UNIVERSIDAD DE GRANADA

Práctica 2.b

Técnicas de búsqueda basadas en poblaciones para el problema del aprendizaje de pesos en características (APC).

Mario Román García

2 de mayo de 2018

- DNI: 77145669N
- Email: mromang08@correo.ugr.es

- Grupo 1 (Lunes de 17:30 a 19:30)
- Algoritmos: AGE-{BLX,CA}, AGG-{BLX,CA}, AM-{10,0.1,mej}.
- Metaheurísticas, 2017-2018.

Índice

1. Formulación del problema

Trataremos un problema de **aprendizaje de pesos en características** (APC), consistente en la optimización de la simplicidad y precisión de un clasificador 1-NN; es decir, un clasificador que asigna a cada instancia la clase de la instancia más cercana a él, para una distancia euclídea modificada por un vector de pesos. Así, cada solución del problema vendrá dada por un vector de valores reales $w_i \in [0, 1]$ para $1 \leq i \leq n$, donde n es el número de características que tiene el problema. La bondad de un clasificador de este tipo sobre un conjunto de evaluación T vendrá determinada como

$$F(\{w_i\}) = \alpha \text{tasaClas}\,\{w_i\} + (1 - \alpha)\text{tasaRed}\,\{w_i\}\,.$$

En esta fórmula, *tasaClas* debe ser entendida como la **precisión** del algoritmo, midiendo el porcentaje de aciertos del clasificador en el conjunto de evaluación T. Por otro lado, *tasaRed* debe ser entendido como la **simplicidad** de la solución, que mide el número de pesos w_i que quedan por debajo de 0.2, y que consecuentemente no se tienen en cuenta al calcular las distancias. En nuestro caso tenemos determinado un valor de $\alpha = 0.5$ que equipara ambas métricas.

En resumen, el algoritmo deberá tomar un conjunto de datos y producir un clasificador 1-NN con unos pesos asignados a las características que sean sencillos, en el sentido de usar el mínimo número de características, y que sean precisos, en cuanto a su habilidad para clasificar instancias fuera del conjunto de entrenamiento.

2. Aplicación de algoritmos

2.1. Esquema de representación de soluciones

Como hemos tratado anteriormente, una solución del problema viene dada por un vector de pesos. Nuestra representación específica de una solución vendrá dada por una lista de valores reales.

$$\{w_i\}_{0 \leq i \leq n} = (w_1, w_2, \ldots, w_n)$$

Donde n será el número de características que tenga nuestro conjunto de datos y w_i. En nuestra implementación específica, los pesos vendrán dados por un vector contiguo en memoria de valores en coma flotante de precisión doble en 64 bits; para facilitar su tratamiento, en ciertos algoritmos se traducirán a listas enlazadas, pero su interpretación matemática será siempre constante y se describe explícitamente en la función objetivo.

2.2. Clasificador 1-NN, función objetivo

Se define la **distancia con pesos** $\{w_i\}$ entre dos vectores t y s como

$$\text{dist}(w, t, s) = \sum_{i=0}^{n} w_i (t_i - s_i)^2.$$

Y nuestro clasificador kNN para unos pesos $\{w_i\}$ consiste en devolver la clase del punto que minimiza la distancia. Es decir, es una implementación de un clasificador 1NN, que para cada instancia devuelve la clase de su vecino más cercano. En el caso de la función objetivo, al no tener un conjunto de test separado, usamos la técnica de *leave-one-out*.

Algorithm 1 Función objetivo (w : Pesos, T : Training)

1: $\text{Obj}(w, T) = \alpha \cdot \text{precision}(w, T) + (1 - \alpha)\text{simplicity}(w)$
2: $\text{TasaRed}(w) = \text{length}[x < 0.2 \mid x \in w]/\text{length } w$
3: $\text{TasaClas}(w, T) = \sum_{t \in T}(\text{knn}(w, T - t, t) == s.Clase)/\text{length } s$
4: $\text{knn}(w, T, t) = (\text{minimizador}_{t' \in T}(\text{dist}^2(\text{trunca}(w), t', t))).Clase$
5: $\text{trunca}(w) = \{0 \text{ si } w_i < 0.2; \quad w_i \text{ en otro caso } \mid w_i \in w\}$

En el código original esta función objetivo aparecerá implementada dos veces: una vez para el puntuador de los algoritmos y otra vez para la función objetivo. Esta duplicación tiene como ventaja que separa completamente las partes de evaluación del código de los algoritmos, reduciendo la posibilidad de error. Además,

- la implementación para evaluación es corta y es más fácil de verificar que está escrita correctamente;
- mientras que la implementación de la función objetivo está fuertemente optimizada, usando paralelismo, pero en caso de que tuviera cualquier error, eso no se vería reflejado en las puntuaciones de los algoritmos.

Es importante notar que la formulación original del problema no asume que todas las variables sean reales, y en el caso en el que son discretas asume una distancia de Hamming que simplemente indica si las dos características son exactamente iguales.

$$\text{dist}(a, b) = \delta_{ab} = \begin{cases} 1 & \text{si } a = b \\ 0 & \text{si } a \neq b \end{cases}$$

En los conjuntos de datos que trataremos en este análisis, tenemos de hecho que todas las características vienen dadas por reales, y nuestra implementación, aunque sería fácilmente extensible, no tratará el otro caso explícitamente.

2.3. Generación de soluciones aleatorias

La generación de una **solución aleatoria inicial** se realiza llamando repetidas veces a las librerías del lenguaje [Yor16]. Estas proporcionan funciones que nos permiten clacular listas potencialmente infinitas de números reales distribuidos uniformemente en el intervalo $[0, 1]$. Además, usamos replicamos el proceso aleatorio con mónadas para poder generar varios individuos cuidando que sean muestras independientes bajo el mismo generador aleatorio.

Algorithm 2 Solución inicial (t : Training)

1: solInicial(t) = tomaLos (nAttr(t)) primerosDe aleatorioUniforme$(0.0, 1.0)$
2: PoblInicial(t) = replica 30solInicial

3. Pseudocódigo de selección, cruce y mutación

3.1. Torneo binario

Un torneo binario es un proceso aleatorio que genera un individuo o desde una población de soluciones. La población de soluciones llega a este punto del algoritmo en una estructura ordenada por bondad de solución basada en árboles binarios balanceados por tamaño (véase [Lei02] y [Ada96]), por lo que simplemente debemos elegir dos índices y tomar el mayor. El conjunto de entrenamiento no se usa explícitamente en el algoritmo pero es una dependencia necesaria para haber calculado previamente la bondad de los elementos de la población.

Algorithm 3 Torneo binario (p : Población, T : Training)

1: x = randomEntre$(0, \text{size}(p) - 1)$
2: y = randomEntre$(0, \text{size}(p) - 1)$
3: torneo(p) = p[max x y]

En el código implementamos además variantes que realizan varios torneos binarios seguidos para usarlas directamente en los algoritmos.

3.2. Cruce aritmético

El cruce aritmético de dos soluciones es una operación componente a componente que devuelve el centro de gravedad n-dimensional de los dos padres. Es una operación determinista que sólo genera un hijo.

Algorithm 4 Cruce aritmético (a : Individuo, b : Individuo)

1: $media(x, y) = (x + y)/2.0$
2: $ca(a, b)$ = componenteAComponente media a b

3.3. Cruce BLX

El cruce BLX sí es no determinista y sí nos permitirá obtener dos hijos desde una sola pareja de padres. Nuestro código genera un solo hijo y simplemente repite (de nuevo usando mónadas) el procedimiento para asegurarse la independencia de los dos hijos.

En nuestro caso prepararemos el código para tratar uniformemente los dos cruces distintos, haciendo que cada uno devuelva una lista con uno y dos hijos y dejando que cada algoritmo trate estos dos casos.

Algorithm 5 Cruce BLX (a : Individuo, b : Individuo)

1: $blx2(a, b) =$ replica 2 blx(a, b)
2: $blx(a, b) =$ componenteAComponente blxComp(a, b)
3: $blxComp(x, y) =$ aleatorioUniformeEn intervalo(x, y)
4: $intervalo(x, y) = [0, 1] \cap [\text{mín}(x, y) - \alpha|x - y|, \text{máx}(x, y) + \alpha|x - y|]$

3.4. Mutación

La mutación, en su versión más común, procede directamente del operador de generación de vecinos de la búsqueda local. En ocasiones nos interesará mutar cada gen de un individuo con probabilidad 0.001.

Algorithm 6 Mutación (s : Solución)

1: $Muta(s) =$ truncaEntre0y1 map $(\lambda x.\text{if } rand() < 0.001 \text{ then } x + rand() \text{ else } x)$

Mientras que en otras ocasiones nos interesará aplicar un número fijo de mutaciones aleatorias sobre la población completa en lugar de generar un número aleatorio y comprobar si mutamos o no cada uno de los genes.

Algorithm 7 MutaPoblación (p : Población, n : Nº mutaciones)

1: $Muta(p) =$ replica n $MutaUnaVez(p[i], j),$ para $i = rand(), j = rand()$
2: $MutaUnaVezEn(s, j) =$ truncaEntre0y1 map $(\lambda(x, i).x + \delta_{ij}\varepsilon)$ (indexa s)

En esos casos controlaremos el número de mutaciones totales para que se correspondan a las que deberían producirse en caso de que usáramos la esperanza matemática para calcular el número de mutaciones.

4. Pseudocódigo del esquema de evolución y reemplazamiento

En general, nuestro algoritmo genético repite una subrutina denominada *paso generacional* que se corresponderá con un cruce y sustitución en el modelo estacionario y con el avance de una generación en la evolución generacional. El criterio de parada no se determina por el número de generación sino por el número de evaluaciones de la función objetivo, que se almacena con la estructura de datos del algoritmo.

De esta forma podemos modelar las variantes del algoritmo evolutivo uniformemente. Cuando trabajamos gestionando el generador aleatorio, podemos escribir en estilo imperativo (esto se conoce como un "bloque `do`").

4.1. Esquema de evolución estacionario

En el esquema de evolución estacionario, cada iteración hará dos evaluaciones para los dos nuevos hijos creados y que compiten con los individuos de la población anterior. Se

Algorithm 8 EsquemaEvolutivo (pasoEv : Subrutina)

1: $pobl \leftarrow$ poblacionAleatoria()
2: iteraMientras($evaluaciones > 15000$)
3: $\quad pasoEv(env)$

realizarán los torneos necesarios para generar 2 hijos (que serán 2 o 4 según el operador de cruce); luego se mutarán los hijos sin usar la esperanza matemática, usando el primero de los operadores de mutación que describimos. Finalmente, se incluyen los hijos en la población y se eliminan los peores; la estructura de datos ordenada se encargará de esto automáticamente.

Algorithm 9 EsquemaEstacionario (p : Población, cruza : Operador)

1: $padres \leftarrow$ torneosBinarios()
2: $hijos \leftarrow$ toma 2 de cruza(empareja(padres))
3: $mutados \leftarrow$ map $muta\ hijos$
4: $nuevaPopl \leftarrow$ insertaYReemplazaEn$(p, mutados)$
5: $contadorEvaluaciones \leftarrow +2$

Finalmente, el *emparejamiento* se hace tomando el primero con el segundo, tercero con el cuarto, y así sucesivamente. En el caso en el que se necesitaran más parejas, se vuelve a empezar con la lista en el segundo elemento para emparejar el segundo con el tercero y así sucesivamente.

4.2. Esquema de evolución generacional

Las iteraciones del algoritmo generacional generan 21 hijos nuevos; las parejas necesarias para ello (que serán el doble cuando estemos usando un operador de cruce de un solo hijo como el cruce aritmético) se obtendrán generando al principio una nueva población por torneo binario de 30 padres y luego usando el 70 % para generar hijos. Luego aplicaremos de nuevo mutación a toda la población usando esta vez la esperanza matemática y acabaremos incluyendo el mejor de la generación anterior en la nueva población.

Algorithm 10 EsquemaGeneracional (p : Población, cruza : Operador)

1: $mejor \leftarrow \max(p)$
2: $padres \leftarrow$ torneosBinarios()
3: $hijos \leftarrow$ cruza(empareja(70 % de los $padres$))
4: $npopl \leftarrow$ toma 30 de $padresNoCruzados + hijos + padresNoSubstituidos$
5: $npopl \leftarrow$ reemplaza($mejor$) \$ muta($npopl$)
6: $contadorEvaluaciones \leftarrow n(hijos) + n(padresMutados)$

4.3. Reemplazamiento

Explícitamente, el reemplazamiento en la población se realiza insertando y eliminando el mínimo en la estructura ordenada que describimos anteriormente.

Algorithm 11 Reemplazamiento (p : Población, h : individuo)

1: $reemplaza(p, h) = (borraMnimo \circ inserta(h))\ p$

En cuanto al reemplazamiento en la selección del torneo binario, se ha implementado con y sin reemplazamiento (simplemente filtrando índices duplicados), y se usa con reemplazamiento cuando el número de padres no sería suficiente de otra forma.

5. Pseudocódigo de integración de algoritmos meméticos

El haber escrito el esquema evolutivo independiente de la estrategia evolutiva o el operador de cruce concreto que usemos nos permite ahora introducir los algoritmos meméticos con una variación pequeña del código de esquema evolutivo. Nótese que la población aquí considerada será de 10 en lugar de 30.

Algorithm 12 EsquemaMemético (pasoEv : Subrutina)

1: $pobl \leftarrow$ poblacionAleatoria($size = 10$)
2: iteraMientras($evaluaciones > 15000$)
3: Si $generacion\,\%10 == 0$ entonces $pasoM(ev)$ si no $pasoEv(env)$

Aquí, *generación*, que no debe confundirse con el número de evaluaciones, cuenta el número de iteraciones del paso evolutivo, ya sea generacional o estacionario.

5.1. Versiones del algoritmo memético

Sobre este esqueleto se integran las distintas versiones del algoritmo memético. Todas ellas se empiezan basando en una función que aplica búsqueda local sobre los n mejores individuos de la población, pero con una probabilidad p sobre cada uno de ellos. Si llamamos a esta función $pBusqueda(n, p)$, las tres versiones del algoritmo memético buscadas pueden obtenerse como

1. $pBusqueda(size(popl), 1)$, aplicará búsqueda sobre todos los cromosomas de la población;

2. $pBusqueda(size(popl), 0.1)$, aplicará búsqueda para cada cromosoma de un conjunto en el que se selecciona cada uno con probabilidad 0.1;

3. $pBusqueda(size(popl)/10, 1)$, aplicará búsqueda sobre los 0.1 mejores cromosomas de la población.

El código de esta función central incrementará el contador de evaluaciones según las necesite. Nótese que aunque la búsqueda local siempre debe parar cuando se hayan evaluado $2m$ vecinos distintos en cada ejecución, donde m es el número de atributos, el número total de evaluaciones dependerá de la selección aleatoria de cromosomas y no será fijo en general.

Algorithm 13 pBusqueda (n : nº mejores, p : probabilidad, popl : Población)

1: $(noseleccionados, seleccionados) \leftarrow (\text{escogelosNmejores}(n, popl), \text{resto}(n, popl))$
2: $nuevos \leftarrow map\ localSearch\ seleccionados$
3: return $nuevos \cup noseleccionados$

Finalmente, esta *pBusqueda* es la que se usa como implementación de *pasoM*, variando los parámetros en cada uno de los casos.

6. Procedimiento considerado, manual de usuario

Al igual que en la primera práctica, se usa **Haskell** [P+03] y paralelismo con [Les08]. El proceso de validación y generación de los resultados se hace reproducible con [Smi92] y se encuentra en el archivo `makefile`, en el que se declaran las semillas de aleatoriedad (`$SEEDn`) que son las que se envían a los distintos algoritmos.

Además de los ejecutables de validación `bin/fivefold` y `bin/scorer`, así como los algoritmos de la primera práctica, presentamos los ejecutables nuevos de algoritmos genéticos:

- `bin/Ageca`, implementación del algoritmo genético estacionario con cruce aritmético;
- `bin/Ageblx`, implementación del algoritmo genético estacionario con cruce BLX;
- `bin/Aggca`, implementación del algoritmo genético generacional con cruce aritmético;
- `bin/Aggblx`, implementación del algoritmo genético generacional con cruce BLX;

y los ejecutables de algoritmos meméticos:

- `bin/AmAll`, implementación del algoritmo memético sobre genético generacional con cruce BLX, aplicando búsqueda cada 10 generaciones sobre todos los los cromosomas (AM-10,1.0);
- `bin/AmProb`, implementación del algoritmo memético sobre genético generacional con cruce BLX, aplicando búsqueda cada 10 generaciones sobre todos los los cromosomas pero con probabilidad 0.1 (AM-10,0.1);
- `bin/AmBest`, implementación del algoritmo memético sobre genético generacional con cruce BLX, aplicando búsqueda cada 10 generaciones sobre el 10 % de los mejores (AM-10,0.1mej).

Todas las implementaciones reciben como argumento de línea de comandos una semilla aleatoria y leen por la entrada estándar un conjunto de entrenamiento; acabarán devolviendo una solución por salida estándar.

Para el resto de detalles de ejecución nos referimos a la primera práctica.

7. Experimentos y análisis de los resultados

Nuestros algoritmos reciben varios parámetros que fijamos en el código. Explícitamente, el tamaño de población en los genéticos es 30, mientras que es 10 en los meméticos. La probabilidad de cruce será de 1 en el estacionario y 0.7 en el generacional (nótese que en el generacional usaremos la esperanza matemática). La probabilidad de mutación será siempre del 0.001 (en el generacional usaremos de nuevo la esperanza matemática). En la mutación, tomamos $\sigma = 0.3$ al igual que en la primera práctica y para el BLX tomamos $\alpha = 0.3$. El criterio de parada es siempre de 15000 evaluaciones de la función objetivo.

7.1. AGE-CA

Los resultados de nuestra primera versión de un algoritmo genético son comparables con los resultados de la búsqueda local, siendo la única diferencia un incremento en tiempo derivado de que, aunque nuestro criterio de parada nos sigue asegurando la misma cantidad de evaluaciones de la función objetivo, la gestión de la población consume ahora mucho más tiempo.

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
Partición 1	0.641	0.750	0.695	56.817	0.650	0.773	0.711	16.351	0.815	0.659	0.737	53.703
Partición 2	0.719	0.764	0.741	58.385	0.900	0.864	0.882	15.965	0.759	0.659	0.709	54.944
Partición 3	0.750	0.667	0.708	58.459	0.800	0.727	0.764	15.943	0.759	0.682	0.721	55.115
Partición 4	0.812	0.667	0.740	56.819	0.625	0.727	0.676	16.983	0.815	0.591	0.703	55.134
Partición 5	0.812	0.667	0.740	57.562	0.743	0.727	0.735	17.590	0.863	0.750	0.806	53.451
Media	0.747	0.703	0.725	57.608	0.744	0.764	0.754	16.566	0.802	0.668	0.735	54.469

Cuadro 1: Algoritmo AGE-CA en el problema del APC

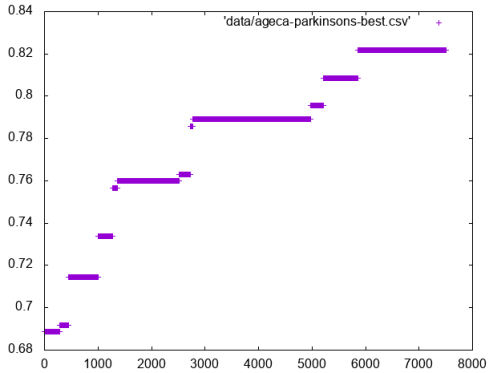

Figura 1: Puntuación del mejor individuo en el tiempo en Parkinsons bajo AGE-CA.

7.2. AGE-BLX

El cambio a un operador de cruce BLX parece dar mejores resultados. Una hipótesis es que la falta de diversidad estaba limitando innecesariamente nuestro algoritmo, haciendo

mucho más lento el saltar de una solución a otra mejor y atascándose innecesariamente en extremos locales. Esta hipótesis podemos comprobarla con los gráficos que añadimos, que muestran cómo varía la mejor solución encontrada conforme avanza el algoritmo. Destacamos que este nuevo operador introduce una componente de aleatoriedad y que permite la exploración más allá de la recta que incluye a las dos soluciones, ambas características que no poseía el cruce aritmético.

Cuadro 2: Algoritmo AGE-BLX en el problema del APC

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
Partición 1	0.672	0.847	0.760	52.604	0.800	0.909	0.855	17.367	0.685	0.841	0.763	35.973
Partición 2	0.781	0.792	0.786	54.268	0.750	0.955	0.852	17.198	0.815	0.682	0.748	37.450
Partición 3	0.734	0.889	0.812	54.032	0.925	0.864	0.894	17.485	0.815	0.750	0.782	37.033
Partición 4	0.703	0.847	0.775	53.836	0.825	0.818	0.822	18.017	0.722	0.795	0.759	36.139
Partición 5	0.828	0.778	0.803	54.688	0.629	0.864	0.746	18.869	0.804	0.886	0.845	35.162
Media	0.744	0.831	0.787	53.886	0.786	0.882	0.834	17.787	0.768	0.791	0.779	36.351

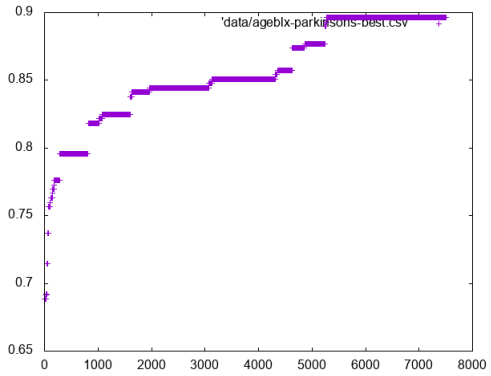

Figura 2: Puntuación del mejor individuo en el tiempo en Parkinsons bajo AGE-BLX.

En este caso, obtenemos un algoritmo que mejora ligeramente en resultados a la búsqueda local, indicando que este es un buen camino a seguir.

7.3. AGG-CA

El cambio a generacional da resultados consistentemente peores que el uso de estacionarios. Nuestra hipótesis aquí es que esto esté causado porque, al seleccionar una nueva población a cada generación con torneos que pueden repetir a los elementos de la élite a cada paso, la convergencia sea más rápida a un extremo local de lo que sería deseable.

7.4. AGG-BLX

En la implementación de un algoritmo genético generacional con operador de cruce BLX obtenemos resultados ligeramente mejores que con su equivalente con el cruce aritmético. De nuevo el uso de un operador de cruce incrementando la posibilidad de

Cuadro 3: Algoritmo AGG-CA en el problema del APC

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
Partición 1	0.641	0.667	0.654	33.080	0.750	0.455	0.602	9.031	0.778	0.636	0.707	30.315
Partición 2	0.828	0.639	0.734	34.266	0.825	0.364	0.594	9.749	0.704	0.614	0.659	30.502
Partición 3	0.703	0.653	0.678	32.630	0.850	0.409	0.630	9.040	0.778	0.659	0.718	30.412
Partición 4	0.797	0.653	0.725	31.224	0.700	0.364	0.532	9.707	0.759	0.568	0.664	30.762
Partición 5	0.719	0.611	0.665	32.990	0.800	0.318	0.559	9.644	0.784	0.659	0.722	30.572
Media	0.738	0.645	0.691	32.838	0.785	0.382	0.583	9.434	0.761	0.627	0.694	30.513

exploración mejora los resultados. Esta mejora es la que nos lleva a implementar los algoritmos meméticos sobre este operador de cruce en lugar de sobre el cruce aritmético.

Cuadro 4: Algoritmo AGG-BLX en el problema del APC

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
Partición 1	0.750	0.667	0.708	58.386	0.725	0.682	0.703	14.646	0.833	0.614	0.723	54.581
Partición 2	0.875	0.653	0.764	58.796	0.775	0.545	0.660	16.239	0.722	0.591	0.657	52.923
Partición 3	0.688	0.611	0.649	63.074	0.725	0.500	0.612	16.086	0.778	0.614	0.696	52.356
Partición 4	0.766	0.681	0.723	58.969	0.700	0.545	0.623	15.972	0.870	0.523	0.697	55.612
Partición 5	0.734	0.597	0.666	60.222	0.800	0.409	0.605	17.354	0.765	0.682	0.723	54.331
Media	0.763	0.642	0.702	59.889	0.745	0.536	0.641	16.059	0.794	0.605	0.699	53.961

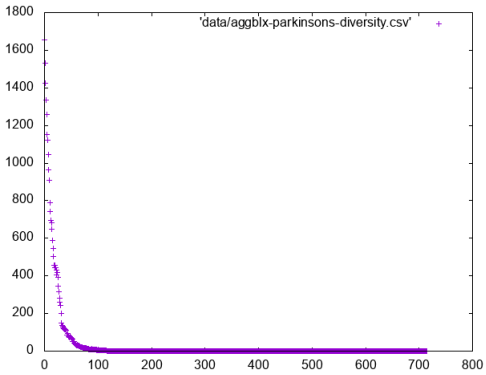

Figura 3: Diversidad de la población en el tiempo en Parkinsons bajo AGG-BLX.

Confirmamos además que tenemos una gran falta de diversidad en nuestro algoritmo, podemos definir una medida de diversidad como la suma de distancias entre los distintos individuos.

$$\text{diversidad}(Popl) = \sum_{a,b \in \text{Popl}} \text{dist(a, b)}.$$

7.5. AM-(10,1.0)

Los algoritmos meméticos representan una importante mejora frente a sus contrapartes genéticas. En particular, en este caso implementamos un algoritmo memético que cada 10 generaciones aplica búsqueda local a toda la población, y que usa el esquema generacional con un operador de cruce BLX. La elección de BLX la justificamos en la superioridad que ha mostrado en los experimentos anteriores.

Cuadro 5: Algoritmo AM-(10,1.0) en el problema del APC

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
Partición 1	0.688	0.806	0.747	56.761	0.850	0.864	0.857	16.310	0.722	0.818	0.770	53.718
Partición 2	0.844	0.833	0.839	56.484	0.825	0.909	0.867	16.429	0.685	0.705	0.695	60.846
Partición 3	0.750	0.833	0.792	57.573	0.875	0.818	0.847	16.525	0.759	0.818	0.789	57.060
Partición 4	0.750	0.819	0.785	57.894	0.750	0.909	0.830	15.903	0.778	0.864	0.821	57.693
Partición 5	0.812	0.792	0.802	58.243	0.743	0.909	0.826	16.496	0.725	0.750	0.738	59.329
Media	0.769	0.817	0.793	57.391	0.809	0.882	0.845	16.333	0.734	0.791	0.763	57.729

En este caso podemos comprobar que una de las causas por las que debe estar mejorando el caso genético es debido a que la diversidad se está incrementando. Tomaremos esta idea como inspiración para implementar una versión propia del algoritmo memético: idealmente nos gustaría poder mantener un cierto nivel de diversidad durante la ejecución completa del algoritmo.

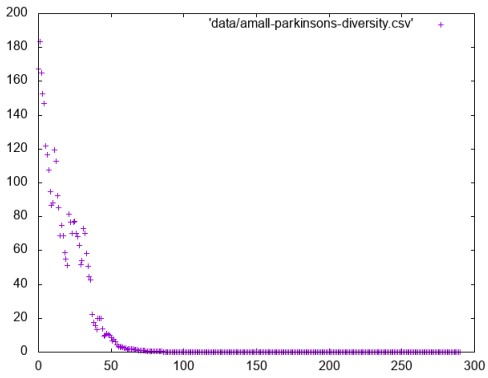

Figura 4: Diversidad de la población en el tiempo en Parkinsons bajo AM-(10,1.0).

En cualquier caso, hemos obtenido un algoritmo basado sobre un generacional pero con resultados mucho mejores que él. Esto nos confirma que aunque los generacionales estaban funcionando de manera pobre, el aplicar búsquedas locales que concreten el extremo local al que se acerca cada individuo mejora los resultados notablemente.

7.6. AM-(10,0.1)

Esta versión de un algoritmo memético vuelve a utilizar el esquema generacional evolutivo y a elegir el operador de cruce BLX. Sin embargo, en lugar de aplicar la búsqueda

local sobre cada uno de los individuos de la población, elegimos de manera aleatoria, con probabilidad 0.1, si aplicarla sobre cada uno de los individuos.

Cuadro 6: Algoritmo AM-(10,0.1) en el problema del APC

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
Partición 1	0.719	0.806	0.762	59.041	0.800	0.818	0.809	19.580	0.722	0.841	0.782	51.512
Partición 2	0.797	0.736	0.766	62.561	0.850	0.909	0.880	19.145	0.685	0.750	0.718	52.803
Partición 3	0.781	0.792	0.786	59.715	0.900	0.818	0.859	19.617	0.722	0.773	0.747	54.172
Partición 4	0.812	0.667	0.740	62.793	0.600	0.864	0.732	19.095	0.796	0.864	0.830	52.732
Partición 5	0.766	0.833	0.799	60.239	0.743	0.909	0.826	18.748	0.765	0.795	0.780	54.859
Media	0.775	0.767	0.771	60.870	0.779	0.864	0.821	19.237	0.738	0.805	0.771	53.216

Los resultados son ligeramente peores, y aunque no parecen suficientemente fuertes como para no atribuirlos a la aleatoriedad, podríamos intuir que gastar menos búsquedas locales no está ayudando al algoritmo memético.

7.7. AM-(10,0.1mej)

La última versión del algoritmo memético es similar a la anterior pero aplica de manera más selecta la optimización por búsqueda local solo en la mejor de las instancias.

Cuadro 7: Algoritmo AM-(10,0.1mej) en el problema del APC

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
Partición 1	0.641	0.875	0.758	56.271	0.875	0.909	0.892	17.944	0.741	0.727	0.734	51.844
Partición 2	0.766	0.806	0.786	59.840	0.925	0.864	0.894	18.073	0.759	0.727	0.743	52.675
Partición 3	0.703	0.722	0.713	64.421	0.750	0.864	0.807	18.174	0.778	0.841	0.809	56.189
Partición 4	0.719	0.792	0.755	59.736	0.725	0.909	0.817	17.677	0.722	0.795	0.759	55.535
Partición 5	0.734	0.736	0.735	58.176	0.714	0.864	0.789	18.276	0.765	0.705	0.735	57.986
Media	0.713	0.786	0.749	59.689	0.798	0.882	0.840	18.029	0.753	0.759	0.756	54.846

Aunque los resultados vuelven a ser una variación de los anteriores, es muy reseñable que en este caso, donde solo aplicamos búsqueda local sobre el mejor, obtengamos resultados similares. Esto sugiere que si necesitáramos reducir las evaluaciones de la función objetivo, podríamos intentar reducir el número de individuos en los que se aplica la búsqueda local sin incurrir necesariamente en pérdidas de calidad importantes.

7.8. AM-div

Nuestra propuesta para solucionar el problema de la falta de diversidad pasa por cambiar el parámetro del cruce BLX a un valor ligeramente mayor ($\alpha = 0.5$), intentando así que los intervalos en los que se pueden mover los valores del cruce sean más amplios; e incrementar hasta 0.01 la posibilidad de mutación.

Cuadro 8: Algoritmo AM-(10,0.1mej) en el problema del APC

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
Partición 1	0.641	0.806	0.723	62.723	0.750	0.909	0.830	19.493	0.741	0.841	0.791	58.903
Partición 2	0.828	0.806	0.817	63.813	0.800	0.909	0.855	18.738	0.685	0.841	0.763	58.608
Partición 3	0.734	0.819	0.777	62.888	0.800	0.818	0.809	19.385	0.852	0.818	0.835	59.931
Partición 4	0.844	0.806	0.825	63.577	0.550	0.909	0.730	19.104	0.796	0.841	0.819	59.331
Partición 5	0.828	0.819	0.824	62.085	0.743	0.909	0.826	19.221	0.725	0.795	0.760	59.402
Media	0.775	0.811	0.793	63.017	0.729	0.891	0.810	19.188	0.760	0.827	0.794	59.235

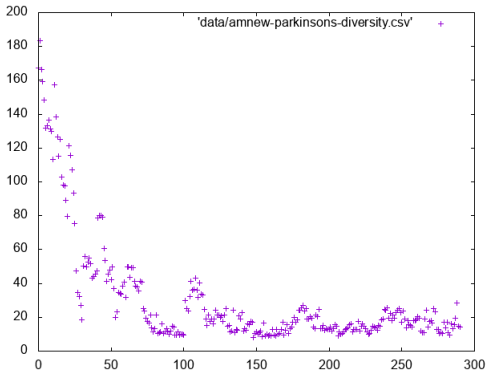

Figura 5: Diversidad de la población en el tiempo en Parkinsons bajo AM-div.

Hemos obtenido un algoritmo tan bueno como el mejor de los meméticos y que incluso lo mejora en el conjunto de datos "Spectf". El valor exacto de los dos ajustes de parámetros lo hemos obtenido empíricamente buscando que facilitaran mantener una ligera diversidad hasta el final de la ejecución; pero una propuesta futura podría ser automatizar esta elección de parámetros tras comprobar cómo funciona sobre varios conjuntos de datos y usando la medida de diversidad propuesta como criterio para la elección.

7.9. Resultados globales y comparación

Cuadro 9: Resultados globales en el problema del APC

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
1NN	0.816	0.000	0.408	0.032	0.783	0.000	0.391	0.008	0.774	0.000	0.387	0.016
RELIEF	0.819	0.014	0.416	0.118	0.794	0.000	0.397	0.024	0.767	0.000	0.383	0.055
BL	0.628	0.969	0.799	15.354	0.391	0.973	0.682	1.192	0.622	0.954	0.788	9.458
BL2	0.738	0.800	0.769	13.177	0.655	0.909	0.782	0.898	0.768	0.855	0.811	7.488
AGE-CA	0.747	0.703	0.725	57.608	0.744	0.764	0.754	16.566	0.802	0.668	0.735	54.469
AGE-BLX	0.744	0.831	0.787	53.886	0.786	0.882	0.834	17.787	0.768	0.791	0.779	36.351
AGG-CA	0.738	0.645	0.691	32.838	0.785	0.382	0.583	9.434	0.761	0.627	0.694	30.513
AGG-BLX	0.763	0.642	0.702	59.889	0.745	0.536	0.641	16.059	0.794	0.605	0.699	53.961
AM-(10,1.0)	0.769	0.817	0.793	57.391	0.809	0.882	0.845	16.333	0.734	0.791	0.763	57.729
AM-(10,0.1)	0.775	0.767	0.771	60.870	0.779	0.864	0.821	19.237	0.738	0.805	0.771	53.216
AM-(10,0.1mej)	0.713	0.786	0.749	59.689	0.798	0.882	0.840	18.029	0.753	0.759	0.756	54.846
AM-div	0.775	0.811	0.793	63.017	0.729	0.891	0.810	19.188	0.760	0.827	0.794	59.235

7.10. Conclusiones

La importancia de alcanzar equilibrio entre exploración y explotación se ha manifestado en los resultados de estas ejecuciones. Un problema común en los algoritmos ha sido la falta de diversidad, algunos pasando demasiado tiempo en un conjunto muy pequeño de las soluciones. Las correcciones en esta dirección han servido para mejorar los resultados.

Referencias

[Ada96] Stephen Adams. Implementing sets efficiently in a functional language. *Journal of Functional Programming*, 1996. https://groups.csail.mit.edu/mac/users/adams/BB/.

[Lei02] Daan Leijen. Data.set. *Hackage*, 2002. https://hackage.haskell.org/package/containers-0.5.11.0/docs/Data-Set.html.

[Les08] Roman Leshchinsky. Data.vector. *Hackage*, 2008. https://hackage.haskell.org/package/vector-0.12.0.1/docs/Data-Vector.html.

[P+03] Simon Peyton-Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[Smi92] Paul Smith. Gnu make. *GNU Operating system*, 1992. https://www.gnu.org/software/make/.

[Yor16] Brent Yorgey. Control.monad.random.class. *Hackage*, 2016. https://hackage.haskell.org/package/MonadRandom-0.5.1/docs/Control-Monad-Random-Class.html.