UNIVERSIDAD DE GRANADA

PRÁCTICA 1.b: Técnicas de Búsqueda Local y Algoritmos Greedy para el Problema del Aprendizaje de Pesos en Características (APC)

Metaheurísticas || Curso 2018-2019

Alumno: José María Sánchez Guerrero
DNI: 76067801Q
Correo: jose26398@correo.ugr.es
Grupo: A3 – Jueves 17:30

Contenidos

Descripción del problema

He elegido el problema del **Aprendizaje de Pesos en Características (APC)**, en el cual tendremos que realizar un clasificador de características partiendo de unos datos iniciales sin procesar. Este clasificador va a ser del tipo **k-NN** (donde k=1), en el que la clase que asociaremos a cada dato, será la clase del vecino más cercano.

El **vecino más cercano** se obtiene a partir de la distancia euclídea entre el propio dato y el vecino. Por tanto, la distancia entre dos ejemplos e_1 y e_2, utilizando por ejemplo d_e para las características y w para los pesos (que ponderan su importancia dentro del contexto), se calcularía de la siguiente forma:

$$d_e(e_1, e_2) = \sqrt{\sum_i w_i \cdot (e_1^i - e_2^i)^2 + \sum_j w_j \cdot d_h(e_1^j - e_2^j)}$$

Los datos iniciales utilizados en el problema tendrán que ser divididos en datos de entrenamiento y datos para testear. Esta división se realizará de tal forma que 80% sea de entrenamiento y el 20% para test, manteniendo la distribución de clases.

Al final de cada algoritmo vamos a devolver cuatro valores que nos van a servir para evaluar tanto la precisión como la complejidad del clasificador. Estos valores son:

- **Tasa_clas** (precisión). Se calculará dividiendo las instancias bien clasificadas en T entre el número de instancias totales de T.
- **Tasa_red** (reducción). Se calculará dividiendo el número de valores de w menores que 0.2 entre el número total de características.
- **Agr** (agregado). Valor de la función objetivo y que calcularemos sumando los dos valores anteriores multiplicados por α y 1-α respectivamente. En nuestro caso, éste va a ser de 0.5.
- **T** (tiempo). Tiempo que tardará en ejecutarse el algoritmo.

Los algoritmos que vamos a implementar van a ser el algoritmo de comparación *greedy* RELIEF y el algoritmo de Búsqueda Local. Realizaremos 5 ejecuciones para cada unos de ellos utilizando los tres ficheros de datos proporcionados y mostraremos los 4 estadísticos anteriores en tablas para poder comparar el rendimiento de los algoritmos.

Descripción de la aplicación de los algoritmos

Lectura de datos

Esta parte en muy simple. Sólo son 3 líneas que servirán para extraer los datos de los archivos (guardados en la carpeta doc/).

```
    df = read_csv('doc/colposcopy.csv')
    data = df.extraer_valores
    datos = quitarPrimeraColumna
```

Quitamos la primera columna porque en los archivos tenemos guardado ahí el número de los datos (o la id de los datos), la cuál no vamos a necesitar.

División de los datos

Gracias a esto vamos a poder dividir los datos en 5 partes iguales conservando la proporción de 80% entrenamiento y 20% test. La división está realizada principalmente por la función ***StratifiedKFold*** perteneciente a la librería de python *scikit learn*.

Ésta proporciona automáticamente índices de entrenamiento y test para después poder dividir los datos. Vamos a verlo mejor con el pseudocódigo:

```
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=73)

x = extraerAtributos(datos)
y = extraerClases(datos)

for indiceTrain, indiceTest in skf.sacarIndicesDe(x,y):
   x_train, y_train = x[indiceTrain], y[indiceTrain]
   x_test, y_test   = x[indiceTest], y[indiceTest]

   train.añadirFinal(x_train, y_train)
   test.añadirFinal(x_test, y_test)
```

La función ***StratifiedKFold*** la he dejado tal cual es, para que se vea cómo selecciono el número de divisiones (n_split=5), shuffle es para que salga aleatorio y random_state es la semilla, por si desea cambiarla a otra.

Despúes, dividimos los datos anteriores en atributos y las clases; y gracias a los índices que nos proporciona la función anterior, añadimos a los vectores ***train*** y ***test*** un dato aleatorio al final de ellos.

Función objetivo (función evaluación)

La función objetivo la he implementado de forma muy similar a la función de evaluar, por lo que la explicaré posteriormente, en la sección del *Algoritmo de comparación* (o el apartado f) en el guión de prácticas).

Esquema de representación de los datos

El esquema de representación de soluciones se obtendrá gracias a la función evaluar(que veremos posteriormente). Al final de cada una de las ejecuciones, esta función guardará en una matriz los resultados obtenidos, correspondiendo cada fila de la matriz a una de ellas. Las columnas de la matriz estaran formadas por los cuatro valores estadísticos mencionados al principio.

A continuación, muestro una tabla en representación de la matriz para verlo mejor visualmente:

Matrix	tasa_clas	tasa_red	Agr.	T
Ejecucion 1				
Ejecucion 2				
Ejecucion 3				
Ejecucion 4				
Ejecucion 5				

Las matrices (o tablas) se guardarán al final en una variable del tipo DataFrame, para poder imprimirlas por pantalla más fácilmente y también para poder sacarlas a un .csv externo y tener un acceso más sencillo a los resultados.

Descripción de los algoritmos

Algoritmo de Búsqueda local

Vamos a ver primero la implementación en pseudocódigo del algoritmo y posteriormente lo explicaremos paso a paso:

```
for ejecucion in range(5):
    tiempoAntes = guardarTiempo()
    w = generarAleatorios(tamañoAtributos)
    ganancia = obtenerGanancia(w, testAtributos, testClases)
    evaluacionesRealizadas = atributosExplorados = 0

    while (evaluacionesRealizadas < 15000) and (atributosExplorados< 20*n):
        wActual = copiar(w)

        for posNuevo in generarAleatorio(tamañoAtributos):
            w[posNuevo] += generarAleatorioNormal(media=0, varianza=0.3²)
            acotarEntre(0, 1, w[posNuevo])
            evaluacionesRealizadas += 1
            gananciaNueva = obtenerGanancia(w, trainAtributos, trainClases)

            if gananciaNueva > ganancia:
                ganancia = gananciaNueva
                atributosExplorados = 0
                break
            else:
                atributosExplorados += 1
                w[posNuevo] = wActual[posNuevo]

            if evaluacionesRealizazdas > 15000 or atributosExplorados > 20*n:
                break

    resultadosBL.guardarResultadoEjecución(evaluar(w, testAtr, testCla, actual))
```

El primer bucle realiza las 5 ejecuciones, asi que realmente no pertenece al algoritmo. Tras este, vemos como activamos el tiempo, generamos la solución inicial *w* aleatoriomente y calculamos su ganancia; e inicializamos a 0 las variables que utilizaremos.

Tras esto ya si nos metemos en el algoritmo. Comenzamos con un *while* que pare de ejecutarse cuando, según indica el guión, realize **15000 evaluaciones** o cuando explore **20*n atributos** (siendo n el número de atributos que tiene cada dato).

Dentro del bucle, lo primero que hacemos es copiar a una variable auxiliar nuestra *w* para modificarla y no perder los datos si esta no mejora la ganancia. Después, generaremos un **vector aleatorio de posiciones** (obviamente con el tamaño de los atributos) y lo recorreremos de la siguiente forma. Generamos un vecino en la propia posición a través de una **distribución normal** de media 0 y varianza 0.3^2, y se lo sumaremos al vector *w*. Una vez hecho esto, calculamos la ganancia que tiene esta *w* modificada.

Si esta **ganancia** nueva **es mejor** que la ganancia calculada al principio, la guardaremos como nueva mejor ganancia y romperemos el bucle, ya que hemos encontrado una *w* mejor.

Al romper el bucle, también tendremos que volver a poner el número de atributos explorados a 0 (volvemos al principio del while con nuevo array aleatorio de atributos).

En caso de que esta nueva **ganancia no sea mejor**, simplemente aumentaremos el número de atributos explorados en 1 y copiaremos el valor del *wActual* guardado anteriormente en el *w* que estamos modificando, ya que es peor.

Después de esta comprobación, también realizaré otra para comprobar lo mismo que se comprueba en el while, es decir, que realice 15000 evaluaciones o que haya explorado 20*n atributos. Esto es para que si ha cumplido alguna de estas condiciones, corte el bucle for y no lo haga completamente.

Por último, guardamos los resultados en una variable utilizando la función **evaluar**, que veremos en el siguiente apartado. Aún así, puedo decir que esta función devolverá una tasa_clas, una tasa_red, un agregado y el tiempo, en función de los parámetros que le pasamos.

Algoritmo *greedy* RELIEF

Tal y como hemos hecho antes, vamos a ver primero la implementación en pseudocódigo del algoritmo y posteriormente lo explicaremos paso a paso:

```
for ejecucion in range(5):
    antes = guardarTiempo()

    for atributoActual in range(numeroDatos):
        amigos = atributosMismaClase(datoActual)
        enemigos = atributosDistintaClase(datoActual)

        tree_am = KDTree(amigos)
        indiceAmigo = tree_am.amigoMasCercano(atributos[datoActual])

        tree_en = KDTree(enemigos)
        indiceEnemigo = tree_en.enemigoMasCercano(atributos[datoActual])

        amigo = amigos[indiceAmigo]
        enemigo = enemigos[indiceEnemigo]

        w + distancia(atributos[datoActual], enemigo)
        w – distancia(atributos[datoActual], amigo)

    w = w / maximo(w)
    acotarEntre(0, 1, w)

    resultadosRELIEF.guardarResultadosEjecucion(evaluar(w, testAtributos,
                                                testClases, antes))
```

El primer bucle, al igual que antes, realiza las 5 ejecuciones, asi que realmente no pertenece al algoritmo. Tras este, a diferencia de antes, sólo activamos el tiempo. La solución inicial *w* la hemos calculado antes, generando 5 arrays (uno para cada ejecución) y así no tener que generarlo cada vez.

Ahora ya nos meteremos dentro del algoritmo, en el cual recorreremos todos los datos uno a uno. Lo primero que hacemos es guardar un vector con todos los **amigos** y otro con todos los **enemigos**, siendo los amigos los datos que tienen la misma clase que el que estamos explorando, y los enemigos los datos con distinta clase.

Una vez tenemos clasificados a los amigos y los enemigos generamos un **KDTree** para cada uno. Estos árboles te permiten buscar rápidamente el índice del vecino más cercano al vector de atributos que le pasaremos como parámetro.

Ya obtenidos tanto el índice del amigo y el índice del enemigo más cercanos, sumaremos a *w* la distancia entre el dato actual y el enemigo, y luego le restaremos distancia entre el dato actual y el amigo.

Cuando hayamos terminado con todos los atributos, **normalizamos** dividiendo entre el valor máximo de *w,* y acotamos entre 0 y 1 para que no haya valores negativos.

Por último, al igual que antes, guardaremos los resultados en una variable utilizando la función **evaluar**, que vermos en el siguiente apartado.

Algoritmo de comparación

Esta función la he "dividido" en dos funciones:

- **evaluar(w, atributos, clases, tiempo)**. Esta función es la que utilizo para devolver los 4 valores estadísticos a partir de los parámetros anteriores.
- **obtenerGanancia(w, atributos, clases).** Esta función es la que utilizo en la búsqueda local para obtener sólo las nuevas ganancias (sin los otros valores estadísticos) y poder compararlas después.

Aclarado esto, vamos a ver la implementación en pseudocódigo de la función:

```
def evaluar(w, atributos, clases, antes):
    wNormalizados = quitarCaracterísticas(0.2, w)
    prod = (atributos * wNormalizados)

    arbol = cKDTree(prod)
    vecinos = arbol.obtenerVecinos(prod, k=2)

    _clas = clases[vecinos] / totalClases           # Hacer la medias
    _red = wMenoresQue(0.2) / totalCaracteristicas
    despues = obtenerTiempo()

    return _clas*100, _red*100, (_clas*0.5 + _red*0.5)*100, despues – antes
```

Esta función está hecha **vectorizando**, así nos evitamos poner bucles. Primero selecciona los datos que sean menores que 0.2 (como indica en el guión) y después multiplicamos los atributos por sus pesos y mandamos el producto a un **KDTree**. Conseguimos una mejora de rendimiento porque al enviar todos los puntos al árbol directamente, podemos obtener los vecinos sin tener que ir calculando las distancias uno a uno en un bucle.

Después obtenemos los dos vecinos más cercanos (k=2) y te quedas con el segundo, porque el primero es el propio nodo evaluado.

Por último, obtenemos los valores estadísticos como explicamos en la "*Descripción del Problema"*. También volveremos a obtener el tiempo actual, ya que en este momento se acabrá de ejecutar el algoritmo. Cuando los devolvemos, los valores están multiplicados por 100 para obtener el porcentaje (menos el timepo que simplemente hacemos una resta entre el inicial y el final).

La función de *obtenerGanancia()* es exactamente igual, exceptuando que no tendremos que pasarle el tiempo como parámetro y los valores que devolveremos no tienen que ser los 4 estadísticos, sino sólo el **agregado**. Éste valor es el único que nos va a hacer falta al comparar ganancias en el algoritmo de Búsqueda Local.

Desarrollo de la práctica y manual de usuario

La práctica está realizada en su totalidad en **python**, sin utilizar los códigos proporcionados en la web, otros códigos u otros frameworks externos.

La aplicación está desarrollada por completo en un script de python(.py), y también se incluye junto a este una carpeta *doc,* en la cual podremos encontrar los .**csv** utilizados para obtener los datos. Estos documentos no están exactamente igual que los .arff proporcionados originalmente, es decir, ya están parseados para no tener que complicar el código innecesariamente (al decir que no están exactamente igual, me refiero a la forma de mostrar los datos; obviamente los valores de estos siguen siendo exactamente los mismos).

En este script se hacen uso de las siguientes **librerías**, las cuales serán necesarias tener descargadas para poder ejecutar el script:

```
- import pandas as pd
- import numpy as np
- import time
- from scipy.spatial import KDTree
- from scipy.spatial import cKDTree
- from sklearn.model_selection import StratifiedKFold
```

En la *línea 14* tendremos el **documento** que analizaremos y que podremos cambiar dependiendo de cuál queramos. Las opciones son: *'doc/colposcopy.csv', 'doc/ionosphere.csv'* y *'doc/texture.csv'*.

La **semilla** está declarada tanto en la *línea 8* como en la *línea 23*. Las pruebas que yo he realizado utilizan la semilla 73 para generar los aleatorios.

Las cuatro últimas líneas del código sirven para **visualizar** los datos obtenidos. Por defecto, yo lo dejo para que muestre por consola los resultados de la búsqueda local. Opcionalmente, si quitamos el comentario a las siguientes, podremos mostrar el resultado del algoritmo RELIEF y también podremos exportar el resultado a un .csv externo con el nombre *'result.csv'*.

Análisis de los resultados

Se han ejecutado los 3 conjuntos de datos proporcionados:

- **Colposcopy.** Conjunto de datos de colposcopias que dipone de 287 ejemplos con 62 características y deben ser clasificados en 2 clases.
- **Ionosphere.** Conjunto de datos de radar que dispone de 352 ejemplos con 34 características y deben ser clasificados en 2 clases.
- **Texture.** Conjunto de datos de imágenes y hay que diferenciar a que pertenece cada una. Dispone de 550 ejemplos con 40 características y deben ser clasificados en 11 clases.

Otros parámetros a tener en cuenta pueden ser: la semilla, para la cual yo he uitlizado el valor 73; el número máximo de evaluaciones y el número máximo de vecinos, que valdrá 15000 y 20*n respectivamente; y la media y la varianza para generar números aleatorios a partir de una distribución normal (0 y 0.3 respectivamente).

Para cada experimento se han realizado 5 particiones y, a continuación, vamos a mostrar las tablas de los resultados obtenidos:

Tabla 5.1: 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	72,88	43,55	58,21	6,10	78,87	50,00	64,44	3,96	81,82	45,00	63,41	4,12
Partición 2	73,68	19,35	46,52	1,68	85,71	50,00	67,86	4,01	80,91	50,00	65,45	2,93
Partición 3	77,19	24,19	50,69	1,53	81,43	44,12	62,77	2,73	82,73	30,00	56,36	5,01
Partición 4	66,67	32,26	49,46	5,30	84,29	32,35	58,32	1,61	87,27	52,50	69,89	3,79
Partición 5	73,68	40,32	57,00	6,35	81,43	47,06	64,24	1,82	79,09	47,50	63,30	5,08
Media	**72,82**	**31,94**	**52,38**	**4,19**	**82,35**	**44,71**	**63,53**	**2,83**	**82,36**	**45,00**	**63,68**	**4,18**

Tabla 5.2: 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	64,41	64,52	64,46	0,60	81,69	5,88	43,79	1,58	90,00	22,50	56,25	3,45
Partición 2	64,91	66,13	65,52	0,59	82,86	2,94	42,90	1,21	89,09	22,50	55,80	3,83
Partición 3	77,19	62,90	70,05	0,55	81,43	5,88	43,66	1,08	83,64	12,50	48,07	3,47
Partición 4	63,16	50,00	56,58	0,56	82,86	5,88	44,37	1,12	90,00	7,50	48,75	3,66
Partición 5	70,18	50,00	60,09	0,53	78,57	5,88	42,23	1,09	81,82	10,00	45,91	3,28
Media	**67,97**	**58,71**	**63,34**	**0,57**	**81,48**	**5,29**	**43,39**	**1,22**	**86,91**	**15,00**	**50,95**	**3,54**

Como podemos observar, la tasa de acierto es bastante similar en los dos algoritmos. Si obtenemos la media entre las tres pruebas realizadas obtenemos:

	Media total			
	%_clas	*%red*	*Agr,*	T
BL	79,18	40,55	59,86	3,73
RELIEF	78,79	26,33	52,56	1,77

Lo que más destaca es la tasa de reducción (y como consecuencia el agregado), que en la Búsqueda Local es bastante más grande que en el RELIEF, lo que quiere decir que descarta más datos de peso menor que 0.2.

Pese a esto, lo verdaderamente importante es la tasa media de aciertos, que en la Búsqueda Local es ligeramente superior que en el RELIEF. Al estar trabajando con aleatorios, estos valores pueden variar, sin embargo, vemos que la tasa de acierto está alrededor de un 80% lo cual es un porcentaje alto de acierto.

Por otra parte, la otra variable más importante a tener en cuenta es el tiempo. Como podemos observar, en el algoritmo RELIEF apenas supone una carga media de 1,77 segundos para realizar las 5 ejecuciones. Esto es porque utiliza operaciones de cálculo sencillas y que apenas tienen cómputo.

En cambio, el algoritmo de Búsqueda Local tiene una media de 3,73 segundos (llegando incluso a los 6 segundos en algunas ejecuciones). Esto se debe a que utiliza operaciones un poco más costosas, como puede ser el máximo de 15000 evaluaciones o los 20*n atributos explorados.

Bibliografía

- Seminaro 2. Problemas de optimización con técnicas basadas en búsqueda local.
- Tema 2. Modelos de Búsqueda: Entornos y Trayectorias vs Poblaciones.
- https://en.wikipedia.org/wiki/Feature_learning
- https://machinelearningmastery.com/feature-selection-machine-learning-python
- https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.cKDTree.html
- https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.KDTree.html#scipy.spatial.KDTree
- https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.KDTree.query.html#scipy.spatial.KDTree.query
- https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html
- https://en.wikipedia.org/wiki/K-means_clustering
- https://docs.scipy.org/doc/numpy/reference/routines.random.html
- https://docs.python.org/2/library/time.html
- https://docs.scipy.org/doc/numpy/reference/generated/numpy.clip.html