

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	1
2	Aplicación de los algoritmos	2
2.1	Descripción de la representación y calculo de distancias	2
2.2	Calculo de la tasa de reducción	3
2.3	Calculo de la tasa de clase	3
2.4	Función de evaluación	3
2.5	Generación de la población inicial	4
2.6	Torneo binario	4
2.7	Cruce BLX	5
2.8	Cruce aritmético	6
2.9	Mutar ge	7
3	Explicación de los algoritmos	7
3.1	Algoritmo del vecino mas cercanos 1-NN	7
3.2	Algoritmo RELIEF	8
3.3	Algoritmo de Búsqueda Local	10
3.4	Algoritmo genético generacional	12
3.4.1	Parámetros	12
3.4.2	Pseudocódigo	13
3.5	Algoritmo genético estacionario	14
3.5.1	Parámetros	14
3.5.2	Pseudocódigo	15
3.6	Algoritmos meméticos	16
3.6.1	Parámetros	16
3.6.2	Pseudocódigo	16
4	Algoritmo de comparación	17
5	Explicación de desarrollo de la práctica	18
6	Análisis de resultados	19
6.0.1	Semilla	19
6.0.2	Valores utilizados	19
6.0.3	Tablas de resultado	19
6.0.4	Análisis de resultados	20
6.0.5	Tiempo de ejecución	21
7	Bibliografía	21

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.

$$(x_1, x_2, \dots, x_n, y)$$

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 el fichero.

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

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:

$$(w_1, w_2, \dots, w_j)$$

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.

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

$$eval_poblacion = (valor_1, valor_2, \dots, valor_i)$$

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:

$$tasa_{reduccion} = 100 \frac{n^{\circ} \text{ de } w_j < 0.2}{w.size}$$

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 realiza en una línea:

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:

$$tasa_{clase} = 100 \frac{n^{\circ} \text{ de instancias bien clasificadas}}{n^{\circ} \text{ instancias totales en test}}$$

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

```
num_aciertos = 0
for i in test
    clase_clasificado = clasificador_1NN(i,pesos)
    if clase(i) == clase_clasificado
        num_aciertos += 1
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 numero 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 población 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 población durante la siguiente generación y el perdedor será eliminado.

La selección de los padres a enfrentar se hará con dos enteros aleatorios que representarán los padres de la población 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[padre1]
            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 población $C_1 = (c_{11}, \dots, c_{1n})$ y $C_2 = (c_{21}, \dots, c_{2n})$ y se crean dos hijos. Para esto cogemos cada gen de los cromosomas C_1 y C_2 y se generan dos nuevos que serán introducidos 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 .

$$C_{\max} = \max \{c_{1i}, c_{2i}\}$$

$$C_{\min} = \min \{c_{1i}, c_{2i}\}$$

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

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

3. Con esto generamos dos aleatorios en el rango

$$[C_{\min} - I \cdot \alpha, C_{\max} + I \cdot \alpha]$$

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

Esto se realizará 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
end

```



```

        return hijo1, hijo2
    end

```

2.10 Mutar ge

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

```

function 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 buscare 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 mismo 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
    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 random 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 numero 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 = uno_nn(train_datos, train_clases
    , test_datos, test_clases, w)

    funcion_mejora = funcionObjetivo(tasa_clase,tasa_reduccion)

    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 = uno_nn(
            train_datos, train_clases, test_datos,
            test_clases, w)
            funcion_mejora = funcionObjetivo(tasa_clase,
            tasa_reduccion)

            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 = obtenemosSigientePosicion(pos_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 población 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 población. En este caso usamos el torneo binario para realizar esta selección. Los individuos con mejor función objetivo tendrán más opciones de ganar un combate y quedar en la nueva población. Los individuos que competirán se eligen aleatoriamente. Si se tienen N individuos en la población se realizarán $N*2$ torneos.

Después realizamos los cruces entre padres para generar nuevos hijos. Estos hijos sustituirán a los padres en la nueva población. Se tiene una probabilidad de que dos padres crucen. Para 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 realizará este número de cruces empezando por los primeros individuos de la población.

El siguiente paso sería realizar las mutaciones en los genes a partir 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 población, se calculará 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 mutará.

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 cambiará el peor individuo de la población actual por el mejor de la anterior. Esto nos asegura que como mínimo tendremos el mejor individuo conocido o alguno mejor en la nueva población.

3.4.1 Parámetros

Los parámetros para nuestro AGG son los siguientes:

1. Tamaño de la población(tam_poblacion): 30 individuos
2. Número de torneos(num_torneo): $2*M$ siendo m el tamaño de la población
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 inividuo
    num_cruces = int(prob_cruce * tam_poblacion)
    num_mutaciones = 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
        //obtenomos 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

    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 numero 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. Numero 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. Numero 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 = random en rango [0,1]
        mutacion2 = random 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 población, 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 será la que si se cumple nos realice la búsqueda local. Esta comprobación será que la generación 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 población.
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 $0.1 \cdot N$ 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 algoritmo 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 población(tam_poblacion): 10 individuos
2. Número de torneos(num_torneo): $2 \cdot M$ siendo m el tamaño de la población
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: $2 \cdot N$ siendo N el número de genes de un individuo de la población.

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 está 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 función 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 prob_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 población es mas pequeña en los memémicos. 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 = random 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
```

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 clasificamos 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(X_train,pesos)
    X_test = eliminarColumnas(X_test,pesos)

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

    X_train = X_train * pesos_sin_minimos
    X_test = X_test * pesos_sin_minimos

    num_aciertos = 0

    #esto indica al clasificador que buscare un unico vecino 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_reduccion,funcion_objetivo
end
}
```

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 tres 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

6.0.1 Semilla

La semilla utilizada ha sido 14 y tengo un parámetro definido al principio del scrip llamado semilla para poder cambiarlo.

6.0.2 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.

6.0.3 Tablas de resultado

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	74.576271	32.258065	53.417168	0.077334	85.915493	2.941176	44.428335	0.122969	94.545455	5.0	49.772727	0.278389
Partición 2	82.456140	27.419355	54.937748	0.076381	87.142857	2.941176	45.042017	0.104845	92.727273	2.5	47.613636	0.238462
Partición 3	78.947368	24.193548	51.570458	0.078437	87.142857	2.941176	45.042017	0.105410	88.181818	20.0	54.090909	0.231540
Partición 4	70.175439	35.483871	52.829655	0.076579	87.142857	2.941176	45.042017	0.105282	94.545455	2.5	48.522727	0.225342
Partición 5	73.684211	24.193548	48.938879	0.078767	94.285714	2.941176	48.613445	0.105393	92.727273	12.5	52.613636	0.220764
Media	75.967886	28.709677	52.338782	0.077500	88.325956	2.941176	45.633566	0.108780	92.545455	8.5	50.522727	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	84.507042	0.0	42.253521	0.030654	91.818182	0.0	45.909091	0.043423
Partición 2	78.947368	0.0	39.473684	0.024994	88.571429	0.0	44.285714	0.030145	91.818182	0.0	45.909091	0.042935
Partición 3	77.192982	0.0	38.596491	0.025259	84.285714	0.0	42.142857	0.030164	87.272727	0.0	43.636364	0.041187
Partición 4	70.175439	0.0	35.087719	0.025259	85.714286	0.0	42.857143	0.030123	92.727273	0.0	46.363636	0.041029
Partición 5	73.684211	0.0	36.842105	0.024933	90.000000	0.0	45.000000	0.030081	93.636364	0.0	46.818182	0.042407
Media	74.237288	0.0	37.118644	0.025222	86.615694	0.0	43.307847	0.030233	91.454545	0.0	45.727273	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	81.355932	53.225806	67.290869	38.819485	94.366197	82.352941	88.359569	27.224013	98.181818	80.0	89.090909	79.786123
Partición 2	89.473684	69.354839	79.414261	125.77766	95.714286	79.411765	87.563025	37.040083	95.454545	77.5	86.477273	38.796238
Partición 3	82.456140	58.064516	70.260328	50.393600	95.714286	82.352941	89.033613	23.723314	88.181818	77.5	82.840909	42.283510
Partición 4	80.701754	62.903226	71.802490	62.182601	97.142857	88.235294	89.747899	41.164473	92.727273	75.0	83.863636	80.049610
Partición 5	84.210526	69.354839	76.782683	94.267057	98.571429	79.411765	88.991597	42.310930	98.181818	75.0	86.590909	37.313678
Media	83.639607	62.580645	73.110126	74.288081	96.301811	81.176471	88.739141	34.292563	94.545455	77.0	85.772727	55.645832

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	75.967886	28.709677	52.338782	0.077500	88.325956	2.941176	45.633566	0.108780	92.545455	8.5	50.522727	0.238899
1-NN	74.237288	0.0	37.118644	0.025222	86.615694	0.0	43.307847	0.030233	91.454545	0.0	45.727273	0.042196
BL	83.639607	62.580645	73.110126	74.288081	96.301811	81.176471	88.739141	34.292563	94.545455	77.0	85.772727	55.645832

Figure 1:

6.0.4 Análisis de resultados

Los resultados obtenidos nos muestra que la búsqueda local es muy superior a las otras en cuanto a obtener mejores resultados y la que mayor tasa de reducción. Esto se debe a que es la que mas opciones comprueba y siempre se va quedando con la mejor. Con esto evita perder buenas soluciones. Ademas al solo aceptar la mutación cuando mejora tiende a mejorar rápido hasta que se cumplen las 20*n iteraciones sin mejora. Estas iteraciones sin mejora son siempre la causa de que termine el algoritmo sin llegar nunca a el máximo de vecinos generado. Suele generar unos 7000 vecinos la vez que mas y normalmente ronda entre 2000 y 4000 en textura por ejemplo.

El siguiente que mejor resultados me suele dar es el RELIEF aunque el 1-nn no se diferencia tanto de el. Esto se trata a que el RELIEF si utiliza los datos de entrenamiento para intentar mejorar mediante la ponderación de los pesos. El 1-nn depende de lo parecidos que sean las particiones de entrenamiento y test puesto que no obtienen ningún tipo de mejora para aproximar sus datos a los de entrenamiento.

Algo curioso es que en los ficheros de IONOSPHERE RELIEF siempre me reduce el mismo numero de pesos. Imagino que podrá ser por pesos que son muy parecidos para todos los elementos de la muestra. Y por eso los tiende a

reducir al no ser significativos.

El mejor conjunto de datos para el ajuste es textura. Llegando a obtener mas de 90% de media para todos los datos. El mas difícil de clasificar correctamente es el colposcopy. Aunque en la función de evaluación mejora mucho colposcopy porque suele ser tener una buena tasa de reducción. Y al darle la misma importancia a la clasificación y la reducido eso hace que se la que mejor función objetivo tenga. Esto no se cumple por ejemplo en el 1-nn al no tener reducción es la que peor función objetivo tiene

En la búsqueda local me sorprende que se capaz de ajustar tanto teniendo tasa reducción tan altas. Entiendo que con eso evitara mucho ruido que dan los datos eliminados

6.0.5 Tiempo de ejecución

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 triple que esta pero tampoco es un tiempo considerable.

La búsqueda local es mucho 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. Llegue a bajar de tardar unos 10 minutos de media partición a 1.25 minutos la que mas me puede tardar. También es verdad que depende de la semilla en algunas condiciones. Siendo los datos en lo que mas tarda los que mas numero de características por elemento tienen.

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.