BÚSQUEDA MULTIARRANQUE

Sergio Padilla López

Práctica 2

Selección de Características

Algoritmos considerados:

- BMB
- GRASP
- ILS

DNI: 26505292-T
Email: splsergiopadilla@gmail.com
Grupo de práctica: Jueves de 17.30 a 19.30h (Grupo de Óscar)

Índice

1.- Descripción del problema.

El problema seleccionado es Selección de características. El problema consiste en maximizar la tasa de éxitos del clasificador K-NN, en este caso para k=3..

Para ello, primero me he encargado de la parte de creación y entrenamiento del 3-NN. Separamos el conjunto de datos en 2 conjuntos repartidos equitativamente por clases. Con ellos utilizaremos la validación cruzada para evaluar los algoritmos y hacer una media de todas las medidas.

2.- Descripción de los algoritmos.

Consideraciones comunes a los algoritmos:

- Solución binaria: Elegido un vector de 0 o 1, donde 1 significa que la característica está seleccionada.
- Funciones objetivos:
    - tasa_clas
    - tasa_red

Pseudocódigos:

```
tasa_clas(vector<int> solucion, training_set, test_set){
        reducimos las características de los conjuntos con las seleccionadas en solución.
        creamos 3-NN únicamente con el conjunto de entrenamiento
        evaluamos el conjunto de test y contamos numero de aciertos

        return 100*nExitos/new_test.size()
}

tasa_red(vector<int> solution){
        for(i in solution)
                si i == 1
                        count++

        return 100*(solution.size() - count)/solution.size()
}
```

- Condición de parada común:
        Realizar 15000 evaluaciones.

- Generación de soluciones aleatorias (para BMB e ILS):

```
sol_random(int n){
        S => empty;

        for(i = 0 hasta n)
                S = S U {random(0,1)}

        devuelve S;
}
```

- Busqueda Local empleada:

```
S = solución aleatoria
S' = S
better = true

mientras ( i < 15000 && better){
        better = false
        order <- vector random();

        for( j in order && !better){
                pos <- j
                flip(S, pos)
                new_rate <- tasa_clas(S)

                si new_rate mejora rate_actual_s
                        rate_actual_s <- new_rate
                        better = true
                si no
                        flip(S, pos)
                i++
        }
}

devolver S;
```

3.- Estructuras de método de búsqueda.

3.1. BMB

```
BMB(){
        S => empty;

        for(i = 0 hasta 25)
                S = S U BL(sol_randon())
```

```
        devuelve mejor(S)
}

mejor(vector<vector<int>> S){
        best => empty;
        cost = -1
        new_cost;

        for(v in S){
                new_cost = tasa_class(v)

                SI new_cost mejor cost{
                        cost = new_cost
                        best = v
                }

        devuelve best;
}
```

3.2. GRASP

```
GRASP(){
        S => empty;

        for(i = 0 hasta 25)
                S = S U BL(SFSR())

        devuelve mejor(S)
}

SFSR(){
        S <- [0,0,0,....,0]
        F <- [1,2,3,4,...,n]
        final <- false
        rate_s = 0
        rates // map<double, int>
        LCR => empty;

        mientras ( i < 15000 y F no vacio y no final){
                for(k in F){
                        Flip(S,k)
                        rates = rates U (tasa_class(S), k) //inserta ordenado
                        Flip(S,k)
                        i++;
                }

                Calculo nu;
```

```
                LCR = calculoLCR(rates, nu)
                shuffle(LRC)

                SI LRC[0].first > best_rate
                        best_rate = LRC[0].first
                        S = S U {LRC[0].second}
                        F = F\{F.find(LRC[0].second)}
                SI NO
                        final <- true
        }
}

calculoLCR(rates, nu){
        // rates esta ordenado, luego LCR se forma desde nu hasta el final
        LCR => empty;

        for(i = nu hasta rates.end())
                LCR = LCR U rates(i)

        devuelve LCR;
}
```

3.2. ILS

```
ILS(){
        S <- solucion random
        best_S = BL(S)
        best_rate = tasa_clas(best_S)

        mientras i < 25{
                mutate_S = mutate(best_S, n)
        }

        possible_S = BL(mutate_S)
        possible_rate = tasa_clas(possible_S)

        SI possible_rate mejor best_rate
                best_rate = possible_rate
                best_S = possible_S

        devuelve best_S
}

// Muta aleatoriamente n caracteristicas
mutate(S, n){
        characteristics = {1, 2, 3, …, S.size()}
        shuffle(characteristics)
```

```
        for i = 0 hasta i < n
            pos = characteristics[i]
            Flip(S, pos)

        devuelve S
    }
```

4.- Algoritmo de Comparación.

```
SFS{
    S <- [0,0,0,....,0]
    F <- [1,2,3,4,...,n]
    final <- false
    rate_s = 0

    mientras ( i < 15000 y F no vacio y no final)
        pos = -1
        for i in F
            S(i) = 1

            si (rate_s < tasa_clas(S))
                rate_s = tasa_clas(S)
                pos_better = i
            S(i) = 0

        si pos != -1
            S(pos_better) = 1
            F \ {pos_better}
        si no
            final <- true
}
```

5.- Desarrollo de la práctica.

Comencé desarrollando el lector de archivo ARFF, a continuación implemente el clasificador 3-NN y por último los algoritmos de búsqueda.
Para la práctica me apoyé en el código proporcionado en clase.
En el archivo main.cpp se pueden encontrar todo el proceso de cómo hice las pruebas y captura de tiempos y tasas para las tablas. Particioné el conjunto de datos total en 2 conjuntos, hice evaluación cruzada y medida de tiempos, y repetí el proceso 5 veces con diferentes semillas, para cada semilla, se utiliza un conjunto como entrenamiento y otro como test y viceversa, realizando así las 10 evaluaciones.

6.- Experimentos y análisis de resultados.

5.1 Semillas

He realizado 5 ejecuciones, con 5 semillas diferentes: 69, 111, 24, 55, 230.

5.2 Resultados Obtenidos

Se adjunta tabla comparativa de datos.

5.3 Análisis de Resultados

Para esta práctica si he implementado la estrategia del leave one out en el 3-NN por lo que los porcentaje de tasa_clas son más relevantes que en la primera práctica, ya que no acertará tan fácilmente en caso de no haber una clase mayoritaria.

Basándonos en los datos recogidos, no podemos decantarnos tan fácilmente por un algoritmo como sí lo hicimos en la práctica 1, en esta ocasión podemos fijarnos que para un conjunto de datos pequeño, el algoritmo ILS es el mejor, ya que da un mayor porcentaje de acierto en menos tiempo que el resto. Cuando el tamaño de datos aumenta, podemos ver como este algoritmo deja de ser tan efectivo pasando a un tercer plano con respecto a los otros dos, es el que más tarda y el que peores resultados da. GRASP y BMB dan porcentajes de acierto muy similares en un tamaño de datos medio pero bastante dispares con un conjunto de datos muy grande, donde podemos ver que el algoritmo GRASP reduce aproximadamente a la mitad el tiempo empleado en el algoritmo y además, da el mejor porcentaje de acierto aumentando en un 9% la tasa media de acierto de BMB.

Fijándonos en %_red, como era esperable, el algoritmo que menos características selecciona en el greedy, sabiendo que el greedy parte de una solución vacía y explora en orden, es esperable que llegue pronto a una buena solución (ya que nunca empeora) y se cumpla pronto la condición de que ningún vecino mejore a la solución actual. Siguiendo al greedy, el algoritmo que menos características selecciona es el GRASP, lo que hace que quizás sea el algoritmo más completo de los 3 que estamos evaluando, ya que además daba los mejores resultados como hemos explicado antes. Además, esto nos da una pista del porqué esa diferencia de tiempos con respecto a los otros algoritmos, ya para el conjunto de datos más grande, GRASP selecciona casi un 40% menos de características que el resto de algoritmos.

Otra diferencia destacable entre GRASP y greedy, es que ambos crecen linealmente en cuanto a tiempo de ejecución, es decir, emplean un tiempo proporcional en la ejecución del algoritmo según sea el volumen de datos que se maneja, en lo que sí difieren bastante, es en el porcentaje de acierto, mientras que el greedy es mucho más inestable, el algoritmo GRASP mantiene un porcentaje más igualado en diferentes volúmenes de datos, no tiene un pico tan grande como experimente Greedy en el archivo “movement_libras”. Esto puede deberse a que GRASP tiene una parte de greedy aleatorizada pero mejorada con una optimización de la solución mediante BL, que como

vimos en la primera práctica, era el mejor algoritmo, por lo tanto hace que GRASP tenga ventajas similares al greedy pero mejorándolas con la BL.

El algoritmo ILS y el algoritmo BMB tienen una tabla muy parecida, ambos apenas difieren en cuanto a tiempo empleado, siendo el ILS siempre un poco más rápido, y tampoco difieren demasiado en el porcentaje de acierto donde BMB da un porcentaje de acierto ligeramente superior. En %_red también mantienen porcentajes muy parejos, no siendo superior la diferencia a un 2%, por tanto estamos antes dos algoritmos que se comportan muy parecidos sea cual sea el conjuntos de datos. Para explicar la diferencia mínima de tiempo de ejecución, podemos suponer que el algoritmo ILS no empeore la solución lo suficiente con las mutaciones como para que la diferencia en tiempo de ejecución de la componente de BL de ambos algoritmos difiera tanto como para que se haga notable en las 25 ejecuciones que ambos algoritmos hacen. Por tanto, si quisieramos un algoritmo diferente, tendríamos que elegir otra estrategia de mutación o mutar más componentes para que la diferencia fuera más apreciable.