UNIVERSIDAD DE GRANADA

Práctica 1b: Algoritmos voraces y búsqueda local para el Aprendizaje de Pesos en Características

Metaheurísticas

Curso Académico 2023-2024

Problema Escogido: Aprendizaje de Pesos en Características (APC)

Algoritmos Considerados: 1-NN, RELIEF, Búsqueda Local

Nombre del Estudiante: Pablo Olivares Martínez

DNI: 24411228V

Email: pablolivares@correo.ugr.es

Grupo: 1, Horario de Prácticas: Jueves, 17:30 - 19:30

Pablo Olivares Martínez

7 de abril de 2024

ÍNDICE

1. DESCRIPCIÓN DEL PROBLEMA

En el ámbito del aprendizaje automático supervisado, se presenta el desafío de optimizar la clasificación de objetos o instancias dentro de un conjunto de datos $\mathscr{D} = (\mathbf{x}_i, y_i)_{i=1}^{M}$, siendo M el número de muestras. Aquí, $\mathbf{x}_i$ representa un vector de características o atributos de la i-ésima instancia, e y_i es la etiqueta o clase asociada a $\mathbf{x}_i$. El objetivo es diseñar un sistema clasificador que, mediante la utilización de estas instancias previamente etiquetadas, sea capaz de clasificar nuevas instancias de manera automática y precisa.
Una de las técnicas más utilizadas para la clasificación es el clasificador k-NN (k vecinos más cercanos), el cual se basa en la premisa de que instancias similares tienden a encontrarse en la misma categoría. A pesar de su simplicidad y efectividad, el rendimiento del k-NN puede verse bastante afectado por la importancia de cada característica. Aquí es donde entra en juego el problema del **Aprendizaje de Pesos en Características** (APC), que busca asignar un peso a cada característica, ponderando su importancia en la decisión de clasificación.
El APC se formaliza como la tarea de encontrar un vector de pesos $\mathbf{w} \in \mathbb{R}^N$, donde cada peso w_i ajusta la contribución de la i-ésima característica en la medida de distancia utilizada por el clasificador, de tal manera que se mejore la precisión del clasificador k-NN. Más generalmente, se busca encontrar una **solución** $\mathbf{w}^* \in \mathbb{R}^N$ que maximice el valor de una **función objetivo** $F : \mathbb{R}^N \to \mathbb{R}$ al que llamaremos **valor fitness**. En nuestro caso, la función objetivo buscará maximizar la precisión de la clasificación, `tasaClas`, así como encontrar la solución más reducida, `tasaRed`, mediante una ponderación de ambos términos. Formalmente, buscamos encontrar

$$\mathbf{w}^* = \arg\max_{\mathbf{w}\in\mathbb{R}^N} F(\mathbf{w})$$

tal que

$$F(\mathbf{w}) = \alpha \cdot \texttt{tasaClas} + (1-\alpha) \cdot \texttt{tasaRed}$$

donde a α le asignaremos un valor entre 0 y 1.
Durante el desarrollo de la práctica, buscaremos encontrar la mejor solución al problema APC de los conjuntos de datos **ecoli**, **parkinsons** y **breast-cancer** mediante el uso de los siguientes algoritmos de búsqueda: 1-**NN**, **RELIEF** y **búsqueda local**.

2. APLICACIÓN DE LOS ALGORITMOS

2.1. REPRESENTACIÓN DE LAS SOLUCIONES

El esquema de representación de nuestras soluciones será similar al comentado anteriormente, pero restringiendo nuestro espacio al intervalo cerrado $[0,1]$. Concretamente, una solución será un vector $\mathbf{w} \in [0,1]^N$, donde N indica el número de atributos del conjunto de datos a tratar. En particular,

$$\mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{pmatrix}$$

donde $w_1, \ldots, w_N$ son números reales entre 0 y 1. Si bien esta representación es la idónea, lo cierto es que los ordenadores no son capaces de representar soluciones reales, pues todos sus tipos numéricos son finitos. Por tanto, nuestro espacio de soluciones se reducirá a trabajar sobre el conjunto de números en coma flotante de precisión con 4 bytes, `float`, entre 0 y 1. La representación vectorial vendrá dada por una estructura de datos de tipo `vector`.
Para una mayor abstracción y mayor flexibilidad, se ha implementado una clase `Solution` que encapsula dicha implementación añadiendo constructores.

2.2. Función objetivo

De igual manera, debido a las limitaciones de los ordenadores y los requisitos del problema, nuestra función objetivo queda restringida a los números en coma flotante entre 0 y 1. Es decir, buscaremos optimizar la función $F : D^N \rightarrow D$ donde el dominio $D = \texttt{float} \cap [0,1]$ y N representa el número de características del conjunto de datos.
Para la obtención del valor fitness utilizaremos dos variables obtenidas a partir de **w**: la tasa de clasificación, `tasaClas`, y la tasa de reducción, `tasaRed`.

Algorithm 1 Calcular tasa de clasificación

```
1:  function CALCULATECLASSIFICATIONRATE(datos, solucion)
2:      prediccionesCorrectas ← 0
3:      solucionReducida ← REDUCIRSOLUCION(solucion)   ▷ Si w_i < 0,1, entonces w_i = 0
4:      for each elemento in datos do
5:          clasePredicha ← 1NNCLASIFICAR(elemento, solucionReducida)
6:          if clasePredicha == elemento.clase then
7:              prediccionesCorrectas ← prediccionesCorrectas + 1
8:          end if
9:      end for
10:     tasaClas ← (prediccionesCorrectas / longitud(datos)) × 100,0
11:     return tasaClas
12: end function
```

La tasa de clasificación representará el valor de precisión obtenido en la clasificación del conjunto a evaluar mediante el algoritmo 1-NN. Este algoritmo de clasificación es un caso particular del ya conocido k-NN, que consiste en encontrar a los k vecinos más cercanos y asignar a la muestra la clase con mayor número de ocurrencias entre los vecinos. En nuestro caso emplearemos una variante donde las distancias entre los atributos de la muestra a clasificar vendrán ponderadas por los pesos de la solución dada. Si bien 1-NN admite el uso de distintas distancias, nosotros emplearemos la **distancia euclídea ponderada** $\text{dist} : D^N \times \texttt{float}^N \times \texttt{float}^N \rightarrow \texttt{float}$ tal que si $\mathbf{u}, \mathbf{v} \in \texttt{float}^N$,

$$\text{dist}(\mathbf{w}, \mathbf{u}, \mathbf{v}) = \sum_{i=0}^{N} w_i \cdot (u_i - v_i)^2.$$

Algorithm 2 Clasificación 1-NN

```
1: function CLASSIFY1NN(elemento, solucion)
2:     distanciaMinima ← ∞
3:     etiquetaMasCercana ← Cadena Vacía
4:     for desde i = 0 hasta longitud(datos) − 1 parallel do
5:         dist ← DISTANCIAEUCLIDEA(datos[i], elemento, solucion)
6:         if dist < distanciaMinima then
7:             distanciaMinima ← dist
8:             etiquetaMasCercana ← datos[i].etiqueta
9:         end if
10:    end for
11:    return etiquetaMasCercana
12: end function
```

Durante el inicio del desarrollo del proyecto se valoró implementar k-NN utilizando $k = 1$. De esta forma, el proyecto sería más flexible y permitiría comparar la eficacia de soluciones para otros valores de k. Sin embargo, se notó una clara diferencia de rendimiento entre ambas implementaciones cuando $k = 1$ por lo que se descartó emplearlo. Sin embargo, la clase sigue implementada como `KNNClassifier` en caso de que se decidiera usar en algún momento. Su implementación es la siguiente:

Algorithm 3 Clasificación k-NN

```
1: function CLASSIFYKNN(instanciaTest, solution)
2:     distancias ← lista vacía
3:     for i = 0 hasta longitud(datos.instancias) − 1 do
4:         instancia ← datos.instancias[i]
5:         dist ← DISTANCIAEUCLIDEA(instancia, instanciaTest, solucion)
6:         Añadir (dist, instancia.clase) a distancias
7:     end for
8:     Ordenar distancias por orden ascendente
9:     contEtiquetas ← nueva tabla hash
10:    for i = 0 hasta mín(k, longitud(distancias)) − 1 do
11:        clase ← distancias[i].clase
12:        contEtiquetas[clase] ← contEtiquetas[clase] + 1
13:    end for
14:    masComun ← GETMAXCOUNTLABEL(contEtiquetas)
15:    return masComun
16: end function
```

Aquí `getMaxCountLabel` es un método sencillo que devuelve la etiqueta más común entre los pares dados.
En cuanto a la tasa de reducción, se obtendrá calculando el promedio entre los pesos con valor menor que 0,1 y la dimensión del espacio del conjunto de datos, N. Esto se debe a que

a la hora de realizar operaciones con nuestra solución, descartaremos tener en cuenta los atributos cuyo peso sea menor a este valor, fomentando así la obtención de soluciones que se centren en encontrar los atributos más importantes.
Para facilitar la comprensión y el análisis de los resultados, así como para intentar minimizar los errores de precisión, multiplicaremos ambas variables por 100, de forma que nuestro valor fitness se encuentre siempre entre 0 y 100, interpretándolas como un porcentaje.

Algorithm 4 Calcular Tasa de Reducción

```
1: function CALCULATEREDUCTIONRATE(pesos)
2:     peso ← pesos
3:     atributosReducidos ← 0
4:     for each peso in pesos do
5:         if peso < 0,1 then
6:             atributosReducidos ← atributosReducidos + 1
7:         end if
8:     end for
9:     tasaRed ← (atributosReducidos / longitud(pesos)) × 100,0
10:    return tasaRed
11: end function
```

Nuestro objetivo en esta práctica consiste en obtener una solución que busque tanto maximizar la precisión de clasificación del algoritmo 1-NN como reducir lo máximo posible el número de características a emplear. Como buscamos que la tasa de clasificación sea más importante que la reducción de la solución, hemos asignado una ponderación α de 0,75 al primero y de $1-\alpha$, es decir 0,25, al segundo.

Algorithm 5 Función objetivo

```
1: function CALCULATEFITNESS(tasaClas, tasaRed)
2:     return α × tasaClas + (1 − α) × tasaRed
3: end function
```

También se ha implementado una leve modificación de la función objetivo que recibe como parámetros el conjunto de datos y la solución, encapsulando la ejecución del cálculo de `tasaClas` y `tasaRed`.

Algorithm 6 Cálculo del Fitness usando datos y solución

```
1: function CALCULATEFITNESS(datos, solucion)
2:     tasaClas ← CALCULATECLASSIFICATIONRATE(datos, solucion)
3:     tasaRed ← CALCULATEREDUCTIONRATE(solucion)
4:     fitness ← CALCULATEFITNESS(tasaClas, tasaRed)
5:     return fitness
6: end function
```

Durante el proceso de búsqueda de soluciones, si se calcula la tasa de clasificación emplean-

do el mismo conjunto de entrenamiento, tendremos que ésta será siempre del 100 %. El motivo es que si intentamos clasificar una instancia mediante 1-NN respecto a un conjunto al que pertenece, su vecino más cercano siempre será él mismo, por lo que la distancia será 0 y siempre acertará. Para solventar este problema, emplearemos el método de validación cruzada *leave one out*, evaluando cada instancia descartándola del conjunto empleado en 1-NN.

Algorithm 7 Validación Cruzada Leave-One-Out

```
1: function LEAVEONEOUTCROSSVALIDATION(dataset, solution)
2:     prediccionesCorrectas ← 0                                   ▷ Atomic
3:     solucionReducida ← REDUCIRSOLUCION(solucion)
4:     for i = 0 to longitud(datos) − 1 parallel do
5:         elemento_i ← dataset.items[i]
6:         clasePredicha ← CLASIFICAR1NNEXCLUYENTE(elemento_i, i, solucionReducida)
7:         if clasePredicha == elemento_i.clase then
8:             prediccionesCorrectas.fetch_add(1)      ▷ Incremento seguro en paralelo
9:         end if
10:    end for
11:    tasaClas ← (prediccionesCorrectas / longitud(datos)) × 100,0
12:    return tasaClas
13: end function
```

Para poder usar 1-NN en la tarea de clasificación durante el entrenamiento, los siguientes métodos se han sustituido realizando la misma función salvo las especificaciones indicadas:

- `calculateFitnessLeaveOneOut` sustituye a `calculateFitness`.
- En `calculateFitnessLeaveOneOut`, `leaveOneOutCrossValidation` sustituye a `calculateClassificationRate`.
- En `calculateFitnessLeaveOneOut`, `classifyExcludingIndex` sustituye a `classify`, simplemente saltando la comprobación de distancia i-ésima indicada.

Además, nótese el uso de directivas de paralelización. Su uso ha sido bastante relevante en el desarrollo de la práctica, pues las numerosas evaluaciones de clasificación y el tamaño del conjunto de datos de entrenamiento suponían un gran coste computacional. Dado que el mayor tiempo de cómputo se centraba en un gran número de llamadas a iteraciones en bucle de lectura, paralelizar dicha parte del código supuso una gran mejora en rendimiento.

3. ALGORITMOS DE BÚSQUEDA

A continuación describiremos los algoritmos de búsqueda de soluciones empleados: 1-NN, el algoritmo greedy RELIEF y la búsqueda local del primer mejor.
Por motivos que comentaremos más adelante en esta memoria, se ha implementado una clase abstracta `Algorithm` de la que son hijos todos los algoritmos a evaluar. Todos ellos contienen el método `run`, que ejecuta el algoritmo para un conjunto de datos dado y devuelve la solución junto a su valor fitness sobre dicho conjunto.

3.1. 1-NN

El primer algoritmo empleado es 1-NN. Si bien este método no consiste realmente en un método de búsqueda de soluciones, nuestro objetivo será emplearlo como una ejecución de referencia. De esta forma, estudiaremos el valor fitness obtenido de la clasificación 1-NN usando la solución que no altere los atributos de la muestra a clasificar. Es decir, la solución trivial donde todos los pesos valen 1. Para ello, se ha imlpementado por homogeneidad la clase `Naive`, que al ejecutarse devuelve dicha solución.

Algorithm 8 Ejecución del Algoritmo 1-NN

```
1: function RUN(dataset)
2:     solucion ← Crear solución con pesos a 1
3:     return EVALUAR(solucion)
4: end function
```

3.2. RELIEF

A continuación buscaremos una solución mediante el uso de un algoritmo voraz denominado RELIEF. Comenzamos inicializando la solución a 0. El algoritmo consiste en encontrar para cada muestra del conjunto de entrenamiento las siguientes dos instancias:

- **Amigo más cercano**: instancia del conjunto de datos de entrenamiento de la misma clase que esté a menor distancia de la instancia actual sin tenerse en cuenta a sí misma.
- **Enemigo más cercano**: instancia del conjunto de datos de entrenamiento de distinta clase que esté a menor distancia de la instancia actual.

En caso de no haber amigos o enemigos significa que, o bien es la única instancia de la clase, o bien el conjunto de datos de entrenamiento solo contiene dicha instancia (lo cual es absurdo). En dicho caso, omitiríamos el cómputo de esa iteración. A continuación, obtendremos la diferencia entre las componentes del amigo y enemigo y las sumaremos al vector de pesos. Tras iterar por todas las instancias, normalizamos el vector de pesos entre 0 y el máximo, descartando los valores negativos.

Algorithm 9 Ejecución del Algoritmo RELIEF

```
1: function RUN(dataset)
2:     solucion ← Crear solución con pesos a 0
3:     for cada instancia en dataset do
4:         Inicializar variables para amigo y enemigo más cercano
5:         for cada otraInstancia en dataset do
6:             if instancia es igual a otraInstancia then
7:                 continue                          ▷ Ignorar la misma instancia
8:             end if
9:             distancia ← DISTANCIAEUCLIDEA(instancia, otraInstancia)
10:            Actualizar amigo o enemigo más cercano según corresponda
11:        end for
12:        if amigo y enemigo más cercanos encontrados then
13:            for i = 0 hasta longitud(solucion) − 1 do
14:                Ajustar pesos de solucion basado en diferencias
15:            end for
16:        end if
17:    end for
18:    NORMALIZAR(solucion)
19:    return EVALUAR(solucion, dataset)
20: end function
```

3.3. BÚSQUEDA LOCAL

A continuación describiremos al algoritmo metaheurístico empleado: la búsqueda local del primer mejor. Esta implementación de la búsqueda local se caracteriza por ir generando vecinos de manera secuencial hasta encontrar uno que tenga un mejor valor fitness. En dicho caso, se descarta la solución anterior y se emplea la nueva para las siguientes iteraciones hasta cumplir con algún criterio de parada. En nuestro caso, detendremos la búsqueda si se han generado ya $20 \times n$ vecinos, donde n es el número de atributos, o si se han realizado ya un total de 15000 evaluaciones.
Comenzaremos generando los pesos de la solución inicial a partir de una distribución uniforme entre 0 y 1 que notaremos por $\mathcal{U}(0,1)$. En cuanto a la generación del vecindario, obtendremos una permutación de los índices de los pesos, iremos generando cada vecino sumando un valor obtenido de una distribución normal entre $\mathcal{N}(0,\sigma)$, con $\sigma = 0{,}3$, en la componente del vector de pesos que indique la posición actual de la permutación.

Algorithm 10 Generación de Vecino

```
1: function GENERARVECINO(solucion, i)
2:     Sumar a solucion[i] un valor aleatorio con distribución N(0, 0,3)
3:     return solucion ajustada entre 0,0 y 1,0
4: end function
```

Algorithm 11 Ejecución del algoritmo Búsqueda Local Primer Mejor

```
1: function RUN(datos)
2:     Inicializar solucionActual con distribución 𝒰(0,1)
3:     evaluaciones ← 0, numVecinos ← 0
4:     while no se alcanzan maxEvaluaciones o maxNumVecinos do
5:         Generar índices aleatorios de características
6:         for cada índice sin mejora do
7:             Generar vecino y calcular su fitness
8:             if fitness de vecino mejora solucionActual then
9:                 Actualizar solucionActual y resetear numVecinos
10:                Establecer indiceConMejora a verdadero
11:            end if
12:        end for
13:        Incrementar numVecinos si no hay mejora
14:    end while
15:    return EVALUAR(solucionActual)
16: end function
```

También se ha implementado la búsqueda local del mejor. Como el espacio de soluciones es teóricamente de variable real, un vecindario tendría infinitas soluciones. En consecuencia, realmente se ha implementado un punto medio entre la búsqueda local del primer mejor y la del mejor, generando un número finito de vecinos. Además, se omite la comprobación de máximo número de vecinos, pues en cada iteración se genera dicha cantidad. La implementación es análoga a la del primer mejor, pero generando $20 \times n$ vecinos y escogiendo el mejor. Para mejorar su eficiencia, se ha aplicado paralelización en el bucle de la búsqueda del mejor vecino. Esta implementación se puede encontrar en la clase `BestLocalSearch`.
En cuanto a la generación aleatoria de números para ambas distribuciones, se ha usado la implementación de la librería `Random.hpp` realizada por el usuario de GitHub *effolkronium*. En ella se proporciona una manera intuitiva y de calidad para trabajar con números pseudo-aleatorios en C++.

4. ALGORITMOS DE COMPARACIÓN

El método de comparación empleado ha sido la validación cruzada. Para cada conjunto de datos se han tomado las particiones aportadas por el profesor y se ha utilizado como conjunto de pruebas la i-ésima partición de la i-ésima ejecución del bucle y el resto como conjunto de entrenamiento, donde $i \in \{1,\dots,5\}$.
En cuanto a la normalización, se ha obtenido el máximo y el mínimo de cada componente del conjunto de datos completo. Si bien una primera aproximación se realizó obteniendo el máximo y el mínimo solamente del conjunto de entrenamiento en cada una de las 5 iteraciones, el hecho de que dichos valores no contemplasen todo el conjunto podía llevar a errores de cálculo en la normalización. Esta **no es una buena práctica**, pues entraríamos en un caso de *data snooping*. En un entorno profesional, el máximo y el mínimo de cada atributo debería

Algorithm 12 Ejecución Principal

```
1: for testIndex ← 0 hasta 4 do                                          ▷ 5 Particiones
2:     for i ← 0 hasta 4 do
3:         if i ≠ testIndex then CARGARDATOS(train, i)
4:         end if
5:         if i = testIndex then CARGARDATOS(test, i)
6:         end if
7:         NORMALIZARDATOS(train, test)
8:         eval ← CREAREVALUACION(train, parametros)
9:         algoritmo ← CREARALGORITMO(nombre, parametros, eval)
10:        PROCESARPARTICION(csvFile, conjuntoDatos, testIndex +
   1, train, test, eval, algoritmo, semilla, logResults)
11:    end for
```

estar contemplado en el conjunto de entrenamiento o en su defecto, ser proporcionado por un experto. Sin embargo, para que nuestros resultados coincidan con los exigidos se realizará de la manera indicada.

Algorithm 13 Procesamiento de Partición

```
1: function PROCESARPARTICION(csvFile, datasetName, fold, train, test, eval,
   algorithm, seed, logResults)
2:     start ← INICIARTEMPORIZADOR
3:     evaluatedSolution ← EJECUTARALGORITMO(algorithm, train)
4:     duration ← FINALIZARTEMPORIZADOR(start)
5:     trainRate ← EVALUARLEAVEONEOUT(train, solution)
6:     testRate ← EVALUARTASACLASIFICACION(test, solution)
7:     reductionRate ← EVALUARTASAREDUCCION(solution)
8:     fitness ← CALCULARFITNESS(testRate, reductionRate)
9:     if logResults then
10:        GUARDARSOLUCION(EvaluatedSolution.solution)
11:        GUARDARREGISTROSDEENTRENAMIENTO
12:        REGISTRARRESULTADOS(csvFile, fold, trainRate, testRate, reductionRate,
   fitness, duration, seed)
13:    end if
14: end function=0
```

La ejecución principal se divide en dos partes. Primero, un bucle que genera las particiones, prepara los datos y crea los objetos de ejecución y evaluación necesarios. En su interior, se llama a una función que se encarga de la ejecución, medición y registro de los resultados de dicha ejecución. Nótese que la ejecución del programa principal de comparación tan sólo considera un único conjunto de datos y algoritmo, dejando otras ejecuciones compuestas al guión `run.sh`, como veremos más adelante.
Dado que almacenamos los datos de ejecución, he decidido no calcular el valor prome-

dio de los resultados para hacerlo directamente en la generación de las tablas mediante `generate_algorithm_table.py` y `generate_global_table.py` en `scripts`.

5. Procedimiento e instrucciones

Para el desarrollo de la práctica se ha empleado el lenguaje de programación C++ [2], en particular la versión C++17, por ser un lenguaje muy eficiente que permite emplear varios paradigmas de programación diferentes. Se ha seguido la estructura típica de directorios de un proyecto de C++, contando con `src`, `include`, `obj`, `bin` y `docs`. Además, se han incluido otros directorios típicos en los experimentos de ciencia de datos para almacenar salidas como `outputs`, almacenar los conjuntos de datos `data` o para contener scripts útiles para el proyecto como `scripts`. Mi objetivo ha sido crear un proyecto lo suficientemente flexible y organizado para permitir realizar modificaciones sobre este alterando la menor parte posible del código. En consecuencia, el proyecto abstrae el mayor número posible de componentes para permitir alteraciones futuras haciendo uso de la programación orientada a objetos, polimorfismo o patrones de diseño como Factory. A continuación se mencionan las principales clases:

- Los datos leidos se abstraen en las clases `DataItem` y `DataSet`.
- La clase `Solution` abstrae la implementación concreta de las soluciones.
- La clase `Algorithm`, que abstrae la lógica de los algoritmos de búsqueda implementados para permitir el uso de polimorfismo. Sus hijos son `Naive` (devuelve la solución trivial de todo 1), `Relief` y `LocalSearch`.
- La clase `Evaluation`, que crea un objeto de evaluación con su clasificador 1-NN a partir de un conjunto de datos y contiene todos los métodos de evaluación necesarios.
- Además, se ha añadido una estructura `EvaluatedSolution` que almacena tanto la solución como el valor fitness. Así el valor devuelto por el algoritmo cumple con las exigencias de la práctica.
- Se han creado diversas utilidades matemáticas y de números aleatorios que abstraen funciones básicas de las librerías `cmath` y `Random.hpp`, permitiendo cambiar su implementación fácilmente si fuese preciso. Entre ellas, encontramos funciones de normalización, cálculo de distintas distancias, etc. Las podemos encontrar en el directorio `utils` de `src` e `include`.

En cuanto a la lectura y escritura de datos se han empleado varias funciones. Para la lectura de los archivos ARFF se ha implementado `readARFF`, que dada una instancia de `DataSet`, inserta las nuevas entradas a continuación. Para la escritura de registros de ejecución se ha implementado `logResult` y para guardar las soluciones asociadas tenemos `saveSolutionToJSON` usando `json.hpp` del usuario de GitHub *nlohmann*. Cada solución tiene un identificador único obtenido a partir del reloj del sistema que se almacena con su ejecución. Finalmente, `saveFitnessRecords` se ha empleado para ir almacenando el valor fitness obtenido en cada

iteración durante la búsqueda de la solución. Para más información sobre las clases, métodos y funciones implementadas consúltese el directorio `include` con sus ficheros ampliamente comentados.
La compilación se realiza a partir de la directiva `make` de GNU Make [6]. En cuanto a la ejecución, es bastante sencilla y personalizable:

```
./bin/metaheuristics --algorithm=ALGORITHM_NAME [--param1=value1 ...]
    --dataset=DATASET_NAME [--log]
```

Aquí los corchetes denotan que los parámetros son opcionales. Como vemos, para ejecutar el programa sólo necesitamos aportar el nombre del algoritmo (`1nn`, `relief` o `local-search`) y del conjunto de datos (`ecoli`, `parkinsons` o `breast-cancer`). En cuanto a los hiperparámetros de cada algoritmo, en caso de no aportarlos se ejecutarían los valores por defecto exigidos en la práctica. El único algortimo que los acepta es `local-search` siendo estos:

- `maxNeighbors`: Número máximo de vecinos a generar según el número de atributos. Por defecto 20.
- `maxEvaluations`: Número máximo de evaluaciones. Por defecto 15000.
- `variance`: Varianza empleada por la distribución normal. Por defecto 0,3.

Más parámetros que se pueden emplear independientemente del algoritmo son `alpha`, que por defecto es 0,75; `reductionThreshold`, que por defecto es 0,1 y `seed`, que en caso de no asignarse se genera de manera aleatoria entre 0 y 1000. Además, si añadimos `--log` almacenaremos todos los datos de la ejecución.
Por último se proporciona `run.sh` en `scripts`. Este guión permite ejecutar de manera todavía más flexible el programa: permitiendo no aportar el conjunto de datos para ejecutar los tres, no aportar el algoritmo para ejecutar los tres con su configuración por defecto o no ofrecer parámetros para ejecutar todo por defecto.

```
./scripts/run.sh [--algorithm=ALGORITHM_NAME] [--dataset=DATASET_NAME]
    [--param1=value1 ...] [--log]
```

Un ejemplo de ejecución sería el siguiente:

```
./scripts/run.sh --algorithm=local-search --dataset=parkinsons
    --maxEvaluations=7000 --log --seed=46
```

El orden de los parámetros no es relevante. Es importante que los archivos ARFF estén en una carpeta llamada `data` en el directorio raíz del proyecto. Si bien en el proyecto original se asume que el binario `bin/metaheuristics` está en el directorio raíz en `run.sh`, este hecho se adapta en la entrega a las exigencias del formato, por lo que la ejecución funciona tal y como se ha descrito sustituyendo `./scripts/run.sh` simplemente por `./run.sh`.

6. EXPERIMENTOS Y ANÁLISIS DE RESULTADOS

Se van a evaluar las soluciones generadas por los algoritmos de búsqueda en los conjuntos de datos ecoli, parkinsons y breast-cancer mediante una validación cruzada de 5 particiones tal y como se ha comentado anteriormente. Los conjuntos de datos constan de las siguientes características:

- **Ecoli** [3]: Conjunto de datos sobre la bacteria *Escherichia coli* para detectar la posición de proteínas. Consta de 366 instancias con 8 atributos y 8 clases.
- **Parkinson** [4]: conjunto de datos de medidas de la voz para la detección de la enfermedad de Parkinson. Consta de 195 instancias con 23 atributos y 2 clases.
- **Breast-Cancer** [7]: Conjunto de datos para la detección de cancer de mama. Consta de 569 instancias con 31 atributos y 2 clases.

Con el fin de evitar sesgar los resultados buscando una semilla que se comporte "bien"para mi ejecución e inspirado por el proyecto LavaRand, he decidido implementar un algoritmo similar al de la ya expirada patente [5] sobre generación de números aleatorios en sistemas caóticos para generarla. Como consecuencia, usaremos la semilla generada por la foto de mi perro con valor 138508859 a través de `chaotic_seed_generator.py` en `scripts`. Es una buena práctica utilizar una semilla diferente en cada ejecución con el fin de emplear secuencias pseudoaleatorias diferentes para cada una. Sin embargo, como mi programa ejecuta las cinco validaciones de manera secuencial y estamos usando un generador pseudoaleatorio de buena calidad, en principio no es necesario el uso de nuevas semillas. Es decir, las cinco ejecuciones por algoritmo y conjunto de datos utilizan secuencias completamente diferentes para cada partición.

6.1. RESULTADOS INDIVIDUALES

Comencemos analizando los resultado obtenidos en la Tabla 6.1. Como podemos apreciar, el clasificador de vecino más cercano muestra una precisión bastante alta de clasificación. Sin embargo, su reducción nula perjudica notablemente al valor fitness, provocando caídas de hasta el 20 % en la bondad de la solución.

	Ecoli				Parkinsons				Breast-cancer			
	% clas	% red	Fit.	T(s)	% clas	% red	Fit.	T(s)	% clas	% red	Fit.	T(s)
Partición 1	80.00	0.00	60.00	7.39e-02	97.50	0.00	73.12	7.38e-02	97.39	0.00	73.04	9.47e-02
Partición 2	72.86	0.00	54.64	3.96e-03	87.50	0.00	65.62	**8.02e-04**	96.52	0.00	72.39	**3.04e-03**
Partición 3	82.35	0.00	61.76	**1.91e-03**	97.50	0.00	73.12	3.51e-03	92.17	0.00	69.13	1.59e-02
Partición 4	83.82	0.00	62.87	1.23e-02	**100.00**	0.00	**75.00**	6.87e-03	**98.26**	0.00	**73.70**	1.47e-02
Partición 5	**85.00**	0.00	**63.75**	2.58e-03	91.43	0.00	68.57	8.95e-03	93.58	0.00	70.18	1.33e-02
Media	80.81	0.00	60.61	1.89e-02	94.79	0.00	71.09	1.88e-02	95.59	0.00	71.69	2.83e-02
Desv. Est.	4.82	0.00	3.61	0.03	5.15	0.00	3.87	0.03	2.60	0.00	1.95	0.04

Cuadro 6.1: Resultados de ejecución de 1-NN

En cuanto a la ejecución de RELIEF, la Tabla 6.2 nos muestra una leve mejoría del valor fitness respecto a 1-NN. Si bien la tasa de clasificación es similar a la proporcionada por la so-

lución trivial, el aumento en la reducción de las soluciones marca la diferencia en los valores ofrecidos por la función objetivo. Este último hecho es especialmente curioso debido a que el algoritmo RELIEF no proporciona ninguna estrategia que se enfoque en la reducción de atributos. Este algoritmo pretende dar más valor a las características similares entre amigos cercanos e ignorar las de enemigos cercanos. En consecuencia, las características que consistentemente no contribuyen a la distinción entre clases cercanas tienden a ser devaluadas, lo cual puede interpretarse como una forma indirecta de reducción que hace que los conjuntos de datos con instancias más similares presenten un mayor grado de reducción. Este hecho se puede apreciar en la baja variabilidad que presenta dicha tasa.

	Ecoli				Parkinsons				Breast-cancer			
	% clas	% red	Fit.	T(s)	% clas	% red	Fit.	T(s)	% clas	% red	Fit.	T(s)
Partición 1	77.14	28.57	65.00	5.26e-01	97.50	0.00	73.12	4.78e-02	**97.39**	3.33	**73.88**	1.48e-01
Partición 2	74.29	28.57	62.86	7.99e-03	90.00	0.00	67.50	7.14e-03	**97.39**	3.33	**73.88**	3.01e-01
Partición 3	79.41	28.57	66.70	1.88e-02	95.00	0.00	71.25	**3.17e-03**	89.57	**13.33**	70.51	3.18e-02
Partición 4	80.88	28.57	67.80	1.06e-02	**100.00**	0.00	**75.00**	9.97e-03	**97.39**	0.00	73.04	**2.95e-02**
Partición 5	**86.67**	28.57	**72.14**	**7.49e-03**	91.43	0.00	68.57	8.42e-03	94.50	0.00	70.87	3.35e-02
Media	79.68	28.57	66.90	1.14e-01	94.79	0.00	71.09	1.53e-02	95.25	4.00	72.44	1.09e-01
Desv. Est.	4.63	0.00	3.47	0.23	4.15	0.00	3.11	0.02	3.41	5.48	1.63	0.12

Cuadro 6.2: Resultados de ejecución de RELIEF

Finalmente encontramos en la Tabla 6.3 los resultados de la búsqueda local realizada. Vemos que los resultados obtenidos son similares e incluso mejores que los anteriores. Si bien *Ecoli* muestra un valor fitness algo menor que en RELIEF, *Parkinsons* y *Breast-cancer* muestran una más que notable mejora. El motivo puede deberse a los pocos atributos de los que dispone *Ecoli*, haciendo más difícil aumentar la tasa de reducción sin que suponga un gran sacrificio en clasificación. Es más, nótese que el gran aumento en reducción de los dos últimos conjuntos ha sido determinante. Otra observación a tener en cuenta es que estos resultados muestran una mayor desviación típica en el valor fitness, sugiriendo que la solución inicial y el método de búsqueda influyen de gran manera al resultado final. Este hecho se ve reforzado por los resultados[1] de la Figura 6.1. Como podemos ver en la Figura 6.2, las soluciones tienen una convergencia a saltos, donde las soluciones evolucionan rápidamente en un muy breve número de ejecuciones y pasan una gran parte de la ejecución en el óptimo local[2].

	Ecoli				Parkinsons				Breast-cancer			
	% clas	% red	Fit.	T(s)	% clas	% red	Fit.	T(s)	% clas	% red	Fit.	T(s)
Partición 1	78.57	0.00	58.93	9.64e-01	97.50	72.73	91.31	4.19e+00	94.78	**90.00**	93.59	2.81e+01
Partición 2	72.86	**14.29**	58.21	**8.11e-01**	87.50	**77.27**	84.94	3.26e+00	**98.26**	**90.00**	**96.20**	**2.72e+01**
Partición 3	80.88	0.00	60.66	1.01e+00	**100.00**	68.18	92.05	3.21e+00	93.04	86.67	91.45	2.77e+01
Partición 4	88.24	**14.29**	69.75	1.18e+00	**100.00**	72.73	**93.18**	**3.04e+00**	93.91	73.33	88.77	2.78e+01
Partición 5	**86.67**	**14.29**	**68.57**	1.67e+00	91.43	**77.27**	87.89	3.28e+00	93.58	83.33	91.02	2.87e+01
Media	81.44	8.57	63.22	1.13e+00	95.29	73.64	89.87	3.40e+00	94.72	84.67	92.20	2.79e+01
Desv. Est.	6.24	7.82	5.51	0.33	5.59	3.80	3.39	0.45	2.08	6.91	2.81	0.54

Cuadro 6.3: Resultados de ejecución de Búsqueda Local del Primer Mejor

[1] Puede encontrar las gráficas de *ecoli* y *Breast-cancer* en el directorio `outputs/figures`.

[2] Las soluciones pueden consultarse en `outputs/solutions` cuyo nombre contiene el ID asociado a la respectiva ejecución de `outputs/results`.

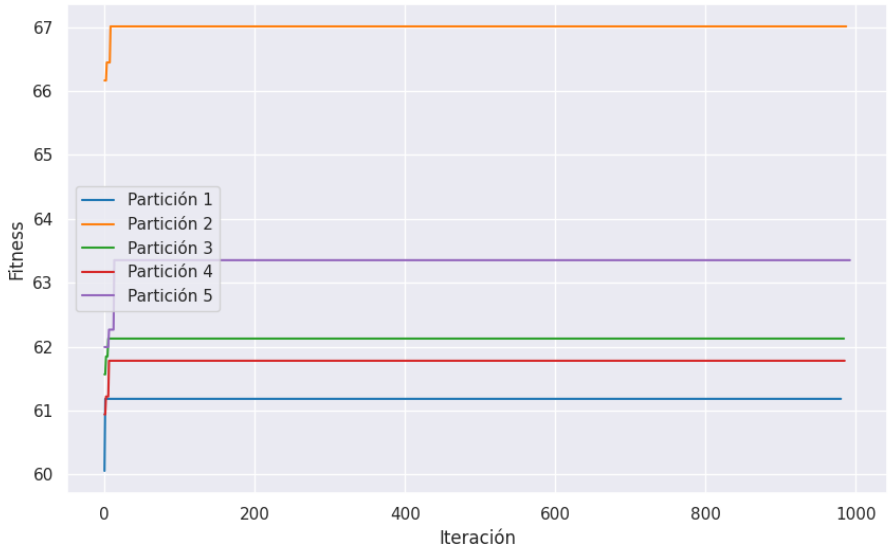

Figura 6.1: Convergencia de fitness de la Búsqueda Local del Primer Mejor en *Ecoli* a través de cinco particiones. Las líneas trazan la mejora del fitness en el entrenamiento, mostrando una fuerte dependencia de los valores iniciales de la solución.

De manera adicional se ha calculado la Tabla 6.4 para la modificación de la búsqueda local implementada. Es interesante ver que a pesar del aumento de evaluaciones por iteración, el algoritmo mantiene e incluso mejora los tiempos de ejecución cuanto mayor sea el número de atributos, consecuencia de la paralelización.
Por otro lado también puede apreciarse el empeoramiento de los resultados. Mi hipótesis es que en la búsqueda del primer mejor, el hecho de no buscar la mejor solución del entorno sino la primera mejor actuaba como regularizador en el método de búsqueda. Ahora, al buscar sobre un gran número de vecinos, el algoritmo tiende a caer más rápido en mínimos locales y por tanto evitando una exploración más estocástica del entorno.

	Ecoli				Parkinsons				Breast-cancer			
	% clas	% red	Fit.	T(s)	% clas	% red	Fit.	T(s)	% clas	% red	Fit.	T(s)
Partición 1	80.00	0.00	60.00	5.94e+00	65.00	95.45	72.61	3.72e+00	83.48	**93.33**	85.94	2.68e+01
Partición 2	72.86	0.00	54.64	**5.81e+00**	75.00	95.45	80.11	3.67e+00	**94.78**	**93.33**	**94.42**	**2.66e+01**
Partición 3	82.35	0.00	61.76	5.82e+00	**82.50**	95.45	**85.74**	3.67e+00	89.57	83.33	88.01	2.68e+01
Partición 4	80.88	**14.29**	64.23	**5.81e+00**	60.00	95.45	68.86	**3.65e+00**	73.91	90.00	77.93	2.73e+01
Partición 5	**88.33**	**14.29**	**69.82**	6.18e+00	80.00	95.45	83.86	3.90e+00	88.99	**93.33**	90.08	2.75e+01
Media	80.89	5.71	62.09	5.91e+00	72.50	95.45	78.24	3.72e+00	86.15	90.67	87.28	2.70e+01
Desv. Est.	5.54	7.82	5.57	0.16	9.68	0.00	7.26	0.10	7.92	4.35	6.09	0.39

Cuadro 6.4: Resultados de ejecución de la Búsqueda Local modificada

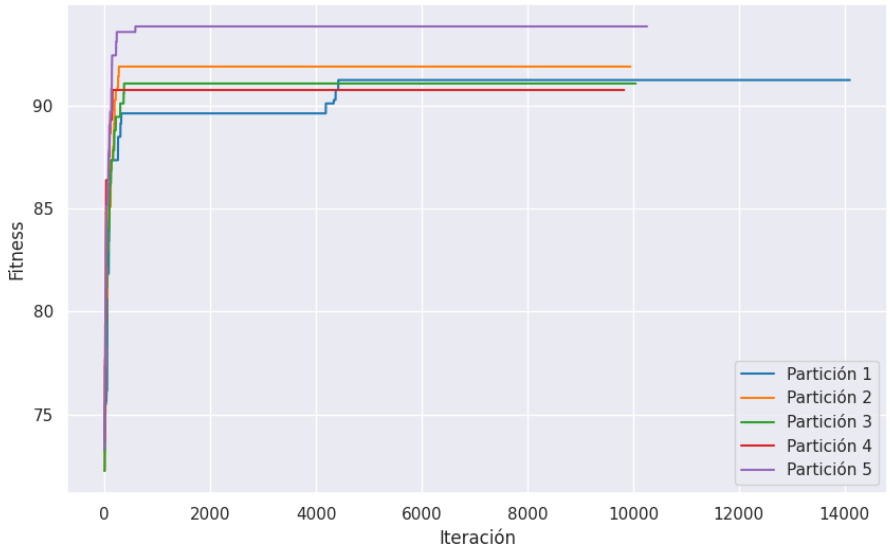

Figura 6.2: Convergencia de fitness de la Búsqueda Local del Primer Mejor en *Parkinsons* a través de cinco particiones. Las líneas trazan la mejora del fitness en el entrenamiento, mostrando una estabilización homogénea tras las iteraciones iniciales.

6.2. Resultados globales

Los resultados medios finales ofrecen una comparativa muy interesante tal y como muestra la Tabla 6.5. Dadas las circuntancias, resulta complicado ofrecer una respuesta contundente a qué algoritmo es mejor.
Nuestra función de evaluación fue escogida de forma que se priorizase una solución que sacrifique parte de la tasa de clasificación por una que aporte una mayor reducción. De esta forma, buscamos una solución cuya inferencia sea lo más rápida posible intentando no perder precisión. En este escenario, vemos que el algoritmo que mejor rinde en general es la búsqueda local. Nótese que en promedio, el factor determinante para obtener el mejor valor fitness es la tasa de reducción. Es curioso que no siempre los algoritmos que han logrado la mayor tasa de clasificación han sido los mejor evaluados en la función objetivo, mostrando que el sacrificio de un poco de precisión a favor de una mayor reducción ofrece una ventaja notable, hecho del que ha sacado partido la búsqueda local frente a sus competidores.

	Ecoli				Parkinsons				Breast-cancer			
	% clas	% red	Fit.	T(s)	% clas	% red	Fit.	T(s)	% clas	% red	Fit.	T(s)
1-NN	80.81	0.00	60.61	**1.89e-02**	94.79	0.00	71.09	1.88e-02	**95.59**	0.00	71.69	**2.83e-02**
RELIEF	79.68	**28.57**	**66.90**	1.14e-01	94.79	0.00	71.09	**1.53e-02**	95.25	4.00	72.44	1.09e-01
LOCAL-SEARCH	**81.44**	8.57	63.22	1.13e+00	**95.29**	**73.64**	**89.87**	3.40e+00	94.72	**84.67**	**92.20**	2.79e+01

Cuadro 6.5: Comparativa de la eficiencia de los algoritmos 1-NN, RELIEF y Búsqueda Local en los conjuntos de datos *Ecoli*, *Parkinsons* y *Breast-cancer*.

Por otro lado, es evidente el aumento de tiempo de ejecución de la búsqueda local frente a la aproximación voraz y la trivial. Tras analizar la ejecución con la ayuda del *profiler* `gprof` [1], vemos que el coste temporal de ejecutar el clasificador 1-NN con *leave one out* en cada iteración es muy alto, pues el número de veces que calcula la distancia es abrumador a pesar de los esfuerzos en paralelización. Todo esto obviando el hecho de que, al ser la búsqueda local del primer mejor una búsqueda secuencial, es complicado encontrar una forma de paralelizarlo sin perder el enfoque original.

REFERENCIAS

[1] FENLASON, J., AND STALLMAN, R. Gnu gprof. *GNU Binutils. Available online: http://www.gnu. org/software/binutils (accessed on 21 April 2018)* (1988).

[2] ISO. *ISO/IEC 14882:1998: Programming languages — C++*. Sept. 1998. Available in electronic form for online purchase at `http://webstore.ansi.org/` and `http://www.cssinfo.com/`.

[3] NAKAI, K. Ecoli. UCI Machine Learning Repository, 1996. DOI: https://doi.org/10.24432/C5388M.

[4] NILASHI, M., IBRAHIM, O., SAMAD, S., AHMADI, H., SHAHMORADI, L., AND AKBARI, E. An analytical method for measuring the parkinson's disease progression: A case on a parkinson's telemonitoring dataset. *Measurement 136* (2019), 545–557.

[5] NOLL, L. C., MENDE, R. G., AND SISODIYA, S. Method for seeding a pseudo-random number generator with a cryptographic hash of a digitization of a chaotic system, Mar. 24 1998. US Patent 5,732,138.

[6] STALLMAN, R. M., MCGRATH, R., AND SMITH, P. D. *GNU make.* , 2001.

[7] WOLBERG, WILLIAM, M. O. S. N., AND STREET, W. Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository, 1995. DOI: https://doi.org/10.24432/C5DW2B.

Como referencia fundamental se han seguido las diapositivas de la asignatura.