hijos. Para esto cogemos cada gen de los cromosomas C_1 y C_2 y se generan dos nuevos que seran introducido en los hijos. Los nuevos genes de hijos se calculan de la siguiente manera.

1. Seleccionamos el mayor y el menor de los genes elegidos del C_1 y C_2.

$$C_{\max} = \max\{\mathrm{c}_{11}, \mathrm{c}_{2l}\}$$
$$\mathrm{C}_{\min} = \min\{\mathrm{c}_{1i}, \mathrm{c}_{2j}\}$$

2. Nos quedamos con la diferencia entre el mayor y el menor.

$$\mathrm{I} = \mathrm{C}_{\max} - \mathrm{C}_{\min}, \alpha \in [0, 1]$$

3. Con esto generamos dos aleatorios en el rango

$$[\mathrm{C}_{\min} - \mathrm{I} \cdot alpha, \mathrm{C}_{\max} + \mathrm{I} \cdot alpha]$$

Los dos aleatorios generados pasaran a ser los genes de los hijos.

Esto se realizara tantas veces como genes tienen los padres.

```
function cruceBlx(C1, C2, alpha)
        hijo1 = array[numero_genes]

        hijo2 = array[numero_genes]

        for i in range [0,numero_genes]
                cmin = menor de C1[i] y C2[i]
                cmax = mayor de C1[i] y C2[i]
                I = cmax-cmin

                hijo1[i] = random(cmin-I*alpha, cmax+I*alpha)
                hijo2[i] = random(cmin-I*alpha, cmax+I*alpha)

                if hijo1[i] > 1: hijo1[i] = 1
                if hijo2[i] > 1: hijo2[i] = 1
                if hijo1[i] < 0: hijo1[i] = 0
                if hijo2[i] < 0: hijo2[i] = 0

        end

        return hijo1,hijo2


end
```

2.9 Cruce aritmético

En este caso he realizado un cambio respecto al dado en las diapositivas para poder generar dos hijos por cada padre en vez de uno.

En vez de realizar la media aritmética como nos indican las diapositivas de practicas, realizo una media ponderada. De esta forma obtengo dos hijos por cada cruce.

1. Genero un vector de números aleatorios del mismo tamaño que los padres. Los valores aleatorios estarán entre $[0, 1)$

2. Despues para cada gen de los padres realizo la media ponderada y relleno los hijos con los resultandos de estos.

$$hijo1[i] = C1[i] * ponderado[i] + C2[i] * (1 - ponderado[i])$$
$$hijo2[i] = C1[i] * (1 - ponderado[i]) + C2[i] * ponderado[i]$$

Esto se realiza tantas veces como genes tienen los cromosomas.

```
function cruceAritmetico(cromosoma1,cromosoma2)
        ponderado = array[num_genes]
        ponderado = Se rellena con aleatorios en el rango [0,1)

        hijo1 = array[numero_genes]
        hijo2 = array[numero_genes]

        for i in [0,num_genes]
                hijo1[i] = cromosoma1[i] * ponderado[i] + cromosoma2[i] * (1-
                    ponderado[i])
                hijo2[i] = cromosoma1[i] * (1-ponderado[i]) + cromosoma2[i] *
                     ponderado[i]
        end

        return hijo1, hijo2
end
```

2.10 Cruce Rand

En este caso se seleccionan tres padres para realizar la operación de cruce. La operación se realizara sobre un cuarto individuo de la poblacion. Los tres padres no podrán coincidir con el 4 elemento. Se tendrá una probabilidad para que se acepte el gen de la recombinación de los tres padres(En nuestro caso un 0.5), si no se cumple la probabilidad nos quedaremos con el gen del cuarto individuo. Para obtener los genes de los tres padres que se introducirán al cuarto individuos se realizara la siguiente operación.

$$V_{i,G} = X_{rI,G} + 0.5 \cdot (X_{r2,G} - X_{r3,G})$$

El pseudocódigo es el siguiente:

```
function cruceRand(padre_1, padre_2, padre_3, cuarto_individuo)
        nuevo_individuo = (0,...,0)
        for j in (0,size(cuarto_individuo)-1)
                if rand(0,1) < 0.5:
                        nuevo_individuo[j] = padre_1[j] + 0.5 * (padre_2[j] -
                            padre_3[j])
                        if nuevo_individuo[j] < 0
                                nuevo_individuo[j] = 0
                        end

                        if nuevo_individuo[j] > 1
                                nuevo_individuo[j] = 1
                        end
                else
                        nuevo_individuo[j] = cuarto_individuo[j]
                end

        end



end
```

2.11 Cruce Current to Best

En este caso se seleccionan dos padres para realizar la operación de cruce. La operación se realizara sobre un tercer individuo de la poblacion. Los dos no

podrán coincidir con el tercer individuo. Se tendrá una probabilidad de 0.5 de cruce. En este cruce se utilizara también el mejor elemento de la poblacion para el cruce. Se realizara la siguiente operación con los elementos.

$$V_{i,G} = X_{i,G} + 0.5 \cdot (X_{best,G} - X_{i,G}) + 0.5 \cdot (X_{rl,G} - X_{r2,G})$$

El pseudocódigo es el siguiente:

```
function cruceCurrentToBest(padre_1, padre_2, mejor_individuo,
    tercer_individuo)
        nuevo_individuo = (0,...,0)
        for j in (0,size(tercer_individuo)-1)
                if rand(0,1) < 0.5:
                        nuevo_individuo[j] = tercer_individuo[j] + 0.5 * (
                             mejor_individuo[j] - tercer_individuo[j]) + 0.5
                             * (padre_1[j] - padre_2[j])
                        if nuevo_individuo[j] < 0
                                nuevo_individuo[j] = 0
                        end

                        if nuevo_individuo[j] > 1
                                nuevo_individuo[j] = 1
                        end
                else
                        nuevo_individuo[j] = tercer_individuo[j]
                end

        end



end
```

2.12 Mutar gen

La mutación se realiza en una posición del cromosoma. Ha esta posición se le suma un aleatorio generado con una distribución normal con $\sigma = 0.3$. Se tiene que controlar que la mutación no queda por encima de 1 o por debajo de 0

```
fucntion mutarGen(cromosoma, posicion)
        valor_mutacion = distribucion_normal(0.0, 0.3)
        cromosoma[posicion] += valor_mutacion

        if cromosoma[posicion] > 1; cromosoma[posicion] = 1
        if cromosoma[posicion] < 0; cromosoma[posicion] = 0

        return cromosoma

end
```

3 Explicación de los algoritmos

En esta sección explicaremos los tres algoritmos que hemos utilizado para obtener los datos de clasificación.

3.1 Algoritmo del vecino mas cercanos 1-NN

El algoritmos 1-NN o vecino mas cercano se entrena con las particiones de entrenamiento para poder clasificar las entradas de la partición de test. Se basa en el buscar el vecino mas cercano de las muestras de entrenamiento para una muestra de test. De este modo calcula la distancia euclidea de cada elemento de entrenamiento con respecto a un elemento de test. Selecciona el que mas se acerca y mira la clase de este. Si la clase coincide con la clase del elemento de entrenamiento se da como acertado.

En mi caso he implementado dos uno para obtener los datos del 1-NN en el que no necesitamos pasarle vector de pesos y que utilizo KNeighborsClassifier de la librería de python sklearn como clasificador.

El pseudocódigo es el siguiente:

```
        def K-NN(datos_train,datos_test){

                X_train = datos(datos_train)
                X_test = datos(datos_test)
                Y_train = datos(datos_train)
                Y_test = clase(datos_test)

                num_aciertos = 0
                #esto indica al clasificador que buscara uun unico vecion
                     cercano
                clasificador = crearClasificador(n=1)
                clasificador = clasificador.entreno(X_train,Y_train)

                num_elementos = getNumeroElementos(X_test)

                for i in [0,num_elementos]
                        clase = clasificador.predice_clase(Y_test[i])
                        if clase == Y_test[i]
                                num_aciertos += 1
                        end
                end
                tasa_acierto = 100 * (num_aciertos / Y_test.size)

                #como no trabaja con pesos el valor de tasa_reduccion sera 0
                return tasa_acierto,funcionObjetivo(tasa_acierto,0)
        }
```

3.2 Algoritmo RELIEF

El algoritmo RELIEF es una solución greedy que se basa en buscar el enemigo y vecino mas cercano para para mejorar el vector de pesos. El vector de pesos se inicia a 0.

La formula utilizada para calcular el nuevo vector de pesos es la siguiente:

$$pesos = pesos + |e_i - e_e| - |e_i - e_a|$$

Esto se realiza para cada elemento del conjunto de entrenamiento.

Una vez terminado se calcula la función objetivo clasificando mediante uno-NN pero antes hay que normalizar el vector de pesos por si alguno ha sobrepasado el uno o es menor que 0. De esta manera se coloca a 0 los menos que este y nos quedamos con el valor mayor del vector para normalizar respecto a este todos los valores.

```
        def RELIEF(X_train,Y_train,X_test, Y_test)
                num_elementos = getNumeroElementos(X_test)
                pesos = zeros(num_elementos)

                #creo una matriz de distancia para no tener que
                #estar calculando consantemente distancias para calcular
                #y repetir calculos
                distancias = disntacia_euclidea(X_train)

                #Recorremos la matriz de distancia comparando
                for i in [0,num_elementos]
                        mejor_enemigo = int()
                        valor_enemigo = max_float
                        mejor_amigo = int()
                        valor_amigo = max_float

                        for j in [0,num_elementos]

                                if Y_train[i] == Y_train[j]

                                        if valor_amigo < distancias[i][j]
                                                #Se evita que se el mimso el
                                                     mejor amigo
                                                if i != j
                                                        mejor_amigo = j
                                                        valor_amigo =
                                                             distancias[i][j]
                                                end

                                        end

                                else

                                        if valor_enemigo < distancias[i][j]
                                                mejor_enemigo = j
                                                valor_amigo = distancias[i][j
                                                     ]
                                        end

                                end

                        end

                        pesos = pesos + |X_train[i]-X_train[mejor_enemigo]| -
                        |X_train[i]-X_train[mejor_amigo]|
                end

                #normalizamos el vector de pesos si es menor

                max = obtenerMayor(pesos)

                for i in peso
                        if i < 0
                                i = 0
                        else
                                i = i / max
                end

                return uno-nn(X_train,Y_train,X_test,Y_test,pesos)

        end
```

3.3 Algoritmo de Búsqueda Local

En la búsqueda local iremos generando vecinos por mutación hasta un máximo de 15000. El pseudocódigo para la mutación es el siguiente. En mi caso he usado la librería randon de numpy. Si el peso supera 0 o 1 se trunca el peso.

```
        def mutacion(posicion,pesos)
                Z = np.random.normal(0.0, 0.3, None)
                pesos[posicion] += z
                if pesos[posicion] > 1
                        peso[posicion] = 1
                end
                if pesos[posicion] < 0
                        peso[posicion] = 0
                end

                return pesos
        end
```

El valor 0.3 es la varianza que utilizaremos para la mutación. Para generar el primer vector de pesos se tiene que realizar de forma aleatoria con valores entre [0,1]. Para eso utilizo también la librería numpy.random de python con la opción rand que genera valores entre 0 y 1

```
        def generar_vector(num_valores)
                pesos = np.random.rand(num_valores)
                return pesos
        end
```

La búsqueda local ira mutando un componente del vector y quedara este si mejora los valores que el anterior. Si no se da la mejora volverá al anterior y mutara el siguiente valor. Por cada fallo en la mutación se contara un erro de mejora. Si se obtiene 20*n errores en la mejora de forma consecutiva se para la búsqueda. El valor n corresponde al número de características que tienen un elemento de nuestro dataset.

De esta forma ya tenemos dos criterios de parada, el generar 15000 vecionos o no mejorar en 20*n mutaciones de forma consecutiva.

Para mejorar un poco los tiempos antes de realizar una comprobación de si los pesos son mejores que los anteriores miro dos cosas.

La primera es que si antes de mutar una posición del vector de pesos esa posición era menor que 0.2 y al mutar sigue siendo menor que 0.2, sabemos que no tendrá mejora, por lo que no realizamos la comprobación.

La segunda opción que nos asegura que no se da mejora es cuando al mutar un gen el valor anterior era igual que el valor mutado. Esto parece poco probable pero cuando el valor de ese peso antes de la mutación ya era 1 si nos sale en la mutación un valor positivo no mejorara ese 1 puesto que es el valor máximo. En ese caso tampoco hago la comprobación de mejora.

El pseudocódigo es el siguiente;

```
        def BL(X_train,Y_train,X_test, Y_test)
                w = generar_vector(getNumeroCaracteristicas(X_train))
                pos_w = 0
                num_veciones = 0
                sin_mejora = 0

                #Se comprueba el valor de la solucion
                tasa_clase, tasa_reduccion = evaluate(train_datos,
                     train_clases, w)

                funcion_mejora = funcionObjetivo(tasa_clase,tasa_reducion)

                mejor_valor_w = funcion_mejora
                mejor_w = w
                mejor_tasa_clase = tasa_clase
                mejor_tasa_reduccion = tasa_reduccion

                while num_vecions < 15000 y sin_mejora < 20 * num_pesos
                        num_veciones += 1
                        anterior_peso = w[pos_w]
                        pesos = mutacion(pos_w,pesos)

                        #comprobamos si hay que calcular la mejora
                        if comprobarSiCalculamosMejora(anterior_peso,w[pos_w
                             ]):
                                w[pos_w] = anterior_peso
                                sin_mejora += 1

                        else
                                tasa_clase, tasa_reduccion = evaluate(
                                     train_datos, train_clases, w)
                                funcion_mejora = funcionObjetivo(tasa_clase,
                                     tasa_reducion)


                                if mejor_valor_w < funcion_mejora:

                                        #Si mejora actualizamos los mejores
                                             valores
                                        mejor_w = w
                                        mejor_valor_w = funcion_mejora
                                        mejor_tasa_clase = tasa_clase
                                        mejor_tasa_reduccion = tasa_reduccion
                                        sin_mejora = 0


                                else:
                                        #volvemos al valor anterior y
                                             contamos una pasada sin mejora
                                        w[pos_w] = anterior_peso
                                        sin_mejora += 1
                                end

                        end


                        pos_w = (pos_w+1) % size(w)

                end

                return uno-nn(X_train,Y_train,X_test,Y_test,pesos)
        end
```

3.4 Algoritmo genético generacional

El AGG (algoritmo genético generacional) parte de una poblacion inicial y en cada época cambia completamente la población actual por una población nueva. Esto se realiza mediante el proceso de selección en el que compiten los padres por continuar en la poblacion. En este caso usamos el torneo binario para realizar esta selección. Los individuos con mejor función objetivo tendrán mas opciones de ganar un combate y quedar en la nueva poblacion. Los individuos que competirán se eligen aleatoriamente. Si se tienen N individuos en la poblacion se realizaran N*2 torneos.

Después realizamos los cruces entre padres para generar nuevos hijos. Estos hijos sustituirán a los padres en la nueva poblacion. Se tiene una probabilidad de que dos padres crucen. Parar ahorrar generaciones de números aleatorios y como el proceso de torneo binario es aleatorio, calcularemos el número de cruces que se podrían dar en una época y se realizara este número de cruces empezando por los primeros individuos de la poblacion.

El siguiente paso seria realizar las mutaciones en los genes apartir de una probabilidad de mutación que se tendrán por gen. En este caso como se tiene un número muy elevado de genes en toda la poblacion, se calculara el número de mutaciones que se darían con la probabilidad especificada y se genera por cada mutación dos números aleatorios. Uno para el individuo a mutar y otro para el gen que mutara.

Para no perder la mejor solución de una época a otra, si el mejor individuo de la nueva época es peor que el mejor individuo de la época anterior, se cambiara el peor individuo de la poblacion actual por el mejor de la anterior. Esto nos asegura que como mínimo tendremos el mejor individuo conocido o alguno mejor en la nueva poblacion.

3.4.1 Parámetros

Los parámetros para nuestro AGG son lo siguiente:

1. Tamaño de la poblacion(tam_poblacion): 30 individuos
2. Número de torneos(num_torneo): 2*M siendo m el tamaño de la poblacion
3. Probabilidad de cruce(prob_cruce): 0.7
4. Probabilidad de mutación(prob_mutacion): 0.001
5. Número de evaluaciones(max_evaluaciones): 15000

3.4.2 Pseudocódigo

```
function AGG(X,y)
        num_genes = numero de genes por inidividuo
        num_cruces = int(prob_cruce * tam_poblacion)
        num_mutacions = int(prob_mutacion * (tam_poblacion * num_genes))
        //Se inicia la poblacion y calculamos su funcion objetivo
        poblacion[][] = inicializarPobalcion(tam_poblacion,num_genes)
        eval_poblacion[] = evaluarPoblacion(X,y,poblacion)

        evaluaciones = tam_poblacion
        //Nos quedamos el mejor actual por si hay que volver a meterlo en la
             poblacion
        mejor_acutal = ObtenerPosicionMejorIndividuo(eval_poblacion);
        mejor_individuo[] = poblacion[mejor_actual]
        mejor_valor = eval_poblacion[mejor actual]

        while evaluaciones es menor que max_evaluaciones
                //obtenemos la nueva poblacion mediante torneo
                poblacion, eval_poblacion = torneoBinario(poblacion,
                     eval_poblacion,tam_poblacion)
                for i in [0,num_cruces]
                        Sustituimos los padres i*2 y i*2+1 por los nuevos
                             hijos dados poru no de los dos cruces(aritmetico
                              o blx)

                        //Se ponen los valores a -1 para mas adelante
                             recalcularlos.
                        eval_poblacion[i*2] = -1
                        eval_poblacion[i*2 + 1] = -1

                end

                for i in [0,num_mutaciones]
                        pos_individuo = aleatorio entero entre [0,
                             tam_poblacion]
                        pos_gen = aleatorio entero entre [0,num_genes]

                        poblacion[i] = mutarGen(poblacion[pos_individuo],
                             pos_gen)
                        eval_poblacion[pos_individuo] = evaluate(poblacion[
                             pos_individuo],X,y)
                        evaluaciones = evaluaciones + 1
                end
                //Calculamos la funcion objetivo de los cruzados
                     anteriormente
                for i in [0,num_cruces]
                        if eval_poblacion[i] == -1
                        eval_poblacion[i] = evaluate(poblacion[i],X,y)
                        evaluaciones = evaluaciones + 1
                end

                mejor_actual = ObtenerPosicionMejorIndividuo(eval_poblacion)

                if mejor_valor > eval_poblacion[mejor_actual]
                        Sustituimos el peor de la poblacion por el
                             mejor_individuo
                else
                        mejor_individuo = poblacion[mejor_actual]
                        mejor_valor = eval_poblacion[mejor_actual]
                end

        end
        return poblacion, eval_poblacion

end
```

3.5 Algoritmo genético estacionario

El AGE tiene un funcionamiento muy parecido al AGG. El cambio principal es que en vez de cambiar toda la poblacion en cada época, añade dos nuevos hijos si son mejores que los dos peores de la poblacion actual.

Esto se consigue realizando dos torneos binarios para obtener los dos padres que cruzaremos. A continuación cruzamos los padres y nos quedamos con los dos hijos que obtenemos. Se realizan las mutaciones en la poblacion actual y obtenemos los dos peores individuos de la población. En este caso la probabilidad de mutación no merece la pena hacerla como en el caso del agg. Lo que haremos es obtener la probabilidad que tiene un individuo de la poblacion de mutar. Para esto multiplicamos el número de genes de un individuo por la probabilidad de mutación. Con un aleatorio se vera si muta o no y si es menor a la probabilidad que tiene de mutar se obtendrá otro aleatorio para la posición de gen a mutar. Esto se repite para los dos hijos.

$$prob_mutacion_cromosoma = prob_mutacion * n_genes_cromosoma$$

Des entre los hijos y los peores individuos de la poblacion nos quedamos los 2 mejores y serán lo que formen parte de la poblacion.

3.5.1 Parámetros

Los parámetros para nuestro AGE son prácticamente iguales que los del AGG cambiando la probabilidad de cruce unicamente:

1. Tamaño de la poblacion(tam_poblacion): 30 individuos
2. Número de torneos(num_torneo): 2*M siendo m el tamaño de la poblacion
3. Probabilidad de cruce(prob_cruce): 1. Siempre cruzan los padres seleccionados.
4. Probabilidad de mutación(prob_mutacion): 0.001
5. Número de evaluaciones(max_evaluaciones): 15000

3.5.2 Pseudocódigo

```
function AGE(X,y)
        num_genes = numero de genes por inidividuo
        prob_mutacion_cromosoma = prob_mutacion * n_genes
        //Se inicia la poblacion y calculamos su funcion objetivo
        poblacion[][] = inicializarPobalcion(tam_poblacion,num_genes)
        eval_poblacion[] = evaluarPoblacion(X,y,poblacion)
        evaluaciones = tam_poblacion
        //Nos quedamos el mejor actual por si hay que volver a meterlo en la
            poblacion
        mejor_acutal = ObtenerPosicionMejorIndividuo(eval_poblacion);
        mejor_individuo[] = poblacion[mejor_actual]
        mejor_valor = eval_poblacion[mejor actual]

        while evaluaciones es menor que max_evaluaciones
                //obtenemos la nueva poblacion mediante torneo
                nueva_poblacion[][], valor_poblacion = torneoBinario(
                    poblacion,eval_poblacion,2)

                //Se realiza el cruce aritmetico o blx y se guardan los hijos
                nueva_poblacion[0],nueva_poblacion[1] = cruce aritmetico o
                    blx

                mutacion1 = randon en rango [0,1]
                mutacion2 = randon en rango [0,1]
                if mutacion1 <= prob_mutacion_cromosoma
                        pos_gen = random entre [0, num_genes)
                        nueva_poblacion[0] = mutarGen(nueva_poblacion,pos_gen
                            )
                end
                if mutacion1 <= prob_mutacion_cromosoma
                        pos_gen = random entre [0, num_genes)
                        nueva_poblacion[0] = mutarGen(nueva_poblacion,pos_gen
                            )
                end

                valor_poblacion[0] = evaluate(nueva_poblacion[0],X,y)
                valor_poblacion[1] = evaluate(nueva_poblacion[1],X,y)
                evaluacions = evaluaciones + 2

                //En el 0 tendremos el indice del peor y en el 1 el indice
                    del segundo peor
                pos_peores[] = obtenemos los indices de los dos peores de la
                    poblacion
                //tendremos los cromosomas de los dos mejores y los valores
                    de estos

                if valores[0] > eval_poblacion[pos_menores[0]]
                        poblacion[pos_menores[0]] = nueva_poblacion[0]
                        eval_poblacion[pos_menores[0]] = valores[0]
                        if valores[0] > eval_poblacion[pos_menores[1]] :
                                auxiliar = pos_menores[0];
                                pos_menores[0] = pos_menores[1]
                                pos_menores[1] = auxiliar
                        end
                end

                if valores[1] > eval_poblacion[pos_menores[0]]
                        poblacion[pos_menores[0]] = nueva_poblacion[1]
                        eval_poblacion[pos_menores[0]] = valores[1]
                end

        end
        return poblacion, eval_poblacion

end
```

3.6 Algoritmos meméticos

Sigue la estructura de los AGG añadiendo búsqueda local a determinados individuos de la poblacion, cada ciertas épocas. En este caso se ha usado el cruce BLX que fue el que mejores resultados nos dio para el AGG. El pseudocódigo es el mismo que para AGG solamente que incluimos una nueva comprobación justo debajo de la mutación que sera la que si se cumple nos realice la búsqueda local. Esta comprobación sera que la generaciones sea un múltiplo de 10, puesto que la búsqueda local se hará cada 10 generaciones. Tendremos 3 formas distintas de realizar la búsqueda local.

1. AM-(10,1.0): Se aplica la búsqueda local a todos los elementos de la poblacion.
2. AM-(10,0.1): Se tiene una probabilidad de 0.1 para ver si un individuo de la población recibe búsqueda local o no.
3. AM-(10,0.1mej): Se aplica la búsqueda local a los 0.1*N mejores individuos de la población. N es el número total de individuos.

A la hora de calcular el número de evaluaciones que realizamos tendremos que tener en cuenta también las usadas por el algoritmos de búsqueda.

3.6.1 Parámetros

Los parámetros son muy parecidos al AGG con los cambios en el Tamaño de la población y

1. Tamaño de la poblacion(tam_poblacion): 10 individuos
2. Número de torneos(num_torneo): 2*M siendo m el tamaño de la poblacion
3. Probabilidad de cruce(prob_cruce): 1. Siempre cruzan los padres seleccionados.
4. Probabilidad de mutación(prob_mutacion): 0.001
5. Número de evaluaciones(max_evaluaciones): 15000
6. Número de vecinos BL: 2*N siendo N el número de genes de un individuo de la poblacion.

3.6.2 Pseudocódigo

Como he mencionado anteriormente el pseudocódigo es exactamente el mismo que el AGG con el cambio de que añadimos las búsquedas después de las mutaciones. Voy a realizar solo el pseudocódigo de cada tipo de memético y no incluiré todo el código, puesto que ya esta bien explicado en el apartado de AGG. Para utilizar la búsqueda local en los meméticos he realizado unas modificaciones que nos devuelve el número de evaluaciones que ha realizado, puesto que en algunas pasadas no necesita realizar la evaluación. La bl devuelve el vector de pesos, la funcion objetivos y el número de evaluaciones realizadas.

1. AM-(10,1.0)

```
n_generacion = num_generacion + 1
if num_generacion % 10 == 0
        for i in [0,tamano_poblacion]
                poblacion[i], eval_poblacion[], eval_realizadas = BL(X,
                    y,poblacion[i])
                evaluacioens += eval_realizadas
        end
end
```

2. AM-(10,0.1)

```
n_generacion = num_generacion + 1
if num_generacion % 10 == 0
        for i in [0,tamano_poblacion]
                prob_busqueda = random entre [0,1]
                if pro_busqueda <= 0.1
                        poblacion[i], eval_poblacion[], eval_realizadas
                            = BL(X,y,poblacion[i])
                        evaluacioens += eval_realizadas
                end
        end
end
```

3. AM-(10,0.1mej)

```
n_generacion = num_generacion + 1
        if num_generacion % 10 == 0
                ordenados[] = indices ordenados de la poblacion.
                    Primero los que mejor funcion objetivo tienen.
                for i in [0,truncar(0.1*tamano_poblacion)]
                        poblacion[ordenados[i]], eval_poblacion[
                            ordenados[i]], eval_realizadas = BL(X,y,
                            poblacion[i])
                        evaluacioens += eval_realizadas
        end
end
```

Después he realizado también un pequeño cambio en la forma de seleccionar las mutaciones puesto que la poblacion es mas pequeña en los memémticos. He realizado la probabilidad de mutación como en los AGE utilizando la probabilidad de mutación por cromosoma. El pseudocódigo es el siguiente.

```
prob_mutacion_cromosoma = num_genes * pro_mutacion
for i in [0,tam_poblacion]
mutacion = randon en rango [0,1]
        if mutacion <= prob_mutacion_cromosoma
                pos_gen = random entre [0, num_genes)
                nueva_poblacion[0] = mutarGen(nueva_poblacion,pos_gen)
        end
end
```

3.7 Algoritmo Enfriamiento Simulado(ES)

El enfriamiento simulado viene dado por un parámetro temperatura que sera el que represente si una solución es buena o no. En nuestro caso este parámetro temperatura lo iniciaremos con la siguiente formula:

$$tem_inicial = 0.3 * eval(mejor_solucion)/(-log(0.3))$$

En este caso eval(mejor_solución) viene dado por la solución inicial que se generara aleatoriamente.

Tenderemos que tener un valor de temperatura final que en nuestro caso sera el 10^{-3}.

El algoritmo en cada iteracion creara un nuevo vecino desde la solución anterior, usando la mutación explicada en el apartado 2. Cuando se genera el nuevo vecino se tendrá en cuenta dos posibilidades para que este pase a ser la solución actual o se descarte:

1. Si mejora a la solución de la pasada anterior esta sera la nueva solución.
2. Si la solución no es mejor se tendrá una posibilidad de que esta pase a ser la solución. Esta probabilidad vendra dada por la siguiente formula:

   $$probabilidad = exp((eval(nuevo_vecion) - eval(solucion_actual))/temperatura)$$

Siempre se mantendrá la mejor solución encontrada.

Cada vez que se generen un numero de vecinos igual a 10 veces el numero de genes de un individuo se actualizara la temperatura. La temperatura sera actualizada con la siguiente formula:

$$temperatura = temperatura/(1 + beta * temperatura)$$

El parámetro beta viene dado por:

$$beta = (temperatura\_inicial - tem\_final)/(M * temperatura\_inical * temperatrua\_final)$$

En este caso $M = (iteraciones\_maximas/(10 * numerodegenes))$

Nuestro algoritmo tendrá varios opciones de parada:

1. Numero evaluaciones: El algoritmos parara cuando evalué 15000 vecinos.
2. Temperatura: Cuando lleguemos a una temperatura menor a la temperatura final. En este caso 10^{-3}
3. No mejora: En el caso de que el algoritmo no mejore ninguna vez en las mutaciones dadas en el bucle interior.

Nuestro bucle interior sera el encargado de calcular cuantos veciones tendremos que realizar antes de actualizar la temperatura. Este bucle tiene dos casos de paradas:

1. Numero de vecinos generados: En este caso se ha explicado ya anteriormente que este parámetro tendrá un máximo dado por 10 por el numero de genes que tenga un individuo.

2. Máximo vecinos empeoran: Este caso se da cuando se acepta un nuevo vecino con peor solución. Se podrá aceptar como máximo un numero de vecinos que empeoren igual al número de genes de una solución.

Con esto pasamos a mostrar el pseudocódigo del algoritmo:

```
function ES(X,y):

        num_genes = size(X[0])


        solucion = random(0,1,num_genes)

        valor = obtenerFitness(solucion, train_datos, train_clases)
        mejor_solucion = solucion
        mejor_eval = valor


        #mido su temperatura y la guardo como mejor solucion
        tem_inicial = 0.3 * valor/ (-log(0.3))
        temperatura = tem_inicial
        tem_final = 1e-3



        max_vecinos = 10 * num_genes
        max_exitos = num_genes

        max_iter = 15000
        M = max_iter / max_vecinos
        evaluaciones = 0
        num_exitos = 1


        while evaluaciones < max_iter and temperatura > tem_final and
             num_exitos > 0
                num_vecinos = 0
                num_exitos = 0
                while num_vecinos < max_vecinos and num_exitos < max_exitos:

                        pos_gen = np.random.randint(0,num_genes)
                        vecino = mutarGen(solucion, pos_gen)
                        valor_vecino = obtenerFitness(vecino, train_datos,
                             train_clases)
                        num_vecinos += 1

                        diff = valor_vecino - valor
                        probabilidad = np.exp(diff/temperatura)
                        if diff > 0 or np.random.random() < probabilidad
                                solucion = vecino
                                valor = valor_vecino
                                num_exitos += 1
                                if valor > mejor_eval
                                        mejor_eval = valor
                                        mejor_solucion = solucion
                                end
                        end
                end

                evaluaciones += num_vecinos
                beta = (tem_inicial - tem_final)/( M * tem_inicial *
                     tem_final)
```

```
                temperatura = temperatura / (1 + beta * temperatura)

        end
        return datos_algoritmo
```

4 Algoritmo de comparación

Para la comparación he utilizado el algoritmo uno-nn. Se le pasara tanto los datos de train como los de test. El algoritmo se entrena con los datos de train. Una vez entrenado recorremos los datos de la partición de test y nos clasificar el elemento de forma que pertenecerá a la clase del elemento de train mas cercano.

También ahorro cálculos al quitar de los datos las columnas que tienen valores para los pesos menores que 0.2.

El pseudocódigo es el siguiente:

```
function uno-NN(X_train, Y_train, X_test, Y_test, pesos){

        #Elimino las columnas que estan en la misma posicion que un peso
        #menor a 0.2
        X_train = eliminarColumnas(Xtrain,pesos)
        X_text = eliminarColumnas(Xtrain,pesos)

        #Elimino los pesos que tienen valor menor que 0.2
        pesos_sin_minimos = eliminarInferiores(pesos)

        X_train = X_train * pesos_sin_minimos
        X_text = X_test * pesos_sin_minimos



        num_aciertos = 0

        #esto indica al clasificador que buscara uun unico vecion cercano
        clasificador = crearClasificador(n=1)
        clasificador = clasificador.entreno(X_train,Y_train)

        num_elementos = getNumeroElementos(Y_test)

        for i in [0,num_elementos]
                clase = clasificador.predice_clase(X_test[i])
                if clase == Y_test[i]
                        num_aciertos += 1
                end
        end

        tasa_acierto = 100 * (num_aciertos / Y_test.size)
        tasa_reduccion = tasaReduccion(pesos)
        funcion_objetivo= (tasa_acierto + tasa_reduccion) / 2

        return tasa_acierto,tasa_reducion,funcion_objetivo
end
        }
```

En el caso de obtener los resultados de los algoritmos que trabajan con poblaciones, una vez termina de ejecutar el algoritmo y tengo la poblacion final me quedo con el mejor individuo de esta y le utilizo el uno_nn para obtener su tasa de acierto , reducción y su función objetivo.

5 Explicación de desarrollo de la práctica

La he realizado en python por la gran cantidad de librerías y código ya aportado por estas para el tratamiento y clasificación de este. Aparte de ser un lenguaje que te permite implementar mas rápido los proyectos. Al ser python no hace falta hacer make ni ningún tipo de constructor para ejecutarla simplemente se ejecuta el fichero en el interprete de comandos.

En el main he puesto un for que cicla siete veces para calcular los datos de cada uno de los algoritmos con cada uno de los ficheros de datos dado. Los datos se muestran cuando se han ejecutado las 5 particiones distintas que podremos tener con todos los tres algoritmos de esta practica. Puede tardar en mostrar los datos debido a que los tiempos para la ejecución para la búsqueda local tardan mas que los demás. Esto retrasa la muestra de datos.

Si se quiere mostrar los datos de un solo algoritmo de búsqueda solo se tiene que comentar la llamada a este. Al tener las matrices donde almacenado los datos al inicio a 0 no dara error solamente mostrara los datos del algoritmos comentado como todo 0.

6 Análisis de resultados

6.1 Semilla

La semilla utilizada ha sido 1 y tengo un parámetro definido al principio del scrip llamado semilla para poder cambiarlo.

6.2 Valores utilizados

Los valores utilizados han sido los especificados en la practica, no cambien ninguno mas que para hacer algunas pruebas que no he tenido en cuenta para la practica. Los valores los tengo definidos al principio del script mediante variables para que se fácil la modificación sin tener que estar tocando todo el código.

6.3 Analisis de la practica 1

6.3.1 Tablas de resultado practica 1

Separaremos las tablas por los tipos de algoritmos a los que pertenece para ser mas fácil encontrarlas.

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	71.186441	41.935484	56.560962	0.592309	94.366197	2.941176	48.653687	0.122969	95.454545	15.0	55.227273	0.278389
Partición 2	75.438596	33.870968	54.654782	0.053658	81.428571	2.941176	42.184874	0.104845	92.727273	7.5	50.113636	0.238462
Partición 3	68.421053	29.032258	48.726655	0.051919	82.857143	2.941176	42.899160	0.105410	92.727273	2.5	47.613636	0.231540
Partición 4	71.929825	33.870968	52.900396	0.055273	91.428571	2.941176	42.899160	0.105282	96.363636	5.0	50.681818	0.225342
Partición 5	78.947368	32.258065	55.602716	0.071065	88.571429	2.941176	45.756303	0.105393	97.272727	2.5	49.886364	0.220764
Media	73.184657	34.193548	53.689102	0.164845	87.730382	2.941176	45.335779	0.108780	94.909091	6.5	50.704545	0.238899

Tabla 5.1: Resultados obtenidos por el algoritmo 1-NN en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	71.186441	0.0	35.593220	0.025666	90.140845	0.0	45.070423	0.030654	93.636364	0,00	46.818182	0.043423
Partición 2	71.929825	0.0	35.964912	0.024994	81.428571	0.0	40.714286	0.030145	90.909091	0.0	45.454545	0.042935
Partición 3	77.192982	0.0	38.596491	0.025259	81.428571	0.0	40.714286	0.030164	92.727273	0.0	46.363636	0.041187
Partición 4	75.438596	0.0	37.719298	0.025259	91.428571	0.0	45.714286	0.030123	93.636364	0.0	46.818182	0.041029
Partición 5	82.456140	0.0	41.228070	0.024933	85.714286	0.0	42.857143	0.030081	96.363636	0.0	48.181818	0.042407
Media	75.640797	0.0	37.820398	0.025222	86.028169	0.0	43.014085	0.030233	93.454545	0.0	46.727273	0.042196

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	72.881356	72.580645	77.956989	3.865621	85.915493	82.352941	86.533613	2.597900	88.181818	82.5	86.136364	2.905755
Partición 2	68.421053	82.258065	80.042076	3.629750	82.857143	88.235294	90.914800	2.127955	92.727273	82.5	87.613636	5.444470
Partición 3	73.684211	74.193548	75.855540	4.932757	82.857143	88.235294	89.847184	3.453716	85.454545	80.0	86.136364	4.427295
Partición 4	77.192982	69.354839	73.155680	7.055729	88.571429	88.235294	89.088340	1.764488	95.454545	85.0	89.090909	2.453718
Partición 5	73.684211	77.419355	79.361851	15.476370	91.428571	73.529412	82.138371	2.316528	85.454545	82.5	87.500000	3.046445
Media	73.172762	75.161290	77.274427	6.992045	86.325956	84.117647	87.704462	2.452118	89.454545	82.5	87.295455	3.655537

Figure 1:

6.3.2 Análisis de resultados practica 1

Como se ve en las tablas el la BL siempre gana en la función objetivo, esto es debido que obtiene una tasa de reducción muy superior a los otros dos algoritmos. Al tener el valor para calcular la función objetivo con 0.5 se le esta dando la misma importancia a reducir la tasa de acierto y la tasa de reducción. Por este motivo en términos de función objetivo el mejor algoritmos de estos tres es la BL. El problema de la BL es que en tasa de acierto no suele ser el mejor en ninguna de los conjunto de datos. Si no tenemos en cuenta la tasa de reducción no podríamos elegir un mejor algoritmo puesto que en unos conjuntos de datos salen vencedor el RELIEF y en otros el 1-NN, pero la diferencia entre el porcentaje de acierto entre uno u otro suele ser muy pequeño. La búsqueda local en este caso se suele situar muy cercano a los dos superando a algunas veces a uno de los dos. Pero nunca consigue situarse como el que mas tasa de acierto tiene.

6.3.3 Tiempos de ejecución resultados practica1

El tiempo de ejecución de menor a mayor como era obvio el que menos tarda es el 1-nn puesto que los otros dos utilizan a este para la clasificación. Es prácticamente instantáneo

Lo sigue el RELIEF que si el anterior tarda como el doble o triple que esta pero tampoco es un tiempo considerable.

La búsqueda local es la mas lenta. Conseguí reducirla mucho al tener en cuenta que hay ocasiones en las que no hacia falta volver a valorar un peso porque no es posible que este mejore. Siendo los datos en lo que mas tarda los que mas número de características por elemento tienen.

6.3.4 Conclusión final resultados practica 1

No podríamos elegir uno entre los tres puesto que cada uno tiene sus puntos fuertes. Lo bueno de la búsqueda local es que reduciendo mas del doble que el RELIEF no perdemos mas de 5% en el peor caso. Esto nos puede venir bien para saber que características de nuestros conjuntos son las que mas peso tienen a la hora de clasificar. En cambio siempre es superada por uno de los otros dos algoritmos como mínimo a la hora de clasificar los datos.

6.4 Análisis de AGG

6.4.1 Tablas de AGG

Se tendrá una tabla con cada uno de los cruces realizado.

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

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	74.576271	77.419355	75.997813	55.016733	83.098592	91.176471	87.137531	33.473444	87.272727	80.0	83.636364	57.146631
Partición 2	77.192982	77.419355	77.306169	54.350359	88.571429	88.235294	88.403361	44.724478	90.000000	82.5	86.250000	49.411809
Partición 3	71.929825	77.419355	74.674590	46.618906	92.857143	85.294118	89.075630	46.431660	90.000000	85.0	87.500000	54.140635
Partición 4	78.947368	72.580645	75.764007	47.491210	84.285714	88.235294	86.260504	40.511576	90.000000	85.0	87.500000	72.105262
Partición 5	84.210526	72.580645	78.395586	48.204176	91.428571	82.352941	86.890756	47.345805	90.000000	85.0	87.500000	81.482647
Media	77.371395	75.483871	76.427633	50.336277	88.048290	87.058824	87.553557	42.497393	89.454545	83.5	86.477273	62.857397

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

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	76.271186	67.741935	72.006561	52.502099	95.774648	79.411765	87.593206	55.775730	90.000000	80.0	85.000000	100.45792
Partición 2	68.421053	58.064516	63.242784	64.266662	90.000000	70.588235	80.294118	41.457076	87.272727	80.0	83.636364	100.38173
Partición 3	71.929825	70.967742	71.448783	62.303401	87.142857	88.235294	87.689076	48.472584	90.909091	77.5	84.204545	99.990265
Partición 4	63.157895	77.419355	70.288625	54.335206	87.142857	79.411765	83.277311	46.099188	90.000000	82.5	86.250000	79.399986
Partición 5	78.947368	59.677419	69.312394	69.145283	87.142857	91.176471	89.159664	37.097742	90.000000	70.0	80.000000	87.324275
Media	71.745465	66.774194	69.259829	60.510530	89.440644	81.764706	85.602675	45.780464	89.636364	78.0	83.818182	93.510838

Figure 2:

6.4.2 Análisis de resultado AGG

En este caso el cruce BLX es superior en función objetivo al cruce martinico al obtener entre un 3% y 7%. Quitando el conjunto de datos de colposcopy, en los demás los dos algoritmos están muy próximos en tasa de acierto. El problema del cruce aritmético es que reduce un poco peor que el BLX y al estar igualados

en clase el BLX destaca un poco en la función objetivo al obtener mas reducción. BLX reduce mejor al permitir que los genes de los hijos puedan tener valores inferiores al mínimo de los padres. En cambio en el cruce aritmético esto no se puede dar. Para conseguir valores en los hijos menores que 0.2 uno de los genes de los padres tendría que ser menor a 0.2.

En tasa de clase estan muy igualados en todos los conjuntos de datos menos en colposcopy donde el blx le saca un 6%

6.4.3 Tiempos de ejecución de AGG

El tiempo de ejecución del cruce aritmético es superior a tiempo con el cruce blx. Esto es debido a que el cruce aritmético genera un mayor número de aleatorios (uno por cada gen del vector de pesos).

6.4.4 Conclusión final de AGG

En AGG me quedo con el cruce blx puesto que es mas rápido y siempre esta por encima en tasa de reducción y función objetivo. Ademas en los únicos casos en los que el aritmético ha gando al blx en tasa de clasificación ha sido por una diferencia menor a un 1%

6.5 Análisis de AGE

Igual que en el apartado anterior compararemos los dos cruces pero esta vez para los algoritmos genéticos estacionarios.

6.5.1 Tablas de AGE

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

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	76.271186	72.580645	74.425916	58.677481	88.732394	82.352941	85.542668	49.644899	92.727273	75.0	83.863636	78.553590
Partición 2	71.929825	72.580645	72.255235	62.614056	87.142857	88.235294	87.689076	40.788019	90.000000	85.0	87.500000	70.283978
Partición 3	71.929825	75.806452	73.868138	50.355225	91.428571	79.411765	85.420168	43.361625	90.909091	80.0	85.454545	80.117857
Partición 4	63.157895	74.193548	68.675722	48.180324	85.714286	76.470588	81.092437	51.081919	87.272727	82.5	84.886364	62.322908
Partición 5	75.438596	72.580645	74.009621	51.200417	88.571429	73.529412	81.050420	45.263765	90.000000	80.0	85.000000	59.748156
Media	71.745465	73.548387	72.646926	54.205500	88.317907	80.000000	84.158954	46.028045	90.181818	80.5	85.340909	70.205298

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

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	69.491525	75.806452	72.648989	43.543957	85.915493	76.470588	81.193041	37.587543	89.090909	77.5	83.295455	75.954388
Partición 2	68.421053	69.354839	68.887946	47.384384	92.857143	82.352941	87.605042	32.152689	91.818182	75.0	83.409091	71.823383
Partición 3	73.684211	70.967742	72.325976	46.912623	92.857143	58.823529	75.840336	42.165809	90.909091	80.0	85.454545	63.879918
Partición 4	78.947368	70.967742	74.957555	50.897690	84.285714	82.352941	83.319328	26.739946	89.090909	77.5	83.295455	73.409285
Partición 5	75.438596	61.290323	68.364460	64.339729	88.571429	85.294118	86.932773	29.708551	88.181818	80.0	84.090909	67.430925
Media	73.196551	69.677419	71.436985	50.615677	88.897384	77.058824	82.978104	33.670908	89.818182	78.0	83.909091	70.499580

Figure 3:

6.5.2 Análisis de resultado AGE

En el generacional seguimos teniendo al cruce blx por encima del aritmético. Aunque en este caso están mucho mas próximo con una diferencia máxima en

la función objetivo de un 2%. En este caso la tasa de clase esta muy igualada, lo que marca esa pequeña diferencia es que el blx sigue teniendo una mejor tasa de reducción.

6.5.3 Tiempos de ejecución de AGE

En este caso los tiempos de ejecución son menores en el cruce aritmético que en el blx. Esto me hace pensear dos posibilidades. Esto podría darse por dos razones.

1. En las mediciones del AGGAritmético el procesador estaba sobrecargado por algunos procesos externos a la ejecución del scirpt.

2. Generar un gran número de aleatorios de forma continuada al tener muchos mas cruces en el AGG que en el AGE se sobrecarga el procesador y por eso en el AGE reduce el tiempo. Si fuera esta opción podría tardar mas el cruce blx en el AGE debido a que tiene una mayor carga de operaciones y comprobaciones que el aritmético.

6.5.4 Conclusión final de AGE

En este caso los dos algoritmos están muy igualados y cualquiera de los dos nos podría dar resultados muy parecidos aunque el cruce blx sigue siendo un poquito mejor.

6.6 Análisis de AGE vs AGG

Vamos a comparar los resultados sin tener en cuenta el tipo de cruce elegido. Lo que compararemos es si hay mejora entre el AGG y el AGE con el mismo cruce.

6.6.1 Tablas de AGE vs AGG

Vamos a comparar las medias obtenidas para los cuatro algoritmos.

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Aar.*	T	*% clas*	*%red*	*Aar.*	T	*% clas*	*%red*	*Aar.*	T
AGGBLX	77.371395	75.483871	76.427633	50.336277	88.048290	87.058824	87.553557	42.497393	89.454545	83.5	86.477273	62.857397
AGGAritmetico	71.745465	66.774194	69.259829	60.510530	89.440644	81.764706	85.602675	45.780464	89.636364	78.0	83.818182	93.510838
AGEBLX	71.745465	73.548387	72.646926	54.205500	88.317907	80.000000	84.158954	46.028045	90.181818	80.5	85.340909	70.205298
AGEAritmetico	73.196551	69.677419	71.436985	50.615677	88.897384	77.058824	82.978104	33.670908	89.818182	78.0	83.909091	70.499580

Figure 4:

6.6.2 Análisis de resultado AGE vs AGG

En este caso los algoritmos AGG son superiores en casi todos los caso que los algoritmos AGE. Esto puede darse a que en el AGG tengamos una mayor exploración al no estar obligando al algoritmos a sustituir siempre los hijos nuevos si son mejores que los peores individuos de la poblacion. Esto hace que

puedan entrar soluciones que en principio sean peores pero que puedan sacarnos de algún optimo local.

6.6.3 Conclusión final AGE vs AGG

En este caso me quedaría con los algoritmos AGG y mas específicamente con el AGG con cruce blx que esta un poquito por encima de las otras tres opciones.

6.7 Análisis de AM

6.7.1 Tablas de AM

Tabla 5.1: 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	72.881356	83.870968	78.376162	37.093490	85.915493	88.235294	87.075394	21.077355	88.181818	87.5	87.840909	45.348703
Partición 2	70.175439	83.870968	77.023203	40.463850	85.714286	91.176471	88.445378	20.520685	90.000000	87.5	88.750000	42.794309
Partición 3	66.666667	90.322581	78.494624	32.983060	87.142857	94.117647	90.630252	19.951123	90.909091	82.5	86.704545	59.389557
Partición 4	77.192982	82.258065	79.725523	40.901154	82.857143	91.176471	87.016807	19.578216	87.272727	85.0	86.136364	52.804496
Partición 5	80.701754	82.258065	81.479909	42.949952	88.571429	91.176471	89.873950	20.936393	92.727273	85.0	88.863636	41.896468
Media	73.523640	84.516129	79.019884	38.878301	86.040241	91.176471	88.608356	20.412754	89.818182	85.5	87.659091	48.446706

Tabla 5.1: 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	69.491525	82.258065	75.874795	36.262690	87.323944	91.176471	89.250207	21.149663	84.545455	87.5	86.022727	52.411501
Partición 2	64.912281	87.096774	76.004527	37.005414	81.428571	88.235294	84.831933	20.895147	90.000000	87.5	88.750000	57.305998
Partición 3	71.929825	83.870968	77.900396	38.033948	90.000000	88.235294	89.117647	22.138782	91.818182	85.0	88.409091	49.365058
Partición 4	70.175439	85.483871	77.829655	37.476011	82.857143	91.176471	87.016807	21.444283	90.000000	87.5	88.750000	48.697368
Partición 5	78.947368	77.419355	78.183362	41.702421	85.714286	91.176471	88.445378	20.970878	88.181818	87.5	87.840909	49.450640
Media	71.091288	83.225806	77.158547	38.096097	85.464789	90.000000	87.732394	21.319750	88.909091	87.0	87.954545	51.446113

Tabla 5.1: Resultados obtenidos por el algoritmo AM-(10,0.1mej) en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	67.796610	90.322581	79.059595	33.195340	88.732394	91.176471	89.954432	21.115867	89.090909	85.0	87.045455	50.428138
Partición 2	70.175439	88.709677	79.442558	32.890356	85.714286	91.176471	88.445378	19.812891	93.636364	85.0	89.318182	44.74873
Partición 3	70.175439	83.870968	77.023203	34.630635	90.000000	91.176471	90.588235	22.206109	88.181818	82.5	85.340909	55.398460
Partición 4	68.421053	80.645161	74.533107	39.384717	82.857143	91.176471	87.016807	19.941247	90.000000	85.0	87.500000	47.941728
Partición 5	75.438596	88.709677	82.074137	34.265765	85.714286	91.176471	88.445378	21.686684	89.090909	82.5	85.795455	54.045469
Media	70.401427	86.451613	78.426520	34.873363	86.603622	91.176471	88.890046	20.952560	90.000000	84.0	87.000000	50.512506

Figure 5:

6.7.2 Análisis de resultados AM

En este caso están todos muy igualados. Con una diferencia máxima de in 1% en la función objetivo. Ninguno destaca por encima del otro en clasificación o en reducción.

6.7.3 Tiempos de ejecución AM

En este apartados estamos igual que el apartado anterior. Ninguno destaca por encima de otro para poder decidir que uno de ellos es mas rápidos. Las diferencias que se dan son mínimas y podrían darse por una sobrecarga en le procesador.

6.7.4 Conclusión final AM

No propria elegir uno por encima de los demás puesto que tanto en tiempo como en función objetivo son prácticamente idénticos.

6.8 Análisis AM vs AG

6.8.1 Tablas de AM vs AG

	Colposcopy				Ionosphere				Texture			
	% clas	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T	*% clas*	*%red*	*Agr.*	T
AGGBLX	77.371395	75.483871	76.427633	50.336277	88.048290	87.058824	87.553557	42.497393	89.454545	83.5	86.477273	62.857397
AGGAritmetico	71.745465	66.774194	69.259829	60.510530	89.440644	81.764706	85.602675	45.780464	89.636364	78.0	83.818182	93.510838
AGEBLX	71.745465	73.548387	72.646926	54.205500	88.317907	80.000000	84.158954	46.028045	90.181818	80.5	85.340909	70.205298
AGEAritmetico	73.196551	69.677419	71.436985	50.615677	88.897384	77.058824	82.978104	33.670908	89.818182	78.0	83.909091	70.499580
AM-(10,1)	73.523640	84.516129	79.019884	38.878301	86.040241	91.176471	88.608356	20.412754	89.818182	85.5	87.659091	48.446706
AM-(10,0.1)	71.091288	83.225806	77.158547	38.096097	85.464789	90.000000	87.732394	21.319750	88.909091	87.0	87.954545	51.446113
AM-(10,0.1mej)	70.401427	86.451613	78.426520	34.873363	86.603622	91.176471	88.890046	20.952560	90.000000	84.0	87.000000	50.512506

Figure 6:

6.8.2 Análisis de resultados AM vs AG

En este caso es muy claro que los vencedores en cuanto a función objetivo son los meméticos. Los tres algoritmos meméticos están siempre por encima de todos los algoritmos genéticos en función objetivo debido a que tienen de media un 3% mas de tasa de reducción como mínimo al compararlo con el AGG con cruce blx que es el que mas se les aproxima. Esto se puede deber a que la búsqueda local con la mutación que tiene para generar vecinos conseguía una muy buena reducción. En cambio los AG están igualados o mejoran un poco la tasa de clasificación de los AM.

6.8.3 Tiempos ejecución AM vs AG

En este apartado si son claramente mejores los algoritmos memeticos puesto que las evaluaciones que utilizamos par ala búsqueda local son mucho menos costosas que realizar los torneos binarios, cruces y las mutaciones. Al perder evaluaciones en la búsqueda local realizaremos muchos menos ejecuciones del bucle y ahorramos unos 15 y 20 segundo.

6.8.4 Conclusión final AM vs AG

Los algoritmos meméticos pierden un porcentaje muy pequeño de tasa de clasificación pero ganan en la tasa de reducción y tiempo con una diferencia mucho mayor.

6.9 Análisis final de todos los algoritmos

Con todo lo explicado anteriormente vamos ha realizar una comparación final entre todos los algoritmos por encima.

6.9.1 Tablas de tiempos

Tabla 5.2: Resultados globales en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
RELIEF	73.184657	34.193548	53.689102	0.164845	87.730382	2.941176	45.335779	0.108780	94.909091	6.5	50.704545	0.238899
1-NN	75.640797	0.0	37.820398	0.025222	86.028169	0.0	43.014085	0.030233	93.454545	0.0	46.727273	0.042196
BL	73.172762	75.161290	77.274427	6.992045	86.325956	84.117647	87.704462	2.452118	89.454545	82.5	87.295455	3.655537
AGGBLX	77.371395	75.483871	76.427633	50.336277	88.048290	87.058824	87.553557	42.497393	89.454545	83.5	86.477273	62.857397
AGGAritmetico	71.745465	66.774194	69.259829	60.510530	89.440644	81.764706	85.602675	45.780464	89.636364	78.0	83.818182	93.510838
AGEBLX	71.745465	73.548387	72.646926	54.205500	88.317907	80.000000	84.158954	46.028045	90.181818	80.5	85.340909	70.205298
AGEAritmetico	73.196551	69.677419	71.436985	50.615677	88.897384	77.058824	82.978104	33.670908	89.818182	78.0	83.909091	70.499580
AM-(10,1)	73.523640	84.516129	79.019884	38.878301	86.040241	91.176471	88.608356	20.412754	89.818182	85.5	87.659091	48.446706
AM-(10,0.1)	71.091288	83.225806	77.158547	38.096097	85.464789	90.000000	87.732394	21.319750	88.909091	87.0	87.954545	51.446113
AM-(10,0.1mej)	70.401427	86.451613	78.426520	34.873363	86.603622	91.176471	88.890046	20.952560	90.000000	84.0	87.000000	50.512506

Figure 7:

6.9.2 Análisis de resultados

Con los datos visto hasta ahora los meméticos son los que mejores resultados nos han estado dando en las comparaciones realizadas. En este caso vamos a comparar los meméticos con la búsqueda local.

La búsqueda local esta siempre entre los mejores en todos los conjuntos de datos. Y tiene un muy buen equilibrio entre tasa de clase y de reducción. No tiene nada que envidar en esto a los meméticos. Tienen mas o menos la misma función objetivo.

6.9.3 Tiempos de ejecución

En tiempos de ejecución los algoritmos de la primera practica son muy superiores tardando entre unas 10 o 20 veces menos la búsqueda local que es el mas lento de estos tres.

6.9.4 Conclusiones finales

Visto que la búsqueda local consigue las mismas optimizaciones que los algoritmos meméticos en muchísimo menos tiempo, me quedaría con la búsqueda local.

7 Bibliografía

No he usado nada fuera de las propias paginas de información de python o de las librerías utilizadas como podría ser la de numpy.