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

Alejandro Borrego Megías[1]

07/04/2022

[1]26504975M, alejbormeg@correo.ugr.es, Grupo 3 Martes 17:30-19:30

Contents

1 Descripción del problema APC. 2

2 Descripción de la aplicación de los algoritmos. 3
Métricas 3
Validación Cruzada 4
Clasificador KNN 4
Bases de datos a usar 6
Detalles técnicos 6

3 Descripción de los Métodos de Búsqueda 7
Algoritmo RELIEF 7
Algoritmo de Búsqueda Local 9

Chapter 1

Descripción del problema APC.

El problema que trataremos es el APC (Aprendizaje de Pesos en Características). Antes de explicar el problema tendremos en cuenta los siguientes elementos que se explicarán en el siguiente apartado:

- Conjunto de datos: Se presentará como usa sucesión de elementos sobre los cuales se han llevado a cabo una serie de mediciones o se aporta una serie de características y tienen asociada una etiqueta. Por lo tanto estamos en el ámbito de problemas de Aprendizaje Supervisado.
- Clasificador KNN: Utilizaremos el algoritmo de los K vecinos más cercanos con K=1 para clasificar ejemplos, de esta manera la etiqueta que asignaremos al ejemplo que clasificamos será la del elemento más cercano a él en el conjunto de datos del que disponemos.
- Distancia: Para medir la distancia entre elementos del conjunto usaremos la distancia euclídea ponderada:

$$d(e_1, e_2) = \sqrt{\sum_i w_i (e_1^i - e_2^i)^2}$$

  Como podemos ver, cada elemento del sumatorio viene multiplicado por una constante w_i.

De esta manera, establecido el contexto del problema, nuestro objetivo será hallar ese vector w de pesos que dará prioridad a unos atributos sobre otros, para ello usaremos diversas técnicas aprendidas en clase.

Por lo tanto, el APC es un problema que pretende optimizar el rendimiento del clasificador KNN.

Chapter 2

Descripción de la aplicación de los algoritmos.

Métricas

En todos los algoritmos que empleemos vamos a utilizar un clasificador 1-NN (como se dijo anteriormente). De esta forma las métricas que usaremos para comprobar la bondad del vector de pesos w obtenido por cada algoritmo serán:

- **Tasa de clasificación**: que medirá el porcentaje de instancias correctamente clasificadas aplicando validación sobre el conjunto T de datos correspondiente (lo haremos generalmente sobre el conjunto de datos que usamos para entrenar). Su expresión es la siguiente:

$$tasa_{clas} = 100 \cdot \frac{n\ instancias\ bien\ clasificadas\ en\ T}{n\ instancias\ en\ T}$$

- **Tasa de reducción**: medirá el porcentaje de características descartadas, es decir, aquellas cuyo peso esté cercano a 0. Pondremos el umbral en 0.1. Su expresión es la siguiente:

$$tasa_{red} = 100 \cdot \frac{n\ valores\ <\ 0.1\ en\ W}{n\ caracteristicas}$$

Finalmente, las dos métricas anteriores se combinarán en la que será nuestra **función objetivo**, que intentaremos maximizar, y que tiene como expresión la siguiente:

$$F(W) = \alpha \cdot tasa_{clas}(W) + (1 - \alpha) \cdot tasa_{red}(W)$$

Validación Cruzada

Por otro lado, como sabemos de las técnicas de Aprendizaje Automático, para diseñar un clasificador, es necesario realizar dos tareas: **Aprendizaje y Validación**.

De esta forma, el conjunto de los ejemplos que tratamos se divide en dos, en conjunto de entrenamiento (que usaremos para entrenar el clasificador) y en el conjunto de Validación (que lo usaremos para validarlo).

Para mayor seguridad y garantías de generalización, se suelen realizar varias particiones Entrenamiento-Validación sobre los datos originales, es por ello que nosotros usaremos la técnica denominada **Cross Validation 5-Fold**, que consiste en los siguiente:

- El conjunto de datos se divide en 5 partes (5-fold) disjuntas al 20% con distribución de clases equilibrada.
- Aprendemos un clasificador con el 80% de los datos (es decir, con 4 de las 5 particiones) y se valida en el 20% restante (la partición que no hemos usado). Esto lo realizaremos 5 veces cambiando cada vez el conjunto de validación.
- De esta manera obtenemos 5 valores distintos de pocentaje de clasificación, por lo que para medir la calidad del método se realizará la media de los 5 obtenidos.
- Lo explicado anteriomente es el método general, pero en nuestro caso en cada iteración obtendremos un resultado de $tasa_{clas}$, $tasa_{red}$, del tiempo de ejecución del algoritmo y de la función objetivo. Por lo que nosotros al final promediaremos todas estas cantidades para obtener la bondad de nuestro clasificador final.

Clasificador KNN

Finalmente, como ya se ha comentado, usaremos el **clasificador KNN** con K=1, por lo que en esta última parte de la sección vamos a explicar algunos detalles del clasificador.

El proceso de aprendizaje consiste en manteenr en memoria una tabla con los ejemplos de entrenamiento junto con la clase que tienen asociada, de esta forma, dado un nuevo ejemplo se calculará la distancia con los n elementos de la tabla y se escogen los k elementos más cercanos, de esta forma la clase que asignaremos al nuevo ejemplo sería la mayoritaria entre estas k clases. En nuestro caso como K=1 se simplifica el procesp, pues asignamos la clase del elemento más cercano.

La descripción en Pseudocódigo sería:

```
cmin=clase del primer elemento en entrenamiento e1
dmin=distancia entre e1 y el nuevo ejemplo
```

```
Para i=2 hasta m hacer:
  Se calcula la distancia entre ei y el nuevo ejemplo.

  Se comprueba si la nueva distancia es menor que dmin:
    - Si es menor se elige cmin=clase de ei y se actualiza dmin.
    -Si no es menor se continúa.

Devolver cmin.
```

Por otro lado la distancia que usaremos para el clasificador será la comentada en el apartado anterior, que denominaremos **distancia euclídea ponderada**, recordamos su expresión:

$$d(e_1, e_2) = \sqrt{\sum_i w_i (e_1^i - e_2^i)^2}$$

Finalmente usaremos dos técnicas diferentes para comprobar la precisión de nuestro clasificador según nos encontremos en la fase de entrenamiento o la de validación en cada partición realizada en el 5-Fold Cross Validation:

- Entrenamiento: Usaremos el método **Leave One Out**, su pseudocódigo es el siguiente:

  ```
  Para cada elemento e en Entrenamiento:
    etiqueta_predicha=Clasificador1NN(Entrenamiento-{e},e)

    Si etiqueta_predicha=etiqueta_real de e
      correctos ++

  return correctos/num_datos
  ```

  Es decir, en cada paso eliminamos un elemento del conjunto de datos de entrenamiento y tratamos de clasificarlo, si las etiquetas predicha y real coinciden sumamos un acierto, si no se continúa. Finalmente se devuelve el porcentaje de acierto.

- Validación: En este caso método será intentar clasificar cada elemento del conjunto de validación usando los del conjunto de entrenamiento:

  ```
  Para cada elemento e en Validacion:
    etiqueta_predicha=Clasificador1NN(Entrenamiento,e)

    Si etiqueta_predicha=etiqueta_real de e
      correctos ++

  return correctos/num_datos
  ```

Para este método es muy importante que los datos estén normalizados entre [0,1] para no priorizar unos atributos sobre otros. Por ello, para cualquier conjunto

de datos que utilicemos, lo primero que se hará será normalizar los datos y mezclarlos (para evitar que las clases estén desbalanceadas). El algoritmo para normalizar es el siguiente:

```
Para cada atributo a:
  Se calcula el máximo y mínimo de entre todos los datos

  Para cada elemento del conjunto de datos:
    Cambiar valor de atributo a por (x-min)/(max-min)
```

Bases de datos a usar

Detalles técnicos

En nuestro problema, el conjunto de datos se va a representar por medio de un vector de pares (atributos, etiqueta).

El esquema de representación de soluciones será un vector de doubles que representará el vector de pesos w solución, de manera que si una característica tiene asociado un peso de 1, significa que se considera completamente en el cálculo de la distancia, si tiene un 0.1 o menos no se considera y para cualquier otro valor intermedio pondera la importancia del atributo que tiene asociado.

Chapter 3

Descripción de los Métodos de Búsqueda

En esta sección vamos a describir las distintas técnicas que emplearemos para resolver el problema APC.

Algoritmo RELIEF

Se trata de un algoritmo Voraz (greedy) de búsqueda secuencial, en el cual se parte de un vector de pesos inicializado a 0 que incrementa cada componente en función del enemigo más cercano a cada ejemplo y la reduce en función del amigo más cercano a cada ejemplo.

Entendemos el enemigo más cercano como el elemento más próximo al que estemos considerando que tiene asociada una clase distinta a la suya. Del mismo modo consideramos el amigo más cercano como aquel que es más próximo al elemento que estamos considerando y tiene misma clase asociada.

La implementación del algoritmo es la siguiente en pseudocódigo:

```
    Inicializamos w a 0

    Para cada elemento e en el conjunto de entrenamiento E:
    se calcula distancia componente a componente al enemigo más cercano de e en E
    se calcula distancia componente a componente al amigo más cercano de e en E
    se calcula w=w+distancia_enemigo-distancia_amigo componente a componente

  Normalizamos w

  return w
```

Por su lado las implementaciones para las funciones que calculan las distancias al enemigo y amigo más cercano son las siguientes:

```
elemento de entrada a
dmin=distancia euclídea entre a y el primer elemento de entrenamiento
distancia=0

Para cada elemento e en entrenamiento E:
  distancia=distancia euclídea entre e y a

  Si distancia < dmin && clase de e!= clase de a
    dmin=distancia
    enemigo_mas_cercano=e

Calculamos distancias componente a componente entre a y enemigo_mas_cercano

return distancias_componente_componente
```

Para el amigo más cercano sería análogo.

En el código, la estructura de datos que hemos empleado han sido los vectores de la STL. Además hemos tenido que sobrecargar los operadores de suma y resta para realizar sumas y restas componente a componente entre dos vectores.

Algoritmo de Búsqueda Local

En segundo lugar hemos empleado la técnica de búsqueda por trayectorias simples para implementar el algoritmo de **búsqueda local del primer mejor**.

Antes de describir el algoritmo vamos a aclarar algunos puntos necesarios para su comprensión:

- Definimos el entorno de una solución w como el conjunto formado por las solucuiones accesibles desde ella a través de un movimiento, que en nuestro caso será una mutación de una componente del vector w por medio de un valor aleatorio generado por una Distribución normal de media 0 y desviación típica $\sigma = 0.3$.

$$Mov(W, \sigma) = W' = (w_1, ..., w_i + z_i, ..., w_n)$$

$$z_i \sim N(0, \sigma^2)$$

- Para asegurarnos de que tras aplicar una mutación, nuestro vector resultante siga cumpliendo las restricciones del problema debemos truncar la componente modificada para que su valor se encuentre entre [0,1].

El tamaño del entorno de cada solución es infinito por ser un problema de codificación real. Para solucionar este problema vamos a mutar cada componente del vector w en un orden aleatorio y sin repetición hasta que **haya mejora en la función objetivo** o se hayan mutado todas las componentes. Si se produce mejora aceptamos la solución vecina y comenzamos de nuevo. Si no se produce ninguna mejora tras mutar las n componentes del vector se vuelve a repetir el proceso sobre la solución actual.

Este método que hemos descrito se denomina **Búsqueda Local del primer Mejor**, y en nuestro caso concreto partimos de un vector cuyas componentes son valores aleatorios de una distribución uniforme en el intervalo [0,1] y repetiremos el proceso descrito anteriormente hasta que se hayan realizado un máximo de 15000 llamadas a la función de evaluación o bien hasta que se hayan realizado un máximo de $20 \cdot$ n mutaciones sobre la solución actual sin que haya mejora (se han visitado $20 \cdot$ n vecinos sin que haya mejora).

Vamos a presentar a continuación los pseudocódigos del algoritmo y sus funciones.

- Inicialización Búsqueda Local: Es la función para inicializar el vector w antes del algoritmo. Utilizamos el generador de números pseudoaleatorios basado en el algoritmo de Marsenne Twister, que funciona muy bien como generador de números aleatorios[1].

```
InicializacionBL(int dimension, int i){
  Declaramos el vector w
```

[1] se denomina mt19937 debido a que se basa en el primo $2^{19937} - 1$

```
  Inicializamos el generador mt19937 con la semilla i
  Inicializamos la distribución uniforme en [0,1]

  Mientras que i<dimension:
      elem_generado=dist(gen)
      añadimos a w elem_generado

  return w
}
```

- Movimiento: es la función que realiza la generación de vecinos en el entorno de la solución actual, volvemos a usar el generador de números pseudoaleatorios Marsenne Twister.

```
Mov(vector w, double sigma, int pos, int i){
  Inicializamos el generador mt19937 con la semilla i
  Inicializamos la distribución Normal(0,sigma^2)

  z=elemento generado por la distribucion Normal
  w[pos]=w[pos]+z

  Si (w[pos]>1.0){
      w[pos]=1.0
  }

  Si(w[pos]<0.0){
      w[pos]=0.0;
  }
}
```

- Algoritmo de BL: Como aclaración al pseudocódigo, el vector w viene ya inicializado con la función anterior.

```
BusquedaLocal(vector de pares datos, vector w, semilla){
  Establecemos semilla(semilla)
  creamos vector con orden de mutaciones
  contador_evaluaciones=contador_mutaciones=0
  mejpra=false
  Vector w_mutado=w

  Mientras(contador_evaluaciones<15000 && contador_mutaciones<20*tam(w))
    mezclar orden de mutaciones
    mejora=false

    for(i=0; i<tam(w)&& mejora==false; i++)
      w_mutado=w
      Mov(w_mutado,0.3,orden_mutaciones[i])
      contador_mutaciones++
```

```
      Obtenemos precision en entrenamiento con LeaveOneOut
      Obtemenos tasa de reduccion
      Obtenemos valor de función evaluación
      contador_ev++

      Si mejora la funcion evaluacion
        w=w_mutado
        mejora=true

        contador_mutaciones=0
}
```