Práctica 3 de Metaheurísticas

Universidad de Granada

Antonio Manuel Fresneda Rodríguez

antoniomfr@correo.ugr.es
77447672-W

Enfriamiento simulado, Evolución diferencial y Busqueda Local Iterativa
Grupo: A3 (Miércoles de 17:30 a 19:30)

9 de Junio de 2018

Índice

1 Descripción del problema

El problema del APC consiste en optimizar el rendimiento de un clasificador basado en vecinos mas cercanos a partir de añadir pesos asociados a cada características del problema que modifican su valor en el momento de calcular las distancias entre los ejemplos. Vamos a usar el 1-NN (consideramos un solo vecino). La variante del problema del APC que confrontaremos busca optimizar tanto la precision como la complejidad del clasificador. Asi, se puede formular como:
Maximizar $F(w) = \alpha * tasa\_clas(W) + (1 - \alpha) * tasa\_red(W)$ con:

$$tasa\_clasificacion = 100 * \frac{n^o\_instancias\_bien\_clasificadas}{n^o\_instancias\_T}$$

Sujeto a que: $w_i = [0, 1], 1 \leqslant i \leqslant n$, donde:

- $w = (w1, ..., wn)$ es una solución al problema que consiste en un vector de números reales w1 [0,1] de tamaño n que define el peso que pondera o filtra a cada una de las características fi.
- 1-NN es el clasificador k-NN con k=1 vecino generado a partir del conjunto de datos inicial utilizando los pesos en W que se asocian a las n características.
- T es el conjunto de datos sobre el que se evalúa el clasificador, ya sea el conjunto de entrenamiento (usando la técnica de validación leave-one-out) o el de prueba.
- $\alpha \in [0, 1]$ pondera la importancia entre el acierto y la reducción de la solución encontrada.

2 Descripción de la aplicación

Las soluciones se van a devolver en tres listas y un vector:

- La primera contendrá la tasa de clasificación para cada partición que hemos hecho de los datos.
- La segunda contendrá la tasa de reducción para cada partición que hemos hecho de los datos.
- La tercera contendrá el tiempo empleado en cada partición.
- El vector de pesos obtenido.

La función objetivo a maximizar va a ser: $F(w) = \alpha * tasa\_clas(w) + (1 - \alpha) * tasa\_red(w)$
Para el clasificador, he usado el clasificador KNN implementado en la librería scikit-learn:

- KNN=KNeighborsClasiffier(vecinos=1,métrica=dist,argumentos=w)
- KNN.fit(X_train,Y_train)

Si queremos usar evolución diferencial, la llamada será así:

- DE(X_train, y_train, X_test, y_test, alpha, CR, F)

Si queremos usar ILS, la llamada será así:

- DE(X_train,Y_train,alpha,sigma)

Si queremos usar enfriamiento simulado, la llamada será así:

- SA(X_train, y_train, x_test, y_test, alpha, nu, fi, Tfinal, sigma, pmaxaciertos, M)

Las particiones de los datos para 5-fold cross validation he usado una función de scikit-learn llamada StratifiedKFold. A esa función tiene como parámetro el numero de particiones (en nuestro caso 5), el conjunto de X y de Y, y devuelve dos listas por cada partición. En la primera están los indices de X y en la segunda los de Y.

```
1: folds=StratifiedKFold(particiones=5)
2: folds.datos(X,Y)
3: for indice_train, indice_test en folds do
4:     Particion_i_X_train=X[indice_train]
5:     Particion_i_Y_train=Y[indice_train]
6:     Particion_i_X_test=X[indice_test]
7:     Particion_i_Y_test=Y[indice_test]
8:     Llamada_algoritmo_KNN(Particion_i_X_train,.....)
```

Otro operador común que tenemos es la normalización de los datos. He usado una función de sk-learn que se encarga de normalizar los datos:

```
normalizer=MinMaxScaler()
normalizer.fit(X_train)
normalizer.transform(X_train)
normalizer.transform(X_test)
```

La función de valoración (o función objetivo), que calcula la tasa de clasificación, reducción y puntuación total:

```
function VALORACION(X,Y,w,KNN,porcentaje_clas,porcentaje_red)
    Y_vecinos=KNN.kneighbors()
    tasa_clas=Cuenta_iguales(Y_vecinos,Y)/tamaño_train
    tasa_red=cuenta_ceros(w)/tamaño_w
return (porcentaje_clas*tasa_clas)+(porcentaje_red*tasa_red),clas,red
```

Para la representación de las soluciones en evolución diferencial, he usado una clase con 4 atributos llamada Datos:

- Punt: Valor de la función objetivo
- Clas: Tasa de clasificación con el w correspondiente
- Red: Tasa de reducción con el w correspondiente
- w: Vector de pesos.

Esta clase datos tiene una función llamada actualizar(punt,clas,red,w) que se encarga de sobrescribir los datos de un objeto de la clase por los pasados por parámetro

3 Descripción del pseudocódigo

Algorithm 1 RELIEF

```
function RELIEF(X,Y,w))
    vecinos=vecinos_mas_cercanos(X,Y,metrica=euclidea)
    for xi,yi en X,Y do
        indice_vecinos_etiqueta1=vecinos.etiqueta(1)
        indice_vecinos_etiqueta2=vecinos.etiqueta(2)
        if yi==1 then
            amigo=indice_vecino_etiqueta1[0]
            enemigo=indice_vecino_etqueta2[0]
        else
            amigo=indice_vecino_etiqueta2[0]
            enemigo=indice_vecino_etqueta1[0]

        w=w+valor_absoluto(xi-X[enemigo])-valor_absoluto(xi-X[amigo])
    w_max=max(w)
    for wi en w do
        if wi<0 then
            wi=0
        else
            wi=wi/w_max
```

Algorithm 2 Greedy

```
function GREEDY(X,Y)
    tiempo1=time()
    w=vector_ceros(tamaño=n_características)
    w=RELIEF(X,Y,w)
    KNN=ClasificadorKNN(n_vec=1,metrica=Dist,args=w,X,Y)
    Y_vecinos=knn.vecinos()
    tasa_clas=(numero_componentes_iguales(Y_vecinos,Y))/tam_muestra
    tasa_red=numero_de_ceros(w)/n_características
    tiempo2=time()
return tasa_clas,tasa_red,tiempo2-tiempo1,w
```

Aquí si uso la métrica con los pesos para comprobar cómo de bueno es el w que hemos obtenido con el método RELIEF.

Algorithm 3 1NN

```
function 1NN(X,Y)
    tiempo1=time()
    KNN=ClasificadorKNN(N_vecinos=1,metrica=euclidea, X,Y)
    Y_vecinos=KNN.vecinos()
    KNN=ClasificadorKNN(n_vec=1,metrica=Dist,args=w,X,Y)
    Y_vecinos=knn.vecinos()
    tasa_clas=(suma_componentes_iguales(Y_vecinos,Y))/tamaño_muestra
    tasa_red=numero_de_ceros(w)/n_características
    tiempo2=time()
return tasa_clas,tasa_red,tiempo2-tiempo1,w
```

Algorithm 4 Iterative Local Search

```
function ILS(x_train, y_train, x_test, y_test, alpha, sigma)
    pred = alpha
    pclas = 1 - alpha
    w = np.random.uniform(0, 1, x_train.shape[1])
    tinicial = time()
    KNN = KNN.fit(x_train, y_train)
    w = BL(x_train, y_train, 0.3, alpha, KNN, w)
    for i in range(1, 15) do
        w = mutacion(w, sigma)
        w = BL(x_train, y_train, 0.3, alpha, KNN, w)
    tfinal = time() return w
```

Algorithm 5 Búsqueda Local

```
function BL(X_train, Y_train, sigma, alpha, KNN, w_ini)
    indices = lista(de 0 a n_caracteristicas - 1, de 1 en 1)
    indexes = list(indices)
    w = w_ini
    w[w < 0.2] = 0
    puntuacion_padre = Valoracion(w
    op = random.normal(loc=0, scale=sigma, size=n_caracteristicas)
    for i in range(1, 1000) do
        index = indexes.pop()
        w_ant = w[index]
        w[index] = w[index] - op[index]
        vecinos_generados += 1
        total_ant = total_red
        if w[index] < 0.2 then
            w[index] = 0
        puntuacion_hijo = Valoracion(w)
        if puntuacion_hijo > puntuacion_padre then
            puntuacion_padre = puntuacion_hijo
            vecinos_generados = 0
        else
            w[index] = w_ant
            total_red = total_ant
        if not indexes then
            op = random.normal(loc=0, scale=sigma, size=n_caracteristicas)
            indexes = list(indices)
            total_red = n_caracteristicas - np.count_nonzero(w)
        if vecinos_generados == 20 * n_caracteristicas then
            break
    return w
```

Algorithm 6 Enfriamiento Simulado

```
function SA(X_train, y_train, alpha, nu, fi, Tfinal, sigma, pmaxaciertos, M)
    KNN=KNN.fit(X_train,y_train)
    w=random.uniform(high=1,low=0,size=n_caracteristicas)
    punt=Valoracion(w)
    indexes=lista(desde=0,hasta=n_caracteristicas-1,de 1 en 1)
    op=np.random.normal(loc=0, scale=sigma, size=n_caracteristicas)        ▷ Vector para llamar solo 1 vez al generador de números aleatorios
    aciertos=0
    tot=1
    T=(nu * punt)/-log(fi)                                                 ▷ Temperatura inicial
     B = (T - Tfinal) / ((M/max_vecinos) * T * Tfinal)
    tiempo1=time()
    while T>=Tfinal do
        for i in range (0,max_vecinos) do
            if indexes.empty() then
                indexes=lista(desde=0,hasta=n_caracteristicas-1,de 1 en 1)
                op=np.random.normal(loc=0,scale=sigma,size=n_caracteristicas)
            index=indexes.pop()
            w_ant = w[index]
            w[index]=w[index]+op[index]
            if w[index]<0.2 then w[index]=0
            if w[index]>1 then w[index]=1
            puntuacion=Valoracion(w)
            tot+=1                                                         ▷ total de evaluaciones
            df = best_punt-puntuacion
            if df<0 or random(0,1)<=exp(-df/T) then
                aciertos+=1
                if puntuacion>best_punt then
                    best_punt=puntuacion
                    best_w[index]=w[index]
                if aciertos>=aciertos_maximos thenbreak
            else
                w[index]=w_ant
        if aciertos==0 then break
        T=T/(B*T)
                ▷ Esquema de enfiramiento, también se ha probado con T=0.99*T
    return w
```

Algorithm 7 Evolución diferencial

```
function DE(x_train, y_train, x_test, y_test, CR, F)
    KNN=KNN.fit(X_train, y_train)
    w=random.uniform(low=0, high=1, size=n_caracteristicas)
    P=GenerarPoblacionInicial(Tamaño=50)
    EvaluacionesFOBJ=50
    ListaVectores=[]
    for i=0;i<15000;i++ do
        for actualParent in P do
            parents=PickUpParents(n_parents, P)
                        ▷ En el caso de rand/1 n_parents=3, en actual-to-best =2
            cromosoma_calculado=vector(size=w.size)
            for actualGene in actualParent.w;cont++ do
                if random(entre 0 y 1)<CR then
                    aux = MutacionDiferencial(Parents,F,actualGene.index)
                     ▷ En el caso de actual-to-best se le pasa también actualGene
                else
                    aux=actualGene
                if aux>1 then
                    aux=1
                else if aux<0.2 then
                    aux=0
                cromosoma[cont]=aux
            ListaVectores.push(cromosoma)
        P_sig,total_evals=valorarW(ListaVectores)
        TotalEvaluaciones+=total_evals
        ListaVectores.clear()
        for i,j in P,P_sig do
            if i.punt>=j.punt then
                P_intermedia.push(i)
            else
                P_intermedia.push(j)
        best,worst=buscarMejorPeor(P_intermedia)
        P_intermedia.pop(worst)
        P_intermedia.push(best)
        P=P_intermedia
        if totalEvaluaciones>=15000 then
            break
return best
```

Algorithm 8 Mutación Diferencial

```
1: function MutacionDiferencial(Parents,F,gen)
2:     if rand1 then
3:         ret=parents[0].w[gen]+F*(parents[1].w[gen] - parents[2].w[gen])
4:     else
5:         ret=parentGen+(F*(best.w[gen]-parentGen))+(F*(parents[0].w[gen]-
   parents[1].w[gen]))
       return ret
```

4 Procedimiento para realizar la práctica

Para realizar la practica, he usado el lenguaje de programación Python usando las siguientes librerías:

- **Numpy**: Librería para agregar mejor soporte para vectores y matrices
- **Scikit-learn**: Librera de machine learning para Python
- **StratifiedKfold**: Función para realizar K-fold cross validation (en nuestro caso K=5)
- **KneighbrosClassifier**: Clasificador KNN. Buscando optimizarlo, cambie la metrica diseñada en la practica anterior por una metrica ya programada en sklearn que es wminkowski. Básicamente es igual que la métrica minkowski pero añadiendo un vector de pesos que se tiene en cuenta en el cálculo de la distancia. Para hacer que la minkowski se comporte como la metrica euclidea he colocado el parametro p=2 (dejo tanto la pregunta que hice en stackoverflow como la referencia a la documentación de la sklearn en el apartado de la bibliografía).
- **Train_test_split**: Función para separar en entrenamiento y test el conjunto de datos.
- **MinMaxScaler**: Para realizar el normalizado del conjunto de datos.
- **Time**: Librería para medir tiempos.

A la hora de realizar la practica he usado Anaconda (una distribución de Python para instalar mas fácilmente librerías de terceros) y como IDE he usado Spyder.
La práctica la he realizado usando el sistema operativo Ubuntu 16.04.
Para instalar Anaconda y Spyder (vienen en el mismo script), lo he descargado de su página web.
Una vez instalado, para instalar scikit-learn, en un terminal he ejecutado conda install scikit-learn.
Hecho esto, ya estaría toda la librería de scikit-learn instalada.
Finalmente, como he mencionado anteriormente, he usado el clasificador KNN que viene instalado y la función para la división de los datos.
Estas funciones están implementadas en el script Practica3.py
Para el uso de las mismas, tenemos que hacer un import del mismo fichero al script que vamos a usar.
Para el generador de números aleatorios, he usado el proporcionado en la librería de numpy, fijando la semilla en el valor 1.
En la carpeta de Fuentes están tanto el archivo Practica3.py como cada script para cada conjunto de datos usando cada tipo de algoritmo.
Los ficheros de datos son los mismos que los de la plataforma.

5 Experimentos y análisis

Los casos que hemos usado para la practica han sido:

- **Ozone**: Conjunto de datos para la detección del nivel de ozono. 73 características, 2 clases y 320 muestras
- **Parkinsons**: Conjunto de datos orientado a distinguir entre la presencia y la ausencia de la enfermedad del Parkinson. 22 características, 2 clases y 195 muestras.
- **Spectf-heart**: Conjunto de datos para la detección de enfermedades cardíacas. 44 características, 2 clases y 267 ejemplos.

Los parámetros que hemos usado han sido:

- **Alfa**: Parámetro para la función objetivo que le da una mayor importancia a clasificar bien o a reducir al máximo. En nuestro caso, alfa es igual a 0.5 (misma importancia para ambos).
- **Sigma**: Para metro para el operador de vecino. Este se generara mediante una distribución normal de media 0 y desviación sigma. En nuestro caso, sigma es igual a 0.3
- **La semilla** para la generación de números aleatorios se ha fijado a 1.

Parámetros de Enfriamiento simulado:

- **M**:Número de iteraciones para el enfriamiento simulado. Se ha fijado a 15000
- **fi y nu**:Fi es igual a la probabilidad de aceptar una solución un nu por ciento peor que la inicial. Fi y nu se han fijado a 0.3.
- **Tfinal**:Temperatura final. Fijada a 10e-3.
- **sigma**:Parámetro para ver cual va a ser el radio del operador $Mov(W, \sigma)$. Se ha fijado a 0.3.
- **pmaxaciertos**:Porcentaje del número máximo de vecinos. Indica el máximo de soluciones aceptadas (ya sean por metrópolis o porque haya se haya logrado una mejor valoración) descendientes de un mismo padre. Se ha fijado a 0.1.
- **max_vecinos**: Numero de vecinos máximos que se van a generar de un único padre. Se ha fijado al 10% del número de características

Parámetros de Evolución diferencial

- **F**:Constante que acompaña a la mutación diferencial. Fijado a 0.5.
- **CR**:Probabilidad de cruce. Fijado a 0.5.

Parámetros de ILS:

- **Numero iteraciones BL**:Fijado a 1000.

- **Iteraciones de la ILS**:Fijado a 15.
- **Porcentaje de mutación del vector**: Porcentaje que se va a mutar del vector una vez que termine la BL. Fijado a 0.1.
- **sigma**:Parámetro para ver cual va a ser la varianza a la hora de generar los aleatorios usando una normal(0,sigma) Se ha fijado a 0.4 para la mutación del vector una vez terminada la BL y a 0.3 para el operador $Mov(W, \sigma)$.

Estos parámetros se han considerado iguales para todas las particiones.

5.1 Resultado de las ejecuciones

GREEDY	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	0,51	0,00	0,26	0,63	0,25	0,00	0,13	0,15	0,73	0,00	0,37	0,79
Partición 2	0,50	0,00	0,25	0,62	0,74	0,00	0,37	0,20	0,73	0,00	0,37	0,75
Partición 3	0,49	0,00	0,25	0,61	0,74	0,00	0,37	0,15	0,73	0,00	0,36	0,74
Partición 4	0,48	0,00	0,24	0,62	0,74	0,00	0,37	0,16	0,73	0,00	0,36	0,77
Partición 5	0,50	0,00	0,25	0,63	0,75	0,00	0,38	0,15	0,73	0,00	0,36	0,75
Media	0,50	0,00	0,25	0,62	0,64	0,00	0,32	0,16	0,73	0,00	0,36	0,76

1NN	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	0,77	0,00	0,39	0,01	0,91	0,00	0,46	0,002	0,80	0,00	0,40	0,01
Partición 2	0,68	0,00	0,34	0,01	0,91	0,00	0,46	0,002	0,78	0,00	0,39	0,01
Partición 3	0,73	0,00	0,37	0,02	0,91	0,00	0,46	0,002	0,79	0,00	0,40	0,02
Partición 4	0,71	0,00	0,36	0,01	0,91	0,00	0,46	0,002	0,77	0,00	0,39	0,10
Partición 5	0,76	0,00	0,38	0,02	0,90	0,00	0,45	0,002	0,75	0,00	0,38	0,50
Media	0,73	0,00	0,37	0,01	0,91	0,00	0,45	0,002	0,78	0,00	0,39	0,13

DE/rand/1	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	0,80	0,93	0,87	410	0,90	0,90	0,90	71	0,81	0,93	0,87	343
Partición 2	0,68	0,94	0,81	405	0,90	0,91	0,91	74	0,76	0,93	0,85	340
Partición 3	0,75	0,94	0,85	414	0,92	0,91	0,92	72	0,84	0,90	0,87	349
Partición 4	0,76	0,93	0,85	414	0,92	0,91	0,92	72	0,85	0,93	0,89	338
Partición 5	0,75	0,92	0,84	414	0,88	0,91	0,90	73	0,83	0,91	0,87	343
Media	0,75	0,93	0,84	411	0,90	0,91	0,91	72	0,82	0,92	0,87	343

DE-Current-to-best	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	0,76	0,74	0,75	518	0,82	0,91	0,87	85	0,83	0,77	0,80	417
Partición 2	0,73	0,76	0,74	518	0,94	0,86	0,90	86	0,77	0,86	0,82	428
Partición 3	0,81	0,80	0,81	520	0,86	0,86	0,86	85	0,74	0,90	0,82	428
Partición 4	0,70	0,83	0,77	530	0,88	0,86	0,87	86	0,78	0,86	0,82	429
Partición 5	0,73	0,72	0,72	521	0,90	0,82	0,86	85	0,75	0,77	0,76	433
Media	0,74	0,77	0,76	521	0,88	0,86	0,87	85	0,77	0,83	0,80	427

ILS	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	0,78	0,80	0,79	571	0,93	0,86	0,90	35	0,86	0,81	0,84	321
Partición 2	0,70	0,76	0,73	563	0,95	0,77	0,86	44	0,78	0,75	0,77	325
Partición 3	0,70	0,75	0,73	577	0,95	0,81	0,88	37	0,79	0,90	0,85	344
Partición 4	0,75	0,75	0,75	579	0,91	0,72	0,82	37	0,81	0,81	0,81	338
Partición 5	0,73	0,73	0,73	570	0,91	0,86	0,89	34	0,80	0,84	0,82	344
Media	0,73	0,76	0,74	572	0,93	0,80	0,87	37	0,81	0,82	0,82	334

ES-Cauchy	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	0,75	0,32	0,54	3	0,93	0,60	0,77	0,60	0,81	0,27	0,54	2
Partición 2	0,74	0,30	0,52	3	0,93	0,72	0,83	0,70	0,85	0,32	0,59	3
Partición 3	0,72	0,40	0,56	3	0,91	0,50	0,71	0,70	0,80	0,36	0,58	3
Partición 4	0,77	0,32	0,55	3	0,80	0,54	0,67	1,00	0,86	0,43	0,65	2
Partición 5	0,74	0,31	0,53	4	0,91	0,72	0,82	0,65	0,81	0,40	0,61	2
Media	0,74	0,33	0,54	3	0,90	0,62	0,76	0,73	0,83	0,36	0,59	2

ES-Proporcional	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	0,80	0,50	0,65	11	0,89	0,81	0,85	3	0,76	0,57	0,66	10
Partición 2	0,72	0,38	0,55	13	0,92	0,72	0,82	6	0,88	0,45	0,67	41
Partición 3	0,72	0,45	0,59	17	0,98	0,72	0,85	3	0,82	0,50	0,66	25
Partición 4	0,81	0,41	0,61	10	0,93	0,77	0,85	3	0,83	0,47	0,65	10
Partición 5	0,73	0,36	0,55	11	0,91	0,68	0,80	4	0,84	0,55	0,70	14
Media	0,76	0,42	0,59	12	0,93	0,74	0,83	4	0,83	0,51	0,67	20

Tabla 5.2: Resultados globales en el problema del APC

	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
1-NN	0,73	0,00	0,37	0,01	0,91	0,00	0,45	0,002	0,78	0,00	0,39	0,13
RELIEF	0,50	0,00	0,25	0,62	0,64	0,00	0,32	0,16	0,73	0,00	0,36	0,76
DE/rand/1	0,75	0,93	0,84	411,40	0,90	0,91	0,91	72,40	0,82	0,92	0,87	342,60
DE-Current-to-best	0,74	0,77	0,76	521,40	0,88	0,86	0,87	85,40	0,77	0,83	0,80	427,00
ILS	0,73	0,76	0,74	572,00	0,93	0,80	0,87	37,40	0,81	0,82	0,82	334,40
ES-Cauchy	0,74	0,33	0,54	3,26	0,90	0,62	0,76	0,73	0,83	0,36	0,59	2,45
ES-Proporcional	0,76	0,42	0,59	12,40	0,93	0,74	0,83	3,84	0,83	0,51	0,67	20,00

5.2 Análisis de datos

Voy a comenzar comentando unos cambios que he realizado. He usado una función de normalizado que viene en la librería debido a que así el código queda más limpio, es un poco más rápido y porque podemos normalizar el test con los mismos valores (máximo y mínimo) del conjunto de entrenamiento. Esto provoca que hagamos los mismos cambios en test que en entrenamiento y así podremos ver de una forma más fiable como funciona el clasificador. Estos cambios los he realizado tanto en el algoritmo RELIEF como en el 1NN y he actualizado los resultados en la tabla.
Ahora vamos a comentar un poco los algoritmos de comparación. Si nos fijamos en lo resultados, vemos que el tiempo de ejecución de ambos algoritmos es el más pequeño. Esto es obvio ya que 1-NN es simplemente la inicialización del clasificador (esto se va a repetir para cada uno de los algoritmos) y Greedy que da tantas iteraciones como numero de muestras haya en CADA partición y no tenemos bucles añadidos aparte de los que tenga internamente el clasificador implementados. Respecto a clasificación, podemos ver que en media, los resultados de Greedy son los peores. Esto puede venir provocado por la modificación de valores del Greedy. El incremento/decremento de los valores se hace componente a componente sin tener en cuenta el resto de componentes, y si se "equivoca" no tiene forma de solventarlo, no como el resto de algoritmos, y ese error lo arrastrará hasta el final.

Vamos a comenzar comentado los resultados del Simulated Annealing. Tenemos dos tablas para este algoritmo: usando el esquema de enfriamiento Cauchy modificado y el proporcional (usando $\alpha = 0.99$). Vemos como, en media, SA con el esquema proporcional se ha comportado bastante mejor que el esquema Cauchy modificado. Tras la experimentación (y si nos fijamos en las tablas de tiempo) se detectaba como Cauchy converge demasiado rápido, por lo que no tiene tiempo para explorar y explota demasiado la solución mientras que el proporcional ha disminuido más lentamente la temperatura, ha explorado más el vecindario y por eso ha obtenido mejores resultados. Comentando brevemente los tiempos de ejecución, vemos que son sorprendente-mente bajos y eso nos puede dar pié a aumentar la temperatura inicial y así intentar buscar mejores resultados.

Con respecto a la ILS vemos que obtenemos también unos resultados muy buenos. La búsqueda local implementada en la primera práctica funcionaba muy bien y es de esperar que ILS se comportara al menos como la BL. Podemos ver que hay bastante variación dependiendo de las particiones de datos del tiempo de ejecución o resultados de clasificación y de reducción. Esto viene provocado porque en algunas de estas particiones ha encontrado un óptimo local antes que en otras y por ello han tenido menos tiempo de ejecución. Se puede ver claramente en las particiones 2 y 3 del dataset *Spectf-heart* que se han obtenido unos resultados peores que en el resto de particiones, aunque en este caso los tiempos han salido parecidos (lo más probable sea que alguna búsqueda local de las 15 haya tenido un mayor tiempo de ejecución que el resto).

Finalmente vamos a comentar los resultados de Evolución Diferencial.
Los resultados obtenidos con este algoritmo han sido los mejores (y con diferencia) del resto de algoritmos usados en las prácticas anteriores.
En los resultados claramente se ve que el DE con el operador Rand-1 ha obtenido mejores

resultados que el operador Current-to-best. Si nos fijamos en las tablas completas vemos como DE/Rand-1 parece que convergió al mismo vector de pesos (o al menos a eliminar las mismas características) en todas las particiones de todos los dataset, mientras que en DE/Current-to-best vemos que no fue capaz de converger y ello se traduce en una mayor variación de la tasa de reducción y de la clasificación.
Esto nos indica que, en este problema, este algoritmo se ve más beneficiado por la exploración del vecindario que por la explotación.
Respecto a los tiempos, este ha sido el algoritmo que más ha tardado de los realizados.

Personalmente, me ha sorprendido el funcionamiento del algoritmo SA, ha conseguido unos resultados buenos (no tanto como Evolución Diferencial) en un tiempo bastante bajo. Como he comentado antes, si aumentamos la temperatura inicial, cabe la posibilidad de que obtengamos unos mejores resultados.
Respecto a Evolución Diferencial se han obtenido unos resultados que eran de esperar. Son muy buenos algoritmos para problemas con variables reales y así se ha reflejado en los resultados de sus ejecuciones.
ILS es un algoritmo bastante simple y con unos resultados muy buenos, pero esos resultados van a ser buenos únicamente si tenemos una buena búsqueda local para el problema.

6 Bibliografía

- Numpy
- Scikit-learn/neighbors
- Scikit-learn/KNN
- Scikit-learn/Kfold
- Scikit-learn
- Pregunta 1 en stack-overflow
- Anaconda
- Pregunta 2 en stack-overflow
- Paralización
- Métricas