Práctica 3.b: Enfriamento Simulado, Búsqueda Local Reiterada y Evolución Diferencial para el para el Problema del Aprendizaje de Pesos en Características

Metaheurísticas

Nombre	Miguel Lentisco Ballesteros
DNI	XXXXXXXXX
Correo	XXXXXXX@XXXXXX
Grupo	Grupo 1 (M 17:30-19:30)
Problema	Aprendizaje de Pesos en Características
Algoritmos	ES, ILS, DL
Curso	2018-2019

Índice general

1 Descripción del problema

Sean $e_1, \ldots, e_N$ (N nº de datos) una muestra de objetos ya clasificados donde cada objeto pertenece a una de las clases $C_i, \ldots, C_M$ (M nº de clases); y cada objeto d_i tiene ascociado un vector $(x_{1_i}, \ldots, x_{n_i}) \in \mathbb{R}^n$ (n nº de características/atributos); luego hacemos la asociación y representamos cada e_i como el vector de atributos asociado.

Nuestro objetivo es obtener una función $f : \mathbb{R}^n \to \{C_1, \ldots, C_M\}$ llamada **Clasificador**, que asocie correctamente cada objeto e_i con su clase correspondiente C_j.

Como ya conocemos a priori las clases existentes que queremos clasificar y sabemos la clase a la que pertenece cada objeto del conjunto de datos, estamos claramente en una situación de aplicar **aprendizaje supervisado** para obtener este clasificador que buscamos.

De entre las muchas técnicas usaremos el método **K-NN (k-nearest neighbors)**, que lo que hace es clasificar la clase de un punto p según los k puntos más cercanos a p con la distancia euclidea (junto a la Hamming si son atributos nominales pero asumimos que son todos atributos numéricos por la simplicidad). Una vez encontrados estos puntos, se obtiene la clase que predomina en esos k puntos y se asigna esa clase al punto p. Obviamente para evitar problemas con la escala de cada atributo se suelen normalizar todas los atributos al intervalo [0, 1], así ninguna variable influye más que otra en la distancia.

Obviamente este clasificador tiene en cuenta todos los atributos, sin embargo sabemos que a la hora de clasificar objetos no tiene que darse el caso de que todas los atributos importen lo mismo a la hora de clasificar; es decir, hay atributos que pueden ser mas decisivos que otros para saber de que clase es el objeto (puede haber atributos que no influyan en nada o muy poco, atributos que realmente dependan de otros que ya estén incluidos...).

Por ello vamos a asignar un valor $w_i \in [0, 1],\ i \in \{1, \ldots, n\}$ a cada carcterística y representamos los pesos como $W = \{w_1, \ldots, w_n\}$. Por tanto nuestro problema es en encontrar unos "buenos" pesos (más adelante se explica cual es el criterio de bondad), es decir el problema de **Aprendizaje de Pesos en Características (APC)**. Es decir ahora a la hora de calcular las distancias euclideas tendremos en cuenta el peso:

$$d_e(e_1, e_2) = \sqrt{\sum_i w_i \cdot (e_i^i - e_2^i)^2}$$

.

Para obtener nuestra solución a los algoritmos que van a buscar nuestra solución al problema necesitamos un conjunto de entrenamiento y tambien otro de prueba para evaluar como de buena es la solución obtenida. Para obtener estos conjuntos usaremos **k-fold cross validation**, que consiste en dividir el conjunto total de datos en k particiones disjuntas del mismo tamaño manteniendo la distribución de clases equilibrada (para que sea una muestra representativa del total), entonces utilizaremos el algoritmo k veces obteniendo k soluciones tomando cada vez una partición distinta como conjunto de prueba y agrupando el resto de particiones como conjunto de entrenamiento. Finalmente la calidad será la media de los resultados de las k soluciones.

En nuestro caso vamos a usar **1-NN** y **5-fold cross validation** (el 5 y 10-fold cross validation son los "mejores" para validar clasificadores asi que la elección es buena; por otro lado el usar 1-NN o cualquier otro k-NN habría que verlo repitiendo los resultados con distintas k y viendo cuales ofrecen mejores resultados).

La **función de evaluación** (como de buena es la solución obtenida) va a ser

$$F(W) = \alpha \cdot tasa_{clas}(W) + (1 - \alpha) \cdot tasa_{clas}(W)$$

donde $\alpha \in [0, 1]$ es el que pondera cual de dos valores es más importante y:

- $tasa_{clas} = 100 \cdot \dfrac{\text{instancias bien clasificadas en T}}{N}$ es la tasa de clasificación: indica el porcentaje de acierto del clasificador, la tasa de clases correctamente asignadas al conjunto de prueba T. Cuanto más alto se dice que el clasificador es mas **preciso**.
- $tasa_{red} = 100 \cdot \dfrac{|\{w_i : w_i < 0,2\}|}{n}$ es la tasa de reducción: indica el nº de características descartadas, es decir las características que no influyen casi nada a la hora de clasificar porque tienen pesos muy cercanos a 0. Cuanto mas alto más **simple** es el clasificador.

En nuestro caso usaremos $\alpha = 0,5$ queriendo así obtener soluciones que sean buenas clasificando con el mínimo número de características posible.

2 Descripción de la aplicación de los algoritmos empleados al problema

Antes de explicar los algoritmos realizados, describiré los tipos, métodos comunes... de todos los algoritmos. Como anotación, en algunas funciones no se especificarán algunas cosas completamente para evitar obscurecer el pseudocódigo; y en cosas menos importantes como la lectura de datos o creación de particiones no se incluye pseudocódigo al no ser el objetivo de la práctica.

2.1. Distribución del codigo

El código se encuentra distribuido en diferentes módulos:

- `Base`: los tipos de estructura para todos los módulos y unas pocas funciones básicas
- `Main`: código principal, manejo general
- `Lectura`: para leer los datos del fichero y crear la estructura de datos
- `CrossV`: forma las particiones de CrossValidation
- `KNN`: implementa clasificador 1nn
- `Ejecutar`: ejecuta los algoritmos y formatea los resultados
- `Utils`: funciones auxiliares comunes
- `P1`: algoritmos de la P1
- `P2`: algoritmos de la P2
- `P3`: algoritmos de la P3

2.2. Notación de pseudocódigo

He usado la explicación de Pablo Baeyens Fernández (disponible en GitHub) que también hizo las prácticas en Haskell y se entiende bastante bien para gente nueva a este tipo de notación:

Usaré notación parecida a la usada en Haskell para que se entiendan mejor las cosas aunque será una versión muy simplificada ya que hay cosas como para generar números aleatorios que se complican en Haskell que omitiré. Los argumentos se pasan a las funciones separados por espacios y el tipo de la función se indica después de su nombre

seguido de dos dobles puntos `::`. Además para evitar poner paréntesis se puede cambiar por `$`, es decir `f(g(z * y))` equivale a `f $ g $ z * y`.

Aclaro que todo se "pasa" por valor, no por referencia ni por punteros, luego se asume como en C++ como si fuera todo paso por valor, por tanto el resultado de una operacion no se puede alterar.

Para mostrar el estilo del pseudocódigo incluyo un ejemplo de pseudocódigo para una función en C:

```
int suma(int a, int b){
  int c = a + b;
  return c;
}
```

Y su equivalente en el estilo de pseudocódigo que voy a utilizar:

```
suma :: Int → Int → Int
suma a b = c
  where c = a + b
```

Otra alternativa sería:

```
suma :: Int → Int → Int
suma a b =
  let c = a + b
  in c
```

También explico algunas funciones que se usan bastante:

- `juntaCon f l1 l2 ... ln` toma n listas y una función de n argumentos y devuelve una lista tal que la posición `i` tiene el elemento `f l1_i l2_i ... ln_i`
- `map f [x1, ..., xn]` toma una función `f` y una lista `[x1, ..., xn]` y devuelve la lista `[f x1, ..., f xn]`
- `acumula (·) i [x1, ..., xn]` acumula los elementos de la lista usando la función `·`. Devuelve: `i·x1·x2···xn`.
- `\x1 x2 ... xn → expr` es una función lambda (sin nombre) que toma `x1 x2 ... xn` como argumentos y devuelve `expr`
- `repite n x` crea una lista de n copias de `x`.

2.3. Representación de datos

- Un **atributo** se representa con `Double`.
- Un **vector de atributos** (`Punto`) se representa con `Vector Double`.
- Una **clase** (`Clase`) se representa con `String`.
- Un **dato** (`Dato`) se representa como una tupla `(Punto, Clase)`.

- Un **conjunto de datos** (`Datos`) se representa como `[Dato]` (lista de `Dato`), pudiera ser un conjunto de entrenamiento, prueba, o cualquier agrupación de datos.
- Una **partición** (`Particion`) se representa como una tupla con el conjunto de entrenamiento y el de prueba `(Datos, Datos)`, se refiere a una de las agrupaciones obtenidas de aplicar el k-fold cross validation, no solo a una de las particiones en las que se dividen el conjunto total de datos.
- Un **conjunto de particiones** (`Particiones`) se representa como una `[Particion]`.
- Un **algoritmo** (`Algoritmo`) se representa como una función `Datos -> Pesos` que toma el conjunto de entrenamiento y devuelve la solución (los pesos).

Para la búsqueda local, algoritmos genéticos... implemento un tipo extra llamado `Solucion` que encapsula la solución propiamente dicha (los pesos) y guarda su valor de la función objetivo (para evitar evaluarla de nuevo en comparaciones) y también el nº de vecinos que ha creado que se tendrá en cuenta para la condición de parada cuando se requiera.

Además hay otro tipo especial llamado `Estado a` que he tenido que realizar debido a como funciona Haskell, en términos muy sencillos lo que representa es una función que parte de un estado `s` a una tupla `(a,s)` es decir pasa de un estado a otro y devuelve un resultado, en este caso el estado siempre es fijo pero lo que devuelve puede variar según queramos.

Aqui el tipo `Solucion`:

```
data Solucion = Solucion {
  getPesos :: Pesos,
  getFit :: Double,
  getNVecinos :: Int
}
```

Y el tipo `Estado a`:

```
type Estado a = State (StdGen, Int) a
```

El estado será una tupla de un generador de nº aleatorios y un número que representará el nº de evaluaciones de la función objetivo a lo largo de la ejecución del algoritmo de búsqueda local. Tengo que llevar un generador como estado debido a la transparencia referencial de Haskell (una función siempre devuelve lo mismo con los mismos parámetros de entrada) por lo que para ir generando nº aleatorios un generador devuelve un nº aleatorio y un nuevo generador; en cualquier caso esa es la idea general, que tengo un estado **global** por decirlo de alguna manera al que puedo acceder, realizar cosas y devolver otro estado nuevo.

2.4. Lectura del archivo .arff y normalización

La lectura del archivo es simple: primero se ignora todo hasta que se llega `@data`. Entonces para cada linea leo todos los valores excepto el último y creo un `Punto` con esos valores y con el último formo la `Clase` y con ambos ya tengo un `Dato`.

Paso un filtro para eliminar valores repetidos ya que entiendo que queremos entrenar el clasificador para que pueda clasificar valores **nuevos**, el hecho de dejar valores repetidos y que caigan en particiones distintas va a ocasionar que acierte siempre y realmente no nos dice nada nuevo.

A la hora de aplicar el algoritmo en conjuntos de datos no vistos si observamos un dato que es idéntico (en todos sus atributos) a un dato que ya tenemos clasificado en nuestra base de datos obviamente podemos decir que tienen la misma clase y ya hemos acabado. Si no fuese así entonces es que al menos existe un atributo desconocido que no hemos medido en ninguno de nuestros datos y nos encontrariamos un problema mucho mayor por lo que por simplificar la cuestión vamos a eliminar los repetidos para que no metan ruido.

Ahora se procede a normalizar los datos en el intervalo $[0, 1]$ aplicando la función que ya se conoce. Si para un atributo la diferencia entre su maximo y su minimo es 0 se entiende entonces que el atributo es **constante** y por tanto no aporta nada a la hora de clasificar por lo que se normalizan a 0 todos sus valores.

Finalmente se aplica un `shuffle` para cambiar el orden a la hora de crear las particiones.

2.5. Creación de particiones

Para crear las particiones queremos que mantegan una proporción equilibrada de clases, entonces primero dividimos los datos agrupándolos por su clase con `separarClases`. Al resultado se aplica un `shuffle` para y pasamos el resultado a `eleccionCruzada` (k-fold cross validation) con $k = 5$.

La creación de las particiones es simple: para cada lista de datos de clase se divide en k particiones iguales junto al resto que va aparte, y después se juntan partición a partición y los restos se unen. Si quedasen restos entonces se mezclarían, se dividirían en k particiones y se juntarían partición a partición con el resultado anterior; hasta que o bien no quedasen o bien no se pudieran dividir entre k cuando entonces se repartirían uniformemente (uno a cada partición).

Este criterio intenta mantener un **equilibrio de clases** a la vez que mantiene que las k particiones tengan el **mismo tamaño** excepto restos.

2.6. 1-NN

La clasificación k-nn con $k = 1$ anteriormente explicada, implementada con la modificación **APC** para tener en cuenta los pesos. Toma el conjunto de prueba, el de entrenamiento y los pesos y devuelve el % de acierto clasificando los puntos del conjunto de prueba.

Para ello para cada punto del conjunto de prueba se le aplica `dist2P` con todos los puntos del conjunto de entrenamiento (excluyendo el punto al que se le está aplicando para los casos donde el conjunto de entrenamiento sea igual que el de prueba ~ **leave-one-out**, y como ya he quitado repetidos al procesar los datos no hay ningun problema al hacer esto) y obtenemos la clase del punto con menor distancia y el acierto será si coincide esta clase con la del conjunto de prueba.

```
clas1nn :: Datos -> Datos -> Pesos -> Float
clas1nn train test pesos =
  let distanciasA p = map (\x -> dist2P x p pesos) (quita p train)
      aciertos = map (\p -> claseDe p == claseDe $ min $ distanciasA p) test
  in nAciertos aciertos / sizeOf aciertos
```

Cabe mencionar que como solo estamos buscando el mínimo de la distancia y no nos interesa su valor real, como $f(x) = \sqrt{x},\ \forall x \in \mathbb{R}$ es una función creciente no afecta el hecho de calcular el máximo sin la raíz cuadrada (y nos ahorramos calculos para mejorar el tiempo de ejecución):

```
-- Distancia euclidea considerando pesos
dist2P :: Dato -> Dato -> Pesos -> Float
dist2P p1 p2 pesos = suma $ juntaCon (\x y w -> w * (y - x) * (y - x)) p1 p2 pesos
```

2.7. Función objetivo

Como ya se ha explicado la función objetivo en la descripción del problema. En general se pasan la tasa de reducción y la tasa de acierto y tenemos que:

```
-- Función objetivo
fEvaluacion :: Float -> Float -> Float -> Float
fEvaluacion alpha tAcier tRed = alpha * tAcier + (1 - alpha) * tRed
```

Existe una variante para cuando se quiere evaluar f sobre el conjunto de entrenamiento:

```
evaluarF :: Datos -> Pesos -> Float
evaluarF datos pesos =
  let pReduccion = selecciona (< 0.2) pesos / sizeOf pesos
      pAcierto   = clas1nn datos datos (reducePesoss pesos)
  in fEvaluacion 0.5 pAcierto pReduccion
```

2.8. Resultados de algoritmos

Tendremos una lista de distintos algoritmos que queremos aplicarles todos nuestras particiones y obtener los resultados, entonces para cada algoritmo y cada partición se ejecuta el algoritmo con esa partición que nos devuelve unos pesos y el tiempo tardado en obtenerlos. Vemos el porcentaje de reducción y reducimos los pesos para poder aplicar 1-NN y ya obtenemos el porcentaje de acierto y podemos sacar el valor de la función objetivo. Cuando tenemos las 5 particiones hacemos los valores medios y pasamos al siguiente algoritmo.

Finalmente se escribe en el archivo "nombreFichero_resultados.txt" los resultados de todos los algoritmos con las 5 particiones y los valores medios.

2.9. Creación de una solución aleatoria

Es muy común en todas las prácticas crear una solución aleatoria, y se haría de esta manera:

```
pesosIniRand :: Datos -> Estado Solucion
pesosIniRand datos = do
  let listaRands = listaInfinitaRandoms (0.0, 1.0)
  let pesos = toma (nCaract datos) listaRands
  return crearSolucion datos pesos
```

2.10. Búsqueda Local (BL)

El uso de BL para distintos algoritmos merece una explicación en este apartado, como algoritmo usado por otros, el armazón general es:

```
busLoc :: StdGen -> Algoritmo
busLoc gen datos = getPesos solRes
  where solRes = evalua hastaQueM (nIter >= 15000 || nVecinos >= 20 * nCaract datos)
    (crearVecino datos) (pesosIniBL datos)) (gen, 0)
```

Donde obviamente cada algoritmo cambiará las condiciones según necesite (ej en vez de 15k, 1k evaluaciones; o sin parada por generación máxima de vecinos), en nuestro caso general hasta 15k iteraciones o cuando el nº de vecinos generados sin cambiar la solución sea 20 * nº características.

La solución inicial, con los índices posibles a mutar (inicialmente todos):

```
pesosIniBL :: Datos -> Estado (Solucion, [Int])
pesosIniBL datos = return (pesosIniRand datos, [0..(nCaract datos - 1)])
```

Y cada iteración creamos un vecino, de manera que si es mejor lo sustituimos y creamos todos los índices posibles a mutar enteros; si no es mejor se devuelve la solución actual sin el índice que ha mutado, y si se ha quedado sin índices que mutar se refrescan todos.

```
-- Creo un nuevo vecino a partir de una solución
crearVecino :: Datos -> (Solucion, [Int]) -> Estado (Solucion, [Int])
crearVecino datos (sol, indices) = do
  let (solNueva, indNuev) = obtenerVecino 0.3 datos indices sol
  let solAct = aumentaVecino sol
  let indNuev' =
    if solNueva > solAct || null indNuev then [0..(nCaract datos - 1)] else indNuev
  return (max solNueva solAct, indNuev')
```

Finalmente la obtención de un vecino es aplicar el operador de mutación en un índice aleatorio (modificar con un valor normal con desviación típica 0.3 y media 0.0), devolviendo la nueva solución y quitando el índice que se ha modificado de la lista de índices disponibles.

```
obtenerVecino :: Datos -> [Int] -> Solucion -> Estado (Solucion, [Int])
obtenerVecino datos indices sol = do
  let inds = aleatorio (0, length indices - 1)
  let ind = indices !! inds
  let z = rNormal 0.3 0.0
  let pesosN = U.imap (\i x -> if i == ind then restringe $ x + z else x) $ getPesos sol
  return (crearSolucion datos pesosN, delete ind indices)
```

3 Algoritmos de búsqueda

3.1. Enfriamento Simulado (ES)

Este algoritmo basado en trayectorias (en concreto permite empeoramientos de la solución actual), es muy similar a la búsqueda local pero añade más diversidad (exploración) a la búsqueda permitiendo tomar vecinos peores. Basado en el recocido del acero, añade una probabilidad de tomar vecinos peores basada en la temperatura (cuanto más alta más probabilidad de aceptar), que empieza siendo muy alta para ir variando en la búsqueda y en cada iteración va enfriandose hasta llegar a la temperatura final, donde se va aceptando menos y menos vecinos (se incrementa la explotación).

De tipos simplemente he hecho un renombre para que sea mas legible, `Temp` es un `Double` y hace referencia a un dato que sea una temperatura.

El esqueleto general sería:

```
eS :: StdGen -> Algoritmo
eS gen datos = getPesos solMej
  where solMej = evaluar (hastaQue (nIter >= 15000 || nExitos == 0)
    (iterEnfriamento datos) (solIniES 0.3 0.3 datos)) (gen, 0)
```

Creamos la sol inicial:

```
-- Crea la solución inicial que consta de: SolAct + + TActual + MejorSol + T0 + Tf
solIniES :: Double -> Temp -> Datos
  -> Estado (Solucion, Temp, Solucion, Temp, Temp, Int)
solIniES mu phi datos = do
  let solIni = hastaQue (getFit solIni > - log phi * mu * 0.001) (pesosIniRand datos)
  let tempIni = mu * getFit solIni / (- log phi)
  return (solIni, tempIni, solIni, tempIni, 0.001, 1)
```

Considerando que la temperatura inicial es, con $\mu = \phi = 03$:

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

Creamos varias soluciones iniciales hasta que una de ellas tenga una temperatura inicial válida (que sea mayor que la final, $(f(sol) > -0.001 \cdot log(\phi)\mu)$, y ponemos la temp final

a 0.001. La estructura que deuelve sirve para llevar la siguiente información: (solucionActual, temperaturaActual, mejorSolucion, T_0, T_F, nExitos). T_0, T_F serán constantes para usarlas al enfriar la temperatura, nExitos marcará el nº de vecinos aceptados en cada iteración (para comprobar si nExitos == 0 y cortar), mejorSolucion será la mejor solución encontrada durante toda la ejecución, y solucionActual y temperaturaActual la solución y temperatura actual en cada iteración.

Cada iteración del bucle será:

```
-- Iteración principal se hace la busqueda en el vecindario y se enfria la temperatura
iterEnfriamento :: Datos -> (Solucion, Temp, Solucion, Temp, Temp, Int)
  -> Estado (Solucion, Temp, Solucion, Temp, Temp, Int)
iterEnfriamento datos (solAct, tAct, mejSol, t0, tf, _) = do
  let nMaxVec = nCaract datos * 10
  let (solNueva, solMejNueva, nVecExi)
    = hastaQueM (nIter >= 15000 || nVec >= nVecMax || nVecExi >= (nVecMax * 0.1))
    (exploraVecindario datos tAct) (solAct, mejSol, 0, 0)
  let m = redondea (15000 / nMaxVec)
  return (solNueva, enfriaCauchy m tAct t0 tf, solMejNueva, t0, tf, nVecExi)
```

Tomamos el nº máximo de vecinos a generar `nMaxVec` como nº de características por 10, a continuación exploramos el vecindario hasta que lleguemos a las 15k iteraciones, o el nº de vecinos generados llegue al tope, o se hayan aceptado el nº máxito de éxitos; entonces nos devuelve la nueva solución (puede ser la misma que la actual), la mejor solución (puede ser la misma que la actual) y el nº de vecinos aceptados, pasamos toda esta información para la siguiente iteración si procede, enfriando además la temperatura mediante el esquema Cauchy-Mejorado tomando en cuenta que habrá `15000/nMaxVec` iteraciones aproximadamente:

```
enfriaCauchy :: Int -> Temp -> Temp -> Temp -> Temp
enfriaCauchy m t t0 tf = t / (1 + beta * t)
  where beta = (t0 - tf) / (m * t0 * tf)
```

El esquema es:

$$T_{k+1} = \frac{T_k}{1 + \beta T_k}, \quad \beta = \frac{T_0 - T_F}{M T_0 T_F}$$

Finalmente, la exploración del vecindario se hace de la siguiente manera:

```
exploraVecindario :: Datos -> Temp -> (Solucion, Solucion, Int, Int)
  -> Estado (Solucion, Solucion, Int, Int)
exploraVecindario datos tAct (solAct, mejSol, nVec, nVecExi) = do
  let solNueva = obtenerVecino 0.3 datos solAct
  let diferencia = getFit solAct - getFit solNueva
  if diferencia == 0
    diferencia = 0.005
```

```
  let numR = aleatorio (0.0, 1.0)
  if diferencia' < 0 || numR <= exp (- diferencia / tAct) then
    return (solNueva, max solNueva mejSol, nVec + 1, nVecExi + 1)
  else
    return (solAct, mejSol, nVec + 1, nVecExi)
```

Generamos un nuevo vecino y obtenemos la diferencia de evaluación de la sol actual menos la nueva, si la diferencia es 0 (que suele pasar) le ponemos 0.005 para que no se acepte siempre incluso con temperaturas bajas. Entonces generamos un nº aleatorio y si la diferencia es negativa (el vecino es mejor) o $rand \leq exp(-\frac{-diferencia}{T_{act}})$ entonces aceptamos el vecino como solución actual, comprobamos si es mejor que la mejor solución actual y aumentamos el nº de vecinos aceptados; en caso contrario no hacemos nada. En cualquier caso aumentamos en 1 el nº de vecinos creados.

Finalmente para crear un vecino, aplicamos el mismo operador de mutación de BL:

```
obtenerVecino :: Datos -> Solucion -> Estado Solucion
obtenerVecino sD datos sol = do
  let ind = aleatorio (0, nCaract datos - 1)
  let z = rNormal 0.3 0.0
  let pesosN = imap (\i x -> if i == ind then restringe $ x + z else x) $ getPesos sol
  return crearSolucion datos pesosN
```

3.2. Búsqueda Local Reiterada (ILS)

Este algoritmo basado en trayectorias, subtipo multiarranque, se basa en arrancar muchas veces la búqsueda local de la siguiente manera: tomamos una solución inicial y le aplicamos búsqueda local, después aplicamos un bucle que consiste en mutar la solución actual, aplicarle la búsqueda local y quedarnos con la mejor de la soluciones y volver a repetir.

Como usaremos BL llamamos al esqueleto de BL modificado solo para que pare al hacer 1000 evaluaciones (no se decía nada sobre si parar al generar nMaxVecinos asi que he decidido hacer 1k iteraciones solamente), y sigue el mismo esquema que el explicado en las consideraciones generales.

El esquema general sería:

```
iLS :: StdGen -> Algoritmo
iLS gen datos = getPesos sol
  where sol = evalua (hastaQue (nIter >= 15) (iteracionILS datos)
    (solIniILS datos)) (gen, 0)
```

Considerando que haremos 15 iteraciones (contando con la solución inicial) a BL tendremos 15000 evaluaciones (igual que en otros algoritmos).

La solución inicial es la misma pero aplicando BL, y llevando la cuenta del nº de iteraciones:

```
solIniILS :: Datos -> Estado (Solucion, Int)
solIniILS datos = return (bL datos (pesosIniRand datos), 1)
```

Cada iteración de ILS será así:

```
iteracionILS :: Datos -> (Solucion, Int) -> Estado (Solucion, Int)
iteracionILS datos (solAct, nIter) = do
  let solMutada = mutarSolILS datos solAct
  let solBL = blILS datos solMutada
  return (max solAct solBL, nIter + 1)
```

Tomamos la sol actual, le aplicamos la operación de mutación, y después BL. Finalmente devolvemos la mejor solución para seguir aplicando el algoritmo y aumentamos en 1 el nº de iteraciones. Como considerabamos el criterio de elección como el mejor no hace falta llevar la cuenta de la mejor solución, en cualquier caso se podría hacer una pequeña modificación si se quisiera cambiar el criterio.

La mutación de la solución:

```
mutarSolILS :: Datos -> Solucion -> Estado Solucion
mutarSolILS datos solAct = do
  let nMut = min 1 $ redondea $ 0.1 * (nCaract datos)
  let pesosMutados = repiteNM nMut (mutarPesos 0.4) (getPesos solAct)
  return crearSolucion datos pesosMutados
```

Esta mutación es parecida a la de siempre pero más agresiva, en este caso usamos desviación tipica 0.4 en vez de 0.3 y además mutamos el 10 % de los genes (y ponemos un mínimo de mutación de 1 gen para q mute al menos una vez) usando el operador normal de siempre aplicado `nMut` veces.

3.3. Evolución Diferencial (DE)

Este algoritmo basado en algoritmos genéticos, hace más énfasis en la mutación, que va antes de la recombinación/cruce. El esquema general es generar una población inicial de N cromosomas y hasta que se cumpla la condición de parada seguir el siguiente bucle: para todo individuo de la población se genera un hijo nuevo tomando de base los pesos del padre (individuo i) y de manera que por cada gen del nuevo hijo hay una probabilidad de se aplique el operador de mutación; en cualquier caso cuando se forma el nuevo hijo entero si es mejor que el padre se reemplaza.

Como las dos variantes serán usando un operador de cruce distinto, explicaré en general todo y después veremos las dos fórmulas diferentes usadas en los dos algoritmos. De esta manera el operador de cruce (Rand o CurrentToBest) está unido al de recombinación

(binominal), y después de aplicarse a todos los genes del hijo se aplica el de reemplazo (compara hijo-padre)

Se ha incluido el tipo `EsqMutar` que hace referencia al esquema de mutar que es el siguiente `type EsqMutar = Poblacion -> Int -> Int -> [Int] -> Estado Double`. Tomará la población, los índices i y j (i sobre la población y j sobre el nº)

Empezamos con el esquema general, donde variará segun el esquema de mutación que usemos:

```
dE :: EsqMutar -> StdGen -> Algoritmo
dE esqMut gen datos = getPesos (maximo pobSol)
  where pobSol = evaluar (hastaQue (nIter >= 15000) (iterDE datos esqMut)
    (crearPobIni 50 datos)) (gen, 0)
```

La población inicial se crea repitiendo n veces creando cromosomas con pesos aleatorios:

```
-- Crea nPob individuos aleatorios
crearPobIni :: Int -> Datos -> Estado Poblacion
crearPobIni nPob datos = repite nPob (crearCromIni datos)
```

Cada iteración del bucle será aplicar la misma operación a cada cromosoma de la población, actualizando la población, y empezando con i = 0:

```
iterDE :: Datos -> EsqMutar -> Poblacion -> Estado Poblacion
iterDE datos esqMutRecom pobActual = repite (sizeOf pobActual)
  (actualizarPoblacion datos esqMutRecom) (pobActual, 0)
```

Por cada cromosoma generamos unos pesos inicialmente a 9 (da igual) y tomamos 3 índices distintos entre sí y de i que se los pasaremos al esquema de mutación según los necesiten. Al aplicar el bucle interno obtendremos los pesos del nuevo hijo, creando el nuevo cromosoma hijo y finalmente si el hijo es mejor que el padre se sustituye por él en la población (reemplazo) y devolvemos la población actualizada:

```
actualizarPoblacion :: Datos -> EsqMutar -> (Poblacion, Int) -> Estado (Poblacion, Int)
actualizarPoblacion datos esqMut (pob, i) = do
  let vectorIni = replica (nCaract datos) 0
  let indices = tomaIndRand 3 (delete i [0..(sizeOf pob - 1)])
  let hijoNuevo = crearCromosoma datos $ repiteNM (nCaract datos)
    (mutReemp esqMut pob i indices) (vectorIni, 0)
  if getFit hijoNuevo > getFit pob[i] then pob[i] = hijoNuevo
  return (pob, i + 1)
```

Es muy fácil tomar índices distintos tomando una posición de la lista de índices y repitiendo quitando ese índice de la lista y así:

```
tomaIndRand :: Int -> [Int] -> Estado [Int]
tomaIndRand nInd indices = repite nInd tomaIndice (indices, [])
  where tomaIndice (inds, res) = do
```

```
        let k =  (0, sizeOf inds - 1)
        return (delete inds[i] inds, res ++ inds[i])
```

Por último el esquema de mutación/combinación:

```
mutReemp :: EsqMutar -> Poblacion -> Int -> [Int] -> ([Double], Int)
  -> Estado ([Double], Int)
mutReemp esqMut pob i indices (pesos, j) = do
  let r = aleatorio (0.0, 1.0)
  if r <= (0.5 :: Double)
    then pesos[j] = esqMut pob i j indices
    else pesos[j] = pob[i].pesos[k]
  return (pesos, j + 1)
```

Tomamos un nº aleatorio en $[00, 10]$ y si es menor que CR, que en este caso es 0.5 entonces se el valor del gen j-ésimo valdrá lo que nos dia el operador de mutación; si no se toma lo que valga el gen j-ésimo del padre. Como CR vale 0.5 y por tanto se espera una mutación del 50 % de los genes, y no se indicaba nada en el esquema general he decidido no hacer la condición $j = j_{rand}$ ya que sería asegurar la mutación de al menos un gen y estaría bien para casos con CR bajo pero en este caso añadir una mutación mas o menos no lo veo necesario.

3.3.1. DE - Rand

El esquema es aleatorio, se toma como mutación 3 padres aleatorios (distintos entre sí y del individuo i-ésimo) y se toma una suma de ellos de una forma (y se restringe al intervalo $[00, 10]$):

```
mutRand :: EsqMutar
mutRand pob _ j (i1, i2, i3) = return $ restringe $ pob[i1].pesos[j]
  + 0.5 * (pob[i2].pesos[j] - pob[i3].pesos[j])
```

Por tanto:

```
dERand :: StdGen -> Algoritmo
dERand = dE mutRand
```

Obviamente en este caso no se tiene en cuenta al padre i-ésimo y además es una combinación aleatoria pura sin tener en cuenta nada.

3.3.2. DE - CurrentToBest

El esquema toma en consideración el padre i-ésimo, el mejor individuo de la población y dos padres aleatorios (distintos entre sí y del padre i-ésimo), además se restringe el valor al intervalo $[00, 10]$:

```
mutCurrentBest :: EsqMutar
mutCurrentBest pob i j i1 i2 = do
  let iMejor = maximum pob `indiceDe` pob
  return $ restringe $ pob[i].pesos[j] + 0.5 * (pob[iMejor].pesos[j]
    - pob[i].pesos[j] ) + 0.5 * (pob[i1].pesos[j] - pob[i2].pesos[j])
```

Por tanto:

```
dECurrentBest :: StdGen -> Algoritmo
dECurrentBest = dE mutCurrentBest
```

En este caso se tiene en cuenta más quien es el mejor y tira hacia él (explotación) añadiendo también un poco de exploración con los padres aleatorios pero teniendo en cuenta la base de la que parte (el padre i-ésimo).

4 Casos de comparación

4.1. P1

Se nos pide que comparemos estos algoritmos con los originales 1-NN (PESOS UNO) y RELIEF, como 1-NN simplemente es poner todos los pesos a 1.0, volvemos a explicar RELIEF solo.

4.1.1. RELIEF

Este algoritmo basado en técnica greedy es muy sencillo, empezamos inicializando el vector de pesos W a cero y usando el conjunto de entrenamiento, por cada punto de este actualizamos los pesos coordenada a coordenada de la siguiente manera: buscamos el punto más cercano de la misma clase (amigo) y le restamos la distancia 1 coordenada a coordenada, igual buscamos el punto más cercano que no sea de su clase y sumamos la distancia 1 coordenada a coordenada.

Finalmente truncamos a 0 los valores de los pesos negativos, y después normalizamos todos los valores en el intervalo $[0, 1]$.

La función principal:

```
relief :: Algoritmo
relief dEntrenamiento =
  let wCero        = repite (nCaract dEntrenamiento) 0.0
      wRes         = acumula (\w p -> actualizaPesos p trainData w) wCero dEntrenamiento
      wPos         = map (\w -> if w < 0.0 then 0.0 else w)
      (wMax, wMin) = (max wPos, min wPos)
  in map (normaliza w pMax pMin) wRes
```

La idea es simple, creamos los `pesosCero` que son los iniciales (de tamaño n), aplicamos para cada punto la función que lo actualiza y vamos realizándolo con cada punto hasta terminar. Finalmente normalizamos, trucando primero a 0 los valores negativos, sacando el máx y mín después y normalizando a $[0, 1]$.

La función para ir actualizando los pesos es:

```
actualizaPesos :: Dato -> Datos -> Pesos -> Pesos
actualizaPesos p dEntrenamiento pAcumulados =
```

```
  let dist             = ordena $ map (\x -> dist1 p x) (quita p dEntrenamiento)
      amigo            = buscar (\x -> claseDe x == claseDe p) dist
      enemigo          = buscar (\x-> claseDe x != claseDe p) dist
      sumaPeso p e a w = p' `dist1c` e - p' `dist1c` a + w
  in juntarCon4 sumaPeso (valoresDe p) enemigo amigo pAcumulados
```

Simplemente tomando un punto fijo, sacamos la distancia de ese punto a todos del conjunto de entrenamiento y luego lo ordenamos; como queremos sacar el aliado hemos sacado de las distancias el propino punto `p` y ahora buscamos el primero que aparezca en la lista con la misma clase que p, igual que con los enemigos solo que buscamos el primero que tenga distinta clase. Finalmente aplicamos coordenada a coordenada aplicando los pesos acumulados (de ir aplicando a los cada punto) junto a la suma de la distancia enemiga y resta distancia amiga.

Las distancia 1 y distancia 1 coordenada a coordenada son:

```
dist1 :: Dato -> Dato -> Float
dist1 p1 p2 = suma $ juntaCon dist1c p1 p2
```

```
dist1c :: Float -> Float -> Float
dist1c x y = abs (y - x)
```

4.2. P2

4.2.1. DE-Best

He añado una variante de DE, DE-Best, que el operador de mutación es el Best-1, implementado es:

```
mutBest :: EsqMutar
mutBest pob _ j (i1:i2:_) = do
  let (iMejor:_) = maximo pob `indiceDe` pob
  return $ restringe $ pob[iMejor].pesos[j]
    + 0.5 * (pob[i1].pesos[j] - pob[i2].pesos[j])
```

Y por tanto:

```
dEBest :: StdGen -> Algoritmo
dEBest = dE mutBest
```

5 Procedimiento considerado para desarrollar la práctica.

Todo el código está implementado en Haskell, sin usar ningún framework. El código está en la carpeta `FUENTES`. Para compilar he dejado un archivo make en `BIN` de manera que para instalar el compilador y sus dependencias solo hay que instalar `stack` aquí se puede consultar como instalarlo stack. Para compilar el archivo solo hay que hacer `make build`, aunque la primera vez tardará porque tiene que descargar el compilador de Haskell y las dependencias. Una vez terminado se puede ejecutar `make run` para ejecutarlo sin argumentos.

Los distintos modos de ejecución serían:

- Sin argumentos: entonces se leen los 3 dataset que se encuentran en `BIN` y se ejecutan con una seed aleatoria.
- Un argumento: se pasa como argumento el nombre del fichero que se quiere leer y se ejecuta con una seed aleatoria.
- Dos argumentos: se pasa como argumento el nombre del fichero que se quiere leer y el nº de seed, se ejecuta el archivo con la seed que se ha pasado.

Que se resume en `./P3bin [nombreFichero] [seed]`.

He comprobado que la compilación y ejecución es satisfactoria en Fedora 28.

6 Experimentos y análisis de resultados

6.1. Descripción de los casos del problema

Todos los dataset se han leído en formato `.arff` y como se explicó al principio elimino datos repetidos. Todas las características son reales.

6.1.1. Colposcopy

Colposcopy es un conjunto de datos de colposcopias anotado por médicos del Hospital Universitario de Caracas. El fichero tiene 287 ejemplos (imágenes) con 62 características (con distintos datos como valores medios/desviación estándar del color de distintos puntos de la imagen) de que deben ser clasificados en 2 clases (buena o mala).

Sin elementos repetidos seguimos teniendo 287 ejemplos efectivos; 71 elementos de la clase "0" y 216 elementos de la clase "1" habiendo un claro desequilibrio de clases.

6.1.2. Ionosphere

Ionosphere es un conjunto de datos de radar recogidos por un sistema en Goose Bay, Labrador. Se intentan medir electrones libres en la ionosfera y se clasifican en 2 clases (bad o good). Se cuenta con 354 ejemplos y 34 características (representan valores de las señales electromagnéticas emitidas por los electrones) cada dato.

Con 4 elementos repetidos nos quedan 350 ejemplos efectivos, quedando en 125 elementos de la clase "b" y 225 elementos de la clase "g" donde no hay equilibrio de clases.

6.1.3. Texture

Texture es un conjunto de datos de extración de imagénes para distinguir entre las 11 texturas diferentes (césped, piel de becerro prensada, papel hecho a mano...). Se tienen 550 ejemplos con 40 características que hay que clasificar en 11 clases.

Hay un elemento repetido que nos deja en 549 ejemplos donde todas las clases tienen 50 ejemplos excepto la clase "12" que tiene 49. Este conjunto de datos está equilibrado.

6.2. Resultados obtenidos

Siguiendo el formato indicado, incluyo aquí una tabla por cada algoritmo recogiendo los resultados de cada algoritmo sobre cada dataset; y una tabla global recogiendo los resultados medios y desviación típica para cada dataset y cada algoritmo. Las particiones son las mismas para cada algoritmo y los valores están redondeados a las 2 decimales más significativos.

La descripción de la tabla es:

- Nº : nº partición
- Clas: tasa de clasificación (%)
- Red: tasa de reducción (%)
- Agr: función agregada
- T: tiempo que ha tardado el algoritmo en calcular los pesos (s)

Los datos resultantes se han generado con el generador de nº aleatorios "225938972 1", ejecutados en un ordenador con SO Fedora 28, IntelCore i5-7300HQ CPU @ 2.50GHz x 4.

6.2.1. Algoritmos P1 (Aleatorios, 1NN, RELIEF, BL)

Resultados de la P1.

Tabla para los pesos todo uno (1NN normal) 6.1

Tabla para RELIEF 6.2

Tabla 6.1: Resultados obtenidos por el algoritmo 1NN (pesos uno) en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	74.14	0.00	37.07	0.00	84.29	0.00	42.14	0.00	90.00	0.00	45.00	0.00
2	81.03	0.00	40.51	0.00	85.71	0.00	42.86	0.00	91.82	0.00	45.90	0.00
3	82.46	0.00	41.23	0.00	92.86	0.00	46.43	0.00	95.46	0.00	47.73	0.00
4	68.42	0.00	34.21	0.00	84.29	0.00	42.14	0.00	90.00	0.00	45.00	0.00
5	78.95	0.00	39.47	0.00	90.00	0.00	45.00	0.00	94.50	0.00	47.25	0.00
$\bar{x}$	77.00	0.00	38.50	0.00	87.43	0.00	43.71	0.00	92.36	0.00	46.18	0.00
σ	5.13	0.00	2.56	0.00	3.43	0.00	1.72	0.00	2.26	0.00	1.13	0.00

Tabla 6.2: Resultados obtenidos por el algoritmo RELIEF en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	79.31	25.80	52.56	0.01	81.43	2.94	42.19	0.01	96.36	37.50	66.93	0.03
2	77.59	59.67	68.63	0.01	87.14	2.94	45.04	0.01	95.45	47.50	71.48	0.03
3	84.21	30.65	57.43	0.01	91.43	2.94	47.19	0.01	97.27	45.00	71.14	0.03
4	70.18	24.19	47.19	0.01	84.29	2.94	43.61	0.01	91.82	42.50	68.17	0.03
5	87.72	27.42	57.57	0.01	90.00	5.88	47.94	0.01	96.33	40.00	68.16	0.03
$\bar{x}$	79.80	33.55	56.67	0.01	86.86	3.53	45.19	0.01	95.45	42.50	68.97	0.03
σ	6.00	13.24	7.00	0.00	3.66	1.18	2.15	0.0	1.90	3.54	1.80	0.00

6.2.2. Algoritmos P3 (ES, ILS, DE)

Resultados de la P3.

Tabla para ES 6.3

Tabla para ILS 6.4

Tabla para DE-Rand 6.5

Tabla para DE-BestToCurrent 6.6

Tabla para DE-Best 6.7

Tabla 6.3: Resultados obtenidos por el algoritmo ES en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	74.14	83.87	79.01	116.51	87.14	91.18	89.16	81.27	90.00	85.00	87.50	221.00
2	70.69	80.65	75.67	115.13	84.29	88.24	86.26	69.66	90.91	85.00	87.96	219.29
3	73.68	79.03	76.36	86.58	85.71	91.18	88.45	61.83	88.18	80.00	84.09	222.18
4	80.70	77.42	79.06	116.56	84.29	88.24	86.26	81.24	89.09	82.50	85.80	228.49
5	77.19	87.10	82.15	115.56	85.71	91.18	88.45	62.32	89.00	85.00	87.00	230.29
$\bar{x}$	75.28	81.61	78.45	110.07	85.43	90.00	87.71	71.26	89.43	83.50	86.47	224.25
σ	3.40	3.48	2.41	15.94	1.07	1.44	1.22	8.62	0.94	2.00	1.39	4.33

Tabla 6.4: Resultados obtenidos por el algoritmo ILS en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	70.69	87.10	78.89	116.44	81.43	94.12	87.77	112.62	90.91	82.50	86.71	310.85
2	70.69	85.48	78.09	116.02	78.57	91.18	84.87	111.70	91.82	87.50	89.66	310.46
3	68.42	85.48	76.95	116.01	81.43	91.18	86.30	110.86	83.64	87.50	85.57	306.36
4	75.44	90.32	82.88	116.13	90.00	91.18	90.59	111.10	90.91	85.00	87.96	307.40
5	75.44	83.87	79.66	117.20	90.00	91.18	90.59	111.71	91.74	85.00	88.37	313.53
$\bar{x}$	72.14	86.45	79.29	116.36	84.29	91.77	88.03	111.60	89.80	85.50	87.65	309.72
σ	2.82	2.19	2.01	0.45	4.78	1.18	2.29	0.61	3.11	1.87	1.40	2.60

Tabla 6.5: Resultados obtenidos por el algoritmo DE-Rand en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	62.07	93.55	77.81	121.92	87.14	91.18	89.16	115.27	90.00	87.50	88.75	303.98
2	72.41	93.55	82.98	121.14	88.57	91.18	89.87	113.49	91.82	87.50	89.66	318.04
3	71.93	90.32	81.13	128.13	91.43	91.18	91.30	117.88	88.18	87.50	87.84	311.89
4	75.44	95.16	85.30	131.28	84.29	94.12	89.20	123.22	92.73	87.50	90.11	316.17
5	73.68	91.94	82.81	128.51	85.71	91.18	88.45	122.27	92.66	87.50	90.08	319.49
$\bar{x}$	71.11	92.90	82.00	126.20	87.43	91.77	89.60	118.43	91.08	87.50	89.29	313.92
σ	4.68	1.65	2.49	3.87	2.46	1.18	1.70	3.49	1.75	0.00	0.88	5.59

Tabla 6.6: Resultados obtenidos por el algoritmo DE-CurrentToBest en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	81.03	79.03	80.03	125.01	91.43	88.24	89.83	117.63	91.82	85.00	88.41	318.98
2	67.24	67.74	67.49	127.95	84.29	79.41	81.85	121.93	90.91	80.00	85.46	314.67
3	75.44	74.19	74.82	126.60	90.00	88.24	89.12	117.86	89.09	85.00	87.05	317.25
4	78.95	64.52	71.73	127.48	81.43	88.24	84.83	120.58	92.73	80.00	86.36	323.57
5	68.42	70.97	69.70	127.09	80.00	79.41	79.71	117.01	92.66	85.00	88.83	315.34
$\bar{x}$	74.22	71.29	72.75	126.83	85.43	84.71	85.07	119.00	91.44	83.00	87.22	317.96
σ	5.52	5.04	4.37	1.01	4.55	4.33	3.96	1.91	1.36	2.45	1.26	3.18

Tabla 6.7: Resultados obtenidos por el algoritmo DE-Best en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	75.86	70.97	73.41	129.32	90.00	73.53	81.77	121.61	90.91	77.50	84.21	310.18
2	70.69	72.58	71.64	122.57	81.43	88.24	84.83	116.79	94.55	80.00	87.27	313.55
3	78.95	79.03	78.99	123.24	85.71	82.35	84.03	115.29	85.46	80.00	82.73	319.06
4	75.44	83.87	79.66	128.91	84.28	85.29	84.79	117.99	91.82	80.00	85.91	319.44
5	70.18	75.81	72.99	129.84	82.86	82.35	82.61	124.19	89.91	80.00	84.95	329.91
$\bar{x}$	74.22	76.45	75.34	126.78	84.86	82.35	83.61	119.18	90.53	79.50	85.01	318.43
σ	3.33	4.63	3.31	3.18	2.94	4.92	1.22	3.26	2.97	1.00	1.53	6.71

6.2.3. Resultados globales

Tabla global media comparativa 6.8 y de desviación típica 6.9

Tabla 6.8: Resultados globales medios en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nombre	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1NN	77.00	0.00	38.50	0.00	87.43	0.00	43.72	0.00	92.35	0.00	46.18	0.00
RELIEF	79.80	33.55	56.67	0.01	86.86	3.53	45.19	0.01	95.45	42.50	68.97	0.03
ES	75.28	81.61	78.45	110.07	85.43	90.00	87.71	71.26	89.43	83.50	86.47	224.25
ILS	72.14	86.45	79.29	116.36	84.29	91.77	88.03	111.60	89.80	85.50	87.65	309.72
DE-RAND	71.11	92.90	82.00	126.20	87.43	91.77	89.60	118.43	91.08	87.50	89.29	313.92
DE-CURBST	74.22	71.29	72.75	126.83	85.43	84.71	85.07	119.00	91.44	83.00	87.22	317.96
DE-BEST	74.22	76.45	75.34	126.78	84.86	82.35	83.61	119.18	90.53	79.50	85.01	318.43

Tabla 6.9: Resultados globales desviación típica en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nombre	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1NN	5.13	0.00	2.56	0.00	3.43	0.00	1.72	0.00	2.26	0.00	1.13	0.00
RELIEF	6.00	13.24	7.00	0.00	3.66	1.18	2.15	0.0	1.90	3.54	1.80	0.00
ES	3.40	3.48	2.41	15.94	1.07	1.44	1.22	8.62	0.94	2.00	1.39	4.33
ILS	2.82	2.19	2.01	0.45	4.78	1.18	2.29	0.61	3.11	1.87	1.40	2.60
DE-RAND	4.68	1.65	2.49	3.87	2.46	1.18	1.70	3.49	1.75	0.00	0.88	5.59
DE-CURBST	5.52	5.04	4.37	1.01	4.55	4.33	3.96	1.91	1.36	2.45	1.26	3.18
DE-BEST	3.33	4.63	3.31	3.18	2.94	4.92	1.22	3.26	2.97	1.00	1.53	6.71

6.3. Análisis de resultados

6.3.1. Fiabilidad de los algoritmos

En vista de los resultados y sus desviaciones típicas engeneral podemos ver que no varían demasiado en los resultados, quizás DE-CurrentToBestda unos resultados un poco más dispersos en `Colposcopy` y `Ionosphere` pero no mucho más; también DE-Rand en `Ionosphere` en cuanto a la tasa de clasificación varía mucho. Lo que si se nota mucho más es la dispersión en el tiempo de ES en `Colposcopy` y después en `Ionosphere`, ya que la condición de parada incluye que se corte si no ha aceptado ningún vecino en una iteración y provoque estas dispersión tan alta en el tiempo.

En cualquier caso los resultados son consistentes en general.

6.3.2. Tiempo

La mayoría de algoritmos de la P3 usan la condición de evaluar hasta 15000 veces, por lo que los tiempos son muy parecidos entre sí, aquí ES varía debido a que tiene una condición extra de parada que hace que sea más rapido en algunos momentos con ventajas de 50/60s en `Ionosphere` y `Texture`; también podría añadirse la condición a `ILS` si quiesemos mejorar el tiempo. En cualquier caso si el factor de tiempo es indispensable podemos ver que `ES` da unos buenos resultados, no los mejores pero si se necesita contar con el tiempo entonces es la opción mas deseable.

6.3.3. Análisis Algoritmos P3

En este caso tenemos 3 tipos de algoritmos, ES de trayectorias simples, ILS de trayectorias múltiples y DE (y sus variantes) de búsqueda evolutiva.

Sobresale en los 3 dataset un algoritmo claramente por su valor de evaluación y es DE-Rand, ganando por varios puntos de margen en `Colposcopy` y un poco menos en `Ionosphere` y `Texture`, en todo caso hay que tener cuidado porque si bien es claramente mejor tanto en clasificación como en reducción en `Ionosphere` y `Texture`, en `Colposcopy` que es donde saca más puntos hay una diferencia brusca: por un lado tenemos una gran reducción frente a los otros (92.90) pero que tiene peor clasificación que todos (71.11), en los otros dos dataset tenemos clara elección por este algoritmo pero según las necesidades podría no ser el mejor para elegir en `Texture`.

Las otras dos variantes de DE, DE-CurrentToBest y DE-Best, se quedan bien cerca de los otros algoritmos en `Ionosphere` y `Texture` pero se alejan mucho en `Colposcopy`, y no hay opción clara entre estos dos ya que ganan en distintos dataset; lo que parece indicar los datos es que parece que sale más beneficioso usar un operador de mutación que amplifique la búsqueda (exploración) de la evolución y no la explotación (Best, CurrentToBest), probablemente porque se fuerza mucho la convergencia de la población.

Por otro lado tanto ILS como ES se quedan cerca de los resultados respecto DE-Rand por lo que tampoco son malas opciones, quedando ILS un poco por encima de ES en todas los dataset aunque por diferencias de un punto solamente. Parece que el hecho de ir usando BL, mutar y volver a arrancar hace que no caiga rápidamente en máximos locales; al igual que ES que la técnica de enfriamento le permite salirse de esos puntos locales y explorar más el vecindario hasta mejores soluciones.

Como ya he comentado antes cabe añadir la consideración del tiempo, que entonces queda bastante claro que ES es una opción buena, no se aleja mucho de las otras mejores, pero lo bueno que tiene es que puede acabar mucho más rapido que el resto de algoritmos siendo un factor muy atractivo para ciertas situaciones que lo necesitasen.

6.3.4. Análisis global

En vista de los resultados, queda claro que en función del valor de evaluación los nuevos algoritmos ganan sin duda por diferencias de 40/20 puntos, siendo una mejoría bastante grande. Si bien es cierto que 1NN y RELIEF tienen en general mucha mejor puntación en clasificación (algunos puntos más), esa pequeña pérdida de clasificación de los nuevos algoritmos se compensa grandemente con una mejora muy notable de reducción donde aqui si que se nota la diferencia abismal (obviamente 1NN tiene 0 en reducción): en `Colposcopy` se pasa del 33.55 al 92.90 (60 puntos), en `Ionosphere` del 3.53 al 91.77 (88 puntos) y en `Texture` del 42.50 a 87.50 (45 puntos).

Desde luego perder unos cuantos puntos de clasificación es malo, pero poder seguir clasificación casi igual de bien y encima solo usando en un caso hasta el 12 % solo de las variables cuando necesitabas el 96 % para hacerlo en un principio es una mejora descomunal (imagina la cantidad de datos que se pueden ahorrar en obtener, guardar, clasificar...).

Por tanto queda claro la eficacia de las heurísticas frente a las aproximaciones ingenuas.

6.3.5. Conclusión

Pese a las distintas maneras y formas que tienen las heurísticas de solucionar el problema, y que si puedo exprimir un poco más de mejora con una o con otra; queda sin lugar a dudas que el uso de información que se tiene, del poder evaluar el algoritmo, el poder entrenarlo en un conjunto de entrenamiento y saber como lo hace, frente a aproximaciones que no la usan (1NN) o usan aproximaciones estáticas, que no dependen del conjunto, que no se pueden adapatar y probar; es abrumador. Los resultados globales nos indican que podemos obtener mejorías inimaginables a un coste bajo relativo de tiempo (que incluso los algoritmos no están hiperoptimizados: se podría hacer paralelismo, tener mejores prestaciones, ...) y realmente cualquier algoritmo que se basa en estas ideas ofrece buenos resultados, la cuestión de perfilar cual es el mejor para cada caso queda abierta a experimentación.

Finalmente, dejando aparte 1NN y RELIEF, parece claro que entre estos algoritmos la mejor opción para resolver nuestro problema sería DE-Rand aunque teniendo en cuenta que ES siempre es una buena opción dado las tasas de tiempo mejor que tiene, aunque ILS siempre sería una buena alternativa también para DE-Rand.

7 Bibliografía

La plantilla latex de esta memoria es de Pablo Baeyens Fernández (disponible en GitHub).