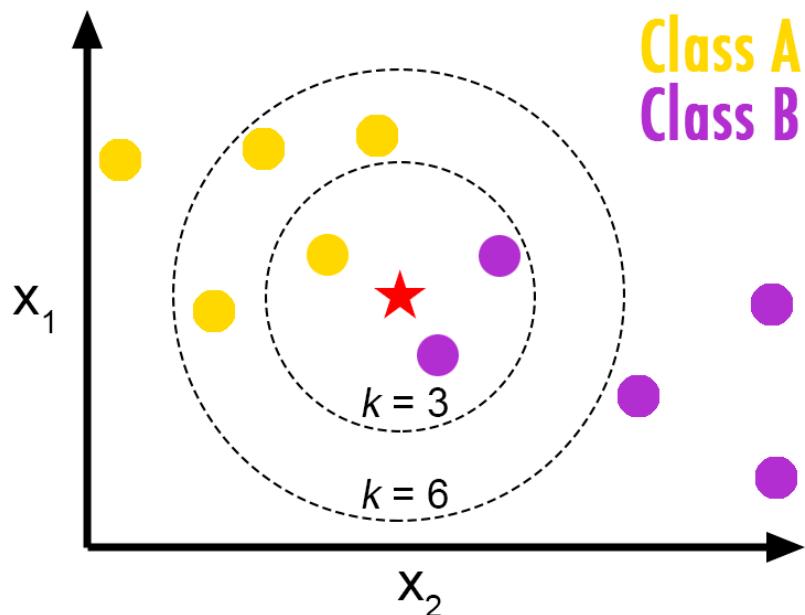


Práctica 1.b: Técnicas de Búsqueda Local y Algoritmos Greedy para el problema del Aprendizaje de Pesos en Características



APC | KNN , RELIEF, LOCAL RESEARCH

Autor: Sergio Aguilera Ramírez

DNI: 77446084R

Correo: sergioaguilera@correo.ugr.es

Grupo y horario: grupo 2 , miércoles 17:30 – 19:

Curso: 2018-2019

Índice:

	PÁGINA
1. Descripción del problema	2
2. Descripción general de los algoritmos	3-4
a. Representación de soluciones	
b. Representación de la función objetivo	
3. Algoritmos	5-6-7
a. Busqueda Local (Local Research)	
4. Algoritmo de comparación	7-8-9
5. Procedimiento de desarrollo de la práctica	10
6. Experimentos y análisis de los resultados	10 a 16
a. Casos del problema	
b. Parámetros utilizados en la práctica	
c. Resultados obtenidos	
d. Análisis de los resultados	
7. Bibliografía	16

1- Descripción del problema

Este problema se basa en la clasificación a partir del entrenamiento de datos y su validación mediante conjuntos de datos prueba. Disponemos de diferentes tipos de datos: colposcopias (colposcopy.arff), datos de radar (ionosphere.arff) y datos sobre diferentes tipos de texturas (textura.arff). A partir de estos datos y de los algoritmos desarrollados debemos encontrar el mejor valor para la función objetivo. Además buscamos reducir el número de características para seleccionar las que nos proporcionen un resultado más óptimo. Para el entrenamiento y el test de los algoritmos, utilizamos el método de validación 5-fold cross validation, el cual consiste en dividir los datos en 5 particiones manteniendo la distribución de las clases.

Nuestro clasificador es un K Nearest Neighbors (KNN), k=1, sirve para calcular la tasa de clasificación de los datos del test, después del entrenamiento.

Por un lado, haremos uso del clasificador KNN en los diferentes algoritmos desarrollados para encontrar un vector de pesos. Este nos permite dividir los datos según su clase de forma más óptima. Para conseguir esto, debemos dar una mayor importancia a las características con mayor relevancia (dividen mejor los datos en sus respectivas clases). Por otro lado para la búsqueda de dicho vector utilizaremos algoritmos como Relief o búsqueda local, los cuales consisten en la modificación del vector de pesos para aumentar el porcentaje de aciertos en la clasificación.

$$\text{Tasa_clas} = 100 * \text{num_aciertos en T} / \text{num_instancias en T}$$

$$\text{Tasa_red} = 100 * \text{num_carac_nulas} / \text{num_carac}$$

$$F(w) = \text{Alpha} * \text{tasa_clas}(w) + (1-\text{alpha}) * \text{tasa_red}(w)$$

2- Descripción general de los algoritmos

Para esta práctica vamos a obtener los diferentes resultados de un clasificador KNN, aplicando dos algoritmos para la búsqueda de un vector de pesos:

KNN: este clasificador no utiliza ninguna técnica de reducción de características ni ningún peso sino que se encarga de tomar todas las características y en función a estas encontrar el vecino mas cercano y calcular su predicción.

Relief: en este algoritmo se calcula un vector de pesos que se multiplica por los datos (características). Esto será útil para dar más importancia a aquellas que se consideren mas relevantes, también puede ser que elimine (reducción) una característica por no ser de utilidad a la hora de predecir.

Búsqueda Local: este algoritmo consiste en encontrar una solución “óptima”, aunque no es probable que se encuentre, mediante la modificación de cada peso del vector de pesos para así ir mejorando el porcentaje de aciertos en conjuntos de datos test. Sin embargo no es probable que se encuentre.

El clasificador KNN usado en mi práctica es el clasificador de la librería scikit-learn de Python. Así mismo, también hago uso de esta librería para la división de las particiones (StratifiedKFold), usada para dividir el conjunto de datos en “n” subconjuntos manteniendo la distribución de clases. Por otro lado empleo la librería time para la medición de tiempo y el paquete MinMaxScaler proporcionado por la librería scikit-learn para la normalización de los datos. Finalmente en la lectura de los ficheros de datos .arff utilizo el paquete arff del módulo scipy.io.

Como he comentado en el apartado 1 los datos serán divididos siguiendo el método de k-fold cross validation (para nosotros el número de particiones será 5), para esto he utilizado el paquete StratifiedKFold. Este genera 5 particiones disjuntas al 20% manteniendo la distribución de clases.

La representación de las soluciones en los diferentes algoritmos son: w es el vector de pesos el cual nos proporcionara una mejora en la clasificación de los datos en el KNN. Dicho vector de pesos tiene tantos valores como características presenta nuestros datos. Los pesos menores a 0.2 serán truncados a 0.0 eliminando la totalidad de su importancia y de la misma manera, los pesos superiores a 1.0 serán truncados a 1.0. En cada iteración de cada partición fragmentamos los datos en trainx, trainy, testx y testy que contendrán los distintos datos para entrenamiento y validación. Por último calcularemos la tasa_clas y tasa_red para el posterior cálculo de la función objetivo. El objetivo de los algoritmos desarrollados es la maximización de dicha función a través de la obtención de un vector de pesos lo mas eficaz posible para la partición precisa de los datos.

En esta práctica vamos a emplear 3 ficheros, colposcopy.arff, ionosphere.arff y textura.arff. Cada fichero será dividido en 5 particiones (comentado anteriormente) y serán ejecutadas por cada algoritmo implementado. Al final se va a obtener la media de los resultados obtenidos en cada ejecución (tasa_clas_media, tasa_red_media, agr_medio y tiempo_medio). Cada fichero presenta una cantidad de características y clases diferentes para así comprobar la eficacia de clasificación de los algoritmos.

$$\text{Tasa_clas} = 100 * \text{num_aciertos en T} / \text{num_instancias en T}$$

$$\text{Tasa_red} = 100 * \text{num_carac_nulas} / \text{num_carac}$$

$$F(w) = \text{Alpha} * \text{tasa_clas}(w) + (1-\text{alpha}) * \text{tasa_red}(w)$$

3- Descripción y pseudocódigo de los algoritmos

Funciones Comunes a todos los Algoritmos:

```
/* En este algoritmo particionamos los conjuntos train y test de cada  
partición */
```

```
Prepara_particiones(i,j,x,y)
```

```
    Iteramos en i
```

```
        Trainx = x[i]
```

```
        Trainy = y[i]
```

```
    Iteramos en j
```

```
        Testx = x[j]
```

```
        Testy = x[j]
```

```
/* Esta función calcula el porcentaje de aciertos obtenidos en el conjunto  
de datos pasado como argumento, entrenando primero con los datos train  
y validando con los datos test, clasificador KNN */
```

```
Calcula_porcentaje_acertado(trainx, trainy, testx, testy)
```

```
    Clasificador = KNeighborsClassifier(n_neighbors=1)
```

```
    Clasificador.fit(trainx,trainy)
```

```
    For j in len(testx)
```

```
        Prediccion = clasificador.predict(testx[j])
```

```
        Si testy[j] == predicción
```

```
            Cont_aciertos ++
```

```
    Return (100 * cont_aciertos / len(testx))
```

BÚSQUEDA LOCAL

```
Max_val = -99999.9
Inicio = time()
W = np.random.uniform(0.0,1.0,len(x[0]))
Trainx, trainy, testx,testy = prepara_particiones()

Mientras que vecinos < 20 * len(w) y it < 15000

    Para cada wi en w
        Z = np.random.normal(0.0,0.3)
        It++
        w_aux = wi

        w[l] += z
        si wi < 0.2
            wi = 0.0
            cont_reduccion ++
        si wi > 1.0
            wi = 1.0

        /* multiplicamos los datos por los pesos obtenidos
        trainx_a = trainx * w
        testx_a = testx * w

        tasa_clas = calcula_porcentaje_acertados(trainx_a,
trainy, testx_a,testy)
        tasa_red = 100*cont_reduccion/len(w)
        valor_funcion = 0.5 * tasa_clas + (1-0.5) * tasa_red

        si valor_funcion > max_val
            entonces max_val = valor_funcion y vecinos = 0 ,
            break
        si no
            wi = w_aux
            vecinos++
```

```

trainx *= w
testx *= w
tasa_clas = calcula_porcentaje_acertados(trainx, trainy, testx,testy)
tasa_red = 100*cont_reduccion/len(w)
función_objetivo = 0.5 * tasa_clas + (1-0.5) * tasa_red
tiempo = time() - inicio

return tasa_red, función_objetivo, tasa_clas, tiempo

```

-Este algoritmo consiste en la búsqueda de un vector de pesos que sea superior a todos los calculados anteriormente hasta llegar al límite de superar un número de vecinos determinado o un número máximo de iteraciones, en cada iteración modificamos una componente del vector de pesos sumándole un valor normal entre [0.0,0.3] y comprobamos su clasificación, si esta es mejor que la anterior guardamos su valor y establecemos el contador de vecinos a 0 y volvemos a iterar sobre el nuevo conjunto de pesos, si este valor no supera al anterior retrocedemos al vector de pesos anterior y pasamos a la siguiente componente, hasta llegar a un número máximo de exploraciones de vecinos o iteraciones, cuando salgamos del bucle tendremos el vector de pesos que mejor resultados a obtenido, usamos dicho vector para la clasificación final del algoritmo.

4- Algoritmos de comparación

KNN

```

Incio = time()
Trainx, trainy, testx, testy = preparer_particiones(i,t,x,y)
Tasa_clas = calcula_porcentaje_acertados(trainx,trainy,testx,testy)
Función_objetivo = 0.5 * tasa_clas + (1-0.5) * 0.0

```

```
Tiempo = time() – inicio
```

```
Return tiempo, tasa_clas, función_objetivo
```

-Este clasificador consiste en dividir los datos en entrenamiento y prueba. Primero se calcula la tasa_clas mediante el clasificador KNN de scikit-learn y después resolvemos la función objetivo. La tasa_red es 0.0 ya que no se utiliza ningún truncamiento ni eliminación de características. Este clasificador tampoco consta de ningún tipo de vector pesos de características por lo tanto, su predicción puede resultar menos acertada que otros algoritmos que se emplean para la obtención de un vector de pesos. La Alpha para todos los algoritmos será de 0.5.

RELIEF

```
inicio = time()
```

```
/* vector de pesos inicializado a 0
```

```
w = np.zeros(len(x[0]))
```

```
cont_reducción = 0
```

```
trainx, trainy, tesx, testy = prepara_particiones()
```

```
Para cada i en trainx
```

```
/* Algoritmo que devuelve el vecino mas cercano de la misma
```

```
clase y el vecino mas cercano de diferente clase
```

```
amigo,enemigo = buscar_vecino_amigo_enemigo(trainx,trainy,i)
```

```
/* Actualizamos el vector de pesos
```

```
w = w + | trainx[i] – enemigo | - | trainx[i] – amigo |
```

```
/*obtenemos el peso de mayor valor
```

```
maximo_peso = max(w)
```

```
Para cada wi en W
```

```
si w[t] < 0.0
```

```
entonces w[t] = 0.0
```

```

    si no

        w[t] = w[t]/máximo_peso

        si w[t] < 0.2

            cont_reducción ++

        trainx *= w

        trainy *= w

    /* calculamos el porcentaje de aciertos que obtenemos mediante el
    clasificador KNN

    tasa_clas=calcula_porcentaje_acertados(trainx,trainy,testx,testy)
    tasa_red = 100 * cont_reducción / len(w)
    función_objetivo = 0.5 * tasa_clas + (1-0.5) * tasa_red

    return tasa_red, función_objetivo, tasa_clas, tiempo

```

-Este algoritmo consiste en buscar el vecino mas cercano de la misma y diferente clase de cada conjunto de características de los datos trainx, es decir, calcular la distancia “euclídea” a cada vecino y elegir el de menor valor. Además actualizamos w en cada iteración mediante la fórmula expresada anteriormente. También recalcar que en la función buscar_vecino_amigo_enemigo () se tiene en cuenta el leave one out. Este consiste en no considerar como vecino a el mismo.

```

/* Método que obtiene amigo y enemigo

buscar_vecino_amigo_enemigo(trainx,trainy,d)

    id_amigo = id_enemigo = 0
    val_amigo = val_enemigo = 99999.0

    /* Procedimiento leave one out

    q = np.delete(trainx, d, 0)
    sumatoria = sum(trainx[d])

    Para cada qj en q

```

Distancia = (sumatoria – sum(qi))^2

Si trainy[qi] == trainy[d] and distancia < val_amigo

Id_amigo = qi

Val_amigo = distancia

Si trainy[qi] != trainy[d] and distancia < val_enemigo

Id_enemigo = qi

Val_enemigo = distancia

Return trainx[id_amigo] , trainx[id_enemigo]

5- Procedimiento considerado para el desarrollo de la práctica

Esta práctica esta realizada en Python. Este lenguaje proporciona módulos para la normalización, partición, lectura y clasificación de los datos. Por otro lado para la implementación del clasificador KNN, como he comentado en los apartados anteriores, he empleado el clasificador del módulo scikit-learn. Los distintos módulos utilizados para esta práctica son:

- scipy para la lectura de los ficheros .arff.
- sklearn.neighbors para el clasificador KNN.
- sklearn.model_selection para las particiones.
- sklearn.preprocessing para la normalización de los datos.
- time para el calculo de los tiempos.
- numpy para el control de los arrays.

6- Experimentos y análisis

a- Casos del problema:

Se han utilizado 3 conjuntos de datos para la ejecución de los distintos algoritmo, estos datos son:

1. Colposcopy (colposcopia): procedimiento ginecológico basado en la exploración del cuello del útero. Este dataset consta de 287 ejemplos, los cuales están compuestos por 62 características y existen 2 clases.
2. Ionosphere: datos de radar, los cuales se basan en la medición de electrones libre en la ionosfera. Este dataset esta compuesto por 352 ejemplos, cada ejemplo consta de 34 características y como en el dataset anterior existen dos clases.
3. Texture: Este datasets recoge características de los diferentes tipos de texturas, como por ejemplo: papel, césped, piel etc. Este conjuntos dispone de 550 ejemplos compuestos por 40 características cada uno y este a diferencia de los demás dataset consta de 11 clases.

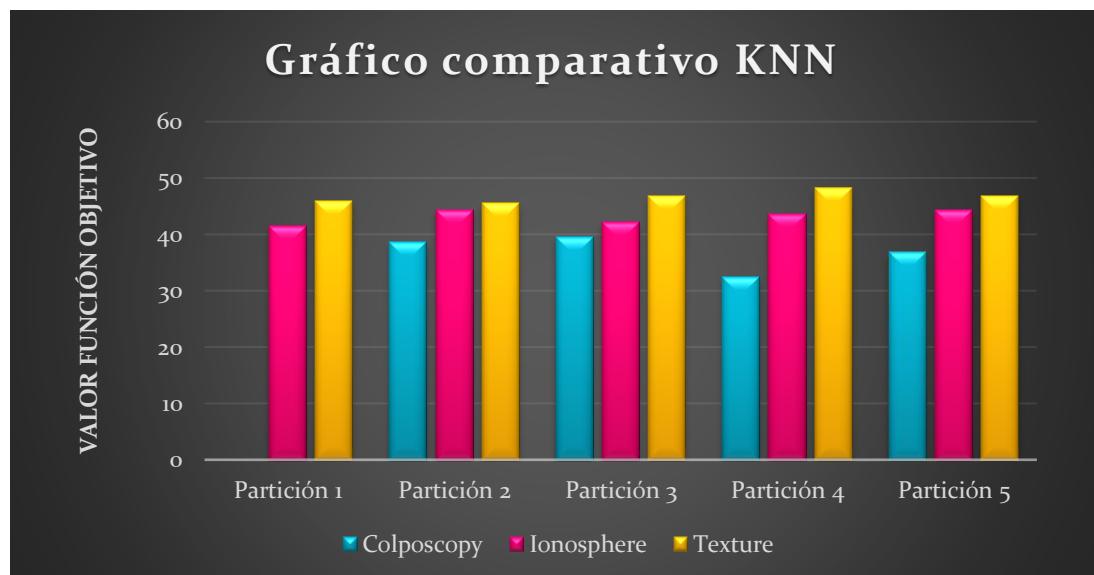
b- Parámetros utilizados en la práctica:

Para la práctica he empleado un valor de Alpha para el cálculo de la función objetivo de 0.5 tal y como indica el guion de prácticas. Además el número de vecinos utilizado en el clasificador es 1. En el algoritmo de búsqueda, se ha tomado como condición de parada tanto un número máximo de iteraciones (15000) como de exploraciones de vecinos ($20 * \text{len}(w)$). La semilla escogida es 2, se podría utilizar cualquier semilla mientras que en todos los algoritmos se utilice la misma. La semilla la fijo al principio del “main” pero al utilizar la opción `shuffle=True` en la generación de la particiones (`StratifiedKFold`) se debe pasar como parámetro el número de la semilla (`random_state = 2`).

c- Resultados obtenidos:

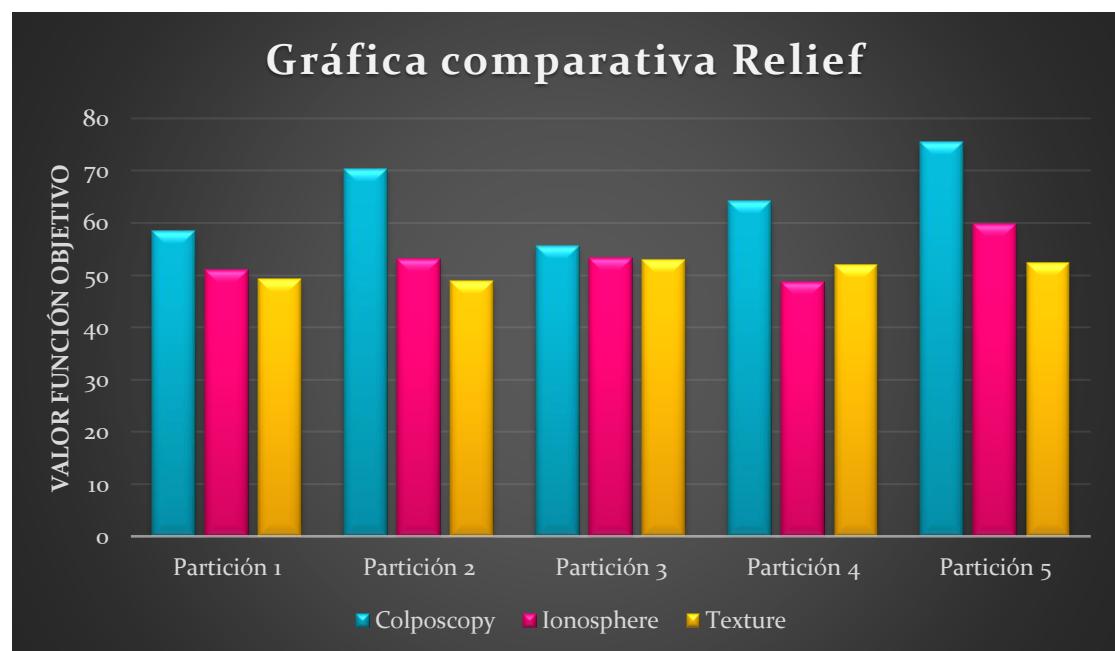
Resultados obtenidos por el algoritmo KNN en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T
Partición 1	71,18	0,0	35,6	0,63	83,09	0,00	41,55	0,04	91,82	0,00	45,91	0,03
Partición 2	77,19	0,0	38,60	0,10	88,57	0,00	44,29	0,03	90,91	0,00	45,45	0,05
Partición 3	78,94	0,00	39,47	0,07	84,29	0,00	42,15	0,03	93,64	0,00	46,82	0,05
Partición 4	64,91	0,00	32,46	0,03	87,14	0,00	43,57	0,03	96,36	0,00	48,18	0,06
Partición 5	73,68	0,00	36,84	0,03	88,57	0,00	44,29	0,03	93,64	0,00	46,82	0,05
Media	73,18	0,0	36,84	0,06	86,33	0,00	43,17	0,03	93,27	0,00	46,64	0,05



Resultados obtenidos por el algoritmo Relief en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T
Partición 1	76,27	40,32	58,30	1,99	87,32	14,70	51,01	0,83	93,64	5,00	49,32	1,97
Partición 2	82,46	58,06	70,26	2,45	91,43	14,71	53,07	0,80	92,73	5,00	48,86	1,98
Partición 3	75,44	35,48	55,46	2,09	85,71	20,59	53,15	0,85	93,36	12,50	52,93	1,95
Partición 4	68,42	59,68	64,05	0,90	91,43	5,88	48,65	0,83	96,36	7,50	51,93	2,16
Partición 5	68,42	82,28	75,35	0,92	90,00	29,41	59,71	0,83	94,55	10,00	52,27	2,15
Media	74,20	55,16	64,68	1,67	89,18	17,06	53,12	0,83	94,13	8,00	51,06	2,04



Resultados obtenidos por el algoritmo BL en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T
Partición 1	83,05	29,03	56,04	191,31	90,14	47,06	68,60	50,23	89,09	40,00	64,55	69,46
Partición 2	75,44	33,87	54,66	276,14	91,43	14,70	53,06	65,94	92,73	35,00	63,86	74,94
Partición 3	77,20	37,10	57,15	90,56	87,14	23,53	55,33	33,08	93,36	32,50	62,93	72,13
Partición 4	75,44	32,26	53,85	60,28	94,29	32,35	63,32	49,93	97,27	37,50	67,39	77,40
Partición 5	82,46	20,96	51,71	82,59	91,43	38,24	64,83	65,94	95,45	32,50	63,98	91,12
Media	78,72	30,64	54,68	140,18	90,89	31,17	61,03	53,02	93,58	35,50	64,54	77,01

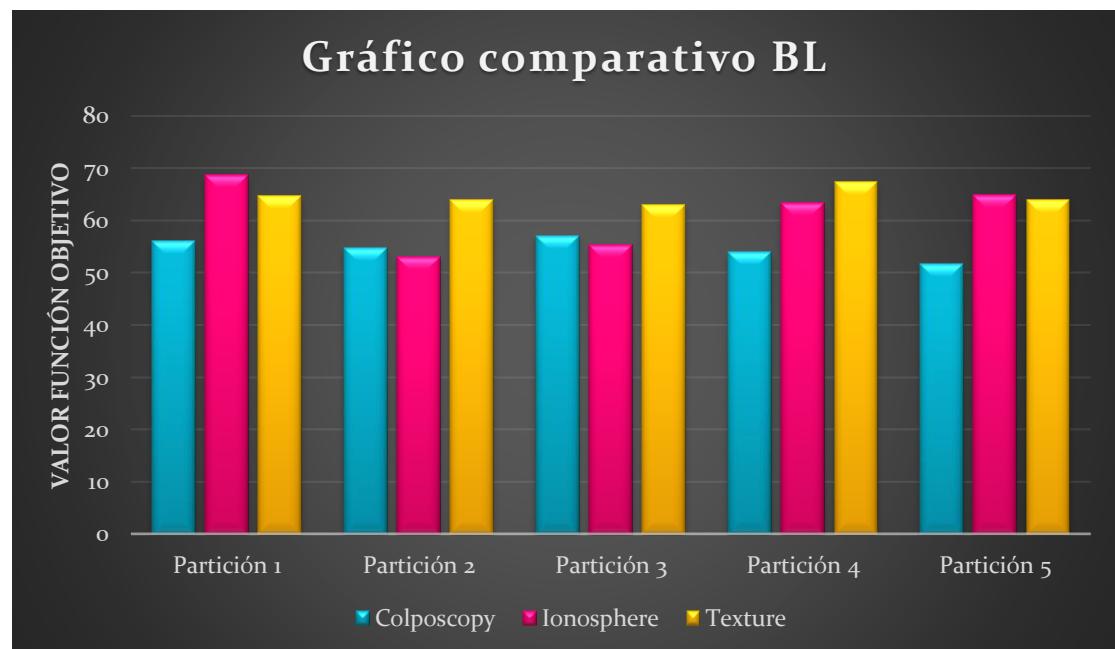


Tabla comparativa

	Colposcopy				Ionosphere				Texture			
	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T	%_clas	%red	Agr.	T
1-NN	73,18	0,00	36,84	0,06	86,33	0,00	43,17	0,03	93,27	0,00	46,64	0,05
RELIEF	74,20	55,16	64,68	1,67	89,18	17,06	53,12	0,83	94,13	8,00	51,06	2,04
BL	78,72	30,64	54,68	140,18	90,89	31,17	61,03	53,02	93,58	35,50	64,54	77,01

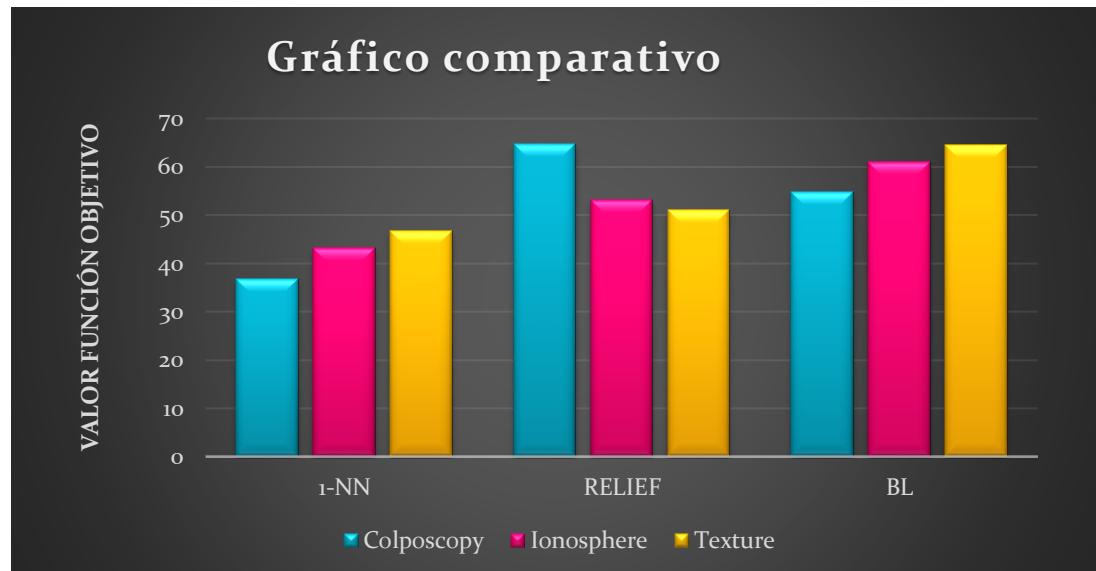
d- Análisis de los resultados:

En el algoritmo KNN observamos que el mejor resultado es obtenido en el dataset textura, ya que este clasificador no tiene en cuenta ningún valor de pesos para las características. Su resultado se ve afectado por la cantidad de datos que el conjunto presenta, porque podrá estudiar mejor las variaciones entre clases. Por lo tanto, en el dataset textura al ser el que más ejemplos y clases contiene, obtiene mejores resultados. En este clasificador la tasa de reducción es 0.0 ya que no desprecia ningún valor de las características. En valores de tiempo es el que mejor resultados obtiene de todos los algoritmos lo cual es obvio ya que clasifica los datos sin ningún tipo de cálculo de ponderación.

Respecto al algoritmo Relief podemos observar una mejora tanto en la tasa_clas como en la función objetivo, debido a que este algoritmo emplea un vector de pesos el cual se obtiene a partir del cálculo de vecinos amigos y enemigos. En este algoritmo aparece la tasa_red ya que los valores menores de 0.0 los trunca a 0.0 y los valores mayores que 1.0 los trunca a 1.0. Esto se ve reflejado en la función objetivo, viendo que es mucho mayor que la obtenido en el KNN sin pesos.

Para terminar, el algoritmo de búsqueda local incrementa la función objetivo respecto a los anteriores descritos, es decir, este algoritmo busca la solución mas “buena” (que no tiene por que ser la óptima) de entre todas las exploradas, por lo tanto también se ve reflejado en la tasa de aciertos (tasa_clas) y en la tasa de reducción (tasa_red). Esto es debido a que esta explora una cantidad definida de vecinos o iteraciones en

busca de una solución “óptima”. Esto se verá reflejado en el tiempo de ejecución del algoritmo, ya que es con diferencia el que más tiempo requiere de ejecución.



7- Bibliografía

- Información sobre las particiones: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html
- Información sobre el clasificador scikit-learn: <https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html>
- Información sobre la lectura de ficheros arff:
<https://discuss.analyticsvidhya.com/t/loading-arff-type-files-in-python/27419>
- Información sobre generación de gráficos