Práctica 1.b: Técnicas de Búsqueda Local y Algoritmos Greedy 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. 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.

3.2. 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 Casos de comparación

He añadido dos métodos simples para comparar: uno que es `pesosUno` con todos los pesos a 1 que equivale al clasificador 1-NN normal; y otro `pesosRand` que saca sus pesos de una distribución uniforme entre $[0, 1]$, es decir una solución aleatoria.

Así tenemos:

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

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

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 `./P1bin [nombreFichero] [seed]`.

He comprobado que la compilación y ejecución es satisfactoria tanto en Windwows 10 como 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, 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). Como el análisis de la P1 depende de los anteriores resultados, los dejo intactos aunque si comento que claramente la Búsqueda Local supera enormemente los resultados de los otros algoritmos, dando unas buenas puntuaciones. Para las comparaciones de resultados globales en otras prácticas si que incluiré los resultados obtenidos corregidos con la misma semilla.

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 “478721762 1”, ejecutados en un ordenador con SO Windows 10, IntelCore i7-6700K CPU @ 4.00GHz x 4.

Tabla para los pesos aleatorios 6.1

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

Tabla para RELIEF 6.6

Tabla para Búsqueda Local 6.4

Y la tabla global media comparativa 6.5 y de desviación típica 6.6

6.3. Análisis de resultados

Antes de analizar propiamente los resultados me gustaría analizar ciertas cosas que hemos fijado de antemano pero que realmente podíamos haber variado:

- La función objetivo: sabemos que la hemos fijado con $\alpha = 0,5$ dándole la misma importancia a reducir pesos que a acertar. En primer lugar cabría hacerse la pregunta de para que querer reducir los pesos.

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	70,69	90,32	80,56	48,56	88,57	85,29	86,93	30,62	9,09	100,00	54,55	97,45
2	75,86	83,87	79,87	37,41	78,57	94,11	86,34	26,17	46,36	95,00	70,68	75,23
3	73,68	75,80	74,74	34,05	81,43	94,11	87,77	29,73	25,45	97,50	61,48	96,29
4	71,93	91,94	81,94	56,53	84,29	91,17	87,73	28,86	78,18	90,00	84,09	129,83
5	71,93	74,20	73,06	32,47	35,71	100,0	67,86	27,10	69,73	87,50	78,61	65,75
$\bar{x}$	72,82	83,22	78,02	41,80	73,71	92,94	83,32	28,50	45,76	94,00	69,88	92,71
σ	1,80	7,26	3,48	9,26	19,29	4,78	7,75	1,65	26,01	4,64	10,81	22,11

Tabla 6.5: 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	72,82	83,22	78,02	41,80	73,71	92,94	83,33	28,50	45,76	94,00	69,88	92,71

Tabla 6.6: 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	1,80	7,26	3,48	9,26	19,29	4,78	7,75	1,65	26,01	4,64	10,81	22,11

Está claro que por varias razones, sabemos que ir tomando valores de una muestra (los datos que obtenemos) puede ser costoso por razones de tiempo, dinero... y tener que recolectar muchos y con muchas características afecta, por tanto si podemos seguir clasificando con menos características (nos olvidaríamos de las características con peso nulo) obtendríamos un beneficio sustancial.

Por otro lado aunque no tan importante (ya que el mayor beneficio de esto es lo comentado anteriormente) se podría ganar en tiempo de cálculo al reducir la dimensión del problema quitando las características que no importan.

Todo esto esta muy bien pero no hay que perder de vista que queremos hacer un clasificador que lo haga bien, aunque está claro que dependiendo de las circunstancias del problema vamos a querer que lo haga bien con mayor o menor margen de error. Por eso ajustar con el mismo peso la reducción que el acierto puede hacer que se tiendan a soluciones simples pero no con toda la precisión que buscamos ya que probablemente queramos algo que tenga valores altos de acierto.

Así pues puede que para otros problemas tuviéramos que subir ese α para darle más importancia al acierto pues pueden pasar cosas como ya veremos en BL.

- 1-NN: hemos fijado el clasificador 1-NN pero realmente podríamos haber cogido cualquier k, no había ninguna restricción al respecto. La realidad es que probablemente haya que hacer un estudio previo para ver un entorno de k óptimo;

  Por estadística sabemos que hay un balance entre la **flexibilidad** y el **error**; cuanto más flexible (mejor se adapta a los datos) provoca el famoso **sobreajuste** que hace que se ajuste tan bien a los datos de entrenamiento que aumente el error en los datos de prueba/reales y además se vuelve menos interpretable; por otro lado cuanto menos **flexible** más simple es y tiene mejor interpretabilidad pero a cambio suele dar muchos errores a la hora de clasificar

  En nuestro caso $k = 1$ produce los menores errores posibles (es muy flexible) pero por ello los límites no están tan claros y es muy sensible al sobreajuste. En otros casos deberíamos hacer pruebas distintas cambiando la k pero teniendo en cuenta que pudiera afectar de distinta manera a los algoritmos que hemos implementado.

Por otro lado hay una consideración extra que es el tiempo. En principio no tenemos ninguna restricción nada más que no tarde un tiempo razonablemente alto, ya que una vez producidos los pesos lo único que hay que hacer es aplicar 1NN cuando se quiera clasificar algo; entonces con una ejecución, aunque tarde mucho, si solo hay que hacer una se podría considerar un tiempo alto si nos va a dar una buena solución mejor que el resto. Por otro lado si estamos en un problema que puede que necesiten calcular varias veces, con restricciones de tiempo puede que nos interese otro algoritmo.

La conclusión es que dependiendo de las condiciones del problema un algoritmo puede ser mejor que otro y en otras condiciones pudiera ser al revés (no tiene por qué, claro); y por supuesto puede que varíen drásticamente al cambiar los hiperparámetros; la función

objetivo es la que marca la bondad de las soluciones y en ella lo que influye principalmente las restricciones de reducción/acierto de nuestro problema concreto.

6.4. Pesos aleatorios

Me ha sorprendido bastante que poniendo unos pesos aleatorios siga dando muy buenos resultados, por un lado ya que estamos tomando valores de una uniforme $[0, 1]$ es esperable que la reducción esté en torno al 20 % y efectivamente se refleja así en los resultados; por otro lado los altas tasas de acierto sobre todo en "Texture" y en "Ionosphere" aunque ya varía un poco en "Colposcopy", pero en general son buenas soluciones en torno al 50 de función objetivo y con resultados parecidos entre particiones (dispersión baja).

El resultado era esperado en verdad, recordando que estamos usando 1-NN ya de base es un buen clasificador de por si, con los pesos menores lo que hacemos es "acercar" las distancias en esa dirección (bajando la dimensión del problema), la cuestión son las posibles dependencias que haya entre las variables entre sí, con las clases... de entre un conjunto prometedor grande de características puede que quites aleatoriamente algunas pero el resto te sigan sirviendo para clasificar.

Para que quede claro, cualquiera dudaría de una solución aleatoria de buenos resultados pero la verdad es que sobre estos pesos aleatorios subyace el 1-NN que no es aleatorio ni es trivial, de ahi que aun asi siga dando buenos resultados; de hecho una solución aleatoria de clasificación sería poner todos los pesos a cero, todos tendrían distancia 0 y se escogería la clase por el orden aleatorio de los datos (y aun así según el criterio que hemos usado para la función objetivo tendríamos como mínimo 50 de puntuación por lo que nos hace sospechar que igual deberíamos ajustar el valor de α)

El problema reside en que siempre vamos a reducir en torno a un 20 % y puede que queramos buscar un conjunto mucho más pequeño de características para seguir clasificando correctamente; pero como aproximación completamente ilusa está bastante bien.

6.5. Pesos uno (1-NN)

El clasificador normal 1-NN, que como era de esperar ofrece unos resultados parecidos a los pesos aleatorios, en unos databases acierto mejor que los aleatorios y en otros al revés; pero parece que la tendencia son unos resultados aproximados. Sin embargo lo "malo" de este algoritmo es que no reduce nada los pesos (son todos 1.0) y como ya he comentado los beneficios de rebajar los pesos está claro que no nos interesa mucho y se ve claramente se refleja en que obtiene valores de función objetivo menores que los pesos aleatorios.

Tanto como 1-NN como los aleatorios mantienen los resultados muy parecidos entre particiones (dispersión baja) y el tiempo obviamente es nulo ya que solo es generar n números.

6.6. Relief

Este es el primer algoritmo "inteligente" (que tiene una estrategia no trivial) y observamos que su tasa de clasificación es buena, y aunque mantiene pocas diferencias con los pesos aleatorios la tasa de reducción aumenta en 2 de los dataset que llega hasta 42 % de reducción aunque en "Ionosphere" por ejemplo obtiene un 3,53 %.

Como clasificador sigue manteniendo una buena tasa un poco por encima de los aleatorios/1nn pero tiene una buena mejora en la reducción lo que nos hace mejorar sustancialmente su función de evaluación en "Colposcopy" y "Texture" pero cae siendo peor que los aleatorios en "Ionosphere".

En cuestión de tiempo tarda bastante poco 0,01 a 0,03s; lo cual nos indica que es un buen algoritmo si queremos un clasificador bueno y que reduzca moderadamente los pesos en un valor próximo a 0 segundos; pero desde luego no nos da la solución óptima o mejor en todos los casos, va variando entre aleatorios y relief aunque es un buen resultado mejor que arriesgarse a tomar valores aleatorios.

Por otro lado los datos se mantienen más o menos estables entre particiones, la dispersión típica es baja (más o menos igual que los otros algoritmos).

6.7. Búsqueda local

Varias cosas destacan de la búsqueda local:

- La tasa de clasificación: aunque en "Colposcopy" y en "Ionosphere" ofrece unos resultados medianamente aceptables (son buenos excepto por una partición) son inferiores en resultado a los algoritmos anteriores; hasta el extremo que son mucho peores en "Texture" con un 45,76 % de media.
- La dispersión de los datos: siguiendo con la tasa de clasificación vemos que en "Texture" los resultados de cada partición son muy dispares, varía totalmente de una partición a otra lo que nos provoca una desviación típica altísima en "Texture" lo que provoca que la tasa de acierto oscile en un intervalo muy amplio, haciendolo poco fiable; eso sí, para los valores de reducción se mantienen cercanos. En "Ionosphere" también hay alta debido a una partición que saca una valor bastante fuera del rango pero igual habría que hacer más ejecuciones para ver a donde tiende la desviación.
- Los valores de reducción: si aunque en dos dataset los valores de clasificación se mantienen parecidos y en el otro se vaya, en cuestión de reducción obtenemos unos valores altísimos en torno al 90 % y en algunos casos llegando al 100 %.
- Los tiempos de ejecucción: si antes teniamos algoritmos que prácticamente no tardaban, ahora es todo lo contrario, obtenemos tiempos de ejecución por partición de 30s hasta 90s (minuto y medio).

Como ya he comentado antes, el tiempo de ejecución será importante o no según el problema en el que estemos, en nuestro problema actual desde luego 1 minuto y medio por ejecución es alto respecto a los otros pero realmente no es nada si pensamos que una vez ejecutado ya tenemos nuestra solución mejor que las otras.

Que por otro lado vemos que en algunos casos el acierto es el mismo pero en "Texture" llega a ser un desastre; sin embargo recordemos que estamos dandole igual valor a la reducción que al acierto y como hemos conseguido tanta reducción obtenemos el valor más alto de la función objetivo incluso siendo más o menos consistente en "Texture" debido a la desviación típica baja de la reducción.

Desde luego si queremos una solución que reduzca lo más posible los pesos la búsqueda local nos da una solución que mantiene buenos resultados y los reduce mucho más que los otros algoritmos; aunque habría que estudiar bien el dataset concreto ya que podríamos encontrarnos con unos resultados como el de "Texture".

Para corregir errores como el de "Texture" habría que considerar cambiar el α para darle más peso al acierto y hacer que en la búsqueda se tienda a darle prioridad al acierto.

6.8. Resultados globales

Queda claro que tanto en "Colposcopy" como en "Ionosphere" la búsqueda local nos da una solución con una tasa de acierto parecida al resto pero que destaca altamente en su tasa de reducción, dando unos valores de agregación muy por encima del resto de algoritmos; eso sí a coste de un tiempo de ejecución mayor. Además en estos conjuntos podemos decir que los resultados son mas o menos consistentes por sus desviaciones típicas.

Sin embargo en "Texture" está claro que con un tiempo de ejecución tan alto nos dan unos resultados poco fiables (muy dispersos y con poco acierto), con mucha reducción si pero que en cuestión de función de evaluación se quedan casi iguales que su compañero RELIEF por lo que en este dataset convendría mejor usar los pesos RELIEF, que son mucho mas fiables.

Está claro que en ciertos dataset como "Texture" habría que tener cuidado con la función objetivo y priorizar el acierto para evitar estos resultados; por otro lado el hecho de que un vector de todos 0 nos de al menos un valor de 50 siempre nos cabe considerar de que posiblemente debamos subir α para priorizar el acierto.

En cualquier caso si disponemos de un tiempo razonable de tiempo, y ajustando bien los hiperparámetros BL podría darnos buenos resultados y en el caso de que no, el algoritmo RELIEF pudiera servirnos. Desde luego para ambos algoritmos se consiguen en general mejores resultados que las aproximaciones simples (1NN y aleatorios).

Nota: los resultados pueden variar ligeralmente usando semillas distintas (las particiones son distintas para cada semilla ya que hay un mezclado de datos previo) pero sobre todo

cambian en la búsqueda local y los pesos aleatorios, un buen estudio sería hacer varias ejecuciones de BL para ver el intervalo de resultados.

6.9. Comparación con solución BL dada

Se nos daba la siguientes soluciones 6.7.

Comparando con la solución BL que he obtenido yo en "Colposcopy" como en "Ionosphere" desde luego las tasas de acierto se mantienen iguales o un poco peores pero no mucho; en cambio las tasas de reducción se nota bastante, teniendo mi solución unas tasas bastante más altas de reducción lo que nos da un balance de función objetivo alta.

Por otro lado en "Texture" que sigue teniendo mucha más reducción, tienen casi los mismos valores de función objetivo ya que aunque no reduzca mucho la solución dada, sus aciertos son muy buenos, del 90 %; lo cual hace que esta solución sea bastante mejor (y mas fiable) que la mía en el conjunto "Texture".

Por otro lado los tiempos de mi algoritmo BL son en general más rápidos notandose más en "Colposcopy" y "Ionosphere".

Tabla 6.7: Resultados obtenidos por el algoritmo BL en el problema del APC (Solución dada)

	Colposcopy				Ionosphere				Texture			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	80,20	25,99	53,10	100,00	73,21	25,99	49,60	100,00	91,92	25,99	58,96	100,00
2	80,20	32,70	56,45	100,00	73,21	32,70	52,96	100,00	91,92	32,70	62,31	100,00
3	80,20	45,22	62,71	100,00	73,21	45,22	59,22	100,00	91,92	45,22	68,57	100,00
4	80,20	50,00	65,10	100,00	73,21	50,00	61,61	100,00	91,92	50,00	70,96	100,00
5	79,15	37,62	58,39	215,00	73,21	37,62	55,42	215,00	91,92	37,62	64,77	215,00
$\bar{x}$	79,99	38,31	59,15	132,00	73,21	38,31	55,76	123,00	91,92	38,31	65,11	123,00

7 Bibliografía

La plantilla latex de esta memoria es de Pablo Baeyens Fernández (disponible en GitHub). Debido a que era mi primera vez programando en Haskell le agradezco mucho la ayuda para realizar las cosas, así como también otras cuestiones de menor importancia para instalación de las dependencias, compilación...