Práctica 2.b: Búsquedas Multiarranque para el Problema de la Selección de Características
BMB
GRASP
ILS

Gustavo Rivas Gervilla DNI: 75570417F
gustavofox92@correo.ugr.es
5º Doble Grado en Ing. Informática y Matemáticas
Grupo 3

Índice

1 Descripción/formulación del problema abordado

Lo que intentamos hacer con nuestros 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) como 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 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 porcentaje de aciertos
```

Nuestros algoritmos irán explorando, según su mecanismo, el espacio de soluciones empleando la función anterior para tomar decisiones sobre qué movimientos se realizan en dicho espacio de búsqueda. Para poder generar las soluciones vecinas a una dada empleamos el operador de Flip, el cual funciona del siguiente modo:

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

Cuando finalice su proceso de búsqueda los algoritmos nos devolverán la solución que ellos han elegido (junto con el score dentro de la muestra de entrenamiento calculada como hemos dicho antes). Entonces una vez tenemos la solución nos queda por evaluar cómo de bien se clasifican los datos empleando un clasificador 3NN construido sólo con aquellas características seleccionadas por nuestra solución:

```
claficador = construir clasificador con los datos de entrenamiento solo con las
    caracterisiticas seleccionadas por nuestra solucion
return el porcentaje de acierto de este clasificador etiquetando los datos de test de los
     que consideramos solo las caracteristicas seleccionadas por la solucion
```

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 explicaremos cómo hemos generado las particiones). Para aquellos algoritmos que empleando algún tipo de aleatoriedad en sus decisiones inicilizaremos la semilla aleatoria antes de la ejecución de dicho algoritmo.

3 Descripción en pseudocódigo de los algoritmos

3.1 Búsqueda Multiarranque Básica (BMB.py)

```
for 25 veces:
        s_r = sol. aleatoria
        s = sol. encontrada por las Busqueda Local partiendo desde s_r

        if s es mejor que la mejor sol. encontrada hasta el momento:
                mejor_s = s

return mejor_s
```

3.2 GRASP (GRASP.py)

```
for 25 veces:
        s_greedy = sol. encontrada por el greedy aleatorio
        s = sol. encontrada por la Busquda Local partiendo desde s_greedy

        if s es mejor que la mejor sol. encontrada hasta el momento:
                mejor_s = s

return mejor_s
```

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:

```
sol_actual = partimos de una solucion aleatoria

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

        sol_actual = mejor vecino encontrado en la iteracion
        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:
                numero de iteraciones sin mejora = 0
                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 evaluación 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 **sklearn** 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`. 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

Los algoritmos han sido implementados en Python 2.7.6 para su ejecución son necesarios tener instalados los módulos compatibles con esta versión de Python siguientes:

- numpy (v 1.8.2) **sudo apt-get install python-numpy**
- scikit **sudo apt-get install python-scikits-learn**
- El resto de módulos empleados suelen venir con las distribuciones básicas de Python (random, sys y time).

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: 3NN, 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 por ejemplo los experimentos para el Enfriamiento Simulado lo que haremos es *python main.py ES* y se ejecutará el algoritmo sobre todas las particiones de datos que tenemos.

En el directorio **make_partitions** tenemos tanto los archivos arff como los códigos necesarios para elaborar las particiones que serán utilizadas por nuestros algoritmos. Esto no es necesario para ejecutar los programas puesto que en el directorio **partitions** tenemos almacenadas todas las particiones construidas. No obstante si se quisiera replicar el procedimiento de desarrollo de particiones lo que haríamos sería: eliminar los archivos con extensión .npy del directorio **make_partitions** (para tener seguridad de que no

se corrompan dichos archivos), ejecutar el loader con la instrucción **python loader.py** y a continuación elaborar las particiones con **python partition_maker.py** (dentro de este directorio que es donde se encuentran ambos códigos). Esto generará diversos archivos de extensión .npy con nombres que empiezan por arr, wdbc o libras (según la base) seguida de un número y la palabra test o training.

Estos archivos deberán reemplazar a aquellos que se encuentran en el directorio **partitions** y ya podremos ejecutar los algoritmos con dichas particiones desde el main.

6 Experimentos y análisis de resultados

6.1 Descripción de los casos del problema empleados y de los valores de los parámetros

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 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.

Hemos empleado tres bases de datos distintas, a continuación mencionamos sus tamaños ya que serán interesantes para el posterior análisis de resultados:

- **wdbc**: Tenemos **569** muestras con **30** características cada una procedentes de imágenes digitalizadas de una masa de mama. Estas muestras se clasifican en **2** clases distintas.
- **libras**: Aquí tenemos **360** muestras de **90** características cada una, éstas muestras son datos de distintos movimientos de la mano que se clasifican en **15** clases distintas.
- **arritmia**: Los datos de este conjunto son mediciones para determinar la presencia de arritmia cardiaca o no. Tenemos **386** muestras con **278** caracterísitcas cada una, a clasificar en **5** grupos en base al tipo de arritmia que indican los datos de la muestra.

Lo que hemos hecho con los datos ha sido un preprocesado, en primer lugar todos los datos han sido codificados como flotantes (las etiquetas de wdbc eran cadenas de texto y las hemos cambiado por los valores 0 y 1), también hemos puesto la etiqueta en la última columna de la tabla de datos.

También hemos normalizado cada una de las columnas de datos (sin contar la de etiquetas) de modo que los valores quedaran en el intervalo [0,1] mediantes la fórmula:

$$x_j^N = \frac{x_j - Min_j}{Max_j - Min_j} \text{ (siendo } Max_j \text{ y } Min_j \text{ el máximo y mínimo valor de los datos para la característica j-ésima de las muestras).}$$

El código empleado para este formateo de los datos está en el fichero **loader.py** que se adjunta en la entrega. Como podemos ver para realizar la normalización de los datos hemos usado una utilidad del módulo scikit-learn antes mencionado que se llama MinMaxScaler. Y para futuros usos de estos datos normalizados, y con el objetivo de no tener que realizar tales operaciones cada ocasión que queramos usarlos, hemos almacenado los arrays de numpy donde hemos almacenado los datos en sendos ficheros con extensión .npy usando la función **numpy.save**. Estos ficheros son: **data_wdbc.npy**, **data_libras.npy** y **data_arrhythmia.npy**.

A continuación hemos elaborado las distintas particiones sobre estos datos que serán utilizadas en los experimentos, para ello hemos elaborado el código recogido en el fichero **partition_maker.npy** donde simplemente tomamos una muestra aleatoria de los índices de los arrays antes generados, teniendo en

cuenta que la cantidad de muestras de cada una de las clases en las que se clasifican los datos sea lo más equilibrida posible entre la partición de test y su correspondiente partición de training. La semilla usada para la generación de dichas particiones, y que podemos ver en el fichero mencionado, ha sido la **12345678**. Nuevamente hemos almacenado cada una de dichas particiones en ficheros .npy, los cuáles están en la carpeta **partitions**.

Los parámetros empleados en cada uno de los algoritmos son los que se indican en el guión de prácticas con la salvedad de que, como ya hemos dicho, el máximo número de evaluaciones que le hemos permitido realizar a los algoritmo han sido **5000** en lugar de 15000 por cuestiones de tiempo. Para cada una de las ejecuciones de lo algoritmos inicializamos la semilla aleatoria a **12345678** nuevamente, esto lo podemos ver en el fichero **main.py**.

6.2 Resultados

Adjuntamos también las tablas en un fichero por si no se leen correctamente en el pdf.

Tabla 1: Resultados BMB

	Wdbc				Movement_Libras				Arrhythmia			
	%_clas_in	%_clas_out	%_red	T	%_clas_in	%_clas_out	%_red	T	%_clas_in	%_clas_out	%_red	T
Partición 1-1	98,25	95,42	40,00	3,60	79,44	72,78	51,11	12,98	74,23	65,10	49,64	125,10
Partición 1-2	98,59	95,44	43,33	3,48	77,78	69,44	54,44	13,65	71,88	65,98	50,00	143,26
Partición 2-1	97,89	95,77	56,66	3,70	77,22	67,22	62,22	12,95	74,23	64,06	47,84	141,67
Partición 2-2	98,59	95,09	56,66	3,28	78,33	70,00	57,78	12,29	70,31	64,95	53,96	144,62
Partición 3-1	96,84	96,83	46,66	3,47	77,78	71,67	51,11	12,09	72,16	64,58	53,60	121,83
Partición 3-2	98,94	95,09	43,33	3,04	76,67	71,67	54,44	14,37	72,92	63,92	53,60	164,83
Partición 4-1	98,25	95,42	50,00	3,68	78,89	75,00	48,89	12,08	72,68	67,71	52,88	152,11
Partición 4-2	98,94	95,09	50,00	3,41	76,67	77,22	54,44	12,25	75,00	64,43	54,68	154,49
Partición 5-1	98,25	95,42	43,33	3,18	75,00	77,78	56,67	14,36	73,71	66,15	51,08	123,77
Partición 5-2	98,94	94,74	53,33	3,37	78,89	77,22	61,11	13,00	72,40	61,86	50,00	140,37
Media	98,35	95,43	48,33	3,42	77,67	73,00	55,22	13,00	72,95	64,87	51,73	141,21

Tabla 2: Resultados Globales

Totales	Wdbc				Movement_Libras				Arrhythmia			
	%_clas_out	%_red	T	%_clas_in	%_clas_out	%_red	T	%_clas_in	%_clas_out	%_red	T	T
3-NN	96,412	95,917	0	0	68,106	70,607	0	0	63,625	64,765	0	0
SFS	97,043	94,934	85,996	16,745	74,274	66,774	89,884	62,559	79,214	70,308	97,912	147,213
BL	97,22	95,43	47,00	7,58	70,55	71,00	45,33	16,08	70,15	65,75	53,23	103,23
ES	97,61	95,25	48,66	82,09	71,939	70,939	56,107	173,02	69,477	65,494	48,774	771,943
BT básica	98,765	95,073	51,663	646,0437	76,66	70,941	55,217	436,677	73,31	67,038	53,698	737,61
BT extendida	98,625	95,286	48,998	687,294	74,606	70,051	50,552	506,1102	71,393	65,905	53,338	939,708

6.3 Análisis de resultados

Como podemos ver la tasa de acierto sobre Wdbc es la más elevada con respecto a las otras bases, además las tasas de acierto de todos lo algoritmos sobre esta base son muy parecidas, esto se debe a que, como dijo el profesor, esta base de datos es muy pequeña, es decir, el espacio de búsqueda es muy pequeño (2^{30}) y hay muy poca tasa de mejora entre una solución y otra.

Las tasas de acierto tanto para arritmia como para libras son muy parecidas como podemos ver en la gráfica para cada algoritmo. Es importante señalar cómo la tasa de acierto de los algoritmos usados en mejor que la tasa que da el 3NN original, es decir, usando todos los datos de la muestra. Con lo cual aquí se pone de manifiesto cómo el hecho de reducir el número de características a considerar no merma la

correcta clasificación de los datos, al menos dentro de la muestra ya veremos qué sucede fuera. De hecho reduciendo el número de características no sólo reducimos el cómputo necesario sino también el posible ruido que pueden tener algunos datos.

Es curioso ver cómo el algoritmo greedy produce los mejores resultados en la base de datos con un espacio de búsqueda mayor, de hecho produce mejores resultados en esa base de datos que en la de libras que tiene un espacio menor, probablemente por la naturaleza de los datos, es decir, cómo de distintas sean las muestras que componen las distintas clases entre sí. En mi opinión el hecho de que sea el SFS el que produzca mejores resultados que el resto en el espacio de búsqueda más grande se debe a que al fin y al cabo el greedy en cada momento va construyendo la mejor solución que puede mientras que en cambio el resto de métodos depende más de cómo de buena sea la exploración que realicen. Entonces en un espacio de tamaño 2^{278} el hecho de que impongamos a los algoritmos que sólo pueda hacer 5000 evaluaciones hace que su capacidad de encontrar una buena solución sea pequeña, además de depender un tanto del azar.

En cambio si podemos ver cómo para el espacio más reducido de libras sí que los algoritmos que hacen una búsqueda más extensa en el terreno, dan resultados más próximos a los que da el SFS e incluso superándolo como en las dos versiones de la Búsqueda Tabú. De hecho esta búsqueda da los mejores resultados sobre el conjunto de libras.

Vemos también cómo los algoritmos de BL, ES, BT y BText dan mejores resultados sobre la base de libras que sobre la de arritmia por lo que hemos comentado antes probablemente.

Veamos ahora cómo se comportan los algoritmos con respecto a la clasificación que producen sus características seleccionadas para los datos fuera de la muestra de entrenamiento. Lo primero que señalamos es que en la amplia mayoría de los casos la tasa de acierto es mayor dentro de la muestra que fuera, pero hay algunos casos donde es al revés. Esto se debe a que es probable que haya muestras de entrenamiento de una clase más cercanas a otra muestra pese a no pertenecer a la clase de dicha muestra. Con lo cual se puede dar el caso de que la "votación" sea mala en la muestra y en cambio la muestra fuera la distribución de puntos sea más favorable a una correcta clasificación de los datos. Por lo tanto no tiene que extrañarnos demasiado esto. Quizás si usásemos un 1NN sí que sería más probable que la muestra más cercana a otra perteneciera a la misma clase que ella (decimos la más cercana y no que en el 1NN hubiésemos obtenido una tasa de acierto del 100% dentro de la muestra puesto que usamos para evitar esto el Leave One Out).

Nuevamente el comportamiento para Wdbc es muy similar para todos los algoritmos, lo que era esperable por el razonamiento que hemos dado antes. Pero también observamos que si bien sigue habiendo diferencias entre el comportamiento de los distintos algoritmos para una determinada base de datos ahora las diferencias son menos notables entre unos y otros. Es decir, que aunque hay algunos algoritmos que son capaces de adaptarse mejor que otros para la muestra que le pasamos, luego la solución que nos dan se comporta más o menos del mismo modo fuera de la muestra. Poniendo de manifiesto que no podemos asegurar que una solución que sea buena dentro de los datos de entrenamiento vaya a serlo también fuera.

Algo que también cabe señalar es que en esta ocasión para la base de datos de libras el 3NN sí supera a dos algoritmos, la BText y el SFS, pero no obstante no mejora a todos los algoritmos con lo que no podemos decir que para obtener mejores resultados en datos desconocidos tengamos que usar toda la información de la que dispongamos. Además por ejemplo se comporta mejor que la Búsqueda Tabú Extendida pero no mejor que la Búsqueda Tabú, aquí se pone de manifiesto que depende fuertemente de los datos que se consideren porque la BT y su versión extendida no difieren mucho en comportamiento. Difieren en que la extendida hace una exploración más esparcida del espacio de búsqueda, pero esto con sólo 5000 evaluaciones puede que no se pueda ver tan reflejado en los resultados.

El hecho de que el 3NN con toda la información no mejore a todos los algoritmos puede venir de que no sólo haya características irrelevantes sino que, como hemos señalado anteriormente, haya datos cuya medición haya sido errónea y que simplemente lo que nos estén haciendo sea empeorar el resultado.

Evidentemente la tasa de reducción para el 3NN es 0 pues tomamos todas las características al no tener otro criterio. Ahora el que produce mayor reducción es el greedy lo que es lógico por la forma en la que trabaja, agregando progresivamente propiedades hasta no encontrar mejora ninguna.

Las tasas de reducción para el resto de algoritmos son muy similares entre sí, si lo pensamos dado que nuestra función de coste sólo tiene en cuenta la tasa media de acierto en la muestra de entrenamiento (no tenemos en cuenta también la reducción que consigue) es lógico que esto suceda así, o al menos que no haya un algoritmo que claramente produzca mejores resultados que otro. Partimos de una solución totalmente aleatoria y nos movemos por los vecinos solamente teniendo en cuenta su score y algún término de azar como sucede en el Enfriamiento Simulado. Por lo tanto no podemos decir nada acerca de por qué se comportan así el resto de algoritmos respecto a la reducción y sólo podemos atribuirlo al azar.

Evidentemente el 3NN tarda 0 segundo puesto que su tiempo de búsqueda de solución es 0 pues simplemente devuelve un solución con todos los valores a True, es decir, escoge todas las características sin tener ningún criterio.

Como podemos ver los tiempos de ejecución del SFS y la Búsqueda Local son "similares" puesto que en esencia funcionan del mismo modo, parten de una solución, exploran su entorno y eligen el mejor, continuando hasta que no encuentran mejora. Y si nos fijamos los timepos de la búsqueda local son mejores incluso que los del greedy y esto se debe a lo que mayor tiempo de cómputo consume en todos los algoritmos que hemos diseñado, la evaluación de las soluciones. Y es que mientras que el greedy ha de explorar todo el entorno de la solución, la búsqueda local evalúa vecinos hasta que encuentra uno mejor que la solución actual; con lo que el número de evaluaciones es menor y por tanto el tiempo de ejecución medio también.

Observemos ahora lo que sucede con los tres algoritmos que quedan y que tienen tiempos de ejecución mayores que los anteriores. El enfriamiento simulado presenta menores tiempos de ejecución para las bases de Wdbc y Libras, esto se debe a que el enfriamiento no siempre realiza las 5000 evaluaciones que tiene como máximo ya que tiene otro criterio de salida, que es no aceptar ningún vecino en la iteración. Entonces en la mayoría de casos el enfriamiento saldrá antes de completar las 5000 evaluaciones con lo que el tiempo de cómputo, que como ya hemos dicho es la mayor parte el cálculo del score, se reduce.

Ahora bien como vemos en el caso de arritmia la Búsqueda Tabú Básica si tiene un tiempo de ejecución menor, eso probablmente se deba a que en esta base las soluciones cambian más de unas a otras y por tanto que se acepten más vecinos. Además cuanto mayor sea el tamaño del problema (número de características) más lento enfriará el algortimo, o lo que es equivalente, más lenta bajará la probabilidad de aceptar vecinos peores, con lo que esto hace que se haga un mayor número de evaluaciones, pudiendo llegar a las 5000. Además el cálculo de la exponencial para ver si aceptamos un mal vecino o no también conlleva un cómputo.

Y finalmente si nos fijamos en las búsquedas tabú vemos cómo claramente la extendida conlleva más tiempo de cómputo. Dado que estos algoritmos tienen como único criterio de parada el realizar 5000 evaluaciones ambos realizan el mismo números de cálculo de score, ahora bien la búsqueda tabú extendida consume más tiempo debido al mantenimiento de la lista tabú y los mecanismos de reinicialización. Además observamos otra cosa, y es que como el número de evaluaciones es el mismo sea cuál sea la base de datos actualizada

ahora la ejecución en Wdbc es más lenta que en libras, ya que al haber un número mayor de muestras (569 ¿ 360) lo que más pesa es el mecanismo de Leave One Out que se realiza 569 veces en lugar de 360 (la diferencia de tamaño de las componentes de los vectores para calcular las distancias en ambos casos es de 90-30 = 60 con lo que esto no afecta demasiado al cómputo). No sucede lo mismo para Arritmia que pese a tener un menor número de muestras tarda más esto se debe en el caso de las BT básica a que el cáculo de la distancia puede resultar muy costoso pues ya la diferencia de longitudes entre vectores es de 278-30 = 248. En el caso de la extendida los mecanismos de reinicialización mediante memoria a largo plazo son más costosos cuanto mayor sea el número de características. Lo que acaba nuestro análisis de los datos.

7 Bibliografía

- Documentación del módulo scikit-learn: http://scikit-learn.org/stable/documentation.html
- Documentación de Numpy: http://docs.scipy.org/doc/