Práctica 1.b: Búsquedas por Trayectorias para el Problema de la Selección de Características
SFS
Búsqueda Local de Primero el Mejor
Enfriamiento Simulado
Búsqueda Tabú

Gustavo Rivas Gervilla DNI: 75570417F
gustavofox92@correo.ugr.es
5º Doble Grado en Ing. Informática y Matemáticas
Grupo de los Viernes de 17:30 a 19:30

Índice

1 Descripción/formulación del problema abordado

Lo que intentamos hacer con nuestro algoritmos es encontrar un conjunto de características de unos datos que nos permitan realizar una clasificación suficientemente buena de nuevos datos que nos lleguen con las mismas características.

Hay un problema muy habitual en la vida real y es la de clasificar una serie de elementos en distintas categorías en función de información sobre ellos, esta tarea puede ser realizada por personas o, lo que es más eficiente, por un ordenador. Para realizar tal clasificación es habitual que se recojan multitud de datos sobre los distintos elementos que se quieren clasificar, de modo que en base a esta información podamos decidir si el elemento es de una categoría o de otra. Pensemos por ejemplo en clasificar fruta en base a si se desecha o no. Podemos pensar en recoger datos sobre el tamaño de esa fruta, su color, su textura o su dureza y en base a estas mediciones una máquina debería clasificar la fruta en buena o mala.

El problema está en que normalmente no se conoce tan bien el campo de estudio como para saber a ciencia cierta qué datos recoger, qué datos serán más relevantes a la hora de clasificar elementos de una determinada población. Entonces lo que se hace es recoger gran cantidad de información sobre los elementos para al menos intentar que no haya carencias en la información, esto por supuesto conlleva tanto el coste de adquirir esa información (no sabemos cómo de caro es realizar una determinada medición) y el coste computacional de procesar toda esa información. Entonces lo que nos gustaría es averiguar qué información, de entre toda la que hemos obtenido, es la verdaderamente relevante para la clasificación que queremos realizar.

Entonces partiendo de un conjunto de datos de aprendizaje, valores de características de distintos elementos, queremos ver con qué subconjunto de características podemos hacer una buena clasificación de esos elementos, así si tenemos que cada dato viene dado por una lista de n características $[f_1, f_2, ..., f_n]$ queremos obtener un subconjunto de esas características, de modo que teniendo sólo la información $[f_{s1}, ..., f_{sm}]$, se haga una buena clasificación del conjunto de datos de aprendizaje, del que conocemos por supuesto la clasificación perfecta de dichos datos. Y esperamos que con esa misma información se clasifiquen lo mejor posible nuevos elementos de fuera de la muestra de aprendizaje.

2 Descripción de la aplicación de los algoritmos empleados al problema

Dado que estamos ante un problema de selección las soluciones se representarán como vectores binarios de booleanos de tamaño el número de características a elegir, indicando si una característica se considera o no, así tendremos claramente un espacio de búsqueda de 2^n, siendo el n el número de características a elegir que también lo podemos ver como el tamaño del problema abordado.

Entonces para evaluar como de buena es una determinada solución hacemos lo siguiente:

```
tomamos de cada dato de entrenamiento las caracteristicas
    seleccionada por la solucion.

for cada dato de entrenamiento:
        clasificador_dato = construir clasificador 3NN con el resto
            de datos
        ver si clasificador_dato clasifica correctamente a ese dato

return media del numero de aciertos
```

Para generar las soluciones vecinas a una dada usamos el operador flip que funciona del siguiente modo:

```
def flip(sol, idx):
        cambiar el valor de la pos. idx de la sol por su negado
```

Cuando queramos ver cómo de buena es la solución final que ha obtenido nuestro algoritmo de búsqueda lo que hacemos es:

```
claficador = construir clasificador con los datos de entrenamiento
    solo con las caracterisiticas seleccionadas por nuestra
    solucion
ver el porcentaje de acierto de este clasificador etiquetando los
    datos de test
```

A cada algoritmo le daremos solamente los datos de entrenamiento separados en características y etiquetas pera que con la estragia de búsqueda que implemente nos devuelva la mejor solución posible para él, luego la evaluación de la solución final la haremos fuera del algoritmo con los datos de test (en la sección 8 ya diremos cómo hemos generado las particiones TODO).

3 Descripción en pseudocódigo de los algoritmos

3.1 Búsqueda Local (BL.py)

```
s = solucion inicial aleatoria

while True:
        for cada vecino de s explorados en orden aleatorio:

                if el vecino es mejor que s:
                        s = el vecino

                if se ha generado el numero max total de sols:
                        return s

                if se ha encontrado vecino mejor a la sol actual:
                        break

        if no hemos encontrado ninguna solucion mejor en todo el
            vecindario:
                return s
```

3.2 Enfriamiento simulado (ES.py)

El esquema de enfriamiento que seguimos es el de Cauchy modificado para el que calculamos la nueva temperatura a partir de la anterior del siguiente modo:

```
nueva_T = T_actual/(1+beta*T_actual) #beta es una param. del
    problema
```

y la temperatura inicial se calcula en base al "costo" de la solución inicial, que en este caso es su poder de clasificación para las muestras de entrenamiento de las que partamos, y dos parámtros μ y ϕ que fijamos a 0.3 con lo que el cálculo de la temperatura inicial es tan sencillo como el siguiente:

```
T0 = -mu*coste_sol_inicial/log(fi)
```

hay algo que me sorprende de este hecho y es que dado que el logarirmo neperiano de 0.3 es negativo, en nuestro cálculo, cuanto mejor sea la solución inicial de la que partamos, más elevada será dicha temperatura, es decir, más tardaremos en llegar a la temperatura final.

Pasamos ya a describir el esquema de búsqueda del algoritmo para el enfriamiento simulado:

```
generamos una sol. inicial aleatoria y calculamos los params. de el
     problema

while se acepte algun vecino en el vecindario explorado de la sol
    actual and no se hayan generado 15000 vecinos en total:

        n_exitos = 0 #vecinos generados y aceptados hasta el
            momento
        for hasta el max_vecinos permitidos:
                generamos un vecino aleatorio, cambiando una
                    caracteristica de la sol actual al azar
                n_vecinos_generados++

                delta = diferencia entre coste sol. actual y el
                    vecino

                if (el vecino es mejor que la sol actual or U(0,1)
                    <= exp(-delta/temperatura_actual) #U(0,1) valor
                     aleatorio uniforme en el [0,1]) and delta !=
                    0:
                        sol actual = vecino
                        n_exitos++

                        if la nueva sol es mejor que la mejor sol
                            encontrada hasta el momento:
                                mejor_sol = sol_actual

                if n_exitos == max_exitos or hemos generado el
                    numero total de vecinos posibles:
                        break

        actualizar temperatura

return la mejor solucion encontrada
```

Hemos añadido la condición de que delta no sea cero ya que de otro modo la exponencial que usamos para aceptar soluciones peores valdría uno con lo que estaríamos tomando siempre aquellos vecinos con el mismo coste que la solución actual, esto hacía que el algoritmo generase siempre los 15000 vecinos que tiene como tope, tardando media hora para una partición de wdbc.

3.3 Búsqueda Tabú (BT.py)

Aquí tenemos el pseudocódigo para la búsqueda tabú básica, en el también se ve cómo manejamos la lista tabú como una lista cíclica que consultamos para ver si un vecino ha sido generado mediante un movimiento tabú:

```
while no se hayan generado el maximo de vecinos permitido:
        elegimos n vecinos aleatorios a revisar, siendo n el maximo
             permitido por iteracion

        for cada vecino elegido:

                if el movimiento que lo genera esta en la lista
                    tabu(LT):
                        if el vecino es mejor que la mejor sol
                            encontrada and mejor que el mejor
                            vecino explorado en esta iteracion:
                                mejor vecino = nuevo vecino
                else:
                        if mejor que el mejor vecino explorado en
                            esta iteracion:
                                mejor vecino = nuevo vecino

                if hemos generado el maximo de vecinos totales
                    permitidos:
                        break

        sol_actual = mejor vecino encontrado en la iteracion

        if sol_actual mejor que la mejor encontrada hasta el
            momento:
                mejor_sol_encontrada = sol_actual

        LT[posicion_correspondiente] = movimiento que genero el
            mejor vecino de esta iteracion
        posicion_correspondiente = a la siguiente ciclicamente

return mejor_sol_encontrada
```

3.4 Búsqueda Tabú Extendida (BText.py)

El esquema de búsqueda es el mismo que en la BT básica, lo que cambia es que añadimos un esquema de reinicialización en el bucle interno que funciona del siguiente modo:

```
#iniciamos la memoria a largo plazo como un vector con tantos 0
    como caracteristicas haya
while no hayamos generado mas vecinos que el total permitido:
        #todo igual que en la BT basica

        s = mejor vecino
        soluciones_aceptadas++
        para cada car. seleccionada en mejor vecino aumentamos el
            contador de la memoria a largo plazo de ella en 1

        if mejor vecino es mejor que mejor sol. hasta el momento:
                actualizamos la mejor
        else:
                numero de iteraciones sin mejorar++

        if llevamos 10 iteraciones sin mejorar la mejor solucion:
                obtenemos numero aleatorio en el [1,3] con
                    probabilidades [0.25,0.25,0.5]

                if sale 1:
                        sol_actual = una solucion nueva
                            completamente aleatoria
                elif sale 2:
                        sol_actual = mejor_solucion_encontrada
                else:
                        #construimos una nueva sol_actual donde
                            cada caracteristica se rige por lo que
                            sigue
                        sol_actual[i] = U(0,1) < (1 - (numero de
                            veces que la car. i se ha aparecido en
                            las sols. aceptadas/
                            soluciones_aceptadas)

                Elegimos si reducir o aumentar el tamanio de la
                    lista tabu

                if Reducir:
                        LT = una lista con la mitad de tamanio con
                            todas las casillas a -1 (vacia)
                else:
                        LT = una lista con el tamanio de antes + la
                             mitad de ese tamanio con todo a -1

                Se insertaran nuevos elementos empezando desde el
                    inicio de la lista

return la mejor solucion encontrada
```

4 Breve descripción del algoritmo de comparación

Para el algoritmo de comparación, SFS, lo que hacemos es lo que vamos a reflejar en el siguiente pseudocódigo, la principal diferencia es que no tenemos la función usual flip sino que hemos hecho una función especial para poder realizar una función vectorizada en Python (aunque no produce ninguna mejora en tiempo a la versión que tendríamos si simplemente tuviésemos un for que recorriese las características que quedan por añadir hasta encontra la de mayor ganancia), esta función crea una nueva solución poniendo a True una componente de la solución que le pasamos y nos da su porcentaje de clasificación, como hacemos en el resto de algoritmos. Dicho esto pasamos al pseudocódigo:

```
sol = un array binario con todo False #conjunto vacio de
    caracteristicas

while tengamos ganancia and queden caracteristicas por aniadir:
        calcular el score de cada caracteristica por aniadir al
            agregarla al conjunto actual
        tomar la caracteristica que de mejor score en el calculo
            anterior

        if el score aniadiendola es mejor que el de el conjunto
            mejor al que habia:
                se agrega dicha caracteristica al conjunto
                quitamos esa caracteristica de el conjunto de
                    caracteristicas por aniadir
        else:
                no tenemos ganancia y acabamos el bucle

return el conjunto de caracteristicas al que hemos llegado
```

5 Procedimiento considerado para desarrollar la práctica

5.1 Desarrollo

El código usado en prácticas lo he implementado yo a partir de las explicaciones dadas tanto en clase de prácticas como de teoría sobre los distintos algoritmos, siendo de mucha utilidad los pseudocódigos de las transparencia de teoría así como el esqueme de funcinamiento de las búsqueda tabú, también de las transparencia.

La parte que no ha sido implementada por mí es la correspondiente a la implementación de las soluciones, es decir, tanto el KNN como los mecanismos de evaluación de las soluciones. Para ello he usado un módulo de Python llamado scikit cuya documentación la podemos consultar en la siguien web que es la que he consultado yo para poder usar aquello que necesitaba http://scikit-learn.org/stable/documentation.html, para instalar este módulo en Ubuntu hemos usado el siguiente comando de terminal *pip install -U scikit-learn*. Este módulo nos permite tanto crear un clasificador 3NN con los datos que elijamos y posteriormente etiquetar datos u obtener directamente el porcentaje de acierto para un conjunto de datos.

Además tiene una función para hacer las pruebas Leave One Out dándole un conjunto de datos y nos dice los índices para poder separar el conjunto de datos de aprendizaje en una muestra por un lado y el resto por otro y poder calcular la función objetivo como queremos.

5.2 Manual de usuario

Cada uno de los algoritmos están implementados en distintos ficheros de los que podemos ver su contenido en el *leeme* que se nos pide que incluyamos con el código. Los experimentos han sido ejecutados usando el fichero main al que le pasamos por argumentos el nombre del algoritmo a usar: KNN, SFS, BL, ES, BT y BText. La semilla se establece dentro del fichero al inicio de la ejecución del algoritmo para cada una de las particiones. Esto lo hemos hecho así para poder cortar la obtención de datos cuando sea necesaria y poder retomar los experimento desde la partición por las que nos quedáramos, así no se desajusta la semilla con la que hacemos los experimentos al retomar.

Entonces para ejecutar los experimentos de cada algoritmo lo que haremos es *python main.py algoritmo* y se ejecutará el algoritmo sobre todas las particiones de datos que tenemos.

6 Experimentos y análisis de resultados

Estás prácticas han sido implementadas en Python 2.7.6 y ejecutadas sobre un ordenador son S.O. Ubuntu 14.04 LTS, de 8GB de RAM y procesador Intel

Core i7-4790 CPU 3.60GHz con 8 núcleos, aunque dado que el código no ha sido paralelizado estos núcleos no han sido explotados al máximo.

Debido a algunas complicaciones y a la falta de tiempo he ejecutado los algoritmo con un número máximo de evaluaciones de 5000 en lugar de 15000 lo que se ve reflejado tanto en el tiempo de ejecución: por ejemplo la Búsqueda Tabú tarda aproximadamente un tercio del tiempo ya que como no tiene otro criterio de parada que no sea el número de evaluaciones, la reducción se ve afectada justamente por el número máximo de evaluaciones que se hagan.