Metaheurísticas (2016-2017)
Grado en Ingeniería Informática
Universidad de Granada

Practica 2: APC

Samuel Cardenete Rodríguez.
Correo: samuelcr1995@correo.ugr.es
DNI: 75934968P

3 de junio de 2017

Índice

Índice de figuras

1. Formulación del problema. APC

1.1. Introducción

El problema a abordar en esta práctica consiste en el aprendizaje de pesos por características (APC).
Se trata de un problema de búsqueda con codificación real en el espacio n-dimensional, siendo n el número de características.

Estamos delante de un problema de Machine Learning, mediante el cuál dado un conjunto de datos no etiquetados obtengamos una etiqueta para cada uno de los datos (por ejemplo, determinar a partir de determinados parámetros si se tiene o no cáncer) bajo una tasa de acierto (función objetivo), siempre a maximizar.

1.2. Objetivos

Como objetivo principal del problema, ajustaremos un conjunto de ponderaciones o pesos asociados al conjunto total de características, utilizando para ello un conjunto de datos de entrenamiento (ya etiquetados), de tal forma que los clasificadores que se construyan a partir de él sean mejores.

APC asigna valores reales a las características, de tal forma que se describe o pondera la relevancia de cada una de ellas al problema del aprendizaje.
Para la resolución del problema, nos basaremos en optimizar el rendimiento de un clasificador basado en vecinos cercanos a partir de la inclusión de pesos asociados a las características del problema que modifican su valor en el momento de calcular las distancias entre ejemplos. En nuestro caso, el clasificador considerado será el 1-NN (vecino más cercano). Para ello adaptaremos las dos siguientes técnicas metaheurísticas:

- Enfriamiento Simulado: Implementación del Enfriamiento llevando a cabo un esquema de enfriamiento de Cauchy modificado. Para la generación de vecinos se emplea el el movimiento de cambio por mutación normal.
- Búsqueda local reiterada: Implementamos la ILS generando la solución inicial de forma aleatoria y aplicamos búsquedas locales reiteradas, aplicando mutaciones sobre la mejor solución obtenida.
- Evolución diferencial: Implementamos el algoritmo basado en modelo evolutivo. Generaremos dos algoritmos, uno con formula de mutación *DE/Rand/1* y otro *DE/current-to-best/1*, ambos emplearan recombinación binomial.

Tras esto realizaremos un estudio comparativo de los resultados obtenidos por las distintas técnicas metaheurísticas.

2. Función objetivo

Debido a la falta de diversidad en los datos a evaluar, realizaremos modificaciones en el método de evaluación de los datos.

Ahora consideraremos un nuevo criterio basado en la simplicidad del clasificador diseñado basado en la minimización del número de características a utilizar en el clasificador final. De tal forma que ahora nuestra función objetivo consistirá en la agregación del porcentaje de clasificación realizado por la solución y el porcentaje de características (pesos) inferiores a 0.1 obtenidas.
Además, se establece una ponderación entre estos dos agregados, de forma que los dos tengan igual importancia, es decir, $\alpha = 0{,}5$.
La función objetivo quedaría así:

$$tasa\ clasificacion = 100 * \frac{\text{nº instancias bien clasificadas}}{\text{nº instancias totales}}$$

$$tasa\ reduccion = 100 * \frac{\text{nº valores } w_i > 0{,}1}{\text{nº valores totales}}$$

$$funcion\ evaluacion = tasa\ clasificacion * \alpha + (1 - \alpha) * tasa\ reduccion$$

En nuestro caso, como deseamos darle igual importancia dando a la reducción como a la clasificación, indicaremos un $\alpha = 0{,}5$.

Puesto que ahora deseamos que sólo se tengan en cuenta aquellos pesos cuyo valor sea superior o igual a 0.1, será necesario modificar la función 'CalcularDistancia' que emplea 1-nn para que aquellos pesos cuyo valor sea inferior a 0.1 sean ignorados; por lo que el pseudocódigo sería el siguiente:

```
//Calcular la distancia entre dos cromosomas
// teniendo en cuenta el vector de pesos:
Calcular distancia cromosoma A y B:
        Para cada i ∈ Atributos(A) y j ∈ Atributos(B)
                Si Pesos_i >= 0,1
                        distancia = distancia + sqrt(Pesos_i * (i - j)^2)
        Fin para cada

        Devolver distancia
```

3. Mecanismo de validación

El mecanismo empleado para la validación de las soluciones obtenidas por los algoritmos es la validación cruzada.

Consiste en la realización de validación cruzada donde realizamos 5 particiones de los datos, de forma que cada partición es del mismo tamaño que el conjunto originar de

datos.
Ahora, para realizar la validación, para cada partición, emplearemos 4/5 de los datos para train, es decir, para obtener el modelo de aprendizaje; y el conjunto restante (1/5) para evaluar el modelo obtenido.

De esta forma el esquema de evaluación en cada una de las 5 iteraciones de la validación cruzada será el siguiente:

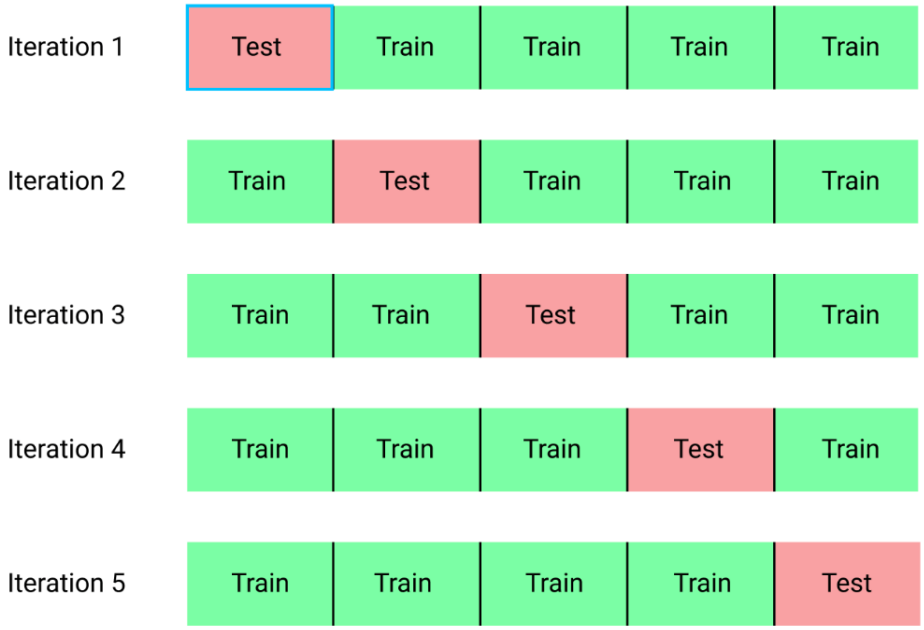

Figura 3.1: 5fold CrossValidation

De esta forma, siguiendo dicho esquema de validación, el pseudo código del particionamiento sería el siguiente:

```
Particionamiento de datos Datos
        //Mezclamos el conjunto de datos para mejorar la variabilidad:
        sample(datos)
        //Dividimos en cinco trozos los datos
        tamFold = Datos/5

        Desde i=0 hasta cinco

                Test_i = Datos[i * tamFold, i * tamFold + tamFold]
                Train_i = Datos - Test_i

                particiones ← (Train_i, Test_i)
        Fin Deste

        Devolver Particiones
fin para cada
```

4. Algoritmos empleados al problema.

4.1. Algoritmos comunes

Representación y explicación de los principales algoritmos comunes (mutación...) así como su estructura en pseudocódigo.

4.2. Generador de soluciones aleatorias

En el ámbito de nuestro problema APC, la representación de una solución corresponde a un vector de pesos W. Para determinados algoritmos como los genéticos o la búsqueda local partimos de una solución aleatoria, es decir, un vector de pesos con componentes aleatorias generadas entre un intervalo $[0, 1]$ (de esta forma ahorramos la normalización). El pseudocódigo sería el siguiente:

```
Desde i hasta numero atributos
        SolucionInicialGenerarAleatorio ⇐ Aleatorio (0,1)
Fin Desde

Devuelve SolucionInicial
```

De esta forma obtenemos un cromosoma aleatorio ya normalizado.

4.3. Normalización de datos.

Dados unos conjuntos de datos a utilizar como base de pruebas para nuestro problema (wdbc, sonar...), para no priorizar unos datos sobre otros es necesario la normalización de los datos en un intervalo $[0, 1]$. Para ello, empleo un script en R, de forma que leemos los ficheros .arff, y utilizamos la función normalizarDatos para guardarlos en un .csv ya normalizados, de forma que dado un valor x_j perteneciente a un atributo j del ejemplo x, y sabiendo que el dominio del atributo j es $[Min_j, Max_j]$, el valor normalizado de x_j es:

$$x_j^N = \frac{x_j - Min_j}{Max_j - Min_j}$$

4.4. Función de evaluación

Implementación de la función de evaluación 1-NN explicada en la descripción anteriormente.
El pseudocódigo es el siguiente:

```
1-NN(train, test, pesos)

//Recorremos el conjunto test:
Para todo a_i ∈ Test
        //inicializamos las distancias minimas al enemigo
        // y al amigo a infinito
        d_min = ∞
```

```
        //Ahora recorremos el conjunto train
        // calculando las distancias, buscando el min

        Para todo b_i ∈ Train
                Si a_i ≠ b_i //leave one out
                        Si (calculaDistancia(a_i, b_i) < d_min^enemigo)
                                d_min = calculaDistancia(a_i, b_i, pesos)

                        Fin Si
        Fin para todo




Fin para todo

tasa = 100*etiquetas bien clasificadas / numero total de etiquetas
//Ahora calculamos la tasa de clasificacion,
// la tasa de reduccion y la func. objetivo:
tasa reduccion = 100 * (nº valores w_i>0,1 / nº valores totales)
funcion evaluacion = tasa clasificacion * α + (1 − α) * tasa reduccion

Devolver func. evaluacion
```

4.5. Generación de vecinos

En el esquema de generación de vecinos, necesario para la Búsqueda local, la Búsqueda local iterativa y el enfriamiento simulado, emplea un movimiento de cambio por mutación Normal, Mov(W,σ), deforma que a una característica del atributo se le suma un valor aleatorio obtenido a partir de una distribución normal de media 0 y desviación típica 0.3.

El pseudocódigo del algoritmo quedaría tal que así:

```
Generar vecino (Pesos, a_i = atributo a modificar)
        //Obtenemos un valor de la distribucion normal mencionada:
        aleatorio = Distribucion normal(media = 0, σ = 0,3)
        //modificamos el atributo a_i de los pesos:
        Pesos(a_i) = Pesos(a_i) + aleatorio

        //truncamos si es necesario para tener los valores normalizados
        //entre [0,1]:

        Si Pesos(a_i) < 0
                Pesos(a_i) = 0
        Si Pesos(a_i) > 1
                Pesos(a_i) = 1

        Devolvemos Pesos
```

De esta forma tendremos un vecino generado.

4.6. Operador cruce BLX

Para el algoritmo genético BLX es necesario la implementación de dicho cruce BLX-α, donde en nuestro caso $\alpha = 0{,}3$. Dados dos cromosomas p_1, p_2 pertenecientes a los padres, generamos dos descendientes h_1, h_2 de la siguiente forma descrita en pseudocódigo:

```
CruceBLX entre p1 y p2
        //Obtenemos los atributos maximos y minimos para cada padre:
        a^1_max, a^1_min ∈ p1
        a^2_max, a^2_min ∈ p2

        //Ahora obtenemos el maximo y el minimo de los dos padres:
        Max(a^1_max, a^2_max)
        Min(a^1_min, a^2_min)

        //Ahora generamos el intervalo I del cual generaremos
            aleatoriamente a los dos hijos:
        I = [Min(a^1_min, a^2_min)-(Max(a^1_max, a^2_max-Min(a^1_min, a^2_min)*0.3, Max(a^1_max, a^2_max)+(
            Max(a^1_max, a^2_max-Min(a^1_min, a^2_min)*0.3 ]

        //Creamos a los dos hijos h1, h2
        Para cada atributo a^i_1 ∈ h1 y a^i_2 ∈ h2
                a^i_1 = Aleatorio (I)
                a^i_2 = Aleatorio (I)
        fin para cada

        devolvemos a1 y a2
```

De esta forma para cada pareja de padres, obtenemos una pareja de hijos.

4.7. Torneo Binario

La función torneo binario realiza un torneo entre dos padres (cromosomas) aleatorios, es decir, de entre esos dos padres, elige aquel que posea una mayor tasa de clasificación, o lo que es lo mismo, el mejor de los dos.
Genera tantos padres por torneo binario como se le indique. El pseudocódigo sería el siguiente, recibiendo una población y sus tasas correspondientes:

```
TorneoBinario Poblacion, Tasas
        Desde i hasta N_padresagenerar
                //Obtenemos dos padres de forma aleatoria:
                p1 = aleatorio(Poblacion)
                p2 = aleatorio(Poblacion) distinto de p1

                //Ahora nos quedamos con el que tenga mejor tasa de los
                    dos:
                mejores padres ⇐ Max(Tasas(p1), Tasas(p2))
        Fin desde

        Devolver mejores padres
```

De esta forma obtenemos al final un número de padres deseados obtenidos mediante torneo binario, que emplearemos para un posterior cruce en los algoritmos genéticos y meméticos correspondientes.

4.8. Cálculo temperatura inicial

Para la generación del esquema de enfriamiento, una de las principales características es la temperatura inicial sobre la cuál comienza a enfriar el algoritmo.
La temperatura inicial la establecemos como la relación entre un porcentaje (μ) del coste de una solución inicial aleatoria y el negado del logaritmo neperiano de la probabilidad de aceptar una solución un μ por 1 peor que la inicial.
El pseudocódigo sería el siguiente:

```
CalculoTemperaturaInicial μ, Solucion inicial, φ, datos
        //calculamos el coste:
        C_solucion inicial = knn(datos, Solucion inicial)

        T_inicial = (μ*C_solucion inicial) / (-ln(φ))

        Devolver T_inicial
```

4.9. Mutación para ILS

Para la ILS emplearé el mismo cambio por mutación descrito anteriormente, pero esta vez realizamos el cambio sobre 'T' genes de la solución a mutar; por tanto, el pseudocódigo será el siguiente:

```
Mutacion ILS (Pesos, atributos a modificar)
        Para cada a_i en 'atributos a modificar'
                //Obtenemos un valor de la distribucion normal mencionada
                    :
                aleatorio = Distribucion normal(media = 0, σ = 0,3)
                //modificamos el atributo a_i de los pesos:
                Pesos(a_i) = Pesos(a_i) + aleatorio

                //truncamos si es necesario para tener los valores
                    normalizados
                //entre [0,1]:

                Si Pesos(a_i) < 0
                        Pesos(a_i) = 0
                Si Pesos(a_i) > 1
                        Pesos(a_i) = 1
        Fin Para Cada

        Devolvemos Pesos
```

5. Estructuras de los principales métodos de búsqueda

5.1. Búsqueda Local

La implementación de la búsqueda local consiste en emplear una técnica de búsqueda mediante explotación aplicada a nuestro problema, de forma que nos permita explotar una solución localmente, permitiendo así maximizar la tasa evaluación de una solución de forma local.

Como **criterio de parada** cuando se ejecute como algoritmo individual, se detendrá tras haber generado 20* (numero veces el número de características) vecinos, o cuando se hayan realizado 15000 evaluaciones (llamadas al 1-NN).
Además, como modificación, parametrizamos el número máximo de evaluaciones, de forma que cuando ejecutemos ILS le indiquemos un máximo de 1000 evaluaciones en cada llamada a BL
El pseudocódigo sería el siguiente:

```
Busqueda Local (Conjunto de datos: Data)
        //Primero generamos una solucion inicial de forma aleatoria
        S_inicial = GenerarSolucionAleatoria()
        //Y calculamos la tasa
        T_inicial = knn(Data, Data, S_inicial)


        //Inicializamos la primera como mejor solucion actual:
        S_mejor = S_inicial
        //Y guardamos la mejor tasa como esta:
        T_mejor = T_inicial

        //Llevamos cuenta del numero de evaluaciones que llevamos:
        numero evaluaciones = 1


        //Empezamos a generar vecindario con la solucion actual
        // hasta que se cumplan los criterios de parada:

        Mientras (numero evaluaciones < MAX EVALUACIONES) y (vecinos
            generados < 20*(Numero genes cromosoma)

                //vamos generando el vecindario:
                Si hay_que_generar_nuevo_vecindario
                        //rellenamos una cola de indices aleatorios desde
                             con los genes a mutar
                        vecinos por generar = Shuffle(Indices de genes)
                        hay_que_generar_nuevo_vecindario = false
                Sino
                        //Generamos un vecino correspondiente sacando
                        // un elemento de la cola anterior para
                        //Indicar que gen mutar:
                        nuevo vecino = generarVecino(S_mejor, vecinos por
                            generar)
```

```
                    vecinos generados +1

                    Si (Tasa(nuevo vecino) > T_mejor)
                            S_mejor = nuevo vecino
                            T_mejor = knn(Data, Data, nuevo vecino)

                            numero evaluaciones+1

                            //Indicamos que genere un nuevo
                                vecindario
                            hay_que_generar_nuevo_vecindario = true
                    Fin Si

            Fin Sino

            //Hemos recorrido un vecindario sin mejora
            Si estaVacia(vecinos por generar)
                    hay_que_generar_nuevo_vecindario = true
            Fin Si

    Fin Mientras

    Devolver S_mejor
```

5.2. Enfriamiento Simulado

El algoritmo de enfriamiento simulado empleará como temperatura inicial el procedimiento descrito en el apartado anterior, además, seguirá el siguiente esquema de enfriamiento:

5.2.1. Esquema de enfriamiento

Para la realización del algoritmo empleamos un esquema de enfriamiento de Cauchy modificado, es decir, una modificación de las dos ecuaciones diferenciales de Cauchy. El esquema tradicional consiste en que sea k el enfriamiento actual, tenemos que:

$$T_k = \frac{T_0}{k+1}$$

En nuestro caso, incluiremos un parámetro β asociado a la temperatura actual, de forma que la temperatura en una iteración posterior sea:

$$T_k + 1 = \frac{T_k}{1 + \beta * T_k}$$

Donde el parámetro β es la relación entre la diferencia de las cotas de temperatura entre el producto de esas cotas y el nº de enfriamientos a realizar:

$$\beta = \frac{T_0 - T_f}{M * T_0 * T_k}$$

5.2.2. Algoritmo ES

Una vez descritos tanto la generación de la temperatura inicial, así como del esquema de enfriamiento a realizar, es pseudocódigo del algoritmo será el siguiente:

```
EnfriamientoSimulado (Datos)
        //En primer lugar generamos una solucion de forma
        //aleatoria (pesos intervalo [0,1]):
        S_inicial = generaSolucionAleatoria()
        S_actual = S_inicial
        S_mejor = S_actual

        //Ahora calculamos la temperatura inicial y la final:
        T_0 = T_k = CalculoTemperaturaInicial(μ=0.3, S_inicial, φ=0.3, Datos)
        T_f = 0.001
        //Definimos el num. max. de vecinos a generar:
        maximos vecinos = 10 * n |n = tamaño del problema
        //Definimos el num. max. de enfriamientos (M):
        M = 15000/maximosvecinos

        //Comenzamos a realizar los enfriamientos, hasta alcanzar el max.
        // de endriamientos o hasta que no se produzcan mas exitos:
        Mientras (enfriamientos < M) y (exitos \ne 0)
                vecinos = exitos = 0

                Mientras (vecinos < maximos vecinos) y (exitos < maximos
                    exitos)
                     vecino = generarVecino(solucion Actual)

                     Si (C(vecino) − C(S_actual) > 0) o
                         (U(0,1) ≤ e^{−(C(vecino)−C(S_actual))/(enfriamientos*T_k)})

                             S_actual = vecino
                             exitos \leftarrow exitos +1

                             Si (C(S_actual) > C(S_mejor))
                                     S_mejor = S_actual
                             Fin Si
                     Fin Si
                Fin Mientras
                //Enfriamos:
                β = (T_0 − T_f)/(M*T_0*T_k)
                T_k + 1 = T_k/(1+β*T_k)
                enfriamientos ← enfriamientos+1

        Fin Mientras

        Devolver S_mejor
```

5.3. Búsqueda local reiterada

La realización del algoritmo de búsqueda reiterada se basa en generar una solución aleatoria y aplicar Búsquedas locales reiteradas, de forma que obtengamos una solución optimizada.
Una vez obtenida dicha solución estudiamos si es mejor que la encontrada hasta el momento y realizamos una mutación siguiendo el esquema $Mov(W, \sigma)$ sobre la mejor de las dos, de forma que volvamos a aplicar la búsqueda local sobre dicha solución.

Respecto al proceso de mutación, realizaremos un cambio brusco, de forma que aplicamos la mutación sobre un conjunto de t características escogidas aleatoriamente en la solución.
Para la llamada a la búsqueda local, le indicaremos que realice un máximo de 1000 iteraciones por búsqueda local, de esta forma, el máximo de iteraciones a realizar por nuestro algoritmo ILS serán 15, de forma que si en cada iteración realizamos una búsqueda local con 1000 llamadas a la función 0bjetivo, en total realizaremos 15000 llamadas a la función objetivo.

5.3.1. algoritmo ILS

Siguiendo el proceso y esquema de mutación descrito anteriormente el pseudocódigo será el siguiente:

```
ILS (Datos)
        //En primer lugar generamos una solucion de forma
        //aleatoria (pesos intervalo [0,1]):
        S_inicial = generaSolucionAleatoria()
        S_actual = BL(S_inicial)
        S_amutar = S_actual
        S_mejor = S_actual

        //Al haber hecho una BL adelantada, iteramos hasta 14:
        Desde i=1 hasta 14
                //mutamos un conjunto de t atributos aleatorios
                S_mutada = mutacionILS(S_amutar)

                //Aplicamos BL:
                S_actual = BL(S_mutada)

                //Si la solucion mejora, actualizamos:
                If(C(S_mutada) > C(S_actual))
                        S_actual = S_mutada
                        S_amutar = S_actual
                Fin If
        Fin Desde

        Devolver \text{Mejor solucion encontrada}
```

5.4. Evolución diferencial

Se trata de un modelo evolutivo para optimización con parámetros reales que enfatiza en la mutación y en el operador de cruce a posteriori.
Para al obtención del vector de mutación emplearemos dos modelos de generación:

- **DE/Rand/1**: Para la mutación, en este caso obtenemos tres padres (tres individuos de la generación actual) escogidos aleatoriamente, de forma que siendo r1, r2 y r3 dichos individuos, G el índice de la generación actual y F un factor (porcentaje) que multiplica la diferencia entre dos padres: calculamos dicho vector de la siguiente forma:
$$V_{i,g} = X_{r1,G} + F * (X_{r2,G} - X_{r3,G})$$
- **DE/current-to-best**: Para calcular el vector de mutación esta vez obtenemos dos padres (aleatoriamente también), pero además ahora incrementamos el gen teniendo en cuenta la mejor solución, de forma que lo calculamos así:
$$V_{i,g} = X_{i,G} + F * (X_{mejor,G} - X_{i,G}) + F * (X_{r1,G} - X_{r2,G})$$

Cuando terminamos de actualizar el vector de mutaciones, comprobamos que los genes se encuentran normalizados en el intervalo [0,1], de no ser así, los normalizamos.

El pseudocódigo resultante de la evolución diferencial para DE/Rand/1 sería el siguiente:

```
Evolucion diferencial Rand (Datos)
        //En primer lugar generamos una poblacion de 50 individuos
        // de forma aleatoria
        Desde i = 0 hasta i < 50
                Poblacion actual <- generaSolucionAleatoria()
        Fin Desde

        //Ahora realizamos una evaluacion de la poblacion:
        Para cada P_i ∈ poblacion actual
                tasas actuales ⇐ knn(Datos, P_i)
        Fin para cada

        //Como condicion de parada tendremos 15000 evaluaciones:
        Mientras (num. evaluaciones < 15000)
                //Recorremos la poblacion actual:
                Desde i=1 hasta 50
                        //Elegimos tres individuos de forma aleatoria de
                            la
                        //poblacion actual:
                        r1 = aleatorio(poblacion actual)
                        r2 = aleatorio(poblacion actual) | r2 ≠ r1
                        r3 = aleatorio(poblacion actual)  r3 ≠ r2 ∧ r3 ≠ r1

                        //Recorremos los genes del individuo I_i
                        Para cada Gen Xj en I_i
                                //mutamos teniendo segun DE/Rand/1
```

```
                        Si(aleatorio <= PROB.CRUCE=0.5 ||
                            aleatorio == i)
                                mutado ← X_{r1,j} + F * (X_{r2,j} − X_{r3,j})
                        sino
                                mutado ← X_j
                Fin para cada

                //Lo evaluamos:
                coste_mutacion = knn(mutacion)

                //normalizamos por si nos hemos salido del
                    intervalo:
                Normalizar(mutado)

                Si coste_mutacion > tasas actuales(I_i)
                        //Lo aniadimos a una nueva poblacion:
                        nueva poblacion ← mutado
                        tasas nueva poblacion ← coste_mutacion
                Sino
                        nueva poblacion ← I_i
                        tasas nueva poblacion ← tasas(I_{i})
                Fin Si
        Fin Desde

        //Actualizamos la poblacion:
        poblacion actual = nueva poblacion

Fin Mientras



Devolver \text{Mejor solucion encontrada}
```

Como podemos observar, el tamaño empleado para la población es de 50 individuos. El pseudo código de la evolución diferencial siguiendo una mutación 'DE/current-to-best' sería el siguiente:

```
Evolucion diferencial DE/current-to-best (Datos)
        //En primer lugar generamos una poblacion de 50 individuos
        // de forma aleatoria
        Desde i = 0 hasta i < 50
                Poblacion actual <- generaSolucionAleatoria()
        Fin Desde

        //Ahora realizamos una evaluacion de la poblacion:
        Para cada P_i ∈ poblacion actual
                tasas actuales ⇐ knn(Datos, P_i)
        Fin para cada

        //Como condicion de parada tendremos 15000 evaluaciones:
        Mientras (num. evaluaciones < 15000)
                //Recorremos la poblacion actual:
                Desde i=1 hasta 50
```

```
                    //Elegimos dos individuos de forma aleatoria de
                        la
                    //poblacion actual:
                    r1 = aleatorio(poblacion actual)
                    r2 = aleatorio(poblacion actual) | r2 ≠ r1


                    //Recorremos los genes del individuo I_i
                    Para cada Gen Xj en I_i
                            //mutamos teniendo segun DE/current-to-
                                best
                            Si(aleatorio <= PROB.CRUCE=0.5 ||
                                aleatorio == i)
                                    mutado ←
                                        X_j + F * (X_{mejor,j} - X_j) + F * (X_{r1,j} - X_{r2,j})

                            sino
                                    mutado ← X_j


                    Fin para cada

                    //Lo evaluamos:
                    coste_mutacion = knn(mutacion)

                    //normalizamos por si nos hemos salido del
                        intervalo:
                    Normalizar(mutado)

                    Si coste_mutacion > tasas actuales(I_i)
                            //Lo aniadimos a una nueva poblacion:
                            nueva poblacion ← mutado
                            tasas nueva poblacion ← coste_mutacion
                    Sino
                            nueva poblacion ← I_i
                            tasas nueva poblacion ← tasas(I_{i})
                    Fin Si
            Fin Desde

            //Actualizamos la poblacion:
            poblacion actual = nueva poblacion

    Fin Mientras

    Devolver \text{Mejor solucion encontrada}
```

5.5. AGG BLX

El AGG BLX se trata de un algoritmo que implementa un esquema de evolución generacional elitista, mediante un esquema de reemplazamiento generacional, es decir, la nueva generación reemplaza en cada iteración completamente a la generación actual.
Como operador de selección se implementará el torneo binario ya descrito anteriormente,

para la elección de los mejores padres a cruzar.

El operador de cruce es el BLX, descrito anteriormente, para generar en cada cruce a partir de dos padres dos descendientes. La probabilidad de cruce será de 0.7.

Para la mutación, se empleará la función generarVecino descrita anteriormente, bajo una probabilidad de mutación en población de un 0.001.

Por último el criterio de parada del algoritmo es no superar las 15000 evaluaciones de la función objetivo (1-NN).

5.6. AM-(10, 0.1mej)

Consiste en que dado el algoritmo AGG-CA mencionado, cada 10 generaciones aplicamos una búsqueda local sobre los 0.1*N mejores cromosomas de la población.
El pseudocódigo sería el siguiente:

```
AM-(10, 0.1mej) (Data)
//Creamos una poblacion de 10 individuos aleatoria:
Desde i = 0 hasta i < 10
        Poblacion actual <- generaSolucionAleatoria()
Fin Desde

Para cada P_i ∈ poblacion actual
        tasas actuales ⇐ knn(Data, Data, P_i)
Fin para cada

//empezamos generacion tras generacion hasta cumplir
//El criterio de parada:
Mientras (num. evaluaciones < 15000)
        //Calculamos el mejor padre de la poblacion actual:
        Padre_mejor = Max(tasas poblacion actual)

        //Obtenemos los padres (parejas) a cruzar,
        //con una probabilidad de cruce del 70%
        parejas a cruzar = 0.7 * 10

        //Llamamos al torneo binario, obteniendo los padres deseados,
        // en este caso, el doble del tamanio de la poblacion actual,
            puesto que
        // el operador de cruce CA devuelve un unico descendiente.
        padres a cruzar = torneoBinario(poblacion actual)
        //Realizamos los cruces, obteniendo dos hijos por cruce
        Desde i = 0 hasta i < parejas a cruzar
                nueva poblacion ⇐ cruceAritmetico(pareja_i, pareja_{i+1})
        Fin Desde

        //Aniadimos los padres que no se han cruzado hasta llegar
        // a una nueva poblacion de 10:
        Mientras Tam(nueva poblacion) < 10
```

```
                nueva poblacion ⇐ padres no se han cruzado
        Fin Mientras

        //Calculamos el numero de individuos a mutar, con
        // una probabilidad de 0.001:
        individuos a mutar = 0.001 * 10

        //Mutamos de forma aleatoria los n individuos a mutar:
        Desde i = 0 hasta i < individuos a mutar
                nueva poblacion_aleatorio = generarVecino (nueva poblacion_aleatorio)
        Fin Desde

        //Calculamos las tasas de la nueva poblacion
        Para cada P_i ∈ nueva poblacion
                nuevas tasas ⇐ knn(Data, Data, P_i)
        Fin para cada

        //Si hemos perdido al mejor padre lo cambiamos por
        //el peor de la poblacion.
        Si (hemos perdido al mejor padre)
                Min(tasas actual) = mejor padre
        Fin Si

        //La nueva poblacion es ahora la actual:
        poblacion actual = nueva poblacion
        tasas actual = nuevas tasas


        //Si ya llevamos 10 generaciones aplicamos la busqueda local:
        Si (generaciones es multiplo de 10)
                Para cada c_i \in 0.1*poblacion actual
                        c_i = BL (Data, c_1)
                        Tasas actual (c_i) = Tasa(c_i)
                Fin para cada
        Fin Si


Fin Mientras


Devolver mejor individuo poblacion actual
```

6. Algoritmo de comparación

En el dominio del problema a tratar, la implementación de un método de comparación se realiza mediante el método RELIEF.
Se trata de una solución de tipo Greedy.
Dicho método partimos de un vector de pesos inicializado a 0 inicialmente y en cada paso modificamos dichos pesos en función, para cada uno de los ejemplos del conjunto de entrenamiento, de su enemigo más cercano y su amigo más cercano.

Definimos el enemigo más cercano como aquel ejemplo con clase diferente que se encuentra a menor distancia euclídea; y el amigo más cercano de la misma forma pero que tiene la misma clase que él. NUNCA COMPARAMOS LA DISTANCIA ENTRE ÉL CONSIGO MISMO (Leave one out), sino obtendríamos siempre una tasa cercana al 100 %
Tras el cálculo de W, y si es necesario, se normaliza W.
El pseudocódigo sería el siguiente:

```
RELIEF(train, test)
        //Inicializamos los pesos a 0:
        Para todo w_i ∈ w
                w_i = 0
        Fin para todo

        //Recorremos el conjunto test:
        Para todo a_i ∈ Test
                //inicializamos las distancias minimas al enemigo
                // y al amigo a infinito
                d_min^amigo = ínf
                d_min^enemigo = ínf

                //Ahora recorremos el conjunto train
                // calculando las distancias, buscando el min amigo
                // y enemigo:

                Para todo b_i ∈ Train
                        Si a_i ≠ b_i //leave one out
                                Si etiqueta(b_i ≠ a_i) && (calculaDistancia(a_i
                                    , b_i) < d_min^enemigo)
                                        d_min^enemigo = calculaDistancia(a_i, b_i)

                                Sino Si(calculaDistancia(a_i, b_i) < d_min^enemigo)
                                        d_min^amigo = calculaDistancia(a_i, b_i)
                        Fin para todo
                Fin para todo

                //Ahora actualizamos los pesos
                w = w + |a_i − a^enemigo| − |a_i − a_amigo|

        Fin para todo


        Devolver W
```

7. Procedimiento para la realización de la práctica

Para la realización de la práctica no he hecho uso de ningúna implementación ni código externos, todo ha sido desarrollado por mi mismo.

7.1. Manual de usuario

Si se desea realizar una réplica de las pruebas aquí descritas serán necesarios los siguientes pasos:

1. En primer lugar, será necesario ejecutar el script en R "practica1.R", cuya función es leer los archivos que se encuentran en formato '.arff', normalizarlos, y escribirlos en ficheros '.csv' (almacenados en la carpeta "datos") para facilitar de esta forma la futura lectura.

2. Una vez tenemos los datos normalizados, ejecutar el makefile incluido, de esta forma obtendremos un ejecutable "practica1".

3. Por último lanzar dicho programa, indicando como argumento el nombre del conjunto de datos que se desea utilizar para la realización de los algoritmos, por ejemplo: "./practica1 cancer", de esta forma utilizaríamos la base de datos de cáncer de mama.

4. Tras esto se ejecutarán TODOS los algoritmos indicando tanto la tasa obtenida para cada uno, como los tiempos, la tasa media de todas las particiones y su correspondiente tiempo medio.

8. Experimentos y análisis de resultados

Para la realización de los experimento utilizaremos 3 conjuntos de datos distintos, uno para predecir si se posee cáncer de mama, otro para detectar posible correo 'spam' y un tercero, para clasificar datos de un sonar.

Cómo método de validación utilizaremos el método 5fold-cross validation (cv) explicado anteriormente.
Cómo medida de validación, utilizaremos la agregación entre la tasa de acierto del clasificador 1-NN sobre el conjunto de prueba y la tasa de reducción con un porcentaje de 0.5 para cada uno. El resultado final será la media de los 5 valores obtenidos, uno para cada ejecución del algoritmo.

También emplearemos cómo método de comparación el tiempo de ejecución empleado por cada uno. Las semillas utilizadas para la realización de todas las pruebas son las siguientes:

```
default_random_engine generator (5); //para los números aleatorios
normal_distribution<double> distribution (0.0,0.3); //para la distribución normal
```

Los tiempos obtenidos y las tasas para cada algoritmo de forma individual son las siguientes:

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

	Sonar				Wdbc				Spambase			
	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T
Partición 1	75,6098	0	37,8049	0,00049	95,5752	0	47,7876	0,003979	77,1739	0	38,587	0,005581
Partición 2	78,0488	0	39,0244	0,000469	95,5752	0	47,7876	0,002606	75	0	37,5	0,003015
Partición 3	90,2439	0	45,122	0,000468	97,3451	0	48,6726	0,002375	79,3478	0	39,6739	0,002652
Partición 4	97,561	0	48,7805	0,000468	96,4602	0	48,2301	0,002004	77,1739	0	38,587	0,002143
Partición 5	78,0488	0	39,0244	0,000467	92,9204	0	46,4602	0,001907	75	0	37,5	0,002143
MEDIA	83,90246	0	41,95124	0,0004724	95,57522	0	47,78762	0,0025742	76,73912	0	38,36958	0,0031068

Tabla 2: Resultados obtenidos por el algoritmo RELIEF en el problema del APC

	Sonar				Wdbc				Spambase			
	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T
Partición 1	82,9268	13,3333	48,1301	0,007617	96,4602	6,66667	51,5634	0,026403	83,6957	49,1228	66,4092	0,035372
Partición 2	75,6098	15	45,3049	0,009754	92,9204	13,3333	53,1268	0,032633	77,1739	28,0702	52,622	0,03582
Partición 3	87,8049	15	51,4024	0,008191	97,3451	13,3333	55,3392	0,03197	85,8696	33,3333	59,6014	0,047381
Partición 4	97,561	16,6667	57,1138	0,011035	95,5752	20	57,7876	0,02703	78,2609	29,8246	54,0427	0,029635
Partición 5	82,9268	13,3333	48,1301	0,006801	94,6903	6,66667	50,6785	0,026813	79,3478	31,5789	55,4634	0,035591
MEDIA	85,36586	14,66666	50,01626	0,0086796	95,39824	11,999988	53,6991	0,0289698	80,86958	34,38596	57,62774	0,0367598

Tabla 3: Resultados obtenidos por el algoritmo SA en el problema del APC

	Sonar				Wdbc				Spambase			
	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T
Partición 1	73,1707	28,3333	50,752	2,9697	92,0354	36,6667	64,351	11,2243	79,3478	26,3158	52,8318	12,3027
Partición 2	82,9268	31,6667	57,2967	2,84238	94,6903	40	67,3451	10,896	78,2609	35,0877	56,6743	12,7241
Partición 3	87,8049	36,6667	62,2358	2,72996	92,0354	40	66,0177	10,9717	88,0435	28,0702	58,0568	12,9502
Partición 4	87,8049	35	61,4024	2,7919	96,4602	46,6667	71,5634	10,7337	82,6087	29,8246	56,2166	13,0629
Partición 5	85,3659	36,6667	61,0163	2,72719	92,0354	40	66,0177	10,3715	75	31,5789	53,2895	12,5308
MEDIA	83,41464	33,66668	58,54064	2,812226	93,45134	40,66668	67,05898	10,83944	80,65218	30,17544	55,4138	12,71414

Tabla 4: Resultados obtenidos por el algoritmo ILS en el problema del APC

	Sonar				Wdbc				Spambase			
	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T
Partición 1	78,0488	20	49,0244	28,1687	93,8053	16,6667	55,236	71,1936	78,2609	24,5614	51,4111	135,797
Partición 2	82,9268	21,6667	52,2967	27,6909	93,8053	26,6667	60,236	70,3562	78,2609	19,2982	48,7796	139,505
Partición 3	90,2439	18,3333	54,2886	28,1743	97,3451	13,3333	55,3392	68,6134	78,2609	21,0526	49,6568	136,002
Partición 4	87,8049	21,6667	54,7358	29,6936	92,9204	30	61,4602	69,0423	80,4348	24,5614	52,4981	142,93
Partición 5	68,2927	23,3333	45,813	29,4333	94,6903	16,6667	55,6785	68,5227	83,6957	38,5965	61,1461	141,203
MEDIA	81,46342	21	51,2317	28,63216	94,51328	20,66668	57,58998	69,54564	79,78264	25,61402	52,69834	139,0874

Tabla 5: Resultados obtenidos por el algoritmo DE RANDOM en el problema del APC

	Sonar				Wdbc				Spambase			
	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T
Partición 1	73,1707	90	81,5854	19,587	92,9204	93,3333	93,1268	72,1019	90,2174	96,4912	93,3543	106,959
Partición 2	87,8049	88,3333	88,0691	18,9681	94,6903	93,3333	94,0118	72,16	86,9565	94,7368	90,8467	97,8624
Partición 3	82,9268	88,3333	85,6301	19,134	93,8053	93,3333	93,5693	71,4551	82,6087	94,7368	88,6728	107,369
Partición 4	85,3659	88,3333	86,8496	19,6857	92,9204	93,3333	93,1268	74,4091	79,3478	94,7368	87,0423	107,251
Partición 5	80,4878	88,3333	84,4106	19,4013	93,8053	93,3333	93,5693	71,7964	82,6087	94,7368	88,6728	94,889
MEDIA	81,95122	88,66664	85,30896	19,35522	93,62834	93,3333	93,4808	72,3845	84,34782	95,08768	89,71778	102,86608

Tabla 6: Resultados obtenidos por el algoritmo DE CURRENT TO BEST en el problema del APC

	Sonar				Wdbc				Spambase			
	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T
Partición 1	75,6098	63,3333	69,4715	21,0931	92,0354	70	81,0177	79,8989	83,6957	63,1579	73,4268	96,1655
Partición 2	85,3659	63,3333	74,3496	19,4221	92,9204	63,3333	78,1268	75,2726	82,6087	64,9123	73,7605	95,3744
Partición 3	90,2439	48,3333	69,2886	22,0457	96,4602	80	88,2301	76,76	81,5217	61,4035	71,4626	94,7781
Partición 4	87,8049	61,6667	74,7358	19,9012	97,3451	66,6667	82,0059	82,9237	72,8261	64,9123	68,8692	93,0355
Partición 5	80,4878	66,6667	73,5772	18,9014	85,8407	80	82,9204	71,6756	81,5217	59,6491	70,5854	92,037
MEDIA	83,90246	60,66666	72,28454	20,2727	92,92036	72	82,46018	77,30616	80,43478	62,80702	71,6209	94,2781

Tabla 7: Resultados obtenidos por el algoritmo BL en el problema del APC

	Sonar				Wdbc				Spambase			
	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T
Partición 1	78,0488	18,3333	48,1911	2,15189	93,8053	20	56,9027	4,29518	79,3478	19,2982	49,323	9,08716
Partición 2	80,4878	18,3333	49,4106	2,10916	92,9204	16,6667	54,7935	4,14705	76,087	19,2982	47,6926	8,93329
Partición 3	78,0488	18,3333	48,1911	2,11307	95,5752	23,3333	59,4543	4,19296	78,2609	17,5439	47,9024	9,0591
Partición 4	90,2439	16,6667	53,4553	2,23855	95,5752	16,6667	56,1209	4,4525	79,3478	24,5614	51,9546	9,23437
Partición 5	75,6098	31,6667	53,6382	2,10487	94,6903	23,3333	59,0118	4,45159	76,087	22,807	49,447	9,22552
MEDIA	80,48782	20,66666	50,57726	2,143508	94,51328	20	57,25664	4,307856	77,8261	20,70174	49,26392	9,107888

Tabla 8: Resultados obtenidos por el algoritmo AGG-BLX en el problema del APC

	Sonar				Wdbc				Spambase			
	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T
Partición 1	68,2927	38,3333	53,313	25,1605	91,1504	50	70,5752	103,422	79,3478	35,0877	57,2178	115,508
Partición 2	82,9268	38,3333	60,6301	24,9759	91,1504	43,3333	67,2419	103,197	78,2609	36,8421	57,5515	115,755
Partición 3	75,6098	40	57,8049	24,8044	91,1504	46,6667	68,9086	103,492	75	38,5965	56,7982	114,817
Partición 4	95,122	33,3333	64,2276	25,0708	92,9204	46,6667	69,7935	102,967	81,5217	35,0877	58,3047	115
Partición 5	87,8049	38,3333	63,0691	24,7738	92,0354	46,6667	69,351	104,408	77,1739	36,8421	57,008	114,847
MEDIA	81,95124	37,66664	59,80894	24,95708	91,6814	46,66668	69,17404	103,4972	78,26086	36,49122	57,37604	115,1854

Tabla 9: Resultados obtenidos por el algoritmo AM (10,01) MEJ en el problema del APC

	Sonar				Wdbc				Spambase			
	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T
Partición 1	75,6098	26,6667	51,1382	27,4915	94,6903	3,33333	49,0118	111,172	83,6957	21,0526	52,3741	136,015
Partición 2	80,4878	16,6667	48,5772	27,755	92,9204	23,3333	58,1268	111,663	78,2609	24,5614	51,4111	129,982
Partición 3	82,9268	30	56,4634	27,3229	94,6903	20	57,3451	111,772	76,087	19,2982	47,6926	128,604
Partición 4	87,8049	26,6667	57,2358	27,7592	94,6903	13,3333	54,0118	110,089	76,087	17,5439	46,8154	128,605
Partición 5	80,4878	18,3333	49,4106	27,2449	95,5752	16,6667	56,1209	112,304	73,913	22,807	48,36	129,634
MEDIA	81,46342	23,66668	52,56504	27,5147	94,5133	15,333326	54,92328	111,4	77,60872	21,05262	49,33064	130,568

Ahora pasemos a realizar una tabla comparativa con las medias, tanto de tiempo como de tasas obtenidas para cada uno de los algoritmos para poder realizar así un análisis de forma global:

Resultados promedio en el problema del APC

	Sonar				Wdbc				Spambase			
	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T
1NN	83,90246	0	41,95124	0,0004724	95,57522	0	47,78762	0,0025742	76,73912	0	38,36958	0,0031068
Relief	85,36586	14,66666	50,01626	0,0086796	95,39824	11,999988	53,6991	0,0289698	80,86958	34,38596	57,62774	0,0367598
ES	83,41464	33,66668	58,54064	2,812226	93,45134	40,66668	67,05898	10,83944	80,65218	30,17544	55,4138	12,71414
ILS	81,46342	21	51,2317	28,63216	94,51328	20,66668	57,58998	69,54564	79,78264	25,61402	52,69834	139,0874
DE_rand	81,95122	88,66664	85,30896	19,35522	93,62834	93,3333	93,4808	72,3845	84,34782	95,08768	89,71778	102,86608
DE_current_to_best	83,90246	60,66666	72,28454	20,2727	92,92036	72	82,46018	77,30616	80,43478	62,80702	71,6209	94,2781
BL	80,48782	20,66666	50,57726	2,143508	94,51328	20	57,25664	4,307856	77,8261	20,70174	49,26392	9,107888
AGG-BLX	81,95124	37,66664	59,80894	24,95708	91,6814	46,66668	69,17404	103,4972	78,26086	36,49122	57,37604	115,1854
AM (10,0.1mej)	81,46342	23,66668	52,56504	27,5147	94,5133	15,333326	54,92328	111,4	77,60872	21,05262	49,33064	130,568

Realicemos ahora un análisis de los datos obtenidos e intentemos razonar el porqué de dichos datos.

Si analizamos a primera vista las tasas obtenidas según los tres distintos conjuntos, lo primero que nos damos cuenta es de la gran importancia que tiene el mero hecho de que el conjunto de datos que usamos sea o no representativo
Por ejemplo, nos damos cuenta de que el conjunto de datos de cáncer (Wdbc) es el que posee los datos más representativos de los tres, lo cuál facilita el aprendizaje. Es por eso que es en este en el que se obtienen las mayores tasas de clasificación, seguidas por las de Sonar y por último las de Spam.

8.1. Clasificación, Reducción y función objetivo

Si observamos únicamente los valores obtenidos respecto a función objetivo, son los algoritmos de evolución diferencial los claros vencedores, elementalmente porque producen una alta tasa de reducción, lo cuál agrega a la función objetivo.
Si analizamos las tasas de reducción generadas por todos los algoritmos analizados, observamos que el claro vencedor es el algoritmo de evolución diferencial que emplea el esquema DE/Rand, superando casi en el doble al segundo mejor algoritmo.
Esto se debe básicamente a la aleatoriedad a la hora de la obtención del vector de mutación.
Respecto a tasas de clasificación, como en la práctica anterior, los valores son muy similares, el vencedor en este caso el la evolución diferencial con modelo DE/current-to-best.

8.2. BL VS ILS

Comparemos estos dos métodos de búsqueda. El segundo método consiste básicamente en aplicar una poca de exploración al algoritmo básico de BL, mediante el empleo de componentes aleatorias y mutaciones sobre la solución. Aunque el grado de exploración no es comparable al resto de algoritmos como DE o AGG, si analizamos la función objetivo tanto de BL como de ILS, observamos que de media tanto ILS como BL obtienen los mismos valores en tasas de clasificación, reducción y función objetivo.
Esto se debe en parte a lo descrito anteriormente, ambos tienen un máximo de 15000 llamadas a la función de evaluación.
La principal diferencia entre los dos es el proceso de diversificación que introduce la ILS, cosa del que BL carece, por eso en parte, las tasas son muy ligeramente superiores en ILS, en cambio el tiempo de ejecución se ve afectado en esta última.

8.3. DE/rand VS DE/current-to-best

Veamos ahora más detenidamente a los claros vencedores, los algoritmos de evolución diferencial. Observando los resultados obtenidos, el claro vencedor es DE/rand. Pero si analizamos con detalle, esto es porque básicamente por que es capaz de obtener una tasa de reducción mucho superior a la obtenida por Rand; debido a la componente de aleatoriedad, sobre el esquema de cruce, lo cuál produce una mayor reducción a la hora de clasificar los datos.

Aunque los dos clasifican con un mismo error los datos, a la hora de elegir un modelo de aprendizaje, se antepone la simplicidad del modelo, es decir, dado dos soluciones iguales (o casi), aquella que sea la más simple siempre es la mejor. Por tanto, DE/Rand sería el modelo a escoger.

Aún así, si deseamos darle más importancia a la simplicidad, lo lógico sería que a la hora de calcular la agregación, aumentaremos el parámetro (porcentaje) de importancia a la tasa de reducción, en vez de dejarla en 0.5.

8.4. Diversidad VS Convergencia

Si comparamos evolutivos y enfriamiento simulado con la búsqueda local reiterada y la búsqueda local estrictamente observando la tabla, podríamos pensar que ambos son iguales en términos de tasa de clasificación, puesto que tanto los tiempos como las tasas obtenidas en los tres conjuntos no difieren apenas en unas décimas o incluso centésimas entre ellos.

Pero si analizamos la funcionalidad de ambos, podemos deducir que la principal diferencia entre los dos es la diversidad que producen.
En ILS así como BL, realizamos en el primero un proceso de diversificación modesto mediante mutación, mientras que en este último unicamente divergencia.

En cambio en los DE, se le da mayor énfasis a el cruce entre individuos de la población, y en ES a la reducción de la temperatura, ampliando así el espacio de búsqueda.
En cualquier caso, una hibridación como la dada en el algoritmo memético es una buena opción para obtener un equilibrio entre diversidad y convergencia.

9. Conclusión

Como hemos visto en el análisis, la elección de un buen algoritmo de búsqueda es importantísima a la hora de la resolución de un problema, y se encuentra muy ligada a el ámbito de el mismo.

DE es un claro vencedor, debido principalmente al énfasis dado a la reducción de parámetros. Como anotación culmen, hemos de enfatizar la importancia de encontrar un equilibrio entre diversificación y convergencia.