METAHEURÍSTICAS
Grado en Ingeniería Informática
Universidad de Granada

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

Carlos Manuel Sequí Sánchez
DNI: 20 48 69 26
Grupo de prácticas 2.
Horario: viernes, 17:30-19:30

2016-2017

Algoritmos considerados:

- KNN.
- RELIEF.
- Local Search.
- Simulated Annealing.
- Iterative Local Search.
- Differential Evolution: Rand
- Differential Evolution: Current to Best.

Índice

Índice de figuras

1. Descripción del problema: APC

Nota: la partición y normalización de datos han sido realizadas en el lenguaje de programación R, de tal manera que paso los datos a un fichero CSV normalizado con la columna de la clase de cada ejemplo puesta al principio. El resto de código (lectura de fichero CSV, algoritmos principales, programa principal) está hecho en C++.

El problema del aprendizaje de pesos en características (APC) es un problema que optimiza el rendimiento del clasificador KNN mediante la asignación de valores reales (pesos) a las características ponderando de tal manera la relevancia de cada una de ellas en el problema de aprendizaje.
Dichos pesos vienen dados por un vector de valores reales en el intervalo [0,1] con un número de datos igual a la cantidad de atributos de las componentes.

El problema de clasificación consiste en el aprender a clasificar en distintas clases un conjunto de datos a partir de unos datos de entrenamiento (Train). El conjunto de datos que pretendemos clasificar es el conjunto de datos Test, con el cual podremos sacar más tarde la tasa de acierto del algoritmo utilizado para aprender los pesos de las características.
El objetivo principal del problema es ajustar el conjunto de ponderaciones o pesos asociados al conjunto total de elementos, con el fin de que los clasificadores que se construyan a partir de él sean óptimos.

Como clasificador KNN usaremos el 1NN para calcular la clase de un elemento a a partir del vecino más cercano b, considerando como vecino más cercano al que tenga los valores más próximos para cada uno de los atributos.

2. Código en común

Comenzamos con la descripción del código común a todos los algoritmos considerados en el problema.
Para comenzar, podemos decir que mi código consta de una clase Algortimos.cpp donde se encuentran todos los algoritmos (valga la redundancia) que se piden y demás métodos útiles creados para la realización de la práctica.
La representación de los datos es una simple matriz (matrizDatos) la cual contiene los valores numéricos leídos de los ficheros CSV y, un vector de string (etiquetas) que contiene la clase original de cada una de las características de la matriz, también leídas desde los ficheros CSV.
A continuación describo los métodos usados en común:

2.1. Generador de soluciones aleatorias

No tengo un método en sí para la generación de soluciones aleatorias, simplemente he hecho el siguiente código cada vez que lo he necesitado.

Algorithm 1 Generar solución aleatoria

```
1: for i in cantidadAtributos do
2:     num = generarNumeroAleatorio;
3:     normalizar(num);
4:     solucionInicial.insertar(num);
5: end for
```

FIN(1)

2.2. Normalización de datos

Una vez leídos los datos de los ficheros arff, estos han de ser normalizados dentro del rango [0,1] para que no halla unos atributos con mucho más peso o importancia que otros.

Max_j = máximo valor del atributo j para todas las características.
Min_j = mínimo valor del atributo j para todas las características.
x_ij = atributo j de la característica i

Para la normalización:

$x_ij = (x_ij$ - $Min_j)$ / $(Max_j$ - $Min_j)$

2.3. Cálculo de la Distancia

Método común que se encarga de calcular la distancia Euclidea entre un vecino A y un vecino B teniendo en cuenta la dimensión del vector de pesos W.
En el código lo he desarrollado como un desenrollado de bucles con el fin de mejorar los tiempos de ejecución.

Algorithm 2 Calculo de la distancia

```
1: distancia = 0;
2: if ( thentamaño de W es impar)
3:     distancia += distancia entre A_0 y B_0;
4: end if
5: for cada elemento de A y B do
6:     distancia += distancia euclidea entre A_i y B_i;
7:     distancia += distancia euclidea entre A_{i+1} y B_{i+1};
8: end for
```

4

2.4. Generar Vecino

La generación de un nuevo vecino consiste en calcular un número aleatorio (posModificar) entre 0 y el número de genes del individuo W (siendo los genes la cantidad de elementos que tiene el vector de pesos) y dicha posición será relaculada mediante un número aleatorio generado por una distribución normal en el rango [0,1]. Para cumplir el rango, en caso de que el valor generado sobrepase las cotas, simplemente truncamos para que las cumpla.

Algorithm 3 Generar Vecino

```
1: numeroAleatorio = generarNumero;
2: normalizarNumAleatorio(rango[0,1]);
3: W[posModificar] = numAleatorio;
```

2.5. Creación de particiones

La creación de particiones ha sido realizada teniendo en cuenta la especificación del guión de la práctica, es decir, se han creado 5 pares de particiones Train-Test con el fin de realizar la 5-fold crossValidation debido a la poca cantidad de datos existentes como para realizar una validación de datos creíble y poder comparar así los diferentes algoritmos implementados. Cada partición (cada par Train-Test) tiene un 20 % de datos para el Test y un 80 % para el Train (ambos subconjuntos disjuntos). Los 5 subconjuntos Test obtenidos en las 5 particiones son disjuntos entre sí obteniendo así validaciones con distintos conjuntos de datos en cada partición.

3. Esquema de representación de soluciones:

En esta segunda práctica damos un giro a la hora de realizar la comparación entre algoritmos con las soluciones obtenidas. Esta vez no será la tasa de clasificación la que nos diga si un algoritmo X es mejor que otro algoritmo Y sino, un agregado entre la tasa de clasificación y una tasa de reducción. A continuación, explico los distintos parámetros que influyen a la hora de representar las soluciones:

1. **Tasa de clasificación:** Se define exactamente igual que en la práctica anterior, es decir, el porcentaje de etiquetas acertadas con los pesos aprendidos sobre el conjunto de datos original. Este parámetro nos da una medida del acierto que tiene el algoritmo a la hora de encontrar el vector de pesos.

$$tasa\_clas = 100 \cdot \frac{\text{nº instancias bien clasificadas de } T}{\text{nº instancias en } T},$$

2. **Tasa de reducción:** Esta vez no vamos a considerar todos los pesos. Con el fin de minimizar el número de características empleado a la hora de utilizar el clasificador final, se obviarán todas aquellas características cuyos pesos asociados se encuentren por debajo del valor 0.1, obteniendo así otra medida comparativa: ¿cuál de todos los algoritmos es el que utiliza un menor número de características y por tanto, el más simple?

$$tasa\_red = 100 \cdot \frac{\text{nº valores } w_i < 0.1}{\text{nº características}}$$

3. **Agregado:** Basándonos en ambas tasas obtenemos la **nueva función objetivo**, con la que podemos obtener una comparativa fiel de la calidad de los algoritmos teniendo en cuenta la siguiente fórmula:

$$función\_objetivo = tasa\_clas \cdot \alpha + tasa\_red \cdot (1 - \alpha)$$

   De esta manera tenemos en cuenta a ambas tasas (clasificación y reducción) y, a la hora de realizar la comparativa entre los distintos algoritmos, podremos obtener el algoritmo que mejor clasifica y además, el que menos atributos/características utiliza para ello.
   El valor de σ pondera la importancia entre el acierto y la reducción de la solución encontrada. Consideraremos σ=0.5, dándole la misma importancia a ambos criterios.
   Como es evidente, en nuestro problema el objetivo es maximixar la función objetivo, por lo que el algoritmo que mayor valor de agregado obtenga, mejor será.

4. Principales algoritmos y métodos de búsqueda

4.1. Clasificador 1-NN

El objetivo del clasificador 1NN, tal como indica su nombre, es clasificar los elementos del conjunto de datos T_1 (Train, datos de entrenamiento), con respecto a los elementos del conjunto de datos T_2 (Test, los datos de evaluación), basándose siempre en el vecino más cercano, es decir, el que presente menor distancia Euclidea para cada uno de los elementos. De tal manera, un elemento i del conjunto de datos Train tomará el valor de la clase del vecino más cercano e del conjunto de datos Test.
Tras clasificar el conjunto de datos Train al completo, calculamos los tres parámetros necesarios:

- Tasa clasificación: calculada como los aciertos predichos sobre las etiquetas reales.
- Tasa de reducción: calculada como la cantidad (en porcentaje) de atributos innecesarios para la predicción.
- Agregado: una suma de ambos ponderando cada una de las tasas por el factor α

Algorithm 4 1-NN

```
1: for i in T1 do
2:     for e in T2 do
3:         if i != e then //Leave one out
4:             distanciaVecino = calcularDistancia(i,e);
5:             if distanciaVecino <minimaDistanciaActual then
6:                 vecinoMasCercano = e;
7:                 minimaDistanciaActual = distanciaVecino;
8:             end if
9:         end if
10:    end for
11:    if clase(vecinoMasCercano) == clase(i) then
12:        cantidadElementosBienClasificados++;
13:    end if
14: end for
15: //Calculamos la tasa de clasificación
16: tasaClasificacion = 100*(cantidadElementosBienClasificados/Tamaño(T1))
17: tasaReduccion = calcularTasaReduccion();
18: agregado = (tasaClasificacion * α) + (tasaReduccion * (1-α))
        return agregado;
```

4.2. Local Search

El principal objetivo del algoritmo de búsqueda local es explotar un reducido espacio de búsqueda con el fin de obtener máximos locales (la mejor solución del entorno de la solución actual). Para ello recibe un vector inicial de pesos (W_{ini}) y a partir de este va generando nuevos vecinos (V_i) con el fin de encontrar el mejor en dicho espacio de búsqueda. Mediante la función de clasificación 1NN ya descrita comprobamos sucesivamente si cada uno de los vecinos generados se comporta mejor que la solución actual(S_{act}) mediante sus respectivas tasas de clasificacion: T_{act} y T_v, en dicho caso, la solución actual es actualizada.
La condición de parada consiste en generar en total 20*N vecinos (siendo i el vecino actual) o bien realizar 1000 evaluaciones con el clasificador 1NN.(En este problema, se alcanzará siempre antes la condición de las 1000 evaluaciones)

Algorithm 5 Local Search

```
1: inicializar(S_act = W_ini);
2: while i <20*N && i <15000 do
3:     posModificar = número aleatorio;
4:     V_i = generarVecino(S_act, posModificar);
5:     T_v = 1NN(V_i);
6:     if T_act <T_v then //comparamos tasas
7:         S_act = V_i;
8:         T_act = T_v;
9:     else if hemos generado el vecindario de V_i entero then
10:        //continuamos generando nuevos vecinos, poniendo el
11:        //contador de posiciones modificadas a 0;
12:    end if
13: end while
```

5. Algoritmo de comparación: RELIEF

Consiste en un algoritmo voraz de cálculo del vector de pesos para el APC, digamos, algoritmo de fuerza bruta. Partiendo de un vector de pesos inicializado a cero (W_{ini}), consiste en, para cada elemento T_i del conjunto de datos Train (T_1), calcular su amigo A_i y su enemigo E_i del subconjunto Test (T_2) para, en función de ellos, calcular la i-ésima posición del vector de pesos.

Algorithm 6 RELIEF

```
1: W = inicializar(W_ini);
2: for i in T_1 do
3:     for i in T_1 do
4:         if i != e then
5:             //Leave one out
6:             distanciaVecino = calcularDistancia(i,e);
7:             if clase(i) == clase(e) then
8:                 //para los amigos
9:                 if distanciaVecino <distanciaMinAmigo then
10:                    A_i = e;
11:                    distanciaMinAmigo = distanciaVecino;
12:                else
13:                    //para los enemigos
14:                    if instanciaVecino <distanciaMinEnemigo then
15:                        E_i = e;
16:                        distanciaMinEnemigo = distanciaVecino;
17:                    end if
18:                end if
19:            end if
20:        end if
21:    end for
22:    W[i] = actualizar peso teniendo en cuenta al amigo y al enemigo.
23:    //en el código he creado un método aparte para actualizar W.
24: end for
25: //Antes de devolver W lo normalizamos para que contenga valores en el
26: //intervalo [0,1].
        return W;
```

6. Enfriamiento simulado

El Enfriamiento o Recocido Simulado es un algoritmo de búsqueda por entornos con una función probabilística de aceptación de soluciones basada en Termodinámica.
Es una forma de evitar que la búsqueda local caiga en óptimos locales, de manera que, Simulated Annealing permite viajar hacia una solución peor que la actual.
Aún así, esto ha de ser controlado con el fin de no viajar a una solución peor, cuando realmente estamos en el buen camino hacia la mejor solución del entorno.
Para controlar esto utilizamos una función de probabilidad de escape hacia soluciones peores que irá disminuyendo conforme avanza el algoritmo en la búsqueda del mejor, es decir, a medida que el algoritmo avanza, la probabilidad de aceptar una solución pero que la actual será menor.
Aplicamos el criterio de diversificar al principio e intensificar al final del algoritmo.

6.1. Analogías entre el algoritmo y la termodinámica

- Estados - diferentes soluciones
- Energía - coste
- Cambio de estado - nueva solución en el entorno
- Temperatura - parámetro de control de aceptación de soluciones peores que la actual
- Estado congelado - solución final heurística

La temperatura T0 final se inicializa a un valor alto mediante la siguiente fórmula:

$$T_0 = \frac{\mu \cdot C(S_0)}{-\ln(\phi)}$$

aceptando de esta manera en varias ocasiones soluciones peores a la actual. Los parámetros ϕ y μ los pondremos a 0.3. Como hemos explicado antes, a medida que avanza la ejecución del algoritmo la temperatura va descediendo, haciendo que cada vez se acepte una menor cantidad de soluciones peores que la actual. De esta manera, cuando la temperatura se estabiliza en 0 grados, no se aceptan soluciones peores y, por tanto se procede a la explotación del espacio de búsqueda para hayar la mejor solución heurística.
En cada iteración generamos un nuevo vecino y comprobamos el criterio de aceptación para ver si sustituye o no a la solución actual de la siguiente forma:

1. En caso de que él vecino sea mejor, entonces se sustituye automáticamente por la solución actual.
2. En caso contrario, existe probabilidad de que sustituya al vecino. Dicha probabilidad depende de la diferencia de costes entre la solución actual y la vecina y de la temperatura siguiendo esta fórmula:

$$P_{aceptación} = \exp(-\delta/T)$$

Cuanto menor sea la diferencia entre la solución actual y la vecina, mayor probabilidad habrá de que el vecino pase a ser la solución actual.

Una vez terminada la iteración se enfría la temperatura para disminuir la probabilidad de intercambio con perores vecinos y se pasa a la siguiente iteración. El enfriamiento de la temperatura sigue la siguiente fórmula:

$$T_{k+1} = \frac{T_k}{1+\beta \cdot T_k} \quad ; \quad \beta = \frac{T_0 - T_f}{M \cdot T_0 \cdot T_f}$$

donde M es el número de enfriamientos a realizar(15000/máx_vecinos), T0 es la temperatura inicial y Tf ($10\hat{}(-3)$) es la temperatura final que tendrá un valor cercano a cero 1. El número máximo de vecinos es 10*n, el número máximo de éxitos es 0.1*máx_vecinos.

6.2. Pseudocódigo

Algorithm 7 Enfriamiento Simulado

Require: vector de pesos Train, solución inicial (wini)
Ensure:

```
1: Inicializamos T0
2: Almacenamos la solución inicial wini
3: Indicamos que la mejor solución es la inicial
4: while enfriamientosRealizados <maximoNumEnfriamientos & hayExitos do
5:     for i in maximoVecinosGenerados y numExitos <maxExitos do
6:         vecino = generarVecino(solucionActual);
7:         diff = evaluacionVecino - evaluacionSolActual;
8:         if diff >0 || U(0,1) <= pobabilidadAceptacion then
9:             solucionActual = vecino;
10:            if evaluacionSolActual >evaluacionMejorSol then
11:                mejorSolucion = solucionActual;
12:            end if
13:        end if
14:    end for
15:    T = enfriarTemperatura;
16: end while
    return mejorSolucion;
```

7. Iterative Local Search

La ILS está basada en la aplicación sucesiva durante un número de determinadas veces (15 en esta ocasión) de un algoritmo de Búsqueda Local a una solución inicial que se obtiene mediante la mutación de un óptimo local previamente encontrado.
Elementos necesarios para la ejecución del algoritmo:

- Solución inicial escogida de forma aleatoria.
- Proceso de mutación: en este caso será un proceso brusco, realizando cambios sobre t = 0.1*n características de la solución escogidas aleatoriamente considerando un parametro $\sigma = 0.4$ esta vez.
- Un proceso de Búsqueda local para explotar la solución mutada.
- Criterio de aceptación para decidir cuál de las soluciones es la mejor para seguir mutando sobre ella.
  Con el fin de favorecer la intensificación utilizaremos el criterio del mejor entre la solución actual (S) y la solución mutada (S').

Hay que tener en cuenta que esta vez, el número máximo de evaluaciones en la Búsqueda Local es 1000.

7.1. Pseudocódigo

Algorithm 8 Iterative Local Search

Require: vector de pesos Train, solución inicial (wini)
Ensure:

```
1: S = localSearch(wini)
2: for i in 14 do
3:     S' = mutar(S,t);
4:     solucionAux = localSearch(solucionAux);
5:     if evaluacionSolucionActual <evaluacionSolucionAux then
6:         S = solucionAux;
7:     end if
8:     if evaluacionSolucionActual >evaluacionMejorSolucion then
9:         mejorSolucion = S;
10:    end if
11: end for
    return mejorSolucion
```

8. Differential Evolution: Rand

Es un modelo evolutivo el cual hace destacar la mutación y, más tarde aplica operadores de cruce o recombinación.
Es una técnica basada en la evolución de una población de vetores de valores reales que representan las soluciones en el espacio de búsqueda.
Con el operador de selección que utilizaremos nos aseguramos de escoger siempre la mejor de las soluciones entre el padre y el hijo. Aspectos a tener en cuenta para la realización del algoritmo:

- La población inicial (de tamaño 50) es generada de manera aleatoria.
- Proceso de mutación diferencial el cual añade la diferencia proporcional de dos individuos de la poblaión a un tercer individuo (individuo objetivo). El algoritmo se basa en la siguiente fórmula:

$$\mathbf{V}_{i,G} = \mathbf{X}_{r_1,G} + F \cdot \left(\mathbf{X}_{r_2,G} - \mathbf{X}_{r_3,G}\right)$$

  En la que los índices r1, r2 y r3 de cada individuo i en cada generación G serán escogidos aleatoriamente de forma mutuamente excluyente (incluyendo el vector i-ésimo).

- Tipo de recombinación: binomial.

$$u_{j,i,g} = \begin{cases} v_{j,i,g} & \text{if } \text{rand}_j[0,1] \leqslant Cr \text{ or } j = j_{\text{rand}} \\ x_{j,i,g} & \text{otherwise} \end{cases}$$

- Tipo de reemplazamiento: one-to-one, es decir, nos quedamos con el vector objetivo que tenga mejor resultado entre el padre y el hijo.
- Las componentes del vector solución se mantienen dentro de un rango para no permitir soluciones inválidas.

Hay que tener en cuenta que esta ves, el número máximo de evaluaciones en la Búsqueda Local es 1000. Parámetros a tener en cuenta para la realización del algoritmo:

- Probabilidad de cruce (CR) = 0.5.
- F: Establece el rango de diferenciación entre los individuos r2 y r3 con el objetivo de evitar el estancamiento en el proceso de búsqueda. Valor ideal obtenido experimentalmente es 0.5

9. Pseudocódigo

```
Require: vector de pesos Train, población de individuos(Pop), solución inicial (wini)
Ensure:
evaluacionesPoblacion = evaluarPoblacion(Pop);
while numEvaluaciones <maxEvaluaciones do
    for i in Pop.size() do
        p1, p2, p3 = seleccionar3Padres();
        for e in Pop[i].size() do
            if numAleatorio <CR || i == randJ then
                //aplicamos mutación para el DE RAND
                offspring[e] = mutacionDiferencial(p1,p2,p3,F);
            else
                //nos quedamos con la característica del individuo actual
                offspring[e] = Pop[i][e];
            end if
        end for
        evaluacionOffspring = 1NN(offspring);
        if evaluacionOffspring >evaluacionIndividuoActual then
            nuevaPoblacion.push_back(offspring)
        else
            nuevaPoblacion.push_back(individuoActual)
        end if
    end for
end whilereturn mejorIndividuo;
```

10. Differential Evolution: Current to Best

Consiste, excepto en la forma que utiliza para mutar, en lo mismo que el algoritmo anterior descrito (DERand)
Aspectos a tener en cuenta para la realización del algoritmo:

- La población inicial (de tamaño 50) es generada de manera aleatoria.
- Proceso de mutación basado en la siguiente fórmula:

$$\mathbf{V}_{i,G} = \mathbf{X}_{i,G} + F \cdot (\mathbf{X}_{best,G} - \mathbf{X}_{i,G}) + F \cdot (\mathbf{X}_{r_1,G} - \mathbf{X}_{r_2,G})$$

  En la que los índices r1 y r2 de cada individuo i en cada generación G serán escogidos aleatoriamente de forma mutuamente excluyente (incluyendo el vector i-ésimo).Xbest denota el mejor vector en la generación G.

- Tipo de recombinación: binomial.

$$u_{j,i,g} = \begin{cases} v_{j,i,g} & \text{if } \text{rand}_j[0,1] \leqslant Cr \text{ or } j = j_{\text{rand}} \\ x_{j,i,g} & \text{otherwise} \end{cases}$$

- Tipo de reemplazamiento: one-to-one, es decir, nos quedamos con el vector objetivo que tenga mejor resultado entre el padre y el hijo.
- Las componentes del vector solución se mantienen dentro de un rango para no permitir soluciones inválidas.

Parámetros a tener en cuenta para la realización del algoritmo:

- Probabilidad de cruce (CR) = 0.5.
- F: parámetro cuyo valor ideal obtenido experimentalmente es 0.5.

11. Pseudocódigo

```
Require: vector de pesos Train, población de individuos(Pop), solución inicial (wini)
Ensure:
evaluacionesPoblacion = evaluarPoblacion(Pop);
while numEvaluaciones <maxEvaluaciones do
    for i in Pop.size() do
        p1, p2, p3 = seleccionar3Padres();
        for e in Pop[i].size() do
            if numAleatorio <CR || i == randJ then
                //aplicamos mutación para el DE CURRENT TO BEST
                offspring[e] = mutacion(p1,p2,p3,F);
            else
                //nos quedamos con la característica del individuo actual
                offspring[e] = Pop[i][e];
            end if
        end for
        evaluacionOffspring = 1NN(offspring);
        if evaluacionOffspring >evaluacionIndividuoActual then
            nuevaPoblacion.push_back(offspring)
        else
            nuevaPoblacion.push_back(individuoActual)
        end if
    end for
end whilereturn mejorIndividuo;
```

12. Procedimiento considerado para desarrollar la práctica

La inmensa mayoría del código que entrego está desarrollado por mí mismo, exceptuando la lectura de archivos csv por ejemplo, para la cual me he ayudado un poco de internet, así como el uso de páginas como stackOverflow o cPlusPLus para consultar información sobre contenedores de la STL, generación de números aleatorios, uso de la biblioteca ctime para la toma de tiempos...
. No he utilizado nada de código del proporcionado en la web de la asignatura.
El procedimiento seguido para el desarrollo de la práctica es tal cual el que se indica en el guión: lectura de ficheros .arff, normalización de los datos y exportación a .csv en R, lectura de ficheros .csv en C++, generación de las 5 particiones más los datos para Train y para Test, generación de los algoritmos Enfriamiento simulado, Iterative Local Search y los dos de Differential Evolution. Tras la realización de los algoritmos realizamos la recopilación de datos y posteriormente documentación (incluyendo descripción de algoritmos y pseudocódigos) y análisis.

13. Tablas de resultados de ejecución

13.1. Tabla de resultados 1-NN

Como es evidente, la tasa de reducción del algoritmo 1-NN es 0, debido a que el vector de pesos para la evaluación de dicho algoritmo está inicializado a 1, para tener en cuenta todas las características de igual manera. Es por ello también que el agregado sale tan bajo (ya que es la agregación, valga la redundancia, de la tasa de clasificación junto con la de reducción)

	Sonar				Wdbc				Spambase			
	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T
Partición 1	70,731	0	35,3659	0,001775	93.8053	0	46.9027	0.007316	81.5217	0	40,7609	0,00837
Partición 2	85,3659	0	42,6829	0,001781	95.5752	0	47.7876	0.0073	80.4348	0	40,2174	0,008437
Partición 3	85,3659	0	42,6829	0,001773	94.6903	0	47.3451	0.007359	89.1304	0	44,5652	0,008453
Partición 4	85,3659	0	42,6829	0,001797	97.3451	0	48.6726	0.007295	82.6087	0	41,3043	0,008401
Partición 5	81,8182	0	40,9091	0,001872	94.0171	0	47.0085	0.007479	83.6957	0	41,8478	0,008248
MEDIA	81,7295	0	40,8647	0,0017996	95.0866	0	47.5433	0.0073498	83.4783	0	41,7391	0,0083818

Figura 13.1: Tabla 1-NN

13.2. Tabla de resultados RELIEF

El propio hecho de realizar el calculo de los pesos mediante el uso del amigo y el enemigo más cercanos hace que no haya cabida prácticamente para la reducción de pesos en el algoritmo RELIEF ya que, todos los pesos dependen de otros dos pesos: uno que le sumará cierta cantidad (amigo) y otro que le restará cierta cantidad (enemigo).
Por el mismo hecho, la tasa de agregación apenas llega al 50 %.

	Sonar				Wdbc				Spambase			
	% class	% red	Agr.	T	% class	% red	Agr.	T	% class	% red	Agr.	T
Partición 1	68,2927	6,66667	37,4797	0,007958	94.6903	0	47,3451	0,049285	85.8696	10,5263	48,1979	0,041751
Partición 2	78,0488	10	44,0244	0,008015	98.2301	0	49,115	0,052272	78.2609	12,2807	45,2708	0,041935
Partición 3	85,3659	8,33333	46,8496	0,00804	95.5752	6,66667	51,1209	0,05094	92.3913	14,0351	53,2132	0,042615
Partición 4	87,8049	11,6667	49,7358	0,007998	94.6903	0	47,3451	0,050732	91.3043	8,77193	50,0381	0,042562
Partición 5	72,7273	10	41,3636	0,007656	96.5812	10	53,2906	0,048995	90.2174	21,0526	55,635	0,041842
MEDIA	78,4479	9,33333	43,8906	0,007933	95.9534	3,33333	49,6434	0,050444	87.6087	13,3333	50,471	0,042141

Figura 13.2: Tabla resultados RELIEF

13.3. Tabla de resultados ES

Cabe esperar que este algoritmo sea en cierto modo superior al RELIEF (en el sentido de tasa de agregación y tiempo de ejecución) debido simplemente a la intensificación de las mejores solución obtenidas por enfriamiento simulado al final de la ejecución.

Aunque el tiempo empleado por este algoritmo sea superior a los anteriores descritos, podemos observar como el agregado de las tasas de clasificación y reducción ha aumentado a una media del 60 % (sobre las medias de los 3 conjuntos de datos). Este hecho es seguramente debido a la diversificación inicial con la que trabaja el enfriamiento simulado y la disminución de diversificación en contrapartida con el aumento de intensificación al final de su ejecución debido a las bajas temperaturas.

	Sonar				Wdbc				Spambase			
	% class	% red	Agr.	T	% class	% red	Agr.	T	% class	% red	Agr.	T
Partición 1	75,6098	35	55,3049	7,7213	95,5752	43,3333	69,4543	32,4903	81,5217	35,0877	58,3047	35,5596
Partición 2	90,2439	40	65,1219	7,46853	95,5752	40	67,7876	31,5349	81,5217	35,0877	58,3047	34,567
Partición 3	87,8049	31,6667	59,7358	7,60486	95,5752	36,6667	66,1209	31,5931	83,6957	31,5789	57,6373	34,7239
Partición 4	85,3659	28,3333	56,8496	7,75499	95,5752	43,3333	69,4543	31,8383	82,6087	29,8246	56,2166	35,1344
Partición 5	77,2727	33,3333	55,303	7,39723	96,5812	36,6667	66,6239	31,14	86,9565	26,3158	56,6362	36,6096
MEDIA	83,2594	33,6667	58,463	7,58938	95,7764	40	67,8882	31,7193	83,2609	31,5789	57,4199	35,3189

Figura 13.3: Tabla resultados Enfriemiento Simulado

13.4. Tabla de resultados BL

En la ejecución de este algoritmo, podemos comprobar como la tasa de reducción aumenta de forma considerable. Debido a la localidad de las soluciones que encuentra es evidente que no son necesarios, ni mucho menos, muchas de las características asociadas al vector de pesos, en contraposición al algoritmo RELIEF, que como he dicho antes, no prescinde de la mayor parte de los atributos por las dependencias comentadas. El tiempo, debido a la carga computacioinal del propio algoritmo, excede con creces al utilizado por el RELIEF. Las sucesivas búsquedas de vecinos nuevos y las sucesivas comparaciones y evaluaciones de la función objetivo son los causantes de ello.

	Sonar				Wdbc				Spambase			
	% class	% red	Agr.	T	% class	% red	Agr.	T	% class	% red	Agr.	T
Partición 1	82,9268	51,6667	67,2967	4,73407	91.1504	76.6667	83.9086	19,497	80,4348	63,1579	71,7963	20,1576
Partición 2	90,2439	46,6667	68,4553	4,74838	94.6903	73.3333	84.0118	18,5972	77,1739	66,6667	71,9203	20,7793
Partición 3	75,6098	43,3333	59,4715	5,02253	92.9204	76.6667	84.7935	19,6569	90,2174	57,8947	74,0561	20,8364
Partición 4	90,2439	53,3333	71,7886	4,67194	93.8053	83.3333	88.5693	18,2956	83,6957	56,1404	69,918	20,7283
Partición 5	79,5455	55	67,2727	4,44756	93.1624	83.3333	88.2479	19,0941	88,0435	59,6491	73,8463	20,7011
MEDIA	83,714	50	66,857	0,3862	93.1457	78.6667	85.9062	0,6756	83,913	60,7018	72,3074	0,5872

Figura 13.4: Tabla resultados Local Search

13.5. Tabla de resultados ILS

Mediante la sucesiva aplicación de mutación más intensificación mediante búsqueda local, logramos conseguir tasas altas tanto de clasificación como de reducción, haciendo por tanto que el porcentaje de agregado sea alto y, hasta el momento escojamos el algoritmo de Búsqueda Local Iterativa como el mejor para la resolución de este problema. Eso sí, los tiempos de ejecución con respecto a los la local search son prácticamente 10 veces mayores.
Como es evidente, si ya la búsqueda local obtenia mejoras respecto a RELIEF, la ILS mejora de forma providencial los resultados obtenidos comparado con RELIEF aunque, los tiempos de ejecución en contraste con los de este, son desmesurados.

	Sonar				Wdbc				Spambase			
	% class	% red	Agr.	T	% class	% red	Agr.	T	% class	% red	Agr.	T
Partición 1	75,6098	85	80,3049	70,0934	92.0354	90	91.0177	252,346	84.7826	96,4912	90,6369	332,886
Partición 2	87,8049	86,6667	87,2358	66,4375	94.6903	90	92.3451	267,088	86.9565	91,2281	89,0923	324,291
Partición 3	82,9268	81,6667	82,2967	68,117	89.3805	93.3333	91.3569	288,195	88.0435	89,4737	88,7586	351,987
Partición 4	85,3659	80	82,6829	69,1877	92.9204	93.3333	93.1268	266,958	85.8696	92,9825	89,426	342,335
Partición 5	86,3636	85	85,6818	63,788	93.1624	93.3333	93.2479	266,162	81.5217	91,2281	86,3749	309,373
MEDIA	83,6142	83,6667	83,6404	67,5247	92.4378	92	92.2189	268,15	85.4348	92,2807	88,8577	332,174

Figura 13.5: Tabla resultados Iterative Local Searh

13.6. Tabla de resultados DE: Rand

El éxito alcanzado con la ILS dura poco al llegar a los algoritmos evolutivos.
Concretamente, el algoritmo de evolución diferencial aleatorio consigue mejorar a la ILS en todos los aspectos: tanto en tasa de agregado como en tiempo de ejecución.
Teniendo en cuenta esto, nos quedamos con este algoritmo últra-reductor", el cual consigue reducir hasta el 90 % de las características asociadas a al vector de pesos. Seguramente esta gran reducción vaya asociada al hecho de que para el cálculo de nuevos hijos, se realiza una mutación genética asociada a 3 individuos distintos (con cierta probabilidad de cruce).

	Sonar				Wdbc				Spambase			
	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T
Partición 1	78,0488	90	84,0244	65,4804	87.6106	93.3333	90.472	279.868	88.0435	92,9825	90,513	316,951
Partición 2	82,9268	90	86,4634	64,9372	92.0354	93.3333	92.684	291.487	89.1304	92,9825	91,0564	323,858
Partición 3	85,3659	86,6667	86,0163	63,7065	90.2655	93.3333	91.799	288.847	88.0435	94,7368	91,3902	319,61
Partición 4	78,0488	90	84,0244	64,6616	92.0354	93.3333	92.684	297.773	77.1739	94,7368	85,9554	320,582
Partición 5	72,7273	86,6667	79,697	61,2473	93.1624	93.3333	93.247	287.75	83.6957	94,7368	89,2162	316,251
MEDIA	79,4235	88,6667	84,0451	64,0066	91.0219	93.3333	92.177	289.145	85.2174	94,0351	89,6262	319,451

Figura 13.6: Tabla resultados DE: Rand

13.7. Tabla de resultados DE: Current To Best

En un intento de realizar una mejora sobre los resultados del Differential Evolution aleatorio obtenido en la anterior explicación, nos topamos con que a la hora de realizar las mutaciones teniendo en cuenta al mejor de los individuos, las tasas tanto de clasificación como de agregado disminuyen, lo que hace que, en comparación con el anterior algoritmo, este no sea muy eficaz (además de que hace aumentar levemente el tiempo de ejecución del mismo).

	Sonar				Wdbc				Spambase			
	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T
Partición 1	68,2927	65	66,6463	69,8248	95,5752	76,6667	86,1209	261,705	71,7391	75,4386	73,5889	322,636
Partición 2	85,3659	76,6667	81,0163	69,8083	94,6903	80	87,3451	270,149	81,5217	66,6667	74,0942	321,773
Partición 3	92,6829	65	78,8415	71,5805	88,4956	86,6667	87,5811	271,135	84,7826	71,9298	78,3562	321,256
Partición 4	85,3659	73,3333	79,3496	69,7281	92,0354	93,3333	92,6844	275,944	88,0435	66,6667	77,3551	334,432
Partición 5	72,7273	61,6667	67,197	61,6635	90,5983	83,3333	86,9658	266,581	83,6957	78,9474	81,3215	325,027
MEDIA	80,8869	68,3333	74,6101	68,521	92,2789	84	88,1395	269,103	81,9565	71,9298	76,9432	325,025

Figura 13.7: Tabla resultados DE: Current To Best

13.8. Tabla de resultados AGG-BLX

Nos salimos de los algoritmos evolutivos para comparar comprobar si mediante los géneticos conseguimos alguna ganancia y, no se da el caso.
Observamos en la siguiente tabla la baja tasa de reducción que nos aporta este algoritmo y, por tanto, baja tasa de agregación.

	Sonar				Wdbc				Spambase			
	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T
Partición 1	70,7317	16,6667	43,6992	74,3535	92.9204	23,3333	58,126	310,168	79,3478	28,0702	53,709	342,306
Partición 2	90,2439	18,3333	54,2886	73,8017	95.5752	20	57,7876	310,446	80,4348	19,2982	49,8665	341,984
Partición 3	82,9268	23,3333	53,1301	73,7031	95.5752	30	62,7876	311,124	83,6957	26,3158	55,0057	342,3
Partición 4	87,8049	21,6667	54,7358	73,5615	95.5752	16,6667	56,12	311,47	84,7826	19,2982	52,0404	342,522
Partición 5	86,3636	30	58,1818	71,8428	98.2906	23,3333	60,812	305,513	83,6957	12,2807	47,9882	342,035
MEDIA	83,6142	22	52,8071	73,4525	95.5873	22,6667	59,127	309,744	82,3913	21,0526	51,722	342,229

Figura 13.8: Tabla resultados AGG-BLX

13.9. Tabla de resultados AM-10 mej.

He tenido problemas a la hora de ejecutar este algoritmo, ya que solo he podido ejecutarlo con la base de datos de Sonar. El problema con las otras bases de datos es la dimensión de estas. Por algún motivo que desconozco, la carga computacional para mi ordenador es demasiada y, en 30 minutos que he estado ejecutando el algoritmo para la base de datos WDBC, ni tan siquiera ha sido capaz de ejecutar una sola partición.
Por este motivo descarto totalmente que el uso de este algoritmo sea viable a la hora de resolver el problema.

13.10. Tabla de resultados medios de los algoritmos

	Sonar				Wdbc				Spambase			
	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T	%_class	%_red	Agr.	T
1NN	81,7295	0	40,8647	0,0017996	95.0866	0	47.5433	0.0073498	83.4783	0	41,7391	0,0083818
Relief	78,4479	9,33333	43,8906	0,0079334	95.9534	3,33333	49,6434	0,0504448	87.6087	13,3333	50,471	0,042141
ES	83,2594	33,6667	58,463	7,58938	95,7764	40	67,8882	31,7193	83,2609	31,5789	57,4199	35,3189
ILS	83,6142	83,6667	83,6404	67,5247	92.4378	92	92.2189	268,15	85.4348	92,2807	88,8577	332,174
DE_rand	79,4235	88,6667	84,0451	64,0066	91.0219	93.3333	92.177	289.145	85.2174	94,0351	89,6262	319,451
DE_current_to_best	80,8869	68,3333	74,6101	68,521	92,2789	84	88,1395	269,103	81,9565	71,9298	76,9432	325,025
BL	83,714	50	66,857	0,3862	93.1457	78.6667	85.9062	0,6756	83,913	60,7018	72,3074	0,5872
AGG-BLX	83,6142	22	52,8071	73,4525	95.5873	22,6667	59,127	309,744	82,3913	21,0526	51,722	342,229
AM(10,0.1mej)	78,4479	9,33333	43,8906	0,0079852	95.9534	3,33333	49,6434	0,0505708	87,6087	13,3333	50,471	0,0422822

Figura 13.9: Tabla resultados de las medias de todos los algoritmos

14. Análisis de resultados obtenidos.

Como conclusión final, a la vista de los resultados observados con cada uno de los algoritmos implementados, es fácil elegir el mejor de los algoritmos para el problema de la asignación de pesos a características (APC) basándonos tanto en tiempos de ejecución como en las tasas de clasificación y agregación obtenidas es decir, la tasa de agregación. El algoritmo que mejores resultados obtiene con diferencia es el Differential Evolution aleatorio, el cual predomina sobre el Current To Best tanto en tiempo como en agregado (de forma ligera) seguramente por la diversidad que introduce la aleatoriedad en lugar de encaminar la solución obteniendo hijos a partir de la mejor solución de la población actual.

Lo que diferencia a los algoritmos evolutivos del resto de algoritmos es sin lugar a duda el operador de cruce empleado para la obtención de individuos nuevos. En dicho operador se tiene en cuenta a tres individuos de la población escogidos de manera aleatoria. Uno es utilizado como nodo objetivo al cual se le añade la diferencia proporcional entre los otros dos individuos. Este cruce no es realizado siempre, sino solo en el caso de que se cumpla la probabilidad de cruce. Si esta no se cumple, entonces el descendiente es el mismo individuo que esta siendo evaluadoo en esa iteración, lo que hace que no siempre se obtenga un peso i diferente al del individuo actual
Esto, junto con el operador de selección que escoge el mejor de los individuos entre el descendiente generado y el padre (individuo actual), son las características que hacen que el algoritmo evolutivo aleatorio se superponga sobre cualquier otro de los implementados.

Al algoritmo de evolución diferencial aleatorio le sigue muy de cerca el algoritmo de busqueda local iterativa, el cual consigue aproximarse a las tasas de agregación del DE pero en tiempo se va de la cuenta un poco en comparación con el mejor, es decir, es más lento a la hora de ejecutarlo.

Tras la ILS, de los algoritmos implementados en esta práctica, se situa el enfriamiento simulado, el cual está muy por debajo tanto de la ILS como del DE en cuanto a tasa de agregación pero, el costo computacional no es tan alto, por lo que los tiempos se reducen en gran medida respecto de estos dos algoritmos.

Los algoritmos Implementados en las prácticas anteriores no consiguen buenos resultados ni de lejos comparados con los algoritmos de esta práctica. Esto es debido a que los algoritmos genéticos y meméticos son algoritmos encauzados a la optimización de la tasa de clasificación pero, en este problema, al intentar realizar reducción utilizando estos dos métodos, obtenemos pésimos resultados con agregaciones del 50 %, lo que hace que no sean nada competitivos a la hora de obtener un predictor fiable con el menor número de atributos posibles.