Práctica 2.b: Técnicas de Búsqueda basadas en Poblaciones 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
```

3 Algoritmos de búsqueda

3.1. Consideraciones comunes

Aunque realmente los meméticos son algoritmos genéticos mezclados con BL, y todos los genéticos tienen el mismo esquema solo que variendo las formas de hacer algunos pasos, voy a explicar aparte las partes de cruce, mutación y reemplazamiento que son fundamentalmente distintas según el tipo de algoritmo, y tomarlas como la parte de algoritmos genéticos.

Por otra parte de la parte de algoritmos meméticos solo añadiré las funciones extra para seleccionar cromosomas y aplicar Búsqueda Local a estos.

3.1.1. Representación de datos

Los datos básicos para los algoritmos genéticos:

- Un **gen** (`Gen`) se representa con un `Double` entre $[0, 1]$.
- Un **cromosoma** (`Cromosoma`) será la misma estructura que usamos para el algoritmo BL `Solucion` que nos permite guardar los pesos y el fit, y para los meméticos nos servirá para guardar el nº de vecinos.
- Un **conjunto de cromosomas** (`Poblacion`) será una lista de cromosomas (`[Cromosoma]`).

Ahora, para poder usar una misma estructura general y solo intercambiar las distintas partes según el tipo de algoritmo definimos los siguientes tipos:

- Un **esquema inicial** (`EsqInicial`) es una función que toma los datos de entrada y crea una población `Datos -> Estado Poblacion`.
- Un **esquema de selección** (`EsqSeleccion`) es una función que toma la población y devuelve una lista de padres que van a cruzarse (`Poblacion -> Estado Poblacion`)
- Un **operador de cruce** (`OpCruce`) es una función que toma los datos, dos padres y devuelve los dos hijos nuevos (`Datos -> Cromosoma -> Cromosoma -> Estado (Cromosoma, Cromosoma)`).
- Un **esquema de cruce** (`EsqCruce`) es una función que toma la probabilidad de cruce de una pareja, los padres que van a cruzarse y devuelve la población de hijos cruzados, usando el operador de cruce dado (`Double -> OpCruce -> Poblacion -> Estado Poblacion`).

- Un **operador de mutación** (`OpMutacion`) es una función que toma un cromosoma y un vector de índices donde tiene que mutar los genes del cromosoma y devuelve el hijo mutado en esos genes (`Cromosoma -> [Int] -> Estado Cromosoma`).
- Un **esquema de mutación** (`EsqMutacion`) es una función que toma la probabilidad de mutación de un gen, la población y la muta según el operador de mutación, devolviendo la población mutada (`Double -> OpMutacion -> Poblacion -> Estado Poblacion`).
- Un **esquema de reemplazamiento** (`EsqReemp`) es una función que toma la población actual, los hijos y devuelve la nueva población (`Poblacion -> Poblacion -> Estado Poblacion`).

Finalmente para los algoritmos meméticos se ha implementado un tipo añadido:

- Un **esquema de búsqueda local** (`EsqBL`) es una función que toma los datos y la población para aplicar búsqueda local según algun criterio devolviendo otra población (`Datos -> Poblacion -> Estado Poblacion`).

3.1.2. Explicación general

En los algoritmos genéticos se mantiene una población, un conjunto de cromosomas que simulan las soluciones. Con un procedimiento se crea una poblacion inicial de tamaño el que se quiera y a continuación se hace un ciclo de generación que se repite hasta que se cumpla la condición de parada. Este ciclo intenta simular una población de seres vivos que se reproducen, crean nuevos hijos con mezcla de rasgos de los padres, que pueden mutar individualmente y que finalmente van reemplazando a los padres.

El ciclo consta de varias partes:

- **Selección**: de la población se selecciona de algun método dado un subconjunto de cromosomas que serán los padres que se crucen entre ellos para crear hijos.
- **Cruce**: de los padres se aplica un método para juntar 2 de ellos y crear 2 hijos que tengan una mezcla de genes de los padres.
- **Mutación**: cada hijo resultante del cruce tiene una probabilidad a nivel de gen de mutar.
- **Reemplazo**: finalmente se escoge una manera de reemplazar la población actual con los hijos creados, dando así lugar a la nueva población.

Además, los algoritmos meméticos se comportan igual que los genéticos solo que cada x generaciones incorporan al final de ciclo una aplicación de Búsqueda Local a un subconjunto de la población elegida según el método que se quiera.

3.1.2.1. Genéticos

En nuestra práctica haremos distinción entre 2 operadores de cruce: BLX-alpha y el aritmético; y dos tipos de esquemas genéticos: el **generacional** y el **estacionario**:

- El **generacional** en cada ciclo escoge tantos padres como nº de población (pueden ser repetidos) y los hijos creados reemplazan a la población entera con la única excepción de que si el mejor de la población no sobrevive se incluye automaticamente por el peor de los hijos (**elitismo**).
- En el **estacionario** solo se escogen 2 padres y solo se crean 2 hijos que compiten por entrar en la población actual.

Cabe añadir que la probabilidad de cruce en el generacional no tiene por que ser 1, mientras que en el estacionario si debe ser 1 (si no podria ni haber hijos).

3.1.2.2. Meméticos

Para los meméticos habrá 3 tipos de selección de cromosomas: o bien se cogen **todos**, o con una **probabilidad** P_{LS}, o a los k **mejores** de la población.

3.1.3. Esquema general

El pilar fundamental sobre el que se crean los 7 algoritmos tanto genéticos como meméticos es el siguiente:

```
esqGenetico :: EsqBL -> Int -> OpCruce -> Double -> Int -> EsqMutacion
                  -> Double -> EsqReemp -> StdGen -> Algoritmo
esqGenetico esqBL nPob opCruce pCruce nParejas esqMut pMut esqReemp gen datos =
  getPesos $ maximoDe $
  aplicaEstado (hastaQueM (maxIter 15000) cicloPob (crearPobIni nPob datos, 0)) (gen, 0)
  where
    cicloPob (pob, i) =
      do
        let padres = seleccion nParejas pob
        let hijos = cruce pCruce datos opCruce padres
        let hMutados = esqMut pMut (mutNormal 0.3 datos) hijos
        let nuevaPob = esqReemp pob hMutados
        ++i
        let pobBL = if (i % 10) == 0 then esqBL datos nuevaPob else nuevaPob
        return (pobBL, i)
```

Para crear un algoritmo genético o memético necesitamos lo siguiente:

- Un **esquema de Búsqueda Local**, en el memético hay 3 tipos y para los genéticos se aplica uno que devuelve la misma población inalterada.
- El **nº de la población**, es decir el nº de cromosomas, en nuestro caso 30 para genéticos, 10 para meméticos.
- Un **operador de cruce** que es BLX o aritmetico en nuestro caso.

- La **probabilidad de cruce** que será en nuestor caso 0.7 para el generacional y 1 si es estacionaria.
- El **nº de parejas a cruzar**, la mitad de la población si es generacional y 1 si es estacionaria.
- Un **esquema de mutación**, uno para generacional (fijo) o estacionario (por cromosoma).
- La **probabilidad de mutación** de un gen, en nuestro caso 0.001.
- Un **esquema de reemplazamiento**, el generacional o estacionario.

Pues simplemente se empieza creando una población inicial aleatoria de la cantidad de cromosomas indicada con `crearPobIni`, y empezando i = 0, este contador nos servirá para aplicar Búsqueda Local cada 10 generaciones. Entonces mientras no se cumpla la condición de parada (15k evaluaciones) aplicamos `cicloPob` que sigue un esquema normal de algoritmo genético:

- Primero seleccionamos los padres con el nº de parejas indicadas
- Con estos padres se cruzan segun el op de cruce designado y la prob de cruce.
- Los hijos resultados se mutan con el operador de mutación dado `mutNormal`.
- Se aplica el reemplazamiento según toque y obtenemos la nueva población, incrementando en 1 el nº de generaciones.
- Finalmente cada 10 iteraciones se aplica una búsqueda local, que en caso de los genéticos devuelve la misma población. Como siempre se evalua todo el estado con el estado `(gen, 0)`, y la población final despues de realizar 15k iteraciones se obtiene el máximo (mejor solución de la población) y obtenemos los pesos, que es nuestro resultado del algoritmo.

3.1.4. Población Inicial

Obviamente siguiendo la misma idea que en la P1 para crear la población inicial se repite la función para obtener unos pesos aleatorios entre $[0, 1]$ tantas como indique el nº de cromosomas.

```
crearPobIni :: Int -> EsqInicial
crearPobIni nPob datos = repite nPob (crearCromIni datos)

crearCromIni :: Datos -> Estado Cromosoma
crearCromIni datos =
  do
    let pesos = toma (nCaract datos) $ aleatoriosDe (0.0, 1.0)
    return (crearCromosoma datos pesos)
```

3.1.5. Selección

Para seleccionar los padres ahora hemos tomado la selección por torneo dos a dos. Esto consiste en seleccionar un nº de cromosomas igual a `nParejas * 2` de la siguiente manera: seleccionamos dos cromosomas al azar de la población y escogemos el mejor de ellos; repeitmos esto `nParejas * 2` veces y ya tenemos los padres.

```
seleccion :: Int -> EsqSeleccion
seleccion nParejas pob = repite (nParejas * 2) (torneoN 2 pob)
```

En este caso he implementado `torneoN` para escoger n cromosomas, para ello escogemos n índices aleatorios de cromosomas y entonces vamos sacando el maximo de esos cromosomas empezando con una solución vacía (f = 0) devolviendo el mejor de los n.

```
torneoN :: Int -> Poblacion -> Estado Cromosoma
torneoN n pob =
  do
    let inds = toma n $ aleatoriosDe (0, length pob - 1)
    return (acumula (\acc i -> max acc (pob !! i)) (Solucion vacía) inds)
```

Repetimos esto `nParejas * 2` y tenemos los padres.

3.2. Algoritmos genéticos

Ahora tenemos los 4 tipos distintos de algoritmos que obtenemos según el tipo de operador de cruce o esquema genético.

3.2.1. Operadores de Cruce

Para los operadores de cruce tenemos este esquema común:

```
cruce :: Double -> Datos -> EsqCruce
cruce pCruce datos opCruce padres =
  do
    let nCruces = redondea $ pCruce * sizeOf padres / 2
    let (cruces, noCruce) = divideEn (nCruces * 2) padres
    let (padres1, padres2) = divideEn nCruces cruces
    let hijos = juntaCon (opCruce datos) padres1 padres2
    return (hijos ++ noCruce)
```

Necesitamos la probabilidad de cruce, los datos para crear los cromosomas, el operador de cruce y los padres que van a cruzarse. Primero cogemos el nº de cruces esperado `nCruces` que será igual al nº de parejas (nº padres / 2) por la probabilidad de cruce; obviamente si es estacionario el nº de padres es 2 y por tanto el nº de cruces es igual a la

probailidad de cruce que es 1, luego 1 cruce; en el generacional será 15 (30 / 2) por 0.7 que redondeando son 11 cruces.

Nota: seguimos la estrategia para evitar tener que crear tantos nº aleatorios y fijamos el nº de cruces con el nº de cruces esperado.

Ahora como no todos los padres se van a cruzar hacemos una división de la lista de padres dejando `nCruces * 2` padres la izquierda `cruces` y el resto a la derecha `noCruces`; estos últimos no se cruzarán y por tanto los padres serán los propios hijos. Ahora los que se van a cruzar se dividen por `nCruces` de manera que obtenemos `padres1` y `padres2`, dos listas de padres, cada una con nº `nCruces` padres.

Finalmente juntamos el primer padre de `padres1` con el primer padre de `padres2` y asi hasta el final, es decir en un orden fijo, basándonos en que los padres ya estaban escogidos aleatoriamente por `torneoN`. Cada dos padres se les aplica el operador de cruce conveniente que nos da dos hijos.

Finalmente devolvemos la población uniendo los hijos con los que no se han cruzado.

3.2.1.1. Cruce BLX-alpha

Este operador de cruce va tomando de las mismas posiciones un gen de cada padre, entonces se busca obtener un valor aleatorio en el intervalo de los dos genes pero expandiendo ese intervalo a los lados para darle un poco de exploración a los algoritmos (representando los genes como valores sobre la recta real).

Entonces implementado quedaría:

```
blxAlpha :: Double -> OpCruce
blxAlpha alpha datos p1 p2 =
  do
    let res = juntaCon (cruzarGenes alpha) (getPesos p1) (getPesos p2)
    let (w1, w2) = separarGenes res
    return (crearCromosoma datos w1, crearCromosoma datos w2)
```

Tomamos los pesos de ambos padres y juntamos aplicando al gen de cada padre en las mismas posiciones `cruzarGenes` que lo que hace es lo explicado anteriormente, toma el maximo y el minimo de ambos y toma la longitud del intervalo. Entonces tomamos 2 valores aleatorios de una distribución uniforme sobre $[cMin - l \cdot \alpha, cMax + l \cdot \alpha]$ donde $\alpha \in [0, 1]$ es un parámetro que aumenta o empequeñe la longitud de amplitud de búsqueda.

```
cruzarGenes :: Double -> Gen -> Gen -> Estado (Gen, Gen)
cruzarGenes alpha i1 i2 =
  do
    let (cMax, cMin) = (max i1 i2, min i1 i2)
    let l = cMax - cMin
```

```
    let (a, b) = (cMin - l * alpha, cMax + l * alpha)
    let (g1, g2) = take 2 $ aleatoriosDe (a, b)
    return (restringe g1, restringe g2)
```

Cuanto más alto sea α más exploración y cuanto más pequeño menos exploración y mas se ciñe a lo que proporcionan los padres. En nuestro caso tomaremos $\alpha = 0.3$.

Obviamente tomamos 2 valores del intervalo porque vamos a producir dos hijos, por tanto un gen para cada uno; y además los restringimos al intervalo $[0, 1]$ con `restringe` para que no se pasen.

Como devolvemos tuplas de genes tenemos que usar una función auxiliar `separarGenes` para recomponer los genes y crear los dos hijos:

```
separarGenes :: U.Vector (Gen, Gen) -> (Pesos, Pesos)
separarGenes genes = (h1, h2)
  where (h1, h2) =
    acumula (\(acc1, acc2) (g1, g2) -> (acc1 ++ [g1], acc2 ++ [g2])) ([], []) genes
```

Simplemente creamos 2 listas vacías y conforme vamos iterando por los pares de genes vamos añadiendo cada gen a cada lista, obteniendo al final los 2 pesos de los 2 hijos.

3.2.1.2. Cruce aritmético

Este operador de cruce es más sencillo, consiste en hacer una media ponderada de los genes de cada padre en sus mismas posiciones de manera que la idea detrás seria hacer $h_1 = \alpha \cdot p_1 + (1 - \alpha) \cdot p_2$ y $h_2 = (1 - \alpha) \cdot p_1 + \alpha \cdot p_2$.

Así tenemos que un hijo se parecera más a un padre que al otro, y el otro hijo justo al revés donde cambia según variemos el parametro $\alpha \in [0, 1]$. Desde luego si $\alpha = \{0, 1\}$ entonces los dos hijos serán los dos padres, y en el caso especial $\alpha = 0.5$ tendremos que es la media aritmética de ambos genes: $\frac{g_1 + g_2}{2}$.

Según vayamos moviendo α los hijos se parecerán menos entre si y mas a un padre (cerca de los extremos $[0, 1]$) o tendrán más mezcla de los padres y por tanto los hijos se parecerán más entre sí hasta que sean iguales con $\alpha = 0.5$, que en nuestro caso escogemos este valor por lo que tendremos 2 hijos iguales favoreciendo la convergencia de la población.

La implementación en general es la siguiente:

```
cruceAritmetico :: Double -> OpCruce
cruceAritmetico alpha datos p1 p2 = do
  let cLineal i1 i2 = (alpha * i1 + (1 - alpha) * i2, (1 - alpha) * i1 + alpha * i2)
  let (w1, w2) = separarGenes $ juntaCon cLineal (getPesos p1) (getPesos p2)
  return (crearCromosoma datos w1, crearCromosoma datos w2)
```

Juntamos los pesos de ambos padres con la combinación lineal explicada arriba, que nos da una lista de tuplas de genes (como en BL) y aplicamos la misma función `separarGenes` que nos da los dos pesos para crear los hijos.

3.2.2. Mutación

En ambos casos el operador para mutar es el mismo de la P1, obtener de una distribución normal de media 0 y desviación estandar 0.3 un valor y sumarselo al gen que nos indique.

Por si un mismo cromosoma muta varias veces (caso generacional) pasamos una lista de índices de genes donde tiene que mutar, entonces aplicamos a los pesos del cromosoma `imap` que es igual que `map` solo que nos proporciona el índice en el que estamos. Entonces si el índice `i` está en la lista `iGenes` mutamos ese gen, si no lo dejamos igual.

```
mutNormal :: Double -> Datos -> OpMutacion
mutNormal sD datos hijo iGenes =
  do
    let pesos =
      imap (\i x -> if (i perteneceA iGenes then mutGen sD x else x) (getPesos hijo)
    crearCromosoma datos pesos
```

Para ello simplemente obtenemos un valor de la distribución normal y lo restringimos al intervalo $[0, 1]$ y lo devolvemos. Finalmente se aplica para todos los genes y se crea un nuevo cromosoma.

```
mutGen :: Double -> Gen -> Estado Gen
mutGen sD g =
  do
    let z = distribucionNormal (0, sD)
    return (restringe g + z)
```

3.2.2.1. Caso generacional

Para el caso generacional disponemos de una gran cantidad genes en total por lo que para evitar crear una cantidad ingente de nº aleatorios para nada, vamos a fijar el nº de mutaciones con el nº de mutaciones esperados a nivel de población de la siguiente manera $P_m = P_g \cdot M * n$ donde P_g es la probabilidad de mutación de un gen (0.001) en nuestro caso, M es el tamaño de la población y n el nº de características.

Por tanto tomaremos al azar tantos cromosomas como indique P_m y mutaremos aleatoriamente alguno de sus genes; obviamente podría darse el caso de que se seleccione el mismo cromosoma por lo que tenemos que evitar que mute el mismo gen más de una vez.

La implementación quedaría así:

```
mutGeneracional :: EsqMutacion
mutGeneracional pMutGen opMut hijos =
  do
    let nMuts = max 1 $ redondea (pMutGen * sizeOf hijos * getN hijos)
    let croMuts = agrupa $ ordena $ coge nMuts $ aleatoriosDe (0, length hijos - 1)
    return acumula (mutarCromosoma opMut) hijos croMuts
```

Empezamos viendo el nº de mutaciones esperado (explicado arriba) que lo obtenemos viendo el tamaño de la población y el nº de características dado por `getN` y con `pMutGen`, finalmente aplicamos `max 1` para que haya al menos una mutación por generación. Después tomamos ese nº de mutaciones de los cromosomas, cogiendo los índices de la lista pero como pueden repetirse lo que hacemos es ordenar la lista de índices obtenida y las agrupamos por su coincidencia.

Es decir si tenemos `[3, 1, 4, 2, 1, 3, 4, 4]` obtendriamos `[[1, 1], [2], [3, 3], [4, 4, 4]]`. Por tanto por cada sublista que representa que tenemos que mutar tantos genes como longitud tenga la sublista del cromosoma con índice el nº que indique, así vamos aplicando a los hijos un `acumula` para ir mutando el cromosoma indicado y devolviendo la población con ese cromosoma en concreto mutado para que se aplique el siguiente hasta acabar.

Para ir haciendo esta repetición tenemos `mutarCromosoma`:

```
mutarCromosoma :: OpMutacion -> Poblacion -> [Int] -> Estado Poblacion
mutarCromosoma opCruce pob indices =
  do
    let cro = pob !! primeroDe indices
    let nMuts = sizeOf indices
    let n = sizeOf (getPesos cro)
    let (_, iGenes) = repite nMuts indicesSinRepetir ([0..(n - 1)], [])
    let nuevoCro = opCruce cro iGenes
    return ([nuevoCro] ++ delete cro pob)
```

Como ya hemos dicho, el nº que hay en `indices` indica la posición de cromosoma a obtener, pues lo obtenemos sacando el primero por ejemplo; además, la longitud de los índices nos dice cuantos genes tenemos que mutar. Para ello tendremos que obtener índices aleatorios de `[0..(n - 1)]` siendo n el nº de características pero podrían repetirse. Para ello usamos la función `indicesSinRepetir` que nos devolverá una lista de índices de genes únicos de tamaño nMuts.

Acabamos pasando al operador de cruce el cromosoma y la lista, finalmente eliminamos de la población al antiguo cromosoma y añadimos el nuevo.

Para ver como hemos conseguido los índices sin repetir:

```
indicesSinRepetir :: ([Int], [Int]) -> Estado ([Int], [Int])
indicesSinRepetir (indices, acc) =
  do
```

```
    let i = aleatoriosDe (0, sizeOf indices - 1)
    return (delete (indices !! i) indices, acc ++ [indices !! i])
```

Simplemente cogemos la lista original `indices` que al principio contiene todas los índices de los genes, entonces generamos un índice aleatorio de esa lista, de esta manera eliminamos de `indices` el elemento enn la posición `i` y lo pasamos de nuevo junto a la lista de índices únicos donde añadimos el elemento `i`. Así hasta `nMuts` veces obtenniendo los índices sin repetir.

3.2.2.2. Caso Estacionario

El caso estacionario es más sencillo ya que solo tenemos 2 hijos que mutar, pero claro aquí las probabilidades de mutar como población entera son bastante cercanas a 0 por lo que tenemos que usar las probabilidades de mutar a nivel de cromosoma, esto es $P_m = P_g \cdot n$ siendo P_g la probabilidad de mutar de un gen (0.001) por n el nº de características.

Como solo tenemos 2 hijos no nos cuesta nada generar 2 nº aleatorios, para cada cromosoma si su nº aleatorio es menor que P_m entonces muta, escogemos otro nº aleatorio para el gen que se muta y acabamos.

Implementado sería:

```
mutEstacionario :: EsqMutacion
mutEstacionario pMutGen opMut hijos =
  do
    let pMutCro = pMutGen * getN hijos
    let pMuts = coge 2 aleatoriosDe (0.0, 1.0)
    let iGens = coge 2 aleatoriosDe (0, getN hijos - 1)
    let m1 =
      if pMuts !! 0 < pMutCro then opMut (hijos !! 0) [iGens !! 0] else hijos !! 0
    let m2 =
      if pMuts !! 1 < pMutCro then opMut (hijos !! 1) [iGens !! 1] else hijos !! 1
    return [m1, m2]
```

Tomamos `pMutCro` como habiamos dicho, y luego 2 nº aleatorios para ver si muta cada cromosoma de `pMuts` y otros 2 para los genes que mutan si lo hacen en `iGens`. Entonces si se cumple la condición se llama al operador de mutación con el cromosoma elegido en el índice aleatorio; si no se deja el hijo tal y como estaba.

Y devolvemos la lista de los dos hijos mutados o no según el resultado.

3.2.3. Reemplazo

Vemos los dos tipos de reemplazos posibles: generacional y estacionario.

3.2.3.1. Caso Generacional

En el generacional tomamos la población nueva como los hijos, solo en el caso de que el mejor de la población anterior no sobreviva se intercambia por el peor hijo:

```
reempGeneracional :: EsqReemp
reempGeneracional pActual hijos =
  if mejorP perteneceA hijos
    then return hijos
    else return (delete (minimoDe hijos) hijos ++ [mejorP])
  where mejorP = maximoDe pActual
```

3.2.3.2. Caso Estacionario

Por otro lado, en el estacionario se introducen los dos hijos y se pelean con la toda población, echando a los dos peores que haya, es decir, simplemente metemos a los dos hijos, ordenamos y quitamos los dos peores.

```
reempEstacionario :: EsqReemp
reempEstacionario pActual hijos = return (quita 2 $ ordena $ pActual ++ hijos)
```

3.2.4. BL

En los genéticos no aplicamos Búsqueda Local, por tanto usamos esta función para todos que deja la población igual:

```
noAplicaBL:: EsqBL
noAplicaBL _ pob = return pob
```

Se aplica a todos los algoritmos genéticos de manera que no se aplica BL.

3.2.5. AGG-BLX

Algoritmo genético generacional con cruce BLX-alpha.

```
aggBlx :: StdGen -> Algoritmo
aggBlx = esqGenetico noAplicaBL 30 (blxAlpha 0.3) 0.7 15 mutGener 0.001 reempGener
```

Población de 30 cromosomas con operador de cruce BLX-alpha con $\alpha = 0.3$, probabilidad de cruce 0.7, nº de parejas 15 (mitad de población), con esquema de mutación generacional, probabilidad de mutación 0.001 y esquema de reemplazo generacional.

3.2.6. AGG-CA

Algoritmo genético generacional con cruce aritmético.

```
aggCa :: StdGen -> Algoritmo
aggCa = esqGenetico noAplicaBL 30 (cruceAritm 0.5) 0.7 15 mutGener 0.001 reempGener
```

Población de 30 cromosomas con operador de cruce aritmetico con $\alpha = 0.5$, probabilidad de cruce 0.7, nº de parejas 15 (mitad de población), con esquema de mutación generacional, probabilidad de mutación 0.001 y esquema de reemplazo generacional.

3.2.7. AGE-BLX

Algoritmo genético estacional con cruce BLX-alpha.

```
ageBlx :: StdGen -> Algoritmo
ageBlx = esqGenetico noAplicaBL 30 (blxAlpha 0.3) 1.0 1 mutEstaci 0.001 reempEstaci
```

Población de 30 cromosomas con operador de cruce BLX-alpha con $\alpha = 0.3$, probabilidad de cruce 1.0, nº de parejas 1 (2 padres), con esquema de mutación estacionario, probabilidad de mutación 0.001 y esquema de reemplazo estacionario.

3.2.8. AGE-CA

Algoritmo genético estacional con cruce aritmético.

```
ageCa :: StdGen -> Algoritmo
ageCa = esqGenetico noAplicaBL 30 (cruceAritm 0.5) 1.0 1 mutEstaci 0.001 reempEstaci
```

Población de 30 cromosomas con operador de cruce aritmético con $\alpha = 0.5$, probabilidad de cruce 1.0, nº de parejas 1 (2 padres), con esquema de mutación estacionario, probabilidad de mutación 0.001 y esquema de reemplazo estacionario.

3.3. Algoritmos meméticos

Finalmente los algoritmos meméticos, que siguen la misma estructura de los genéticos comentados aqui arriba pero con la incorporación de la Búsqueda Local. Se van a basar sobre el algoritmo **AGG-BLX** que es el que mejor resultado de los generacionales me ha dado; usamos BL sobre los generacionales ya que estos priorizan la exploración, la diversidad frente a los estacionales que priorizan la explotación, generan mucha presión selectiva.

Por ello usamos la BL cada x generaciones para darles un equilibrio de explotación y ya que los generacionales exploran mucho, el operador de cruce BLX aumenta esa

exploración frente al aritmético que favorece la convergencia (al valor intermedio, mismos hijos) y con tanta exploración puede que converga a una solución peor que el resto de variantes.

En cualquier caso se ha hecho esta modificación de la Búsqueda Local de la P1 (y la función tambien se encuentra en `P1.hs`):

```
busLocMem :: Datos -> Cromosoma -> Estado Cromosoma
busLocMem datos cro =
  hastaQueM condParada (crearVecino datos) (cro, [0..(nCaract datos -1)])
    where condParad sol = nIter >= 15000 || getNVecinos sol >= (2 * (nCaract datos))
```

Esta función es practicamente igual que la Búsqueda Local salvo que la condición de parada ahora en vez de generar como máximo $20 \cdot n$ generamos $2 \cdot n$, siendo n el nº de características. Además en vez de trabajar con una solución aleatoria inicial empezamos con el cromosoma inicial (y también pequeñas modificaciones sin importancia por como funciona Haskell).

Finalmente, para cualquier estrategia para selecionar un subconjunto de cromosomas aplicaremos:

```
aplicaBL :: EsqBL
aplicaBL datos = map (busLocMem datos)
```

Que aplica BL a cada cromosoma del subconjunto tomado.

3.3.1. AM-(10, 1.0)

Este algoritmo memético aplica BL a toda la población, luego simplemente usaremos como esquema de BL:

```
aplicaTodos :: EsqBL
aplicaTodos = aplicaBL
```

Y tendremos el algoritmo:

```
amTodos :: StdGen -> Algoritmo
amTodos = esqGenetico aplicaTodos 10 (blxAlpha 0.3) 0.7 5 mutGener 0.001 reempGener
```

Algoritmo memético de población 10 con operador de cruce BLX-alpha con $\alpha = 0.3$, probabilidad de cruce 0.7, nº de parejas 5 (mitad de población 10), esquema de mutación generacional, probabilidad de mutación de gen 0.001, reemplazamiento generacional y esquema de BL a todos.

3.3.2. AM-(10, 0.1)

El esquema BL se aplica según una probabilidad P_{LS} que es la probabilidad de que a un cromosoma se le aplique BL. Igual que pasaba con las mutaciones para una población grande podriamos fijar el nº de aplicaciones, pero para el tamaño que tenemos que es 10 he decidido aplicar individualmente la probabilidad.

```
aplicaProb :: Double -> EsqBL
aplicaProb pLS datos pob =
  do
    let probs = take (length pob) $ aleatoriosDe (0.0, 1.0)
    let escoge (accE, accR, i) x = if (probs !! i) < pLS then (accE ++ [x], accR, i + 1)
    let (escogidos, rechazados, _) = acumula escoge ([], [], 0) pob
    let aplicados = aplicaBL datos escogidos
    return (aplicados ++ rechazados)
```

Lo que hacemos simplemente es tomar M numeros aleatorios siendo M el tamaño de la población, y ahora por cada cromosoma de la población voy viendo si el nº aleatorio correspondiente es menor que `pLS` que es la pobabilidad de cada cromosoma de ser aplicado BL, si es mejor entonces lo meto en la lista de escogidos, si no en la de rechazados. Finalmente aplicamos BL a los escogidos y el resultado lo mezclamos con los rechazos siendo la nueva población que devolvemos.

```
amProb :: StdGen -> Algoritmo
amProb = esqGenetico (aplicaProb 0.1) 10 (blxAlpha 0.3) 0.7 5 mutGener 0.001 reempGener
```

Algoritmo memético de población 10 con operador de cruce BLX-alpha con $\alpha = 0.3$, probabilidad de cruce 0.7, nº de parejas 5 (mitad de población 10), esquema de mutación generacional, probabilidad de mutación de gen 0.001, reemplazamiento generacional, y esquema de BL con probabilidad 0.1 (nº esperado de cromosomas escogidos 1).

3.3.3. AM-(10, 0.1mej)

El esquema BL se aplica solo a los $r \cdot M$ mejores de la población, siendo M el tamaño de la población y r el ratio que nos indica cuantos se van a escoger.

```
aplicaMejor :: Double -> EsqBL
aplicaMejor rMejor datos pob =
  do
    let nMejores = max 1 $ redondear (rMejor * sizeOf pob)
    let (rechazados, escogidos) = divideEn (sizeOf pob - nMejores) $ ordenar pob
    let aplicados = aplicaBL datos escogidos
    return $ (aplicados ++ rechazados)
```

Simplemente miramos el nº de mejores que vamos a escoger como ya hemos dicho, haciendo `max 1` al resultado para que al menos se escoja un cromosoma de la población.

Cogemos la población y la ordenamos y cortamos de manera que como se ordena de menor a mayor según el valor de la función objetivo queremos dejar a la derecha `nMejores` elementos y por tanto a la izquierda tenemos `sizeOf pob - nMejores`.

Aplicamos BL a los escogidos y juntamos el resultado con los rechazados.

```
amMejor :: StdGen -> Algoritmo
amMejor =
  esqGenetico (aplicaMejor 0.1) 10 (blxAlpha 0.3) 0.7 5 mutGener 0.001 reempGener
```

Algoritmo memético de población 10 con operador de cruce BLX-alpha con $\alpha = 0.3$, probabilidad de cruce 0.7, nº de parejas 5 (mitad de población 10), esquema de mutación generacional, probabilidad de mutación de gen 0.001, reemplazamiento generacional, y esquema de BL con ratio de mejor 01 (se escoge al mejor de la población).

4 Algoritmos de comparación

Se compara con los algoritmos hechos en la P1, cuyo explicación y pseudocódigo vuelvo a incluir aquí (es la misma, aunque BL he cambiado un poco los métodos, hace lo mismo pero ahora es más simple y facil de leer). Además he añadido dos variantes de meméticos para comparar.

4.1. P1

4.1.1. Pesos Aleatorios

Pesos aleatorios de una distribución uniforme $[0, 1]$:

```
pesosRand :: StdGen -> Algoritmo
pesosRand gen datos = toma (nCaract datos) $ randoms gen
```

4.1.2. Pesos Uno (1NN)

El clasificador 1NN normal, con todos los pesos a 1:

```
pesosUno :: Algoritmo
pesosUno trainData = repite (nCaract trainData) 1.0
```

4.1.3. Algoritmo 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)
```

Nota: en las transparencias se dice que para normalizar los pesos simplemente se divida entre los valores positivos el mayor valor de los pesos, obviamente en cuanto haya un valor negativo y se trunque al 0 es cierto este resultado, pero en cualquier caso yo aplico la normalización que no añade coste computacional y así aseguramos que la normalización es correcta.

4.1.4. Búsqueda local

La función principal sería:

```
busLoc :: StdGen -> Algoritmo
busLoc gen dTrain = getPesos $ aplicaEstado (hastaQueAplica fParada (crearVecino dTrain)
   where fParada sol = (getNVecinos sol >= 20 * (nCaract datos)) || (nIter >= 15000)
```

Para simplificar las cosas la idea es empezar con los pesos iniciales y mientras que no se cumpla la función de parada, que el nº de vecinos de la solución actual sea inferior a 20 * nº de caracteristicas o que el nº de evaluaciones de la función objetivo sea inferior a 15000 (obtenemos nIter a través del estado global), pues sacamos el mejor vecino del vecindario de la solución actual (que pudiera ser él mismo o no). Cuando pare obtendremos algo del tipo `Estado Solucion` que recordemos es una función que parte de un estado a otro devolviendo una objeto de tipo `Solucion` (la final), pues entonces partimos del estado inicial `(gen, 0)` el generador inicial de la semilla junto a 0 (0 nº de evaluaciones) y evaluando en la función nos devuelve la solución, de la que obtenemos los pesos.

Para obtener la solución inicial, buscamos unos pesos aleatorios de una distribución uniforme sobre $[0, 1]$, para ello usamos `aleatoriosDe (0.0, 1.0)` que devuelve un lista infinita de nº aleatorios sobre una distribución uniforme $[0, 1]$. Ademas devolvemos una lista de índices `[0..(nCaract datos - 1)]` que nos dirá sobre que posiciones podemos realizar el operador para crear un vecino.

```
pesosIniRand :: Datos -> Estado (Solucion, [Int])
pesosIniRand datos = do
  let pesos = coge (nCaract datos) $ aleatoriosDe (0.0, 1.0)
  ++nIter
  return (crearSolucion datos pesos, [0..(nCaract datos - 1)])
```

Al crear una solución estamos haciendo la estructura de datos y evaluando también ya que:

```
crearSolucion :: Datos -> Pesos -> Solucion
crearSolucion datos pesos =
  do
    ++nIter
    return (Solucion pesos (evaluarF datos pesos) 0)
```

Donde `evaluarF` ya se vio en las consideraciones comunes.

Volviendo a la función principal nos falta ver como se consigue un nuevo vecino:

```
crearVecino :: Datos -> (Solucion, [Int]) -> Estado (Solucion, [Int])
crearVecino datos (sol, indices) = do
  let (solNueva, solAct, indNuev) = obtenerVecino 0.3 datos indices sol
  let indNuev' = if solNueva > solAct || indNuev vacíos then [0..(nCaract datos - 1)] els
  return (max solNueva solAct, indNuev')
```

Tenemos la solución actual y el nº de indices de los genes que podemos seguir mutando, primero obtenemos una solución vecina con `obtenerVecino` (operador para obtener vecino), y además nos devuelve la solución actualizada (nº de vecinos incrementado) y los índices actualizados también. Si la solución nueva es mejor que la actual o nos hemos quedado sin índices que mutar regeneramos los índices y finalmente devolvemos la mejor solución con sus índices, que volverá a comprobar la condición de parada y si aun se ha cumplido vuelve a `crearVecino`.

Finalmente para crear los nuevos pesos mutando una componente:

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

Obtenemos un valor de una distribución normal de media 0 y desviación estándar 0.3, escogemos un índice random de la lista y devolvemos unos pesos sumándole esa modificación en la coordenada i-ésima junto a la lista de índices sin i. Además el valor nuevo del peso lo filtramos entre [0,1] con `restringe` ya que es la condición de que sea solución. `aumentaVecino` incrementa el contador de vecinos creados de esa solución.

Nota: he quitado bastantes cosas acerca de los estados y los generadores para simplificar el código, recuerdo que al crear una solución se aumenta automaticamente el nº de evaluaciones la función objetivo.

4.2. P2

He modificado los meméticos ya que veía que me daban mejores resultados que los genéticos, por tanto era más interesante ver si podían mejorarse algo tocando algun paramétro; además no he cogido el que aplica BL a los mejores ya que daba peores resultados luego parecía mas interesante cambiar los otros dos que daban los mejores valores.

4.2.1. AM-(10, 1.0, 0.7)

Es lo mismo que AM-(10, 1.0) pero el α del operador de cruce BLX-ALPHA vale 0.7

```
amTodosCambiado :: StdGen -> Algoritmo
amTodosCambiado
  = esqGenetico aplicaTodos 10 (blxAlpha 0.7) 0.7 5 mutGener 0.001 reempGener
```

4.2.2. AM-(10, 0.2, 0.7)

Es lo mismo que AM-(10, 0.1) pero la prob de aplicar BL aumenta a 0.2 y el operador de cruce BLX-ALPHA cambia el α a 0.7:

```
amProbCambiado :: StdGen -> Algoritmo
amProbCambiado
  = esqGenetico (aplicaProb 0.2) 10 (blxAlpha 0.7) 0.7 5 mutGener 0.001 reempGener
```

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. También se puede hacer solo `make` y hace todo lo necesario para compilar el ejecutable.

Los distintos modos de ejecución serían:

- Sin argumentos: 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 `./P2bin [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

NOTA: Vi que había una pequeño error en la P1, ya que al evaluar la función objetivo no pasaba los pesos reducidos, de ahí los resultados tan malos en `Texture` (ya que clasificaba muy bien con casi todos los pesos casi en 0.0), efectivamente BL hacía muy buenas puntuaciones. He vuelto a sacar los valores con la semilla indicada más abajo.

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 aleatorios 6.1

Tabla para los pesos todo uno (1NN normal) 6.2

Tabla para RELIEF 6.3

Tabla para Búsqueda Local 6.4

Tabla 6.1: Resultados obtenidos por el algoritmo PESOS ALEATORIOS en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	72.41	25.80	49.11	0.00	90.00	20.59	55.29	0.00	90.91	20.00	55.45	0.00
2	81.03	25.80	53.42	0.00	88.57	20.59	54.58	0.00	88.18	20.00	54.10	0.00
3	80.70	25.80	53.25	0.00	90.00	20.59	55.29	0.00	95.46	20.00	57.73	0.00
4	66.67	25.80	46.24	0.00	85.71	20.59	53.15	0.00	85.46	20.00	52.73	0.00
5	78.95	25.80	52.38	0.00	92.86	20.59	56.72	0.00	92.66	20.00	56.33	0.00
$\bar{x}$	75.95	25.81	50.88	0.00	89.43	20.59	55.00	0.00	90.53	20.00	55.27	0.00
σ	5.59	0.00	2.79	0.00	2.32	0.00	1.16	0.00	3.47	0.00	1.73	0.00

Tabla 6.2: 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 6.3: 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

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

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	72.41	83.87	78.14	34.30	82.86	94.12	88.49	24.64	90.91	85.00	87.96	60.03
2	68.97	83.87	76.42	67.25	81.43	88.24	84.83	14.01	92.73	87.50	90.11	46.06
3	68.42	85.48	76.95	54.38	94.29	91.18	92.73	17.73	86.36	87.50	86.93	58.55
4	70.18	85.48	77.83	35.00	88.57	91.18	89.87	16.27	80.91	80.00	80.46	43.15
5	71.93	82.26	77.09	55.90	90.00	91.18	90.59	15.14	95.41	85.00	90.21	66.17
$\bar{x}$	70.28	84.19	77.29	49.37	87.43	91.18	89.30	17.56	89.26	85.00	87.13	54.79
σ	1.57	1.21	0.62	12.81	4.73	1.86	2.62	3.75	2.96	2.74	3.57	8.75

6.2.2. Algoritmos P2 (Geńeticos y Meméticos)

Resultados de la P2.

Tabla para AGG-BLX 6.5

Tabla para AGG-CA 6.6

Tabla para AGE-BLX 6.7

Tabla para AGE-CA 6.8

Tabla para AM-(10, 1.0) 6.9

Tabla para AM-(10, 0.1) 6.10

Tabla para AM-(10, 0.1mej) 6.11

Tabla para AM-(10, 1.0, 0.7) 6.12

Tabla para AM-(10, 0.2, 0.7) 6.13

Tabla 6.5: Resultados obtenidos por el algoritmo AGG-BLX en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	74.14	69.36	71.75	115.95	90.00	82.36	86.18	105.07	89.09	82.50	85.80	288.38
2	67.24	74.19	70.72	115.69	81.43	82.35	81.89	108.09	90.91	77.50	84.21	288.58
3	75.44	72.58	74.01	117.77	90.00	88.24	89.11	113.90	88.12	77.50	82.84	290.33
4	78.95	69.36	74.15	116.87	84.29	88.24	86.26	112.33	96.36	80.00	88.18	294.62
5	75.44	69.36	72.40	116.73	84.29	82.35	83.32	112.21	89.91	80.00	84.95	292.29
$\bar{x}$	74.24	70.97	72.60	116.81	86.00	84.71	85.35	110.32	90.89	79.50	85.20	292.29
σ	3.85	2.04	1.32	0.66	3.43	2.88	2.52	3.26	2.89	1.87	1.78	4.26

Tabla 6.6: Resultados obtenidos por el algoritmo AGG-CA en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	72.41	69.36	70.88	107.84	88.57	79.41	83.99	109.88	95.46	67.50	81.48	291.66
2	81.03	67.74	74.39	112.45	85.71	85.29	85.50	113.42	87.27	72.50	79.89	288.10
3	68.42	67.74	68.08	110.18	91.43	79.41	85.42	126.96	86.36	65.00	75.68	287.05
4	73.68	66.13	69.91	115.19	84.29	82.35	83.32	115.17	91.82	70.00	80.91	287.78
5	75.44	67.74	71.59	111.62	84.29	76.47	80.38	112.10	90.83	75.00	82.91	289.92
$\bar{x}$	74.20	67.74	70.97	111.46	86.86	80.59	83.72	115.51	90.35	70.00	80.17	288.40
σ	4.13	1.02	2.08	2.44	2.77	3.00	1.87	5.98	3.28	3.54	2.45	1.67

Tabla 6.7: Resultados obtenidos por el algoritmo AGE-BLX en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	75.86	69.36	72.61	111.90	91.43	82.35	86.89	115.04	91.82	82.50	87.16	288.03
2	68.97	67.74	68.35	112.40	95.72	79.41	87.56	115.78	91.82	80.00	85.91	290.27
3	70.18	79.03	74.60	112.57	88.57	82.35	85.46	116.50	84.55	77.50	81.02	286.91
4	70.18	66.13	68.15	109.49	85.71	88.24	86.98	116.67	90.91	82.50	86.71	288.32
5	80.70	72.58	76.64	110.59	87.14	82.35	84.75	113.01	96.33	80.00	88.17	295.40
$\bar{x}$	73.18	70.97	72.07	111.39	89.71	82.94	86.33	115.40	91.08	80.50	85.79	289.79
σ	4.46	4.56	3.37	1.18	3.55	2.88	1.05	1.33	3.78	1.87	2.50	3.01

Tabla 6.8: Resultados obtenidos por el algoritmo AGE-CA en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	77.59	67.74	72.66	113.54	91.43	79.42	86.42	108.52	93.64	72.50	83.07	281.94
2	72.41	62.90	67.66	109.86	84.29	82.35	83.32	108.79	87.27	75.00	81.14	283.13
3	70.18	59.68	64.93	112.35	87.14	76.47	81.81	107.77	90.00	72.50	81.25	280.77
4	78.95	67.74	73.35	112.31	82.86	79.41	81.14	108.89	90.91	75.00	82.96	281.93
5	75.44	66.13	70.78	115.05	85.71	79.41	82.56	110.16	93.58	72.50	83.04	280.63
$\bar{x}$	74.91	64.84	69.88	112.62	86.29	79.41	82.85	108.83	91.68	73.50	82.29	281.68
σ	3.24	3.13	3.16	1.71	2.94	1.86	1.48	0.77	2.39	1.23	0.90	0.91

Tabla 6.9: Resultados obtenidos por el algoritmo AM-(10, 1.0) en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	75.86	82.26	79.06	120.65	87.14	91.18	89.16	115.93	83.64	85.00	84.32	294.27
2	75.86	85.48	80.67	122.70	92.86	88.24	90.55	111.86	94.55	87.50	91.02	297.33
3	68.42	83.87	76.15	121.41	80.00	91.18	85.59	113.55	87.27	85.00	86.14	290.39
4	75.44	87.10	81.27	116.17	85.71	91.18	88.45	112.93	90.00	85.00	87.50	290.52
5	70.18	83.87	77.02	113.39	87.14	91.18	89.16	113.14	88.99	87.50	88.25	291.73
$\bar{x}$	73.15	84.52	78.83	118.86	86.57	90.59	88.58	113.48	88.89	86.00	87.45	292.85
σ	3.20	1.65	1.99	3.51	4.10	1.18	1.64	1.35	3.56	1.23	2.23	2.64

Tabla 6.10: Resultados obtenidos por el algoritmo AM-(10, 0.1) en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	81.04	85.48	83.26	109.98	85.71	91.18	88.45	112.43	90.91	82.50	86.71	286.63
2	72.41	82.26	77.34	111.07	92.86	88.24	90.55	114.96	90.91	87.50	89.21	284.62
3	63.16	83.87	73.51	111.25	87.14	88.24	87.69	109.40	88.18	82.50	85.34	278.04
4	75.44	83.87	79.66	112.10	87.14	91.18	89.20	109.49	91.82	87.50	89.66	281.43
5	71.93	87.10	79.51	111.39	91.43	91.18	91.30	109.42	88.99	82.50	85.75	284.91
$\bar{x}$	72.80	84.52	78.66	111.16	88.86	90.00	89.43	111.14	90.16	84.50	87.33	283.13
σ	5.81	1.65	3.20	0.69	2.77	1.44	1.33	2.24	1.35	2.45	1.78	3.05

Tabla 6.11: Resultados obtenidos por el algoritmo AM-(10, 0.1mej) en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	70.69	80.65	75.67	109.98	84.29	82.35	83.32	113.98	94.55	82.50	88.52	289.12
2	77.59	85.48	81.64	110.92	82.86	91.18	87.02	110.11	90.00	87.50	88.75	285.38
3	57.89	80.65	69.27	111.71	90.00	88.24	89.12	108.22	83.64	85.00	84.32	276.69
4	84.21	83.87	84.04	112.61	90.00	88.24	89.12	110.81	86.36	87.50	86.93	283.65
5	73.68	74.19	73.94	110.79	85.71	88.24	86.98	110.67	91.74	85.00	88.37	280.36
$\bar{x}$	72.81	80.97	76.89	111.20	86.57	87.65	87.11	110.76	89.26	85.50	87.38	283.04
σ	8.73	3.87	5.31	0.89	2.94	2.88	2.12	1.86	3.87	1.87	1.66	4.25

Tabla 6.12: Resultados obtenidos por el algoritmo AM-(10, 1.0, 0.7) en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	68.97	83.87	76.42	111.03	91.43	88.24	89.83	110.48	94.55	85.00	89.77	290.71
2	72.41	82.26	77.34	110.94	85.71	91.18	88.45	110.25	94.55	85.00	89.77	290.53
3	68.42	90.32	79.37	111.97	87.14	91.18	89.16	109.50	82.73	87.50	85.11	288.29
4	77.19	80.65	78.92	111.88	84.29	94.12	89.20	109.52	85.46	85.00	85.23	288.08
5	73.68	83.87	78.78	112.41	87.14	85.29	86.22	109.86	93.58	87.50	90.54	292.12
$\bar{x}$	72.14	84.19	78.17	111.65	87.14	90.00	88.57	109.92	90.17	86.00	88.09	289.95
σ	3.22	3.29	1.11	0.57	2.39	3.00	1.26	0.39	5.05	1.23	2.40	1.54

Tabla 6.13: Resultados obtenidos por el algoritmo AM-(10, 0.2, 0.7) en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	77.59	85.48	81.54	110.23	82.86	88.24	85.55	106.08	94.55	85.00	89.77	286.06
2	74.14	83.87	79.00	110.03	78.57	91.18	84.87	106.20	94.55	87.50	91.02	288.96
3	71.93	83.87	77.90	111.01	94.29	88.24	91.26	106.27	87.27	80.00	83.64	288.24
4	73.68	88.71	81.20	112.07	81.43	94.12	87.77	107.15	88.18	85.00	86.59	293.35
5	68.42	85.48	76.95	111.83	87.14	88.24	87.69	107.23	88.99	87.50	88.25	293.94
$\bar{x}$	73.15	85.48	79.32	111.03	84.86	90.00	87.43	106.57	90.71	85.00	87.85	290.11
σ	3.00	1.77	1.80	0.82	5.47	2.35	2.23	0.50	3.18	2.74	2.58	3.05

6.2.3. Resultados globales

Tabla global media comparativa 6.14 y de desviación típica 6.15

Tabla 6.14: Resultados globales medios en el problema del APC

	Colposcopy				Ionosphere				Texture			
Nombre	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
ALEATORIO	75.95	25.80	50.88	0.00	89.43	20.59	55.00	0.00	90.53	20.00	55.27	0.00
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
BL	70.38	84.19	77.29	49.37	87.43	91.18	89.30	17.56	89.26	85.00	87.13	54.79
AGG-BLX	74.24	70.97	72.60	116.81	86.00	84.71	85.35	110.32	90.89	79.50	85.20	292.29
AGG-CA	74.20	67.74	70.97	111.46	86.86	80.59	83.72	115.51	90.35	70.00	80.17	288.40
AGE-BLX	73.18	70.97	72.07	111.39	89.71	82.94	86.33	115.40	91.08	80.50	85.79	289.79
AGE-CA	74.91	64.84	69.88	112.62	86.29	79.41	82.85	108.83	91.68	73.50	82.29	281.68
AM-TODOS	73.15	84.52	78.83	118.86	86.57	90.59	88.58	113.48	88.89	86.00	87.45	292.85
AM-PROB	72.80	84.52	78.66	111.16	88.86	90.00	89.43	111.14	90.16	84.50	87.33	283.13
AM-MEJ	72.81	80.97	76.89	111.20	86.57	87.65	87.11	110.76	89.26	85.50	87.38	283.04
AM-TDS-MOD	72.14	84.19	78.17	111.65	87.14	90.00	88.57	109.92	90.17	86.00	88.09	289.95
AM-PRB-MOD	73.15	85.48	79.32	111.03	84.86	90.00	87.43	106.57	90.71	85.00	87.85	290.11

Tabla 6.15: 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**
ALEATORIO	5.59	0.00	2.79	0.00	2.32	0.00	1.16	0.00	3.47	0.00	1.73	0.00
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
BL	1.57	1.21	0.62	12.81	4.73	1.86	2.62	3.75	2.95	2.74	3.57	8.75
AGG-BLX	3.85	2.04	1.32	0.66	3.43	2.88	2.52	3.26	2.89	1.87	1.78	4.26
AGG-CA	4.13	1.02	2.08	2.44	2.77	3.00	1.87	5.98	3.28	3.54	2.45	1.67
AGE-BLX	4.46	4.56	3.37	1.18	3.55	2.88	1.05	1.33	3.78	1.87	2.50	3.01
AGE-CA	3.24	3.13	3.16	1.71	2.94	1.86	1.48	0.77	2.39	1.23	0.90	0.91
AM-TODOS	3.20	1.65	1.99	3.51	4.10	1.18	1.64	1.35	3.56	1.23	2.23	2.64
AM-PROB	5.81	1.65	3.20	0.69	2.77	1.44	1.33	2.24	1.35	2.45	1.78	3.05
AM-MEJ	8.73	3.87	5.31	0.89	2.94	2.88	2.12	1.86	3.87	1.87	1.66	4.25
AM-TDS-MOD	3.22	3.29	1.11	0.57	2.39	3.00	1.26	0.39	5.05	1.23	2.40	1.54
AM-PRB-MOD	3.00	1.77	1.80	0.82	5.47	2.35	2.23	0.50	3.18	2.74	2.58	3.05

6.3. Análisis de resultados

6.3.1. Exploración vs explotación

El hecho de usar algoritmos genéticos parte de la base de imitar el comportamento población de como "viven" los seres vivos, al tener una población conseguimos variedad de soluciones, incentivamos el aspecto de **exploración** para nuestros algoritmos; es decir poder explorar en el entorno sin quedarnos atrapados en un máximo local.

De los genéticos tenemos dos variantes, el **generacional** que reemplaza por completo excepto por el mejor de la población incrementado la exploración pero vamos destruyendo la población continuamente esperando que los hijos sean mejores y convergan hacia algo. Por otro lado tenemos el **estacionario** que solo crea 2 hijos en cada generación y entran a competir con la población entera provocando una gran presión selectiva en la población obligando a quedarse con los mejores, por tanto esta variante potencia mas el apartado de **explotación** de soluciones, es decir, buscar los máximos.

Además de esto podemos combinarlos con dos tipos de cruce, **BLX-alpha** que nos permite aportar la información de los padres y además añade un poco de exploración para aumentar la exploracion; por el otro lado el cruce **aritmético** provoca que los dos hijos creados sean iguales y además tiendan a la mitad justa de ambos padres, provocando una convergencia más rapida aumentando la **explotación**.

Finalmente los meméticos intentan equilibrar la buena capacidad de exploración de los AGE mejorando el apartado de explotación, aplicando cada ciertas generaciones BL a un subconjunto de la población para dirigir las soluciones a unas mejores.

6.3.2. Análisis de tiempo y dispersión

Antes de nada comentar que aparte de que en el dataset **Colposcopy** la dispersión aumente un poquito, en general para todos se obtienen resultados parecidos entre particiones, sin mucha dispersión por lo que nos dice que podemos decir que los algoritmos tienen buena consistencia.

Más adelante comentaré si es preferible un algoritmo a otro según el tiempo obtenido, pero como ya dije en la P1 es cuestión de lo que se busque, los tiempos de los meméticos/genéticos salen altos pero no está ni de lejos optimizado (se podrían parelizar cosas) ni tampoco los tamaños de la población ni la condición de parada son muy altos por lo que la cuestión reacae en si tengo un tiempo bastante razonable para obtener una solución pero que sea una buena solución o por el contrario tengo un margen de tiempo ajustado y una buena solución me vale.

En APC las aplicaciones más directas es poder conseguir clasificar cosas correctamente con el menor nº de características, ya que cuantas más tengas mas volumen de información necesitas y en problemas reales seguramente escalen muy rapido de tamaño; por tanto

probablemente se prefiera un algoritmo que tarde (de nuevo, una cantidad razonable) pero que nos de mejores frente a uno que sea más rapido pero un poco peor.

Desde la P1 ya se podía ver con BL que en **Texture** tarda más, efectivamente en los algoritmso genéticos/meméticos tarda un poco menos del triple que en los otros dos dataset, debido al nº mucho más grande de datos que tiene este dataset.

6.3.3. Análisis de los algoritmos genéticos

De los resultados obtenidos podemos ver que los 4 algoritmos genéticos devuelven unos resultados bastante parecidos donde en algunos dataset el orden de mejor y peor cambian entre sí pero hay algo que si queda más evidente.

AGG-BLX y AGE-BLX predominan claramente sobre AGG-CA y AGE-CA, siento mas notable la diferencia en el dataset **Texture** por ejemplo, esto nos indica que el operador de cruce **BLX-alpha** nos ofrece unos mejores resultados que el aritmético; está claro que BLX-alpha permite que los hijos puedan variar en un rango más permisivo que CA y por supuesto añade un margen de exploración extra; además CA crea 2 hijos identicos con genes justo la mitad lo que puede que tire hacia máximos locales peores y de ahi los resultados un poco peores. En cualquier caso la diferencia no es tremendamente elevada.

En cuanto a si es mejor AGG o AGE no queda claro, AGG-BLX y AGE-BLX tienen resultados muy aproximados, teniendo que en **Texture** y **Ionosphere** gane AGE-BLX pero en **Colposcopy** gane AGG-BLX, por lo que podemos decir que son igual de buenos, habría que usar más repeticiones con otras semillas, tocar paramétros para ver si en algunas situaciones mejora más uno que otro, pero a lo que nuestros resultados nos dicen más o menos son igual de buenos (con CA también pasa igual).

Los tiempos entre todos son casi los mismos dado que la condición de parada es la misma, e igual pasa con la tasa de acierto, de reducción y de agregado, como se ve en las tablas.

6.3.4. Análisis de los algoritmos meméticos

Detrás de la idea de mejorar los AGG, quedaba claro que el operador de cruce BLX era el mejor y por tanto los meméticos se han implementado sobre AGG-BLX. Vuelve a pasar que las 3 variantes ofrecen unos resultados muy parecidos entre sí, por ejemplo en **Texture** el resultado es casi identico.

Lo que si se nota es que el algoritmo AM-TODOS (BL aplicado a toda la población) gana en 2 y se queda segundo en 1, lo que nos indica que impulsar todas las soluciones de la población hacia máximos es bastante prometedor, aunque AM-PROB (aplicando BL con probabilidad 0.1) tampoco se queda atrás, y de los 3 parece que AM-MEJOR (aplicar BL a los 0.1 * M mejores) sea un poquito peor. En cualquier caso habría que ver si las diferencias son fruto del azar o hay realmente diferencias significativas, pero desde luego se puede ver que todas las variantes devuelven unas buenas soluciones.

En cualquier caso, arece que es mejor idea mejorar aquellas soluciones que son peores o aleatoriamente que mejorar las que ya son buenas, ya que queremos que las malas no sigan explorando por zonas con soluciones mejores.

Además, se manifiesta el equilibrio que aporta BL al aportar la explotación que necesita AGG-BLX, como explora tanto, BL puede encaminar un poco las soluciones mientras que se sigue explorando el espacio de soluciones; es decir intentamos mover las soluciones a zonas más prometedoras para que explore en esa zona y no en cualquier zona donde puede no haber buenos máximos.

De nuevo, los tiempos obtenidos son casi identicos entre sí, al igual que los resultados como ya he comentado.

6.3.5. Meméticos modificados

Nada reseñable que comentar, se obtienen resultados parecidos a sus partes originales, mostrando una mejoría ligera (0.5 puntos) en **Colposcopy** con AM-PRB-MOD y en **Texture** con AM-TDS-MOD; sin embargo quedan por debajo en **Ionosphere**.

6.3.6. Genéticos vs Meméticos

En la batalla genéticos vs meméticos sin duda alguna ganan los meméticos, desde luego el equilibrio que aportan desbancan a los genéticos en varios puntos sobre la función objetivo. Es deseable desde luego intentar tener balanceados la explotación y la exploración, aunque se podría conseguir mucho más retocando parámetros.

Como de mejores son, en **Colposcopy** ganan locon holgura con entre 4 y 6 puntos, en **Ionosphere** unos 3 y 4 puntos, y en **Texture** unos 2 y 3 puntos; además tienen la ventaja de que tardan igual que los genéticos por lo que sin duda en estos dataset cogeremos alguno de los meméticos, probablemente AM-TODOS O AM-MEJOR para desbancar a todos.

Los meméticos modificados mejoran medio punto en **Colposcopy** y en **Texture**, AM-PRB-MOD y AM-TDS-MOD respectivamente pero en **Ionosphere** no, así que hay que elegir bien según que dataset estemos usando.

6.3.7. Análisis global

De la P1 quedó claro que BL es un buen algoritmo que da muy buenos resultados en poco tiempo y aquí queda de manifiesto más todavia, comparando con los genéticos BL gana sin dudas, por unos pocos puntos pero desde luego es mejor tanto por tiempo (muchísimo más rápido) como por los resultados.

Los meméticos sin embargo consiguen ganar a BL aunque no todos, AM-MEJOR pierde en **Ionosphere** y **Colposcopy** por 1-2 puntos mientras que en **Texture** gana por 0.2 puntos por lo que podemos pensar que BL ofrece mejor rendimiento que AM-MEJOR. Los que salen ganando son AM-PROB y AM-TODOS aunque no por tanto, en **Texture** ganan por décimas, en **Colposcopy** por un poco menos de 1 punto y en **Ionosphere** AM-PROB es una décima mejor mientras que AM-TODOS es peor en casi 1 punto.

Los meméticos modificados ya he comentado como mejoran en 2 dataset pero en cualquier caso hay que tener cuidado, ya que con pequeñas diferencias hay que tener en cuenta quizás la desviación típica (para asegurarnos mejor estabilidad de los resultados) donde AM-TDS-MOD nos ofrece mejores resultados en **Texture** pero por otro lado su desviación típica sube a un 5.05 contra AM-TODOS (segundo mejor) con un 3.56.

Queda claro que los algoritmos de la P1 que no usaban ningún tipo de entrenamiento basado en la función objetivo (1NN, Aleatorios, RELIEF) quedan ya muy atrás dando un salto grande hasta los genéticos y finalmente otro pequeño salto, habiendo una brecha poco clara entre BL y los meméticos.

6.3.8. Conclusión

Concluimos múltiples cosas como que el operador de cruce BLX devuelve mejores resultados que el aritmético; que no hay una diferencia clara entre si es mejor el modelo estacionario que el generacional, mientras que los meméticos, que equilibran explotación y exploración, dan mejores resultados. Además, mejorar aleatoriamente o a todos con BL parece ser mejor que solo a los mejores de la población.

Por otro lado BL sigue siendo mejor que los genéticos y depende de los meméticos, y encima no tarda tanto como ellos así que según el caso probablemente habría que estudiar bien el problema en el que estamos; así BL sigue siendo una opción iteresante según sus buenos resultados y además si hay cierto tiempo límite sin duda alguna tomaría BL por la diferencia abismal de tiempos; si no lo hubiera y necesito exprimir hasta el última mejoría posible y no tengo tiempo iría con AM-TODOS o AM-PROB según cual ofrezca mejores resultados.

En cualquier caso hay que tener en cuenta que podemos tocar muchos paramétros que hemos dejado fijos debido a los recursos limitados que contamos pero podriamos haber hecho una población mucho mas grande, aumentar el límite de evaluaciones de la función objetivo, parelilzar para tardar menos tiempo... podríamos estudiar así si ante una mejoraría de recursos BL no se beneficie tanto como los meméticos/genéticos y consigan dar soluciones todavía mejores.

Queda así reflejado en los meméticos modificados que pueden mejorar sus resultados aunque es cierto que según que dataset, un algoritmo puede devolver un pequeño rendimiento más que otro.

7 Bibliografía

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