Metaheurísticas (2018-2019)
Grado en Ingeniería Informática
Universidad de Granada

Práctica 2.b:

Técnicas de Búsqueda basadas en Poblaciones para el Problema del Aprendizaje de Pesos en Característica

UNIVERSIDAD
DE GRANADA

Antonio Jesús Heredia Castillo
DNI:76069518P
a.heredia.castillo@gmail.com
Grupo MH3: Jueves de 17:30h a 19:30h

6 de mayo de 2019

Índice

1. Practica 1.b **3**
1.1. Descripción del problema 3
1.2. Descripción de los algoritmos comunes 3
1.3. Representación de los datos 4
1.4. Ejecución de métodos de búsqueda 5
1.5. Evaluador de soluciones 5
1.6. Descripción de los métodos de búsqueda 6
1.7. Clasificador 1-NN 6
1.8. Relief 7
1.9. Búsqueda local 8
1.9.1. Resumen 10

2. Practica 2.b **10**
2.1. Esquema de representación de soluciones empleado 10
2.2. Función objetivo 11
2.3. Algoritmo de selección en AGs y operadores de cruce y mutación 11
2.3.1. Operador de cruce BLX 12
2.3.2. Operador de cruce aritmetico 12
2.3.3. Operador de mutación 13
2.3.4. Elitismo 13
2.3.5. Algoritmo genéticos generacionales 14
2.4. Experimentos 15
2.4.1. Clasificador 1-knn 15
2.4.2. Relief 15
2.4.3. Búsqueda local 15
2.4.4. AGG-BLX 16
2.4.5. AGG-AC 16
2.4.6. AGE-BLX 16
2.4.7. AGE-AC 16
2.4.8. AM-(10,1.0) 16
2.4.9. AM-(10,0.1) 17
2.4.10. AM-(10,0.1Mejores) 17

Índice de tablas

1. Practica 1.b

1.1. Descripción del problema

En este problema nos encontramos con un conjunto de datos que previamente han sido clasificados. Estos datos tienen asociado un serie de características, cada una con un valor en concreto. Con estos datos y haciendo uso de diferentes algoritmos y caricaturistas queremos crear un modelo que nos permita clasificar de forma automática cualquier valor nuevo que tengamos. En este caso vamos a comprar el resultado de tres diferentes técnicas.

La primera técnica que usamos es usar un clasificador K-NN, utilizando la técnica de buscar el vecino mas cercano. Este método busca simplemente el elemento que esta mas cercano a el y lo clasifica con la misma etiqueta que el.

La siguiente técnica que vamos a usar es el clasificador RELIEF este algoritmo usando amigos mas cercanos y enemigos mas cercano obtiene un vector de pesos que se puede usar para un clasificador 1-NN para modificar la importancia a la hora de obtener la distancia del vecino mas cercano.

La ultima técnica utilizada es una BL, que modifica los valores del vector de pesos usando el operador de vecino por mutación normal, que consiste en coger un atributo y modificarlo por un valor de la normal.

Todos estas técnicas estarán descritas mas adelante.

1.2. Descripción de los algoritmos comunes

Para cargar los datos simplemente utilizo **scipy** [1], que tiene una función para cargar directamente los ficheros arff. Como lo carga en formato Pandas, los normalizo con **preprocessing** de **sklearn** [2] y los paso a array de Numpy.

Para los grupos de datos, he usado un $datos[F, C]$ donde F es la cantidad de datos y C la cantidad de atributos que tiene esa clase de datos:

Atributos / Datos	atributo0	atributo1	...	atributoC
dato0				
dato1				
...				
datoF				

1.3. Representación de los datos

La representación para para las etiquetas es igual de sencilla en este caso se representa de la siguiente forma, $vector[F]$, un vector de tamaño F donde cada etiqueta de una posición corresponde al dato que se encuentra en la misma posición en el array de datos:

dato	etiqueta
dato0	a
dato1	b
...	
datoC	a

Para representar los pesos que obtenemos tanto en RELIEF como en Local Search, usaremos tambien un vector, en este caso sera $pesos[C]$, donde C sera el numero de atributos que tienen nuestros datos:

peso	valor
peso0	0.22335
peso1	0.78335
...	
pesoF	0.5436

Por otro lado también tendremos datos estadísticos para valorar las soluciones. Para cada uno de los siguientes datos también existirá su correspondiente media de todas las ejecuciones. Estos son:

- tiempo inicial: Recogido antes de inicial la computación de los datos.
- tiempo final: Cuando acaba la computación de los datos
- tasa acierto: Porcentaje de aciertos que ha tenido una ejecución.
- tasa reducción: Porcentaje de pesos que son inferior a 0.2

- función objetivo: Función que intentaremos maximizar en cada clasificador.

1.4. Ejecución de métodos de búsqueda

La forma de ejecutar todos los algoritmos la realizo de la misma forma. Realizo un separación de los datos usando la técnica de validación cruzada de 5 iteraciones. Esto consiste en separar los datos de forma uniforme. Para ello estoy usando la biblioteca **StratifiedKFold** de **sklearn**[2]
En el pseudocodigo del algoritmo 1.5 podemos ver como se usa. Lo primero que se realiza es obtener las 5 particiones de datos diferentes. StratifiedKFold nos proporciona **k** grupos de datos. Cada grupo de datos esta formado dos partes una del 80 % y otra del 20 %. Es decir, la parte del test ira variando de un grupo a otro. En el algoritmo se puede ver cuando obtenemos los tiempos, para cada ejecución. También recogemos los datos que usaremos en para realizar las comparaciones.

Algorithm 1 Algoritom para ejecutar los metodos de clasificacion

```
1: tiempoInicio = getTiempo()
2: tiempoAnterior = tiempoInicio
3: particiones = obtenerParticiones(datos, etiquetas,k=5)
4: for particion en particiones do
5:     datosEstadisticos = clasificador(particion)
6:     datosTotales += datosEstadisticos
7:     tiempoActual = getTiempo()
8:     tiempoAnterior = tiempoActual
9:     mostrarDatosPorPantalla(datosEstadisticos,tiempoActual-
   tiempoAnterior)
10: end for
11: tiempoActual = getTiempo()
12: mostrarDatosPorPantalla(datosTotales, tiempoActual-tiempoInicio)
```

1.5. Evaluador de soluciones

Para evaluar nuevos datos usando los pesos he usado un algoritmos que me ha cedido mi compañero de clase Antonio Molner Domenec. En el se hace uso de KDDTree, el cual genera un árbol con los vecinos. Realizando así una búsqueda de cual es el mas cercano.

Algorithm 2 Algoritmo para la evaluación de las soluciones

```
X = datos * pesos
X = eliminaDespreciables(X)
árbol = generoArbol(X)
vecinos = arbol.consulta(nuevoDato)
aciertos = media(clase(vecinos) == claseNuevoDato)
reducción = media(pesos ¡0.2)
return aciertos, reducción, (aciertos+reducion)/2
```

1.6. Descripción de los métodos de búsqueda

1.7. Clasificador 1-NN

Lo primero que hay que explicar en este apartado es el clasificador 1-NN. Ya que es la base. Para este algoritmo tendremos que tener una función que nos devuelva la distancia entre entre dos datos. En nuestro caso usaremos la distancia euclidea. Que viene definida por la siguiente ecuación, donde e_1 y e_2 es un vector con las características de cada elemento:

$$d_e\left(e_1, e_2\right) = \sqrt{\sum_i \left(e_1^i - e_2^i\right)^2}$$

Algorithm 3 Algoritmo 1-NN

```
claseMinima = etiquetas.primera
distanciaMinima = distancia(datos,datos.primero,nuevoDato)
for indice = 0 hasta tamaño(datos) do
    distancia = distancia(datos,datos[indice],nuevoDato)
    if distancia ¡ distanciaMinima) then
        distanciaMinima = distancia
        claseMinima = etiquetas[indice]
    end if
end for
return claseMinima
```

Una vez que tenemos la función de distancia lo único que tenemos que realizar es un simple bucle en el que recorramos todos los datos, y busquemos con cual tiene menor distancia el nuevo dato. Cada vez que encontramos una distancia menor la guardamos y esa sera la clase de nuestro nuevo elemento. Al terminar lo único que hacemos es devolver la clase de ese dato.

Una vez que tenemos el algoritmo que clasifica lo único que tendremos que realizar es bucle que vaya pasando nuevos datos al clasificador y comparar la solución que nos da. Si es igual a la etiqueta real del dato, tendremos un acierto mas. Por ultimo se devuelve la media de los datos y listo.

Algorithm 4 Algoritmo para contar datos bien clasificados con 1-NN

```
1: aciertos = 0
2: for indice, dato en datos_test do
3:     if clasificador(datos,etiquetas,test) == etiquetas_test[indice] then
4:         aciertos++
5:     end if
6: end for
7: return aciertos / tamaño(datos_test)
```

1.8. Relief

El algoritmo 1.8 de Relief tiene un funcionamiento muy sencillo. Lo primero que realizo es crear una matriz de distancia con sckit-learn[2]. Esto lo realizo para evitar tener que calcularla cada vez que evaluó un dato. Para crear el vector de pesos, tenemos que buscar el amigo y el enemigo mas cercano a cada dato. Para evitar realizar un bucle por cada, realizo una comparacion dentro del for que recorre buscando amigos y enemigos. La primera comparación busca que tengan diferente clase, si es asi y ademas tiene una distancia mejor que el mejor enemigo, ese se convierte en el mejor enemigo. La siguiente comparación busca que tengan la misma etiqueta, pero que sea un dato diferente, ademas miramos si la distancia entre ellos y si es menor, ese dato se convierte en el mejor amigo.
Una vez que hemos encontrado el mejor amigo y enemigo, podemos modificar el vector de pesos según aparece en las transparencias, donde e_i es el dato que estamos comparando y e_e y e_a son mejor enemigo y mejor amigo respectivamente:

$$W = W + |e_i - e_e| - |e_i - e_a|$$

Algorithm 5 Algoritmo RELIEF

```
pesos = vector.zeros(tamaño(datos))
matrizDistancias = crearMatrizDistancias(datos, datos)
for indice, dato en datos do
    mejorEnemigo = inf
    mejorAmigo = inf
    enemigoMasCercano = -1
    amigoMasCercano = -1
    for indice2, dato2 en datos do
        if etiquetas[indice] != etiquetas[indice2] then
            if matrizDistancias[indice,indice2] less mejorEnemigo then
                mejorEnemigo = matrizDistancias[indice,indice2]
                enemigoMasCercano = dato2
            end if
        end if
        if etiquetas[indice] == etiquetas[indice2] and  then
            if matrizDistancias[indice,indice2] less mejorAmigo then
                mejorAmigo = matrizDistancias[indice,indice2]
                amigoMasCercano = dato2
            end if
        end if
    end for
    pesos = pesos + abs(dato-enemigoMasCercano) - abs(dato - amigo-
MasCercano)
end for
maxPeso=maximo(pesos)
for indice en pesos do
    if peso[indice] less 0.2  then
        peso[indice] = 0
    else
        peso[indice] = peso[indice] / maxPeso
    end if
end for
return pesos
```

1.9. Búsqueda local

Este algoritmo de búsqueda local se basa en realizar mutaciones aleatorias en los pesos del modelo. Para esto tiene un máximo de iteraciones de 15000 o que no se haya modificado el vector completo por 20 veces. Las mutaciones

la realiza a cada atributo por separado y si existe una mejora, se queda con ese nuevo peso, si no lo mejora vuelve a los pesos anteriores. La mutación se realiza en base a la Normal centrada en 0 y el valor se coge de forma aleatoria. Este algoritmo tarda mucho mas en ejecutarse que los demás ya que tiene que realizar muchas mas evaluaciones de la solución y esto es un proceso costoso.

Algorithm 6 Algoritmo de busqueda local

```
1: pesos = generarPesosAleatorio()
2: mejorEstadisticos = evaluar(pesos,datos,etiquetas)
3: cantCaractarteristicas = tamaño(peso)
4: cantEjecuciones = 0
5: cantExplorados = 0
6: while cantEjecuciones less MAX and cantExplorados less (20*cantCa-
   ractarteristicas) do
7:     for indice hasta cantCaractarteristicas do
8:         cantExplorados += 1
9:         cantEjecuciones += 1
10:        pesosactual = copia(pesos)
11:        Z = random.normal(0.0, pow(0.3,2))
12:        pesosactual[indice] += Z
13:        if pesosactual[indice] great 1  then
14:            pesosactual[y]=1
15:        else if pesosactual[indice]less0 then :
16:            pesosactual[y]=0
17:        end if
18:        if  pesosactual[indice] != W[y] and pesos[indice] great 0.2 then
19:            estadisticos = evaluar(pesos,datos,etiquetas)
20:            if  estadisticos great mejorEstadisticos then
21:                mejorEstadisticos = estadisticos
22:                pesos = copia(pesosactual)
23:                cantExplorados = 0
24:                break
25:            end if
26:        end if
27:    end for
28: end while
29: return pesos
```

Una vez que el algoritmo ha terminado de ejecutar se evaluara el resultado de los pesos generado con los datos de train y se obtendrán los datos

estadísticos que se muestran.

1.9.1. Resumen

Viendo los datos podemos observar que el que mejor se comporta posiblemente sea el de 1-nn ya que obtiene mas tasa de acierto, y eso se puede a la forma en la que se encuentran nuestros datos. Ya que es posible que las clases están muy separadas entre si y con el vecino mas cercano sea suficiente para predecir la clase a la que pertenece.Ademas al ser un algoritmo lineal es muy rapido de realizar.

Por otro lado tenemos Relief y Busqueda local. Relief es cierto que es mucho mas rápido, pero no obstante los resultado de la búsqueda local son mejores. Esto se debe a que solo se muta si ha mejorado el valor de la tasa de agregación, obtenido así mejores resultado. Ademas con la mutación aleatoria, no cae en ningún mínimo local.

2. Practica 2.b

2.1. Esquema de representación de soluciones empleado

Para el esquema de representación para un cromosoma ha sido muy simple, tengo un vector con los diferentes pesos, que serán los diferentes genes

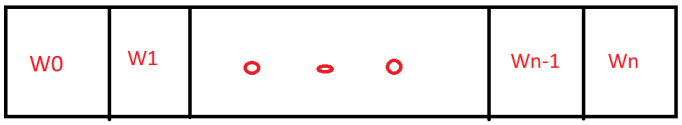

Figura 1:

Por tanto la población sera un conjunto de cromosomas.

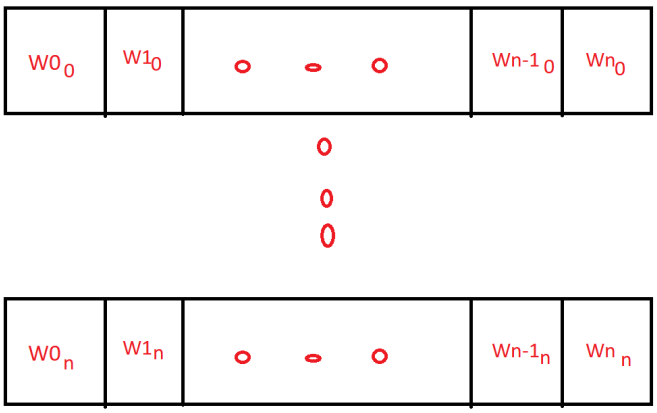

Figura 2:

2.2. Función objetivo

La funcion objetivo en nuestro caso, es una media de dos factores. Primero la tasa de acierto que tiene nuestro clasificador y segundo la tasa de reducción conseguida.

$$Funcionobjetivo = \frac{porcAciertos + porcReduccion}{2} \cdot 100$$

Donde el porcentaje de aciertos es:

$$\frac{Correctos}{TotalDatos}$$

y el porcentaje de reducción es:

$$\frac{pesos < 0{,}2}{TotalDatos}$$

2.3. Algoritmo de selección en AGs y operadores de cruce y mutación

Para el algoritmo de selección se utiliza el conocido "torneo binario ". Este torneo se basa en elegir dos padres de forma aleatoria, evaluarlos, mirar cual es el que mejor función objetivo tiene y añadir ese como padre. En esta practica tendremos dos tipos de torneos binarios uno para el esquema generacional y otro para el esquema estacionario.

Algorithm 7 Torneo binario

```
1: mejores = []
2: mejorValor = 0
3: for indice = 0 hasta tamaño(poblacion) do
4:     primero = poblacion.getAleatorio
5:     segundo = poblacion.getAleatorio
6:     if funcionObjetivo(primero)_a < funcionObjetivo(segundo)_b then
7:         mejorValor = funcionObjetivo(segundo)
8:         mejores.agregar(segundo)
9:     else
10:        mejorValor = funcionObjetivo(primero)
11:        mejores.agregar(primero)
12:    end if
13: end for
14: return mejores
```

2.3.1. Operador de cruce BLX

El operador de cruce BLX, nos recibe dos cromosomas, y devuelve otros dos cromosomas hijo apartir de los padres.

Algorithm 8 Cruce BLX

```
1: h1 = []
2: h2 = []
3: for indice = 0 hasta cantidadGenes do
4:     Cmax = maximo(cromosoma1[indice], cromosoma2[indice])
5:     Cmin = minimo(cromosoma1[indice], cromosoma2[indice])
6:     I = Cmax-Cmin
7:     h1.añadir(random(Cmin-0.3*I,Cmax+0.3*I))
8:     h2.añadir(random(Cmin-0.3*I,Cmax+0.3*I))
9: end for
10: return h1,h2
```

2.3.2. Operador de cruce aritmetico

Este algoritmo tambien devuelve dos cromosomas hijo, aunque estos son iguales entre si.

Algorithm 9 Cruce aritmético

```
c1 = 0.5 * cromosoma1 + 0.5 * cromosoma2
c2 = 0.5 * cromosoma1 + 0.5 * cromosoma2
return c1,c2
```

2.3.3. Operador de mutación

Para la mutación, primero calculo cuantos genes tengo que mutar. Y sabiendo cuantos son necesario, realizo mutaciones a genes de forma aleatoria. El valor de la mutación también es aleatorio, sacado de la normal con $\sigma = 0{,}3$.

Algorithm 10 Mutación

```
for indice = 0 hasta cantidadMutaciones do
    cromosoma = random(Poblacion)
    gen = random(cantGenes)
    poblacion[cromosoma][gen] += normal(0,0.3)
    if poblacion[cromosoma][gen] > 1 then
        poblacion[cromosoma][gen] = 1
    end if
    if poblacion[cromosoma][gen] < 0 then
        poblacion[cromosoma][gen] = 0
    end if
end for
```

2.3.4. Elitismo

El elitismo se basa en que si después de los cruces y las mutaciones, el mejor de la población anterior desaparece, remplazamos el peor de la nueva población por el mejor de la anterior.

Algorithm 11 Elitismo

```
1: encontrado = false
2: peor = null
3: for indice = 0 hasta tamaño(poblacion) do
4:     if poblacion[indice]! = mejor then
5:         encontrado = true
6:     end if
7:     if evaluar(poblacion[indice]) < evaluar(peor) then
8:         peor = poblacion[indice]
9:     end if
10: end for
11: if !encontrado then
12:     poblacion[peor] = mejor
13: end if
```

2.3.5. Algoritmo genéticos generacionales

El esquema de los algoritmo genéticos generacionales es el siguiente. El cruce, lo pongo de forma generica, puede ser tanto BLX como aritmético.

Algorithm 12 Algoritmo Genetico generacional

```
1: poblacion = inicializamosDeFormaAleatoria()
2: mejor = null
3: for indice = 0 hasta cantidadEvalucaiones do
4:     poblacion = torneoBinarioTodos(poblacion)
5:     mejor = mejor(poblacion)
6:     poblacion = cruce(poblacion)
7:     poblacion = torneoBinario(poblacion)
8:     poblacion = remplazoPeorElitimsmo(poblacion,mejor)
9: end for
10: mejor = null
11: for indice = 0 hasta tamaño(poblacion) do
12:     if evalucaion(poblacion[indice]) > evalucaion(mejor)  then
13:         mejor = poblacion[indice]
14:     end if
15: end for
16: return mejor
```

Tabla 5.1: Resultados obtenidos por el algoritmo AGG-BLX en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	71,19	82,26	76,72	21,81	85,92	82,35	84,13	18,26	83,64	82,50	83,07	28,00
Partición 2	75,44	82,26	78,85	21,89	82,86	91,18	87,02	13,17	79,09	87,50	83,30	23,04
Partición 3	73,68	82,26	77,97	25,69	91,43	88,24	89,83	16,07	72,73	82,50	77,61	8,70
Partición 4	73,68	85,48	79,58	19,36	88,57	91,18	89,87	16,69	79,09	85,00	82,05	27,45
Partición 5	71,08	79,03	75,06	27,29	87,14	91,18	89,16	14,15	82,73	85,00	83,86	23,74
Media	73,02	82,26	77,64	23,21	87,18	88,82	88,00	15,67	79,45	84,50	81,98	22,19

Figura 6:

2.4. Experimentos

2.4.1. Clasificador 1-knn

Tabla 5.1: Resultados obtenidos por el algoritmo BL en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	67,80	0,00	33,90	0,11	84,51	0,00	42,25	0,14	94,55	0,00	47,27	0,34
Partición 2	66,67	0,00	33,33	0,11	84,29	0,00	42,14	0,14	93,96	0,00	46,98	0,34
Partición 3	77,19	0,00	38,60	0,10	90,00	0,00	45,00	0,13	91,82	0,00	45,91	0,34
Partición 4	37,68	0,00	18,84	0,11	88,57	0,00	44,29	0,13	93,64	0,00	46,82	0,34
Partición 5	75,44	0,00	37,72	0,10	88,57	0,00	44,29	0,14	91,82	0,00	45,91	0,34
Media	64,96	0,00	32,48	0,10	87,19	0,00	43,59	0,14	93,16	0,00	46,58	0,34

Figura 3:

2.4.2. Relief

Tabla 5.1: Resultados obtenidos por el algoritmo Relief en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	79,66	24,19	51,93	0,22	88,28	2,94	45,61	0,22	82,73	5,00	43,86	0,39
Partición 2	78,95	32,26	55,60	0,10	72,86	2,94	37,90	0,22	81,82	5,00	43,41	0,36
Partición 3	75,44	35,48	55,46	0,10	82,86	2,94	42,90	0,22	87,27	0,00	43,64	0,35
Partición 4	68,42	33,87	51,15	0,10	92,86	2,94	47,90	0,22	85,46	5,00	45,23	0,35
Partición 5	59,65	29,03	44,34	0,10	84,29	2,94	43,61	0,22	83,64	20,00	51,82	0,35
Media	72,42	30,97	51,70	0,12	84,23	2,94	43,58	0,22	84,18	7,00	45,59	0,34

Figura 4:

2.4.3. Búsqueda local

Tabla 5.1: Resultados obtenidos por el algoritmo BL en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	79,66	75,58	77,62	4,01	85,92	85,29	85,60	1,74	82,73	80,00	81,36	7,97
Partición 2	68,42	77,42	72,92	6,28	80,00	85,29	82,65	1,94	81,82	85,00	83,41	8,27
Partición 3	66,67	74,19	70,43	4,25	84,28	88,24	86,26	2,00	81,82	82,50	82,16	4,77
Partición 4	64,91	83,87	74,39	5,76	84,28	88,24	86,26	3,44	80,91	85,00	82,95	4,82
Partición 5	64,91	85,48	75,20	5,03	87,14	82,35	84,75	1,77	77,27	82,50	79,89	3,12
Media	68,91	79,31	74,11	5,07	84,32	85,88	85,10	2,18	80,91	83,00	81,95	0,34

Figura 5:

2.4.4. AGG-BLX

2.4.5. AGG-AC

Tabla 5.1: Resultados obtenidos por el algoritmo AGG-CA en el problema del APC

Colposcopy				Ionosphere				Texture			
% _clas	*% red*	*Agr.*	T	*% _clas*	*% red*	*Agr.*	T	*% _clas*	*% red*	*Agr.*	T
72,88	77,42	75,15	91,30	81,69	82,35	82,02	22,33	80,00	80,00	80,00	47,01
73,68	79,03	76,36	29,61	77,14	85,29	81,22	21,39	77,27	77,50	77,39	50,81
70,18	79,03	74,60	31,21	81,43	85,29	83,36	17,89	76,36	85,00	80,68	43,74
70,18	75,81	72,99	32,77	94,29	82,35	88,32	21,60	85,46	85,00	85,23	43,18
61,40	74,19	67,80	36,69	88,57	82,35	85,46	23,83	84,55	82,50	83,52	38,70
69,66	77,10	73,38	44,32	84,62	83,53	84,08	21,41	80,73	82,00	81,36	44,69

Figura 7:

2.4.6. AGE-BLX

Tabla 5.1: Resultados obtenidos por el algoritmo AGE-BLX en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% _clas	*% red*	*Agr.*	T	*% _clas*	*% red*	*Agr.*	T	*% _clas*	*% red*	*Agr.*	T
Partición 1	71,19	45,16	58,17	49,13	80,28	58,82	69,55	28,10	81,82	50,00	65,91	60,42
Partición 2	77,19	40,32	58,76	48,02	71,43	61,77	66,60	31,75	72,73	50,00	61,36	62,69
Partición 3	77,20	42,94	60,07	48,60	88,57	58,82	73,70	30,21	81,82	55,00	68,41	59,69
Partición 4	68,42	50,00	59,21	40,94	90,00	55,88	72,94	29,97	78,18	52,50	65,34	58,03
Partición 5	63,16	48,40	55,78	49,80	87,14	55,88	71,51	31,96	82,73	55,00	68,86	60,23
Media	71,43	45,36	58,40	47,30	83,49	58,24	70,86	30,40	79,45	52,50	65,98	60,21

Figura 8:

2.4.7. AGE-AC

Tabla 5.1: Resultados obtenidos por el algoritmo AGE-CA en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% _clas	*% red*	*Agr.*	T	*% _clas*	*% red*	*Agr.*	T	*% _clas*	*% red*	*Agr.*	T
Partición 1	77,97	25,81	51,89	51,47	80,28	44,12	62,20	35,23	74,55	32,50	53,52	78,97
Partición 2	73,68	37,10	55,39	49,31	74,29	32,35	53,32	41,71	76,36	42,50	59,43	68,06
Partición 3	70,18	27,42	48,80	53,02	84,29	32,35	58,32	40,38	81,82	73,50	77,66	72,10
Partición 4	80,70	24,20	52,45	59,33	91,43	35,29	63,36	39,57	81,82	30,00	55,91	77,11
Partición 5	59,65	27,42	43,53	58,27	81,43	38,24	59,83	36,75	84,55	35,00	59,77	78,84
Media	72,44	28,39	50,41	54,28	82,34	36,47	59,41	38,73	79,82	42,70	61,26	75,02

Figura 9:

2.4.8. AM-(10,1.0)

Tabla 5.1: Resultados obtenidos por el algoritmo AM-(10,1.0) en el problema del APC

	Colposcopy				Ionosphere				Texture			
	% _clas	*% red*	*Agr.*	T	*% _clas*	*% red*	*Agr.*	T	*% _clas*	*% red*	*Agr.*	T
Partición 1	81,19	80,02	80,60	180,91	92,96	91,18	92,07	77,10	79,09	87,50	83,30	173,23
Partición 2	80,70	85,48	83,09	221,17	84,29	91,18	87,73	79,20	80,91	87,50	84,20	172,16
Partición 3	77,19	90,32	83,76	173,97	87,14	91,18	89,16	75,96	70,91	87,50	79,20	182,85
Partición 4	71,93	87,10	79,51	214,80	78,57	91,18	84,87	82,53	82,73	87,50	85,11	185,23
Partición 5	68,42	90,32	79,37	169,38	95,71	91,18	93,45	11,06	77,27	85,00	81,14	198,57
Media	75,89	86,65	81,27	192,05	87,73	91,18	89,46	65,17	78,18	87,00	82,59	182,41

Figura 10:

2.4.9. AM-(10,0.1)

Tabla 5.1: Resultados obtenidos por el algoritmo AM-(10,0.1) en el problema del APC

Colposcopy				Ionosphere				Texture			
%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
77,97	87,10	82,53	36,10	77,47	91,18	84,32	19,74	82,73	85,00	83,86	41,09
80,70	83,87	82,29	41,97	75,71	88,27	81,99	19,06	78,18	87,50	82,84	38,63
71,93	90,32	81,13	31,53	92,86	91,18	92,02	20,23	75,46	87,50	81,48	41,51
63,16	81,10	72,13	36,46	82,86	91,18	87,02	18,40	86,36	87,50	86,93	39,25
63,16	85,48	74,32	42,92	84,29	91,18	87,73	19,25	85,46	85,00	85,23	42,76
71,38	85,57	78,48	37,79	82,64	90,59	86,61	19,34	81,64	86,50	84,07	40,65

Figura 11:

2.4.10. AM-(10,0.1Mejores)

Tabla 5.1: Resultados obtenidos por el algoritmo AM-(10,0.1mej)en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	72,88	83,87	78,38	40,98	81,69	88,24	84,96	20,46	83,63	85,00	84,32	41,17
Partición 2	80,70	88,71	84,71	32,86	68,57	91,18	79,87	18,70	78,18	87,50	82,84	39,47
Partición 3	77,19	83,87	80,53	40,03	87,14	88,24	87,69	20,76	78,18	87,50	82,84	73,86
Partición 4	66,67	82,26	74,46	41,07	81,43	91,18	86,30	20,07	87,27	82,50	84,89	43,83
Partición 5	66,67	85,48	76,08	44,35	92,86	94,12	93,49	20,54	78,18	90,00	84,09	33,18
Media	72,82	84,84	78,83	39,86	82,34	90,59	86,46	20,11	81,09	86,50	83,79	46,30

Figura 12:

2.4.11. Resultados globales

Tabla 5.2: Resultados globales en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
1-NN	64,96	0,00	33,9	0,1	84,51	0,00	43,59	0,14	94,55	0,00	42,27	0,34
RELIEF	72,41	30,97	51,7	0,12	84,23	2,94	43,58	0,22	84,18	7,00	45,59	0,34
BL	68,91	79,31	74,11	5,07	84,32	85,88	85,10	2,18	80,91	83,00	81,95	0,34
AGG-BLX	73,02	82,26	77,64	23,21	87,18	88,82	88,00	15,64	79,45	84,50	81,98	22,19
AGG-CA	69,66	77,10	73,38	44,32	84,62	83,53	84,08	21,41	80,73	82,00	81,36	44,69
AGE-BLX	71,43	45,36	58,40	47,3	83,49	58,24	70,86	30,4	79,45	52,50	65,98	60,21
AGE-CA	72,44	28,39	50,41	54,28	82,34	36,47	59,41	38,73	79,82	42,70	61,26	75,02
AM-(10,1.0)	75,89	86,65	81,27	192,05	87,73	91,18	89,46	65,14	78,18	87,00	82,59	182,41
AM-(10,0.1)	71,38	85,57	78,48	37,79	82,64	90,59	86,61	19,34	81,64	86,50	84,07	40,65
AM-(10,0.1mej)	72,82	84,84	78,83	39,86	82,34	90,59	86,46	20,11	81,09	86,50	83,79	46,3

Figura 13:

Referencias

[1] SciPy, *Biblioteca open source de herramientas y algoritmos matemáticos para Python*

[2] scikit-learn, *Biblioteca para aprendizaje de máquina de software libre para el lenguaje de programación Python*