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

3ºA Computación y Sistemas Inteligentes
1NN simple, 1NN con pesos aprendidos con greedy RELIEF y con Búsqueda Local

Natalia Fernández Martínez
77449273Q
Correo: nataliafm@correo.ugr.es
Grupo 3 Jueves 17:30 - 19:30

Descripción del problema

El problema abordado en esta práctica es el de Aprendizaje de Pesos en Características (APC). Consiste en crear un clasificador, el cual tiene como input una serie de muestras ya clasificadas, y aprende de ellas como clasificar otras muestras que no tengan clasificación.
Cada muestra está representada como un vector, donde sus elementos son los valores que tiene la muestra para una serie de atributos.
No solo queremos crear un clasificador, sino también optimizar su rendimiento otorgándole pesos a cada característica para distinguir las que son más influyentes a la hora de hacer la clasificación.
La función que se quiere maximizar, ya que va a representar la calidad de nuestro algoritmo es:

$$F(W) = 0,5 * tasa-clas(W) + 0,5 * tasa-red(W)$$

La tasa de clasificación es la precisión que tiene nuestro clasificador, es decir, cuántas muestras están bien clasificadas.
La tasa de reducción es la simplicidad que tiene nuestro clasificador, es decir, cuántas características tienen un peso muy pequeño, lo que significa que tienen menos influencia en el proceso de clasificación y podrían ignorarse.
En la función a maximizar se le da la misma importancia a las dos características, estamos buscando un clasificador que acierte y que además tenga cierta simplicidad.
En caso de que dos algoritmos tengan un valor parecido de la función, se determina cuál es mejor por su tiempo de ejecución.

Descripción general de los algoritmos empleados

En esta práctica se evalúa la comparación entre un KNN sin tener en cuenta los pesos, un KNN obteniendo los pesos con un algoritmo greedy RELIEF, y un KNN obteniendo los pesos por búsqueda local.
Para implementar el KNN, he usado las clases:

- StratifiedKFold para crear las particiones manteniendo la proporción de clases en cada una. Le he especificado un valor de random_state para que sea el mismo en los tres algoritmos.
- KNeighborsClassifier para crear el clasificador KNN. Le he especificado el parámetro de número de vecinos para que sea 1NN, y para el primer algoritmo que no tiene en cuenta los pesos, le he especificado que los pesos son uniformes.

Ambas clases son de la librería scikit-learn.

Otras clase que he usado es la clase arff de la librería scipy para leer los ficheros de datos; además de las librerías numpy y time (esta última para medir los tiempos de ejecución).

En cuanto al tratamiento de los datos, lo he unificado todo en una única función. Se le pasa como argumento los datos tal y como están cuando se cargan con la clase arff, y la función se encarga de:

1. Separar las X de las y.
2. Recorrer las X para normalizar las características fila por fila.
3. Crear el objeto skf de la clase StratifiedKFold y crear las cinco particiones sobre las que se va a iterar después en los algoritmos.

Solo he usado la semilla 42 para generar todos los valores aleatorios.

He implementado la práctica usando Python 3.7 y la IDE Spyder3 en el SO Windows 10.

Algoritmo de Búsqueda Local

Pseudocódigo

```
Se itera 20 * num_características o 15000 veces max:
     Se genera un valor Zi correspondiente a un valor
aleatorio de la distribución normal con varianza 0,3
     Se modifica el vector de pesos con Zi
     Si el nuevo valor del vector es menor que 0,2:
          Se pone a 0 a menos que sea el último valor no nulo
del vector
     Si el nuevo valor del vector es mayor que 1.0:
          Se trunca a 1.0

     Se comprueba si el nuevo vecino generado es mejor que en
el que estamos usando el vector de pesos modificado para
clasificar los inputs de entrenamiento y de prueba
     Se calcula el valor de la función objetivo, que queremos
maximizar
     Si el valor para esta función es mayor que el que
tenemos, hemos encontrado un vecino mejor así que nos movemos
a él
     Si el valor es menor, el vecino es peor así que
deshacemos los cambios al vector de pesos y seguimos
recorriendo el vecindario
```

Algoritmos de comparación

1NN

```
def KNN(X,y,skf):
     Crear los vectores vacíos para las soluciones
     Iterar sobre las particiones:
          Se divide la partición en entrenamiento y prueba
          Se crea un clasificador 1NN usando pesos uniformes
          Se entrena a este con los valores de entrenamiento
          Se clasifican los valores de prueba
          Se obtiene el porcentaje de aciertos de la clasificación
          Se obtiene el tiempo de ejecución de la iteración
     Se devuelven los porcentajes de acierto y los tiempos de
ejecución
```

Greedy RELIEF

```
def RELIEF(X,y,skf):
     Crear los vectores vacíos para las soluciones
     Iterar sobre las particiones:
          Se divide la partición en entrenamiento y prueba
          Se crea un vector de ceros para los pesos
          Para cada elemento del conjunto de entrenamiento:
               Se obtiene el amigo más cercano al elemento
               Se obtiene el enemigo más cercano al elemento
               Se usan para cambiar el vector de pesos
          Para cada elemento del vector de pesos actualizado:
               Si es menor que 0.2, se ignora la
          característica
               Si es mayor, se normaliza usando el valor más
          grande
          Se aplican los pesos a los inputs
          Se entrena al clasificador con los nuevos inputs
          Se clasifican los valores de prueba
          Se obtiene el porcentaje de aciertos de la
clasificación
          Se obtiene el porcentaje de reducción en los pesos
          Se obtiene el tiempo de ejecución de la iteración
     Se devuelven los porcentajes de aciertos y reducción y
los tiempos de ejecución de las cinco iteraciones
```

Para encontrar el amigo y el enemigo más cercanos, uso dos funciones:

```
def encontrarAmigo(X, i, y):
     Borra de X al elemento del cual estamos buscando su amigo
(leave-one-out)
     Obtiene el valor de este elemento
     Recorre todo X (menos el elemento):
          Se obtiene el valor
          Se calcula la distancia entre este elemento y el
original
          Si es la menor distancia registrada y los dos
elementos son amigos, se guarda como posible solución
          Si se ha guardado el elemento, se pone su distancia
como nueva menor distancia registrada hasta el momento
     Devuelve el elemento amigo más cercano

def encontrarEnemigo(X, i, y):
     Obtiene el valor del elemento i
     Recorre todo X:
          Se obtiene el valor del elemento
          Se calcula la distancia entre este elemento y el
original
          Si es la menor distancia registrada y los dos
elementos son enemigos, se guarda como posible solución
          Si se ha guardado el elemento, se pone su distancia
como nueva menor distancia registrada hasta el momento
     Devuelve el elemento enemigo más cercano
```

Procedimiento considerado para desarrollar la práctica

Para desarrollar la práctica, he usado scikit-learn para implementar los aspectos del KNN (los clasificadores y la separación en particiones). La obtención de pesos con greedy RELIEF y Búsqueda Local la he implementado yo usando como guía el seminario 2 de la asignatura.

Análisis de resultados

Con el 1NN básico se obtienen porcentajes de acierto bastante buenos, sobre todo para el dataset Texture, y además su ejecución es casi instantánea.

Con el 1NN usando pesos obtenidos con greedy RELIEF, los porcentajes de acierto empeoran un poco, pero se compensa con altos porcentajes de reducción, lo que hace que tenga mucha más calidad que el 1NN básico. En cuanto al tiempo de ejecución, tarda un poco más que el básico pero no es un inconveniente muy grande.

Con el 1NN usando pesos obtenidos con Búsqueda Local, se obtienen los mejores porcentajes para los datasets Colposcopy e Ionosphere, pero para el dataset Texture son bastante peores que con los otros dos algoritmos. La razón por la que creo que ocurre esto es que, al darle la misma importancia a los aciertos y a la reducción de características, mi algoritmo tiende a favorecer uno de los dos parámetros al máximo.

Esto no es un problema en los datasets con pocas clases, ya que podemos ver que hay particiones con un porcentaje de reducción muy alto, pero esto no hace que baje el porcentaje de aciertos. Si lo pensamos es lógico, ya que al haber dos clases, será más fácil que con un número reducido de características se pueda clasificar bien.
Sin embargo, en el dataset Textura hay 11 clases distintas, y se puede ver en los resultados que en las particiones con un porcentaje de aciertos muy alto hay muy poca reducción o es casi nula, y en las particiones donde hay un porcentaje de reducción muy bajo, es mucho menos preciso. Es bastante razonable que esto pase, porque si tenemos que clasificar muchas clases, el problema es mucho más complejo que si solo tenemos que clasificar dos, y necesitaremos más características.

La conclusión que saco de estos resultados, es que se podría modificar el algoritmo de Búsqueda Local para que tuviera en cuenta el número de clases distintas del problema, y modificar la función objetivo para que favorezca la reducción en casos con pocas clases, y la precisión en casos con muchas.

También hay que tener en cuenta que en el caso de la Búsqueda Local influye mucho la seed elegida para generar números aleatorios. Yo he usado la seed 42, pero con otros valores se pueden conseguir resultados (para estos dataset concretos) tanto mejores como peores que con esta. Esto es porque el “punto de partida” del vector de pesos depende de qué seed se haya usado, y si este está más cerca de un óptimo local que lleva al algoritmo a reducir mucho las características, el algoritmo se irá ahí antes que a otro óptimo que le lleve a tener mucha precisión.

Bibliografía

- https://docs.scipy.org/doc/scipy/reference/generated/scipy.io.arff.loadarff.html
- https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
- https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html#sklearn.model_selection.StratifiedKFold
- https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.normal.html