Memoria práctica 1 de Metaheuristicas:
Problema del Aprendizaje de Pesos en Características (APC)

UNIVERSIDAD DE GRANADA

Antonio Manuel Fresneda Rodriguez
antoniomfr@correo.ugr.es

3-Grupo A3(Miercoles 17:30-19:30)
Los algoritmos seleccionados han sido el de Busqueda Local y Greedy

Índice

Descripción del problema.

El problema del APC consiste en optimizar el rendimiento de un clasificador basado en vecinos mas cercanos a partir de añadir pesos asociados a cada caracteristicas del problema que modifican su valor en el momento de calcular las distancias entre los ejemplos.

Vamos a usar el 1-NN (consideramos un solo vecino). La variante del problema del APC que confrontaremos busca optimizar tanto la precision como la complejidad del clasificador. Asii, se puede formular como:

Maximizar F(w)=α*tasa_clas(W)+(1-α)*tasa_red(W)

con:

- $tasa-clasificacion = \frac{100 \cdot n^o\ instancias\ bien\ clasificadas\ en\ T}{n^o\ instancias\ en\ T}$
- $tasa-reduccion = \frac{100 \cdot n^o\ valores\ w_i < 0{,}2}{n^o caracteristicas}$

Sujeto a que

- wi = [0, 1], 1 ≤ i ≤ n

donde:

- W = (w1, ..., wn) es una solución al problema que consiste en un vector de números reales w1 [0,1] de tamaño n que define el peso que pondera o filtra a cada una de las características f_i.
- 1-NN es el clasificador k-NN con k=1 vecino generado a partir del conjunto de datos inicial utilizando los pesos en W que se asocian a las n características.
- T es el conjunto de datos sobre el que se evalúa el clasificador, ya sea el conjunto de entrenamiento (usando la técnica de validación leave-one-out) o el de prueba.
- α∈[0,1] pondera la importancia entre el acierto y la reducción de la solución encontrada.

Descripción de la aplicación.

Las soluciones se van a devolver en tres listas y un vector:

1. La primera contendrá la tasa de clasificación para cada partición que hemos hecho de los datos.
2. La segunda contendrá la tasa de reducción para cada partición que hemos hecho de los datos.
3. La tercera contendrá el tiempo empleado en cada partición.
4. El vector de pesos obtenido.

La función objetivo a maximizar va a ser:

- F(w)=α*tasa_clas(W)+(1-α)*tasa_red(W)

Para el clasificador, he usado el clasificador KNN implementado en la librería scikit-learn

- KNN=KNeighborsClasiffier(vecinos=1,metrica=dist,argumentos=w)
- KNN.fit(X_train,Y_train)

Si queremos usar el Greedy, la llamada será asi:

Greedy(X_train,Y_train)

Si queremos usar la BL

BL(X_train,Y_train,sigma,alfa)

Si queremos usar el 1NN

1NN(X_train,Y_train)

Operadores comunes a ambos algoritmos podríamos considerar la distancia:

- Dist(X,T,Pesos):

  para cada x_i,y_i,peso_i en X,T,Pesos:

  vector_distancias= (xi-yi)*(xi-yi)*peso_i

  return suma_elemento_a_elemento(vector_distancias)

Las particiones de los datos para 5-fold cross validation he usado una función de scikit-learn llamada StratifiedKFold. A esa función tiene como parámetro el numero de particiones (en nuestro caso 5), el conjunto de X y de Y, y devuelve dos listas por cada partición. En la primera están los indices de X y en la segunda los de Y

- folds=StratifiedKFold(particiones=5)

  folds.datos(X,Y)

  Para cada indice_train,indice_test en folds

  Particion_i_X_train=X[indice_train]

  Particion_i_Y_train=Y[indice_train]

  Particion_i_X_test=X[indice_test]

  Particion_i_Y_test=Y[indice_test]

  Llamada_algoritmo_KNN(Particion_i_X_train,……)

- Otro operador común que tenemos es la normalizacion de los datos. Para ello hacemos lo siguiente:
  - max=BuscarMax(X)
  - min=BuscarMin(X)

    Para cada xi en X
    - Xi=(xi-min)/(max-min)

Descripción del pseudo-codigo

BL(X,Y,sigma,alfa):

```
tiempo1=time()
indices=Genera_lista_desordenada(desde=0,hasta=n_características-1,tamaño=n_características)
op=Genera_lista_uniforme(media=0,desvicacion=sigma^2,tamaño=n_características)
w=Genera_array_distribución_normal(low=0,high=1,tamaño=n_características)
n_vecinos=0
red=w.contar_ceros()
KNN=ClasificadorKNN(n_vec=1,metrica=Dist,args=w,X,Y)
Y_vecinos=KNN.vecinos()
bien_clas=Numero_de_posiciones_iguales(Y_vecinos,Y)
tasa_clas=bien_clas/tamaño_muestra
tasa_red=red/n_características
puntuación_padre=100*alfa*tasa_clas+100*alfa*red
for i to 15000
    indice=indices.pop()
    w_ant=w
    w[indice]=w[indice]-op[indice]
    n_vecinos++
    red=contar_ceros(w)
    Y_vecinos=KNN.vecinos_w_mutado()
    bien_clas=Numero_de_posiciones_iguales(Y_vecinos,Y)
    tasa_clas_h=bien_clas/tamaño_muestra
    tasa_red_h=red/n_características
    puntuación_hijo=100*alfa*tasa_clas_h+100*alfa*red_h
    if puntuación_hijo>puntuación_padre
        puntuación_padre=puntuación_hijo
    else
        w=w_ant
```

```
            if indices.vacia()
                op=Genera_lista_uniforme(0,sigma^2,tamaño=n_características)
                indices=Genera_lista_desordenada(0,n_características-1,tamaño=n_características)
            if n_vecinos==20*n_características
                break
    tiempo2=time()
    return ,tasa_clas,tasa_red,tiempo2-tiempo1,w
```

OP → Lista con las operaciones que se van a aplicar al vector (para no tener así que llamar al aleatorio dentro del for).

Índices → Lista con los índices. Estos están generados mezclados de forma pseudo-aleatoria.

Para el operador de mutación, lo que hago en cada iteración saco un indice de la lista (por el final para que sea más optimo) y ese indice lo uso para sacar el valor de W en ese indice y el valor de OP correspondiente.

Para el cálculo del error, he usado el método de leave one out con la muestra de entrenamiento.

Este KNN no tiene esa opción para calcular el error, pero para hacerlo lo que he echo ha sido calcular los vecinos mas cercanos a cada una de la muestra y las etiquetas de estos y compararlas con las etiquetas del train.

Descripción pseudo-codigo para los algoritmos de comparación

RELIEF(X,Y,w)

```
vecinos=vecinos_mas_cercanos(X,Y,metrica=euclidea)
para cada xi,yi en X,Y
    indice_vecinos_etiqueta1=vecinos.etiqueta(1)
    indice_vecinos_etiqueta2=vecinos.etiqueta(2)
    if yi==1
        amigo=indice_vecino_etiqueta1[0]
        enemigo=indice_vecino_etqueta2[0]
    else
        amigo=indice_vecino_etiqueta2[0]
        enemigo=indice_vecino_etqueta1[0]
    w=w+valor_absoluto(xi-X[enemigo])-valor_absoluto(xi-X[amigo])
w_max=max(w)
para cada wi en w
    if wi<0
        wi=0
    else
        wi=wi/w_max
```

Aquí, los vecinos mas cercanos se obtienen sin tener en cuenta los pesos. Como he usado el KNN de scikit-learn con la métrica euclidea.

Greedy(X,Y)

```
tiempo1=time()
w=vector_ceros(tamaño=n_características)
w=RELIEF(X,Y,w)
KNN=ClasificadorKNN(n_vec=1,metrica=Dist,args=w,X,Y)
Y_vecinos=knn.vecinos()
tasa_clas=(numero_componentes_iguales(Y_vecinos,Y))/tamaño_muestra
tasa_red=numero_de_ceros(w)/n_características
tiempo2=time()
return tasa_clas,tasa_red,tiempo2-tiempo1,w
```

Aquí si uso la métrica con los pesos para comprobar cómo de bueno es el w que hemos obtenido con el método RELIEF.

1NN(X,Y)

```
tiempo1=time()
KNN=ClasificadorKNN(N_vecinos=1,metrica=euclidea, X,Y)
Y_vecinos=KNN.vecinos()
tasa_clas=(suma_componentes_iguales(Y_vecinos,Y))/tamaño_muestra
tasa_red=0
tiempo2=time()
return tasa_cas,tasa_red,tiempo2-tiempo1,0
```

Procedimiento para realizar la práctica

Para realizar la practica, he usado el lenguaje de programación Python usando las siguientes librerías:

1. Numpy: Librería para agregar mejor soporte para vectores y matrices
2. Scikit-learn: Librera de machine learning para Python
    1. StratifiedKfold: Función para realizar K-fold cross validation (en nuestro caso K=5)
    2. KneighbrosClassifier: Clasificador KNN. Para usar métricas definidas por el programador, tenemos que añadir el argumento metric=" métrica " y usar el algoritmo "ball_tree", puesto que es el único que acepta métricas definidas por el programador.
    3. NearestNeighbors: Obtiene las distancias y los vecinos mas cercanos. Se diferencia del KneighborsClassifier en que no podemos predecir, solo obtener los vecinos mas cercanos a un punto. También puede aceptar métricas definidas por el usuario.
    4. Time: Librería para medir tiempos.

A la hora de realizar la practica he usado Anaconda (una distribución de Python para instalar mas fácilmente librerías de terceros) y como IDE he usado Spyder.

La práctica la he realizado usando el sistema operativo Ubuntu 16.04.

Para instalar Anaconda y Spyder (vienen en el mismo script), lo he descargado de su página web.

Una vez instalado, para instalar scikit-learn, en un terminal he ejecutado conda install scikit-learn.

Hecho esto, ya estaría toda la librería de scikit-learn instalada.

Finalmente, como he mencionado anteriormente, he usado el clasificador KNN que viene instalado y la función para la división de los datos.

Estas funciones están implementadas en el script Practica1.py.

Para el uso de las mismas, tenemos que hacer un import del mismo fichero al script que vamos a usar.

Para el generador de números aleatorios, he usado el proporcionado en la librería de numpy, fijando la semilla en el valor 1997.

En la carpeta de Fuentes están tanto el archivo Practica1.py como cada script para cada conjunto de datos usando Greedy,1NN o la búsqueda local.

Los ficheros de datos son los mismos que los de la plataforma.

Experimentos y análisis

Los casos que hemos usado para la practica han sido:

1. Ozone: Conjunto de datos para la detección del nivel de ozono. 73 características, 2 clases y 320 muestras
2. Parkinsons: Conjunto de datos orientado a distinguir entre la presencia y la ausencia de la enfermedad del Parkinson. 22 características, 2 clases y 195 muestras.
3. Spectf-heart: Conjunto de datos para la detección de enfermedades cardíacas. 44 características, 2 clases y 267 ejemplos.

Los parámetros que hemos usado han sido:

1. Alfa: Parámetro para la función objetivo que le da una mayor importancia a clasificar bien o a reducir al máximo. En nuestro caso, alfa es igual a 0.5 (misma importancia para ambos).
2. Sigma: Para metro para el operador de vecino. Este se generara mediante una distribución normal de media 0 y desviación sigma^2. En nuestro caso, sigma es igual a 0.3
3. La semilla para la generación de números aleatorios se ha fijado a 1997.

Estos parámetros se han considerado iguales para todas las ejecuciones.

Resultado de las ejecuciones.

BL	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	0,68	0,23	0,46	452,00	0,84	0,36	0,60	40,00	0,86	0,32	0,59	412,00
Partición 2	0,67	0,30	0,49	447,00	0,85	0,22	0,54	30,00	0,88	0,23	0,56	427,00
Partición 3	0,67	0,23	0,45	636,00	0,84	0,45	0,65	27,00	0,83	0,15	0,49	450,00
Partición 4	0,73	0,23	0,48	530,00	0,90	0,27	0,59	43,00	0,86	0,20	0,53	424,00
Partición 5	0,69	0,23	0,46	560,00	0,90	0,40	0,65	49,00	0,82	0,27	0,55	413,00
Media	0,69	0,24	0,47	525,00	0,87	0,34	0,60	37,80	0,85	0,23	0,54	425,20

GREEDY	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	0,49	0,00	0,25	0,63	0,75	0,00	0,38	0,15	0,72	0,00	0,36	0,79
Partición 2	0,49	0,00	0,25	0,62	0,75	0,00	0,38	0,20	0,72	0,00	0,36	0,75
Partición 3	0,49	0,00	0,25	0,61	0,75	0,00	0,38	0,15	0,73	0,00	0,36	0,74
Partición 4	0,49	0,00	0,25	0,62	0,75	0,00	0,38	0,16	0,73	0,00	0,36	0,77
Partición 5	0,50	0,00	0,25	0,63	0,75	0,00	0,38	0,15	0,73	0,00	0,36	0,75
Media	0,49	0,00	0,25	0,62	0,75	0,00	0,38	0,16	0,72	0,00	0,36	0,76

1NN	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	0,71	0,00	0,36	0,01	0,83	0,00	0,42	0,002	0,86	0,00	0,43	0,01
Partición 2	0,71	0,00	0,36	0,01	0,83	0,00	0,42	0,002	0,86	0,00	0,43	0,01
Partición 3	0,68	0,00	0,34	0,02	0,85	0,00	0,43	0,002	0,82	0,00	0,41	0,02
Partición 4	0,74	0,00	0,37	0,01	0,81	0,00	0,41	0,002	0,86	0,00	0,43	0,10
Partición 5	0,64	0,00	0,32	0,02	0,81	0,00	0,41	0,002	0,80	0,00	0,40	0,50
Media	0,70	0,00	0,35	0,01	0,83	0,00	0,41	0,002	0,84	0,00	0,42	0,13

Tabla 5.2: Resultados globales en el problema del APC

	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
1-NN	0,70	0,00	0,35	0,01	0,83	0,00	0,41	0,002	0,84	0,00	0,42	0,13
RELIEF	0,49	0,00	0,25	0,62	0,75	0,00	0,38	0,16	0,72	0,00	0,36	0,76
BL	0,69	0,24	0,47	525,00	0,87	0,34	0,60	37,80	0,85	0,23	0,54	425,20

Análisis de datos.

Como podemos ver en las tablas, las ejecuciones del 1NN y de Greedy son considerablemente más rápidas.

En particular, el 1NN lo único que realiza es introducir los datos en una matriz y calcular las distancias. Esto se realiza tanto en el Greedy y en la BL, por lo que las ejecuciones con este algoritmo van a ser las mejores. En la tabla se puede que mientras los tiempos de ejecución del 1NN, la mayor media en tiempo ha sido 0.01 segundos, mientras que la mayor media en Greedy ha sido 0,76 segundos y en la BL la mayor media ha sido 525 segundos. Los factores que son relevantes para el tiempo son la cantidad de muestras (se puede ver como en los datos que tienen más muestras los tiempos son mayores) y obviamente el número de iteraciones. En la BL, por cada partición, tenemos que dar 15 mil iteraciones, que en comparación con Greedy que da tantas iteraciones como numero de muestras haya en CADA partición y en el 1NN que no tenemos bucles añadidos a los que tenga internamente el clasificador implementados.

Moviéndonos a la clasificación de los datos podemos ver que en media, los resultados entre el 1NN y la búsqueda local son parecidos, mientras que los del Greedy son algo peores. Esto puede venir provocado por la modificación de valores del Greedy. El incremento/decremento de los valores se hace componente a componente sin tener en cuenta el resto de componentes, y si se “equivoca”, no vuelve atrás (como en la BL) y ese error lo arrastrará hasta el final.

En cuanto a la clasificación entre BL y el 1NN podemos ver que la clasificación no varia de una forma realmente significante.

Si queremos hablar de la tasa de reducción, vemos que ambos algoritmos de comparación no proporcionan ningún tipo de reducción, ya que el 1NN por si mismo no implementa ningún tipo de reducción de características y en el Greedy, como el operador se basa en la distancia, podríamos tener pocos ejemplos de una muestra con una característica que puede ser relativamente importante y podríamos eliminarla

Para decidir que algoritmo es mejor deberíamos de tener en cuenta muchos factores, tales como el tiempo que tenemos de ejecución, la calidad respecto a la clasificación y tendríamos que valorar la necesidad de la eliminación de características, puesto que si no conocemos la dependencia entre las características y queremos reducir el número de las mismas, si o si me declinaría por usar la BL (o algún otro método que estudiemos en las prácticas). Si esto no fuese relevante, me pensaría el uso de la BL por los recursos en tiempo, ya que vemos que puede tardar mucho tiempo en comparación con el resto.

Bibliografía

- https://docs.scipy.org/doc/numpy-1.14.0/reference/
- http://scikit-learn.org/stable/modules/classes.html#module-sklearn.neighbors
- http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier
- http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html
- http://scikit-learn.org/stable/modules/neighbors.html
- https://stackoverflow.com/questions/21052509/sklearn-knn-usage-with-a-user-defined-metric
- https://anaconda.org/anaconda/python