Memoria práctica 2 de Metaheuristicas:
Problema del Aprendizaje de Pesos en Características (APC)

UNIVERSIDAD DE GRANADA

Antonio Manuel Fresneda Rodríguez
antoniomfr@correo.ugr.es

3-Grupo A3(Miércoles 17:30-19:30)
Los algoritmos seleccionados han sido el de Búsqueda Local, Greedy, Genéticos y Meméticos

Índice

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. Asii, se puede formular como:

Maximizar F(w)=α*tasa_clas(W)+(1-α)*tasa_red(W)

con:

- $tasa-clasificacion = \frac{100 \cdot n^o\ instancias\ bien\ clasificadas\ en\ T}{n^o\ instancias\ en\ T}$
- $tasa-reduccion = \frac{100 \cdot n^o\ valores\ w_i < 0{,}2}{n^o caracteristicas}$

Sujeto a que

- wi = [0, 1], 1 ≤ i ≤ 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 f_i.
- 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.
- α∈[0,1] pondera la importancia entre el acierto y la reducción de la solución encontrada.

Descripción de la aplicación.

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

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

La función objetivo a maximizar va a ser:

- F(w)=α*tasa_clas(W)+(1-α)*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 el genético generacional, la llamada será así:

GN(X_train,Y_train,alpha,sigma,pcruce,pmutacion,poblacion)

Si queremos usar el genetico estacionario, la llamada será así:

GNE(X_train,Y_train,alpha,sigma,pcruce,pmutacion, poblacion)

Si queremos usar el memético, la llamada será así:

MEM(X_train,Y_train,alpha,sigma,pcruce,pmutacion)

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

- folds=StratifiedKFold(particiones=5)
  folds.datos(X,Y)
  Para cada indice_train,indice_test en folds
  Particion_i_X_train=X[indice_train]
  Particion_i_Y_train=Y[indice_train]
  Particion_i_X_test=X[indice_test]
  Particion_i_Y_test=Y[indice_test]
  Llamada_algoritmo_KNN(Particion_i_X_train,……)

Otro operador común que tenemos es la normalizacion de los datos. Para ello hacemos lo siguiente:

- max=BuscarMax(X)
- min=BuscarMin(X)
  Para cada xi en X
  - Xi=(xi-min)/(max-min)

En la parte de funciones comunes a los algoritmos tenemos:

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

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),tasa_clas,tasa_red

Para la representacion de las soluciones, he usado una clase con 4 atributos llamada Datos:

1. Punt: Valor de la función objetivo
2. Clas: Tasa de clasificación con el w correspondiente
3. Red: Tasa de reducción con el w correspondiente
4. 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

CruceBLX(P,X,Y,KNN) #Hace el cruce BLX a una población P.

```
evals=0
b_i=-1
b_val=-1
w_i=-1
w_val=100000
for i =0;hasta numero_cruces;i+=2  : #Tamaño R=tamaño_poblacion*pcruce
        c1,c2=BLX(P[i],P[i+1],X,Y,KNN)
        R.añadir(c1,c2)
        evals+=2
best=mejor_punt(c1,c2)
worst=peor_punt(c1,c2)
R=concatenar(R,P)#Eliminando de P los padres que han cruzado. Tamaño_R=Tamaño_poblacion
return R,best,worst,evals
```

BLX(C1,C2,X,Y,KNN): #Hace el cruce a dos cromosomas de P

```
alpha=0.3
index=randint(numero_caracteristicas)
maxi=max([C1.w[index],C2.w[index]])
mini=min([C1.w[index],C2.w[index]])
I=maxi-mini
r=random(low=(mini-(I*alpha)),high=(maxi+(I*alpha)))
if r<0.2:
        r=0
C1.w[index]=r
C2.w[index]=r
punt1,clas1,red1=Valoracion(X,Y,C1.w,KNN,porcentaje_clas,porcentaje_red)
punt2,clas2,red2=Valoracion(X,Y,C2.w,KNN,porcentaje_clas,porcentaje_red)
C1.actualizar(punt=punt1,clas=clas1,red=red1)
C2.actualizar(punt=punt2,clas=clas2,red=red2)
return C1,C2
```

```
CruceAritmetico(P,X,Y,KNN) #Realiza el cruce aritmetico sobre una población P
    evals=0
    for i =0;hasta numero_cruces;i+=2  :
        f1=randint(0,size(P)-1)
        f2=randint(0,size(P)-1)
        c1=Operador_Aritmetico(P[f1],P[f1+1],X,Y,KNN)
        c2=Operador_Aritmetico(P[f2],P[f2+1],X,Y,KNN)
        R.añadir(c1,c2)
        evals+=2
    best=mejor_punt(c1,c2)
    worst=peor_punt(c1,c2)
    R=concatenar(R,P)#Eliminando de P los padres que han cruzado. Tamaño_R=Tamaño_poblacion
    return R,best,worst,evals

Operador_Aritmético(C1,C2,X,Y,KNN):  #Realiza el cruce aritmético a dos cromosomas de P
    r=suma_elemento_elemento(C1.w,C2.w)
    r=dividir_elemento_a_elemento(r,2)
    Hijo.w=menores_de-0.2_iguales-a-0(r)
    punt1,clas1,red1=Valoracion(X,Y,Hijo.w,KNN)
    Hijo.actualizar(punt=punt1,clas=clas1,red=red1)
    return Hijo


def TorneoBinario(P,ncruces):
    winners=[]
    for i in range(0,ncruces):
        i1=randint(0,size(P)-1)
        i2=randint(entre 0 y size(P)-1)
        if i1==i2:
            i2=randint(entre 0 y size(P)-1)
        w1=P[i1]
        w2=P[i2]
        if w1.punt>w2.punt:
            winners.push(w1)
        else:
            winners.push(w2)
    return winners
```

```
def Generar_pobInicial(X,Y,KNN,tam_poblacion):
n_llamadas_objetivo=0
P=[]
        for i in range (0,tam_poblacion):
                w=vectorAleatorio(entre 0 y 1, tamaño=numero_características)
                w=menores_de-0.2_iguales-a-0()
                p,c,r=Valoracion(X,Y,w,KNN)
                n_llamadas_objetivo+=1
                aux=Datos(w=w,clasificacion=c,reduccion=r,puntuacion=puntuacion)
                P.append(aux)
        best=mejor_descendiente(P)
        worst=peor_descendiente®

return P,best,worst,n_llamadas_objetivo

def mutacion(P,X,Y,nmutaciones):

        cromosoma=P[randint(low=0,high=size(P)-1)]
        gen=randint(low=0,high=n_caracteristicas)
        cromosoma.w[gen]=(cromosoma.w[gen]+np.random.normal(loc=0,scale=sigma))
        if cromosoma.w[gen]<0.2:
                cromosoma.w[gen]=0
        elif cromosoma.w[gen]>1:
                cromosoma.w[gen]=1


        val=Valoracion(X_train,Y_train,cromosoma.w,KNN,porcentaje_clas,porcentaje_red)
        total_evaluaciones+=1
        P[cromosoma].actualizar(punt=val[0],clas=val[1],red=val[2])
```

El numero de mutaciones esta para saber cuantas mutaciones se van a hacer. En el código no esta implementado como tal.

Aprovechando la aleatoriedad de los torneos y a que la población no esta nunca ordenada, me ahorro la llamada a la generación de un numero aleatorio para elegir el cromosoma, pero la llamada para ver si hay una mutación o no si la hago.

Descripción del pseudo-codigo

GN(X_train,Y_train,alpha,sigma,pcruce,pmutacion,tam_poblacion) :

```
porcentaje_clas=alpha*100
porcentaje_red=(1-alpha)*100
tiempo1=time()
w=vectorAleatorio(entre 0 y 1, tamaño=numero_características)
w=menores_de-0.2_iguales-a-0()
KNN = KNN( metric='wminkowski',p=2, parametros={'w': w})
KNN.fit(X_train,Y_train)
#Generar poblacion inicial
P,best,worst,evals=Generar_pobInicial(X_train,Y_train,KNN,tam_poblacion):
numero_llamadas_func_obj=evals
for i in range(0,15000):
        P_sig,best_d,worst_d,evals=Cruce(TorneoBinario(P,n_torneos),X_train,Y_train,KNN)[1]
         numero_llamadas_func_obj+=evals
         if best_d.punt>best.punt:
                 best=best_d
         if worst_d.punt<worst.punt:
                 worst=worst_d
         evals=mutacion(P,X_train,Y_train,pmutacion)
         numero_llamadas_func_obj+=evals
         P.pop(worst)
         P.append(best)          #Elitismo, elimino el peor y lo sustituyo por el mejor
         if(numero_llamadas_func_obj==15000):
                 break
```

[1]n_torneos es el parámetro que pasamos a torneo binario para saber el numero de torneos que vamos a lanzar. En el generacional es 30, y en el estacionario es 2 (en el generacional solo cruzamos dos padres).

MEM(X_train,Y_train,alpha,sigma,pcruce,pmutacion,tam_poblacion,BL,iters_BL) :

```
porcentaje_clas=alpha*100[1]
porcentaje_red=(1-alpha)*100
tiempo1=time()
w=vectorAleatorio(entre 0 y 1, tamaño=numero_características)
w=menores_de-0.2_iguales-a-0()
KNN = KNN( metric='wminkowski',p=2, parametros={'w': w})
KNN.fit(X_train,Y_train)
#Generar poblacion inicial
P,best,worst,evals=Generar_pobInicial(X_train,Y_train,KNN,tam_poblacion):
numero_llamadas_func_obj=evals
for n_iters in range(0,15000):
        P_sig,best_d,worst_d,evals=Cruce(TorneoBinario(P,n_torneos),X_train,Y_train,KNN)[1]
          numero_llamadas_func_obj+=evals
          if best_d.punt>best.punt:
                  best=best_d
          if worst_d.punt<worst.punt:
                  worst=worst_d
          evals=mutacion(P,X_train,Y_train,pmutacion)
          numero_llamadas_func_obj+=evals
          P.pop(worst)
          P.append(best)          #Elitismo, elimino el peor y lo sustituyo por el mejor
          if n_iters%iters_BL==0:
                  numero_llamadas_func_obj=BL(P,X_train,Y_train,best,KNN)[1]
          if(numero_llamadas_func_obj==15000):
                  break
```

[1] iters_BL es un parámetro que indica cada cuantas iteraciones se llama a la BL.

[2]Dependiendo del memético, este realizará la BL a todos, al mejor o al 10 por ciento.

Descripción pseudo-codigo para los algoritmos de comparación

BL(X,Y,sigma,alfa):

```
tiempo1=time()
indices=Genera_lista_desordenada(desde=0,hasta=n_características-1,tamaño=n_características)
op=Genera_lista_uniforme(media=0,desvicacion=sigma^2,tamaño=n_características)
w=Genera_array_distribución_normal(low=0,high=1,tamaño=n_características)
n_vecinos=0
red=w.contar_ceros()
KNN=ClasificadorKNN(n_vec=1,metrica=Dist,args=w,X,Y)
tasa_clas,red,puntuacion_padre=Valoracion(X_train,Y_train,w,KNN,porcentaje_clas,porcentaje_red)
for i to 15000
        indice=indices.pop()
        w_ant=w
        w[indice]=w[indice]-op[indice]
        n_vecinos++
      tasa_clas,red,puntuacion_hijo=Valoracion(X_train,Y_train,w,KNN,porcentaje_clas,porcentaje_red):
        if puntuación_hijo>puntuación_padre
                puntuación_padre=puntuación_hijo
                n_vecinos=0
        else
                w=w_ant

        if indices.vacia()
                op=Genera_lista_uniforme(0,sigma^2,tamaño=n_características)
                indices=Genera_lista_desordenada(0,n_características-1,tamaño=n_características)
        if n_vecinos==20*n_características
                break
tiempo2=time()
return ,tasa_clas,tasa_red,tiempo2-tiempo1,w
```

OP→Lista con las operaciones que se van a aplicar al vector (para no tener así que llamar al aleatorio dentro del for).

Índices→ Lista con los índices. Estos están generados mezclados de forma pseudo-aleatoria.

Para el operador de mutación, lo que hago en cada iteración saco un indice de la lista (por el final para que sea más optimo) y ese indice lo uso para sacar el valor de W en ese indice y el valor de OP correspondiente.

Para el cálculo del error, he usado el método de leave one out con la muestra de entrenamiento.

Este KNN no tiene esa opción para calcular el error, pero para hacerlo lo que he echo ha sido calcular los vecinos mas cercanos a cada una de la muestra y las etiquetas de estos y compararlas con las etiquetas del train.

RELIEF(X,Y,w)

```
vecinos=vecinos_mas_cercanos(X,Y,metrica=euclidea)
para cada xi,yi en X,Y
        indice_vecinos_etiqueta1=vecinos.etiqueta(1)
        indice_vecinos_etiqueta2=vecinos.etiqueta(2)
        if yi==1
                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)
para cada wi en w
        if wi<0
                wi=0
        else
                wi=wi/w_max
```

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))/tamaño_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.

1NN(X,Y)

```
tiempo1=time()
KNN=ClasificadorKNN(N_vecinos=1,metrica=euclidea, X,Y)
Y_vecinos=KNN.vecinos()
tasa_clas=(suma_componentes_iguales(Y_vecinos,Y))/tamaño_muestra
tasa_red=0
tiempo2=time()
return tasa_cas,tasa_red,tiempo2-tiempo1,0
```

Procedimiento para realizar la práctica

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

1. Numpy: Librería para agregar mejor soporte para vectores y matrices
2. Scikit-learn: Librera de machine learning para Python
    1. StratifiedKfold: Función para realizar K-fold cross validation (en nuestro caso K=5)
    2. 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)
    3. 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 Practica2.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 1997.

En la carpeta de Fuentes están tanto el archivo Practica2.py como cada script para cada conjunto de datos usando cada tipo de algoritmo genético.

Los ficheros de datos son los mismos que los de la plataforma.

Experimentos y análisis

Los casos que hemos usado para la practica han sido:

1. Ozone: Conjunto de datos para la detección del nivel de ozono. 73 características, 2 clases y 320 muestras
2. 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.
3. 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:

1. 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).
2. 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
3. La semilla para la generación de números aleatorios se ha fijado a 1997.
4. pcruce: Probabilidad de cruce. En los genéticos generacionales se ha considerado 0.7 y en los estacionarios se ha considerado 1
5. pmutacion: Probabilidad de mutación. En todas las ejecuciones se ha considerado como 0.001
6. Población: Tamaño de la población. Tanto en los estacionarios como en los generacionales se ha considerado como 30. En los meméticos se ha considerado 10.

Estos parámetros se han considerado iguales para todas las particiones.

Resultado de las ejecuciones.

AGG-BLX	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	0,76	0,32	0,54	875	0,85	0,45	0,65	128	0,86	0,38	0,62	856,00
Partición 2	0,73	0,33	0,53	886	0,80	0,50	0,65	127	0,88	0,36	0,62	930,00
Partición 3	0,73	0,35	0,54	865	0,87	0,45	0,66	146	0,87	0,38	0,63	944,00
Partición 4	0,73	0,36	0,55	1123	0,83	0,50	0,67	165	0,84	0,36	0,60	973,00
Partición 5	0,73	0,35	0,54	1200	0,83	0,45	0,64	178	0,84	0,38	0,61	1053,00
Media	0,74	0,34	0,54	990	0,84	0,47	0,65	149	0,86	0,37	0,62	951,20

AGG-CA	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	0,68	0,37	0,53	810	0,87	0,45	0,66	126	0,85	0,34	0,60	795
Partición 2	0,64	0,33	0,49	835	0,85	0,45	0,65	131	0,83	0,27	0,55	907
Partición 3	0,74	0,36	0,55	936	0,83	0,45	0,64	138	0,82	0,30	0,56	909
Partición 4	0,74	0,38	0,56	1026	0,83	0,45	0,64	174	0,85	0,32	0,59	986
Partición 5	0,68	0,36	0,52	1085	0,84	0,45	0,65	184	0,86	0,25	0,56	1122
Media	0,70	0,36	0,53	938	0,84	0,45	0,65	151	0,84	0,30	0,57	944

AGE-BLX	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	0,67	0,30	0,49	165	0,84	0,36	0,60	49	0,80	0,30	0,55	141
Partición 2	0,65	0,30	0,48	175	0,82	0,36	0,59	49	0,70	0,30	0,50	143
Partición 3	0,73	0,32	0,53	177	0,85	0,36	0,61	50	0,79	0,27	0,53	151
Partición 4	0,75	0,32	0,54	177	0,78	0,36	0,57	50	0,80	0,30	0,55	189
Partición 5	0,65	0,30	0,48	214	0,85	0,45	0,65	71	0,72	0,30	0,51	197
Media	0,69	0,31	0,50	182	0,83	0,38	0,60	54	0,76	0,29	0,53	164

AGE-CA	Ozone				Parkinsons				Spectf-heart			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	0,66	0,30	0,48	162	0,82	0,40	0,61	51	0,75	0,36	0,56	134
Partición 2	0,69	0,27	0,48	168	0,81	0,36	0,59	51	0,76	0,36	0,56	138
Partición 3	0,67	0,29	0,48	170	0,85	0,40	0,63	53	0,73	0,36	0,55	141
Partición 4	0,67	0,30	0,49	185	0,85	0,40	0,63	69	0,77	0,36	0,57	193
Partición 5	0,71	0,29	0,50	214	0,83	0,40	0,62	70	0,70	0,36	0,53	196
Media	0,68	0,29	0,49	180	0,83	0,39	0,61	59	0,74	0,36	0,55	160

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	0,70	0,00	0,35	0,01	0,83	0,00	0,41	0,002	0,84	0,00	0,42	0,13
RELIEF	0,49	0,00	0,25	0,62	0,75	0,00	0,38	0,16	0,72	0,00	0,36	0,76
BL	0,78	0,76	0,77	268,87	0,88	0,77	0,82	11,94	0,90	0,79	0,85	101,55
GENETICOS												
AGG-BLX	0,74	0,34	0,54	989,80	0,84	0,47	0,65	148,80	0,86	0,37	0,62	951,20
AGG-CA	0,70	0,36	0,53	938,40	0,84	0,45	0,65	150,60	0,84	0,30	0,57	943,80
AGE-BLX	0,69	0,31	0,50	181,60	0,83	0,38	0,60	53,80	0,76	0,29	0,53	164,20
AGE-CA	0,68	0,29	0,49	179,80	0,83	0,39	0,61	58,80	0,74	0,36	0,55	160,40

Análisis de datos.

Búsqueda local, 1NN y RELIEF

Como podemos ver en las tablas, las ejecuciones del 1NN y de Greedy son considerablemente más rápidas.

En particular, el 1NN lo único que realiza es introducir los datos en una matriz y calcular las distancias. Esto se realiza tanto en el Greedy y en la BL, por lo que las ejecuciones con este algoritmo van a ser las mejores. En la tabla se puede que mientras los tiempos de ejecución del 1NN, la mayor media en tiempo ha sido 0.01 segundos, mientras que la mayor media en Greedy ha sido 0,76 segundos y en la BL la mayor media ha sido 268 segundos. Los factores que son relevantes para el tiempo son la cantidad de muestras (se puede ver como en los datos que tienen más muestras los tiempos son mayores) y obviamente el número de iteraciones. En la BL, por cada partición, tenemos que dar 15 mil iteraciones como máximo (ninguna ha llegado a las 15 mil), que en comparación con Greedy que da tantas iteraciones como numero de muestras haya en CADA partición y en el 1NN que no tenemos bucles añadidos a los que tenga internamente el clasificador implementados.

Moviéndonos a la clasificación de los datos podemos ver que en media, los resultados entre el 1NN y la búsqueda local son parecidos, mientras que los del Greedy son algo 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 vuelve atrás (como en la BL) y ese error lo arrastrará hasta el final.

En cuanto a la clasificación entre BL y el 1NN podemos ver que la clasificación no varia de una forma realmente significante.

Si queremos hablar de la tasa de reducción, vemos que ambos algoritmos de comparación no proporcionan ningún tipo de reducción, ya que el 1NN por si mismo no implementa ningún tipo de reducción de características y en el Greedy, como el operador se basa en la distancia, podríamos tener pocos ejemplos de una muestra con una característica que puede ser relativamente importante y podríamos eliminarla

Genéticos

Vamos a empezar comentando la diferencia entre los genéticos generacionales, los estacionarios y los dos operadores de cruce planteados.

Vemos que los estacionarios tardan mucho menos que los generacionales, esto se debe a que al solo cruzar dos padres, la cantidad de recursos que se consumen en los torneos y en los cruces es muy baja en comparación con los generacionales. Pero el precio se paga obteniendo resultados un poco mas bajos que los generacionales, ya que es obvio que un estacionario va a tardar mucho más en sustituir una población entera que el generacional y si esta población es mala (aunque tenga elitismo) puede que necesite varias iteraciones para mejorarla mientras que el generacional sustituye una buena parte de esta por la población obtenida de los ganadores del torneo binario. Ambos algoritmos están implementados con elitismo (la mejor solución se guarda durante las generaciones eliminando al peor de la población). Esto mejora mucho el resultado de los mismos, puesto que cabe la posibilidad de que si no lo guardamos, lo perdamos y tengamos una solución peor que la que teníamos.

En cuanto a los datos, vemos que la diferencia de resultados no es muy grande, incluso hay conjuntos de datos donde los resultados de los generacionales son mejores. Pienso que esto ha ocurrido debido a que el cruce del estacionario (ya que la semilla al ser la misma y la forma de iniciar la población inicial es la misma, la población inicial es exactamente igual) solo usa dos padres y dado que el proceso de selección de los padres es aleatorio, puede ocurrir que el algoritmo haya escogido dos padres que al cruzar han dado una solución muy buena, y al haber elitismo, esta solución no se pierde nunca e incluso puede cruzar con otra y obtener otra solución mejor (no tiene por que ser mejor la solución de un cruce del mejor con otro padre de la

población). Lo interesante es que el estacionario solo obtiene resultados casi iguales al generacional (aritmético) solo si usamos el cruce BLX, del que vamos a hablar ahora. Esto nos revela que el generacional ha sido capaz de mitigar los efectos de usar un operador de cruce peor, pero si nos fijamos en el estacionario esta bastante claro que el cruce BLX es mucho mejor que el aritmético. Quizás, si cambiamos la semilla y le damos más tiempo de ejecución a ambos, seguramente veamos como el operador aritmético se queda atrás.

Los operadores de cruce son una parte vital para que el algoritmo funcione realmente bien.

El operador de cruce aritmético (hace la media de todas las componentes del vector entre dos padres) es un operador que se caracteriza por solo explotar, y ademas de ser un operador desde mi punto de vista bastante malo y explico por qué. Imaginemos que tenemos dos padres: C1, C2 y sus correspondientes w: w_1 y w_2. El operador realizaría la media de todos los w_i de w_1 y w_2. Puede ocurrir que el valor de un w_i en ambos vectores sea por ejemplo 0.9 entonces el resultado seria 0.9. Esto provoca que puede ocurrir que si dos vectores son parecidos, el hijo obtenido sea prácticamente el mismo que los dos padres y no habría evolución (en este caso quizás en otro si). Puede darse el caso de que se comporte mejor debido a que la población inicial no sea especialmente mala y explotándola lo que conseguimos es irnos más rápidamente hacia un máximo **local.**

En cuanto al operador BLX (crea un intervalo entre el menor w_i y el mayor w_j de los dos padres, amplia un alfa por ciento ese intervalo (en nuestro caso 0.3) por los dos extremos y selecciona un valor aleatorio en ese intervalo y sustituye una coordenada aleatoria (la misma para los dos w) por ese valor. Si nos ponemos en el caso anterior en el que tenemos dos vectores parecidos, no tenemos el problema de antes, ya que lo que tenemos es un intervalo de la recta real donde tenemos infinitos puntos para escoger.

Esto tiene una gran ventaja frente al aritmético. Es mucho más difícil estancarse en un mínimo local, ya que el algoritmo puede explotar o explorar el vecindario en busca de nuevas soluciones. Como he mencionado anteriormente, el tiempo de ejecución (que es el mismo para los dos) parece no haber sido suficiente para notar diferencias más notables.

Destaco también que el operador aritmético en el conjunto de datos spectf-heart se ha comportado bastante bien en comparación con el BLX en los estacionarios. Seguramente la población inicial era muy buena y al explotarla ha conseguido alcanzar al BLX.

Respecto a la comparación de los genéticos con la búsqueda local, como he dicho anteriormente, los genéticos han tenido poco tiempo de ejecución para realizar buenas soluciones. Esto quizás se arregle cambiando la semilla o incluso cambiando el generador de aleatorios (y obviamente aumentando el numero de iteraciones para los genéticos). La búsqueda local converge rápidamente porque solo tiene una solución y la va mejorando hasta que llegue un momento que se quede en un optimo y no mejore. Parece que la BL se comporta mejor que el genético pero la realidad es muy distinta. La BL se ha estancado en un óptimo del que es probable que no haya salido, ya que ha explotado al máximo la solución inicial, en cambio, si el genético (con un operador que explore el vecindario) llegase a ese punto, seguro que en un par de iteraciones ya ha conseguido sortear ese óptimo y conseguir una solución mucho mejor que la obtenida con la BL, pero como he dicho anteriormente, parece ser que el tiempo de ejecución para los genéticos no ha sido suficiente para alcanzar a la BL.

No he podido realizar las ejecuciones de los meméticos.

Para decidir que algoritmo es mejor deberíamos de tener en cuenta muchos factores, tales como el tiempo que tenemos de ejecución, la calidad respecto a la clasificación y tendríamos que valorar la necesidad de la eliminación de características, puesto que si no conocemos la dependencia entre las características y queremos reducir el número de las mismas no podríamos usar métodos como el 1NN o RELIEF ya que no implementan por defecto ninguna forma de detectar que características son las menos significativas.

Bibliografía

- https://docs.scipy.org/doc/numpy-1.14.0/reference/
- http://scikit-learn.org/stable/modules/classes.html#module-sklearn.neighbors
- http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier
- http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html
- http://scikit-learn.org/stable/modules/neighbors.html
- https://stackoverflow.com/questions/21052509/sklearn-knn-usage-with-a-user-defined-metric
- https://anaconda.org/anaconda/python

  https://stackoverflow.com/questions/50064632/weighted-distance-in-sklearn-knn
- http://academic.bancey.com/parallelization-in-python-example-with-joblib/
- http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html