

# Práctica 3: Algoritmos genéticos

Antonio Álvarez Caballero 15457968-J  
5º Doble Grado Ingeniería Informática y Matemáticas  
Grupo de prácticas del Viernes 17:30-19:30  
[analca3@correo.ugr.es](mailto:analca3@correo.ugr.es)

8 de mayo de 2016

## Índice

1. Descripción del problema	2
2. Descripción de aplicación de los algoritmos al problema	2
3. Descripción de la estructura del método de búsqueda	2
4. Descripción del algoritmo de comparación	5
5. Desarrollo de la práctica	5
6. Experimentos	7
7. Referencias	9

## 1. Descripción del problema

El problema a resolver es el problema de *Selección de características*. En el ámbito de la *Ciencia de Datos*, la cantidad de datos a evaluar para obtener buenos resultados es excesivamente grande. Esto nos lleva a la siguiente cuestión: ¿Son todos ellos realmente importantes? ¿Podemos establecer dependencias para eliminar los que no nos aportan información relevante? La respuesta es que sí: en muchas ocasiones, no todos los datos son importantes, o no lo son demasiado. Por ello, se intentará filtrar las características relevantes de un conjunto de datos.

La selección de características tiene varias ventajas: se reduce la complejidad del problema, disminuyendo el tiempo de ejecución. También se aumenta la capacidad de generalización puesto que tenemos menos variables que tener en cuenta, además de conseguir resultados más simples y fáciles de entender e interpretar.

Para conseguir este propósito se deben usar técnicas probabilísticas, ya que es un problema *NP-hard*. Una técnica exhaustiva sería totalmente inviable para cualquier caso de búsqueda medianamente grande. Usaremos metaheurísticas para resolver este problema, aunque también podríamos intentar resolverlo utilizando estadísticos (correlación entre características, medidas de separabilidad o basadas en teoría de información o consistencia, etc).

## 2. Descripción de aplicación de los algoritmos al problema

Los elementos comunes de los algoritmos son:

- Representación de las soluciones: Se representan las soluciones como vectores 1-dimensionales binarios (los llamaremos *bits* para poder hacer uso de términos como *darle la vuelta a un bit*):

$$s = (x_1, x_2, \dots, x_{n-1}, x_n); x_i \in \{True, False\} \forall i \in \{1, 2, \dots, n\}$$

- Función objetivo: La función a maximizar es la tasa de clasificación de los datos de entrada:

$$tasa\_clas = 100 \cdot \frac{instancias\ bien\ clasificadas}{instancias\ totales}$$

- Generación de vecino: La función generadora de vecinos es bien simple. Se toma una solución y se le da la vuelta a uno de sus bits, el cual se escoge aleatoriamente.

```
Tomar un vector de características "caracteristica"
indice = generarAleatorio(0, numero_caracteristicas)
caracteristicas[indice] = not caracteristicas[indice]
```

## 3. Descripción de la estructura del método de búsqueda

Veamos los esquemas de cada algoritmo en pseudocódigo:

- Búsqueda Multiarranque Básica:

```
mejor_solucion = [False, False, ..., False]
mejor_tasa = 0
numero_busquedas = 25
```

```
Desde 1 hasta numero_busquedas
```

```

caracteristicas_seleccionadas, tasa = LS(datos_entrenamiento,
                                         etiquetas_entrenamiento, clasificador)

Si tasa > mejor_tasa:
    mejor_tasa = tasa
    mejor_solucion = caracteristicas_seleccionadas

return mejor_solucion, mejor_tasa

```

- GRASP:

```

mejor_solucion = [False, False, ..., False]
mejor_tasa = 0
numero_busquedas = 25

Desde 1 hasta numero_busquedas
    caracteristicas_seleccionadas, tasa = SFSrandom(datos_entrenamiento,
                                                    etiquetas_entrenamiento, clasificador)

    # Lanzamos una Local Search con la solución encontrada
    # con SFSrandom como solución inicial
    caracteristicas_seleccionadas, tasa = LS(datos_entrenamiento,
                                             etiquetas_entrenamiento, clasificador, caracteristicas_seleccionadas)

    Si tasa > mejor_tasa:
        mejor_tasa = tasa
        mejor_solucion = caracteristicas_seleccionadas

return mejor_solucion, mejor_tasa

```

- Iterated Local Search:

```

solucion_inicial = solucionAleatoria()
mejor_solucion = solucion_inicial
num_searchs = 25
mejor_tasa = 0

caracteristicas_seleccionadas, _ = LS(datos_entrenamiento,
                                       etiquetas_entrenamiento, clasificador, solucion_inicial)

Desde 1 hasta numero_busquedas-1
    # Lanzamos una Local Search con la solución
    # anterior mutada como solución inicial
    nuevas_caracteristicas, nueva_tasa = LS(datos_entrenamiento,
                                             etiquetas_entrenamiento, clasificador,
                                             mutacion(caracteristicas_seleccionadas))

    Si nueva_tasa > mejor_tasa:
        mejor_tasa = nueva_tasa
        mejor_solucion = nuevas_caracteristicas

```

```
return mejor_solucion, mejor_tasa
```

Para el GRASP se ha tenido que implementar una versión aleatorizada del SFS, la cual mostramos aquí:

```
caracteristicas_seleccionadas = [False,False,...,False]
mejor_tasa_temporal = 0
peor_tasa_temporal = 0
mejor_caracteristica = 0
mejor_tasa = 0
alpha = 0.3
```

```
Mientras mejor_caracteristica no sea None
    mejor_caracteristica = None
```

```
    caracteristicas_disponibles = Índices de características de
        caracteristicas_seleccionadas que están a False
    tasas = [0,0,...,0]
    caracteristicas_restringidas = ListaVacía
```

```
    # Enumeración devuelve las características con su índice en el vector
    # 0 -> característica 0
    # 1 -> característica 1
    # ...
```

```
    Para indice,caracteristica en enumeración(caracteristicas_disponibles)
```

```
        tasas[indice] = coste al añadir característica a las características seleccionadas
```

```
        Si tasas[indice] > mejor_tasa_temporal
            mejor_tasa_temporal = tasas[indice]
        Si no tasas[indice] < peor_tasa_temporal
            peor_tasa_temporal = tasas[indice]
```

```
    Para indice,caracteristica en enumeración(caracteristicas_disponibles)
        Si tasas[indice] > umbral
            caracteristicas_restringidas.añadir(caracteristica)
```

```
    caracteristica_aleatoria = aleatorio de caracteristicas_restringidas
```

```
    tasa = coste al añadir caracteristica_aleatoria a las características seleccionadas
```

```
    Si tasa > mejor_tasa
        mejor_tasa = tasa
        mejor_caracteristica = caracteristica_aleatoria
```

```
return caracteristicas_seleccionadas, mejor_tasa
```

Para el ILS el operador de mutación es darle la vuelta al 10 % de los bits de la máscara. Se ha implementado así:

```
cambios = EnteroPorArriba(0.1 * longitud(caracteristicas))
mascara = vector aleatorio con "cambios" True y el resto False
caracteristicas_mutadas = XOR(caracteristicas,mascara)
return caracteristicas_mutadas
```

## 4. Descripción del algoritmo de comparación

El algoritmo de comparación es un algoritmo greedy: el *Sequential Forward Selection(SFS)*. La idea es muy simple: se parte del conjunto vacío de características (todos los bits a 0) y se recorren todas las características, evaluando la función de coste. La característica que más mejora ofrezca, se coje. Y se vuelve a empezar. Así hasta que ninguna de las características mejore el coste.

```
caracteristicas_seleccionadas = [False, False, ..., False]
mejor_caracteristica = 0
mejor_tasa = 0

Mientras mejor_caracteristica != -1
    mejor_caracteristica = -1

    caracteristicas_disponibles = índices donde caracteristicas_seleccionadas vale False
    Para cada característica c de caracteristicas_disponibles
        tasa = coste al añadir c a las características seleccionadas
        Si tasa > mejor_tasa
            mejor_tasa = tasa
            mejor_caracteristica = caracteristica
    Si mejor_caracteristica != -1
        caracteristicas_seleccionadas.añadir(mejor_caracteristica)
```

## 5. Desarrollo de la práctica

En primer lugar, comentar que las bases de datos han sido modificadas en su estructura (que no en sus datos) para que sean homogéneas. Así, se han puesto todas las clases como numéricas (en Wdbc no lo estaban) y se han colocado en la última columna.

La práctica se ha desarrollado usando el lenguaje de programación *Python*, ya que su velocidad de desarrollo es bastante alta. Para intentar lidiar con la lentitud que puede suponer usar un lenguaje interpretado, utilizaremos las librerías *NumPy*, *SciPy* y *Scikit-Learn*, que tienen módulos implementados en C (sobre todo *NumPy*) y agilizan bastante los cálculos y el manejo de vectores grandes. Para el KNN con Leave One Out se ha utilizado un módulo que ha desarrollado mi compañero [Alejandro García Montoro](https://github.com/agarciamontoro/metaheuristics)<sup>1</sup>, que usa *CUDA* para agilizar los cálculos usando la GPU.

Usaremos alguna funcionalidad directa de estas bibliotecas:

- *NumPy*: Generación de números aleatorios y operaciones rápidas sobre vectores.
- *SciPy*: Lectura de ficheros ARFF de WEKA.

---

<sup>1</sup><https://github.com/agarciamontoro/metaheuristics>

- *Scikit-Learn*: Particionamiento de los datos, se han usado las particiones estratificadas de la validación cruzada 5x2.
- *ScorerGPU*: Para el KNN con Leave One Out.

Esta elección se ha hecho para poder preocuparme sólo y exclusivamente de la implementación de las metaheurísticas.

Los requisitos para ejecutar mis prácticas son *Python3* (importante que sea la 3), *NumPy*, *SciPy*, *Scikit-Learn* y *CUDA*, por lo que es necesario una gráfica nVidia. En mi plataforma (Archlinux) están disponibles desde su gestor de paquetes.

Una vez instalados los paquetes, sólo hay que ejecutar la práctica diciéndole al programa los algoritmos que queremos ejecutar. La semilla aleatoria está fijada dentro del código como 12345678 para no inducir a errores. Veamos algunos ejemplos de llamadas a la práctica. Primero notamos que los algoritmos disponibles son:

- -SFS: Ejecuta el algoritmo greedy SFS.
- -LS: Ejecuta la Local Search.
- -SA: Ejecuta el Simulated Annealing.
- -TS: Ejecuta la Tabu Search.
- -TSext: Ejecuta la Tabu Search extendida.
- -BMB: Ejecuta la Búsqueda Multiarranque Básica.
- -GRASP: Ejecuta el GRASP.
- -ILS: Ejecuta la Iterated Local Search

```
$ python featureSelection.py -TS
```

Se ejecutará la Tabu Search. Pero no sólo se limita el programa a un algoritmo. Si le pasamos varios, los ejecutará en serie uno detrás de otro. Esto ha cambiado desde la práctica anterior por la entrada de *CUDA*, que hay que iniciarlo debidamente y no es tan sencillo de ejecutar cosas en paralelo.

```
$ python featureSelection.py -BMB -GRASP -ILS
```

Se ejecutarán BMB, GRASP e ILS en serie.

Una vez ejecutado, irán saliendo por pantalla mensajes de este tipo, que proporcionan datos en tiempo real del estado de la ejecución:

```
INFO:__main__:W - TS - Time elapsed: 2265.526112794876.
Score: 98.2394337654. Score out: 95.0877192982 Selected features: 15
```

Este mensaje nos dice todo lo necesario: W es la base de datos (Wdbc), TS el algoritmo, el tiempo transcurrido para esta iteración (recordemos que hay 10), el score de entrenamiento, el score de validación y las características seleccionadas.

## 6. Experimentos

Como se ha comentado antes, la semilla está fija a 12345678 para no tener problemas de aleatoriedad. El número de evaluaciones máxima de todos los algoritmos es de 15000. Por lo demás, todos los demás parámetros propios de cada algoritmo están tal y como se explica en el guión (25 número de búsquedas en BMB e ILS, 10 % de mutación en ILS, etc).

# KNN

	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	96.12676	96.84211	0.0	0.0	66.66667	65.0	0.0	0.0	62.5	66.49485	0.0	0.0
Partición 1-2	96.49123	96.83099	0.0	0.0	65.55556	80.55556	0.0	0.0	61.85567	62.5	0.0	0.0
Partición 2-1	96.12676	96.49123	0.0	0.0	68.88889	74.44444	0.0	0.0	64.0625	64.43299	0.0	0.0
Partición 2-2	95.78947	96.12676	0.0	0.0	75.55556	68.88889	0.0	0.0	61.34021	64.58333	0.0	0.0
Partición 3-1	96.47887	95.4386	0.0	0.0	75.55556	71.66667	0.0	0.0	63.02083	63.91753	0.0	0.0
Partición 3-2	96.84211	96.83099	0.0	0.0	68.88889	65.55556	0.0	0.0	62.37113	64.58333	0.0	0.0
Partición 4-1	97.53521	96.14035	0.0	0.0	66.66667	66.11111	0.0	0.0	65.625	64.43299	0.0	0.0
Partición 4-2	93.33333	97.88732	0.0	0.0	72.77778	72.77778	0.0	0.0	60.30928	64.58333	0.0	0.0
Partición 5-1	96.47887	96.84211	0.0	0.0	75.0	71.11111	0.0	0.0	65.625	65.97938	0.0	0.0
Partición 5-2	97.89474	95.42254	0.0	0.0	70.0	70.0	0.0	0.0	61.85567	63.54167	0.0	0.0
Media	96.30974	96.48530	0.0	0.0	70.55556	70.61111	0.0	0.0	62.85653	64.50494	0.0	0.0

# SFS

	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	97.53521	92.2807	83.33333	0.15793	72.22222	66.66667	93.33333	0.50295	80.20833	70.61856	96.76259	2.41429
Partición 1-2	96.84211	94.01408	86.66667	0.12852	77.22222	66.66667	87.77778	0.91927	73.19588	67.1875	97.48201	1.84876
Partición 2-1	95.42254	91.22807	83.33333	0.15797	84.44444	68.88889	85.55556	1.0978	79.16667	69.58763	97.48201	1.83412
Partición 2-2	97.54386	92.60563	86.66667	0.12911	77.77778	61.66667	93.33333	0.49637	74.2268	64.58333	97.48201	1.88374
Partición 3-1	96.12676	92.98246	90.0	0.09982	83.33333	71.66667	87.77778	0.92519	76.5625	68.5567	98.20144	1.30598
Partición 3-2	97.54386	96.47887	86.66667	0.12927	72.22222	60.0	93.33333	0.50101	71.64948	66.14583	98.20144	1.33008
Partición 4-1	98.23944	96.49123	86.66667	0.12883	70.55556	62.22222	91.11111	0.66223	76.04167	69.58763	97.84173	1.55438
Partición 4-2	94.73684	94.3662	90.0	0.10095	80.55556	65.55556	90.0	0.74934	82.98969	73.95833	97.1223	2.16781
Partición 5-1	94.71831	91.92982	90.0	0.09997	78.88889	66.11111	92.22222	0.57865	77.08333	68.04124	97.48201	1.85395
Partición 5-2	98.94737	93.66197	76.66667	0.21516	76.66667	66.66667	90.0	0.74815	82.98969	71.35417	96.40288	2.75023
Media	96.76563	93.60390	86.00000	0.13475	77.38889	66.61111	90.44444	0.71810	77.41140	68.96209	97.44604	1.89433

# EGA

	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	97.88732	96.84211	46.66667	25.51973	70.55556	69.44444	53.33333	32.1129	66.14584	64.94845	46.40288	137.72354
Partición 1-2	97.54386	95.42254	53.33333	25.33447	70.0	82.22222	54.44444	32.58699	64.94846	64.0625	52.51799	109.34967
Partición 2-1	97.88732	93.68421	40.0	24.82009	68.88889	67.77778	61.11111	31.87575	67.70834	60.82474	51.07914	133.62195
Partición 2-2	97.19299	95.07042	56.66667	24.43126	73.33334	74.44444	50.0	33.51131	65.46392	64.58333	48.56115	109.79654
Partición 3-1	96.47887	96.14035	46.66667	24.29119	72.77778	73.33333	56.66667	32.18128	66.66666	62.8866	52.8777	133.40944
Partición 3-2	98.24561	95.77465	50.0	25.45476	70.55556	70.0	56.66667	33.02999	64.94846	66.66667	52.51799	112.84293
Partición 4-1	97.53521	95.78947	46.66667	24.57323	72.77778	70.55556	52.22222	33.47668	65.625	67.52577	52.15827	135.74265
Partición 4-2	95.08772	94.3662	40.0	24.46908	73.33334	75.0	45.55556	32.8192	67.01031	59.89583	51.07914	110.84738
Partición 5-1	96.83099	95.78947	36.66667	24.56784	74.44444	72.22222	47.77778	33.24316	66.66666	63.91753	55.39568	131.22696
Partición 5-2	97.19299	95.42254	40.0	25.13447	71.66666	79.44444	54.44444	32.43114	65.97938	65.10417	50.71942	108.67263
Media	97.18829	95.43020	45.66667	24.85961	71.83334	73.44444	53.22222	32.72684	66.11630	64.04156	51.33094	122.32337

# GGA

	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	98.59155	96.49123	30.0	28.54975	73.33334	72.77778	44.44444	32.82207	69.27084	62.8866	28.41727	171.97474
Partición 1-2	97.89474	94.71831	43.33333	25.68448	73.88889	82.77778	43.33333	33.94593	69.58763	62.5	31.65468	144.6906
Partición 2-1	99.29578	95.78947	13.33333	31.87766	78.33334	72.77778	38.88889	36.46547	69.27084	63.40206	40.28777	145.52217
Partición 2-2	97.19299	95.42254	70.0	20.04464	75.55556	70.0	56.66667	28.68688	70.1031	62.5	35.97122	128.9489
Partición 3-1	98.23943	96.84211	26.66667	29.04386	78.33334	77.22222	45.55556	33.47387	69.27084	61.34021	33.81295	153.89323
Partición 3-2	96.8421	96.12676	26.66667	29.71426	76.11111	77.77778	40.0	35.69104	72.68041	63.54167	34.53237	136.74538
Partición 4-1	98.23943	96.49123	16.66667	31.48225	74.44444	77.22222	36.66667	36.95578	66.66666	66.49485	29.13669	166.16106
Partición 4-2	97.89474	93.30986	16.66667	31.71025	76.66666	78.88889	43.33333	33.67301	71.13402	65.10417	42.44604	110.06556
Partición 5-1	98.23943	96.14035	16.66667	31.36285	75.55556	68.88889	44.44444	34.03647	71.35416	61.34021	31.65468	168.08526
Partición 5-2	97.54386	96.47887	33.33333	28.12527	77.77778	73.33333	51.11111	31.31469	70.61855	60.9375	36.69065	127.532
Media	97.99740	95.78107	29.33333	28.75953	76.00000	75.16667	44.44444	33.70652	69.99571	63.00473	34.46043	145.36189



## Media

	Wdbc				Movement Libras				Arrhythmia			
	% clas in	% clas out	% red	T	% clas in	% clas out	% red	T	% clas in	% clas out	% red	T
KNN	96.30974	96.48530	0.0	0.0	70.55556	70.61111	0.0	0.0	62.85653	64.50494	0.0	0.0
SFS	96.76563	93.60390	86.00000	0.13475	77.38889	65.61111	90.44444	0.71810	77.41140	68.96209	97.44604	1.89433
EGA	97.18829	95.43020	45.66667	24.85961	71.83334	73.44444	53.22222	32.72684	66.11630	64.04156	51.33094	122.32337
GGA	97.99740	95.78107	29.33333	28.75953	76.00000	75.16667	44.44444	33.70652	69.99571	63.00473	34.46043	145.36189

En primer lugar, comentamos los 2 algoritmos principales: el KNN y el SFS. El primero de ellos obtiene una buena tasa de acierto en la base de datos pequeña, *Wdbc*, y cuanto más grande es la base de datos, más solapamiento se produce y menos tasa de acierto va teniendo.

El SFS se caracteriza por tener una tasa de acierto que sólo mejora en la base de datos más grande a la del KNN, pero tiene una tasa de reducción bastante alta, ya que al partir de una solución sin características y al parar en cuanto no hay mejora, es muy fácil que se quede con muy pocas características.

Ahora evaluamos las tres metaheurísticas implementadas. La BMB funciona bastante bien sobre estas tres bases de datos. En *Wdbc* tiene la segunda mejor tasa de clasificación en el conjunto de test, con una tasa de reducción del 36 %, lo cual no es nada despreciable. En *Movement Libras* es la metaheurística que consigue la mejor tasa de clasificación en el conjunto de test, con una reducción de casi el 50 % de las características. En estas dos bases de datos el tiempo que tarda es muy pequeño, ni siquiera 5 segundos. No es así en el caso de *Arrhythmia*, donde tarda más de un minuto y se queda a mitad de la tabla en cuanto a tasa de acierto, aunque reduce en casi la mitad también el número de características.

Ahora vamos con el GRASP. Es el algoritmo con mejor tasa de acierto dentro de la muestra en las tres bases de datos. Fuera de la muestra se comporta bastante bien, sobre todo en las bases de datos grandes: en *Arrhythmia* consigue una grandísima tasa de acierto de más del 71 % con un 92 % de reducción de las características, tardando además sólo 34 segundos en esta gran base de datos. En las pequeñas también tiene una tasa de reducción bastante notable.

Por último, el ILS. Sólo en *Arrhythmia* se comporta peor que cualquier otro fuera de la muestra, pero en las tres bases de datos es la que menos reduce el número de características.

Posiblemente, el primer algoritmo que probaría de estos 3 en este problema sería un GRASP. Tasa de acierto muy competitiva, tasa de reducción casi insuperable y unos tiempos de ejecución muy razonables.

## 7. Referencias

Las referencias utilizadas han sido:

- *Scikit-Learn*: La propia [documentación](http://scikit-learn.org/stable/modules/classes.html)<sup>2</sup> de la biblioteca.
- *SciPy*: La propia [documentación](http://docs.scipy.org/doc/scipy/reference/)<sup>3</sup> de la biblioteca.

<sup>2</sup><http://scikit-learn.org/stable/modules/classes.html>

<sup>3</sup><http://docs.scipy.org/doc/scipy/reference/>