Práctica 5: Algoritmos híbridos

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

11 de julio de 2016

Índice

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*):

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

- 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

Los tres algoritmos se han implementado con el mismo código. Sólo varía un booleano en la llamada a la función, que indica si es un algoritmo memético, y un real que designa la probabilidad de utilizar la búsqueda local. Si es negativo, no es una probabilidad sino una proporción real.

- Algoritmo genético: Igual al genético de la práctica 3 pero con la componente híbrida. La condición de hibridación se ha modelado de 3 formas. La primera es mejorar todos los cromosomas de la población. La segunda, mejorar un 10 % aleatorio. La tercera, mejorar los 10 % mejores.

```
tamaño_cromosoma = número de características
evaluaciones = 0
max_evaluaciones = 15000

probabilidad_cruce = 0.7
probabilidad_mutacion = 0.001

tamaño_poblacion = 30

Si es generacional, numero_seleccionados = tamaño_poblacion
si no, numero_seleccionados = 2

numero_cruces = EnteroPorArriba(probabilidad_cruce *
                  numero_seleccionados / 2)

# Esta probabilidad es la probabilidad de que un cromosoma mute.
# Esto lo hacemos para evitar trabajar a nivel de gen y generar el
# mínimo número de aleatorios posible, además de homogeneizar
# la implementación de ambas variantes del algoritmo
probabilidad_total_mutacion = probabilidad_mutacion *
        numero_seleccionadas * tamaño_cromosoma

# Inicializar la población y evaluarla
poblacion = Generar "tamaño_poblacion" cromosomas aleatorios
evaluar(poblacion)
# Ordenamos la población por tasa en orden creciente
ordenar(poblacion)

Mientras evaluaciones < max_evaluaciones
  # Cogemos la solución que está en último lugar,
  # ya que la población está ordenada
  mejor_solucion = poblacion[-1]

  # Selección
  Para cada i en [1,2,...,numero_seleccionados]
    eleccion = Coger 2 individuos aleatorios de la población
    ordenar(poblacion)
    mejor = eleccion[-1]
    seleccionados.añadir(mejor)

  # Cruce
  Para cada i en [1,2,...numero_cruces] tomados de dos en dos
    cruce(seleccionados[i], seleccionados[i+1])

  # Mutación
  Si aleatorio() < probabilidad_mutacion_completa
    cromosoma_mutado = aleatorio()
    gen_mutado = aleatorio()

    flip(cromosoma_mutado, gen_mutado)
```

```
    # Actualizar tasas
    evaluar(seleccionados)
    evaluaciones = evaluaciones + numero_seleccionados

    # Componente híbrida
    Para cada i en [1,2,..., numero_seleccionados]
      Si se da la condición de hibridación
        poblacion[i] = BusquedaLocal con solución inicial poblacion[i]

    # Reemplazo
    Para cada i en [1,2,..., numero_seleccionados]
      # Reemplazamos los peores de la población por los mejores seleccionados
      poblacion[i] = seleccionados[-i]
    Si es generacional y mejor_solucion[tasa] > poblacion[0][tasa]
      # Sustituimos la peor solución de la población por la mejor que había
      # Elitismo
      poblacion[0] = mejor_solucion

  ordenar(poblacion)
  mejor_solucion, mejor_tasa = poblacion[-1][cromosoma], poblacion[-1][tasa]
  return mejor_solucion
```

- Búsqueda Local: Se le ha tenido que añadir una componente para almacenar las iteraciones consumidas, ya que al hibridar sólo se puede permitir una.

```
    caracteristicas = generaSolAleatoria()
    fin = Falso
    mejor_tasa = coste(caracteristicas)
    evaluaciones = 0

    Mientras haya_mejora y evaluaciones < MAXIMO_EVALUACIONES
    y iteraciones < maximo_iteraciones
      lista_vecinos = aleatoriza(caracteristicas)
      Para cada caracteristica de lista_vecinos no activa

        flip(caracteristicas, indice_caracteristica)
        tasa = coste(caracteristicas)
        flip(caracteristicas, indice_caracteristica)

        si tasa > mejor_tasa:
          mejor_tasa = tasa
          flip(caracteristicas,indice_caracteristica)
          hay_mejora = Verdadero
    evaluaciones_hechas = evaluaciones
```

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 lenguage 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[1], 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.
- *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:

[1] https://github.com/agarciamontoro/metaheuristics

- -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.
- -EGA: Ejecuta el genético estacionario.
- -GGA: Ejecuta el genético generacional.
- -AM1010: Ejecuta el memético con hibridación total.
- -AM1001: Ejecuta el memético con hibridación del 10 % aleatorio.
- -AM1001mej: Ejecuta el memético con hibridación del 10 % mejor.

```
$ 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 -EGA -GGA
```

Se ejecutarán EGA y GGA 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.

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	0.2367	72.22222	72.77778	93.33333	0.70112	80.20833	68.04124	96.76259	3.34035
Partición 1-2	96.84211	94.01408	86.66667	0.19393	77.22222	67.22222	87.77778	1.28742	73.19588	67.1875	97.48201	2.54349
Partición 2-1	95.42254	91.22807	83.33333	0.23642	84.44444	77.77778	85.55556	1.54551	79.16667	70.10309	97.48201	2.55464
Partición 2-2	97.54386	92.60563	86.66667	0.19242	77.77778	63.88889	93.33333	0.68746	74.2268	66.66667	97.48201	2.58763
Partición 3-1	96.12676	92.98246	90.0	0.149	83.33333	70.55556	87.77778	1.27234	76.5625	68.04124	98.20144	1.80898
Partición 3-2	97.54386	96.47887	86.66667	0.19268	72.22222	65.0	93.33333	0.68529	71.64948	66.66667	98.20144	1.82707
Partición 4-1	98.23944	96.49123	86.66667	0.19197	70.55556	65.0	91.11111	0.91174	76.04167	68.04124	97.84173	2.15922
Partición 4-2	94.73684	94.3662	90.0	0.15017	80.55556	68.33333	90.0	1.03266	82.98969	73.95833	97.1223	2.98434
Partición 5-1	94.71831	91.92982	90.0	0.15063	78.88889	68.88889	92.22222	0.79898	77.08333	68.04124	97.48201	2.57832
Partición 5-2	98.94737	93.66197	76.66667	0.3301	76.66667	68.33333	90.0	1.03194	82.98969	72.39583	96.40288	3.7727
Media	96.76563	93.60390	86.00000	0.20240	77.38889	68.77778	90.44444	0.99545	77.41140	68.91431	97.44604	2.61567

AM1010

	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	98.94366	96.84211	26.66667	59.74466	72.77778	73.33333	32.22222	154.99307	72.91666	63.91753	33.45324	1770.8726
Partición 1-2	97.54386	96.47887	6.66667	58.04732	76.11111	81.11111	45.55556	169.71776	71.13402	68.22917	28.05755	1642.02007
Partición 2-1	96.47887	97.54386	6.66667	52.11255	80.55556	67.77778	41.11111	167.71163	73.4375	63.40206	37.41007	1890.7731
Partición 2-2	98.59649	95.77465	26.66667	60.50266	76.66666	75.55556	51.11111	169.4549	72.68041	65.10417	29.4964	1573.58745
Partición 3-1	97.53521	96.49123	10.0	49.39248	76.66666	69.44444	31.11111	164.08698	73.95834	65.46392	32.3741	1789.77399
Partición 3-2	98.24561	95.42254	20.0	56.38092	79.44444	67.77778	40.0	163.08225	68.5567	65.10417	35.2518	1572.59232
Partición 4-1	97.88732	96.84211	20.0	57.13668	77.22222	70.55556	28.88889	153.94917	70.3125	64.43299	33.09353	1884.23964
Partición 4-2	98.24561	95.07042	23.33333	60.78664	80.55556	65.0	30.0	152.61948	69.58763	63.02083	32.01439	1611.97268
Partición 5-1	98.94366	94.73684	30.0	63.54612	74.44444	72.22222	41.11111	173.82761	71.875	67.52577	26.97842	1790.40634
Partición 5-2	97.89474	97.88732	40.0	64.67783	77.77778	72.77778	34.44444	153.56865	71.64948	66.66667	37.76978	1530.62725
Media	98.03150	96.30900	21.00000	58.23279	77.22222	71.55556	37.55555	162.30115	71.61082	65.28673	32.58993	1705.68654

AM1001

	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	98.94366	96.84211	26.66667	32.95299	73.33334	68.33333	40.0	46.70648	72.91666	64.94845	31.65468	321.38808
Partición 1-2	98.24561	95.07042	36.66667	31.62474	73.88889	80.55556	53.33333	44.39466	68.5567	67.1875	32.73381	284.05299
Partición 2-1	98.59155	94.38596	33.33333	31.6408	76.11111	75.0	52.22222	44.06332	71.875	67.52577	35.97122	333.35355
Partición 2-2	97.89474	95.77465	16.66667	33.87201	81.66666	72.77778	44.44444	44.69038	73.71134	62.5	29.13669	274.26005
Partición 3-1	97.53521	95.4386	16.66667	34.38936	79.44444	74.44444	36.66667	46.74366	73.95834	64.94845	36.69065	319.43827
Partición 3-2	97.54386	96.47887	26.66667	32.89248	77.22222	75.55556	34.44444	46.94561	70.1031	65.625	34.53237	269.39632
Partición 4-1	98.94366	95.08772	40.0	30.49253	78.33334	75.55556	51.11111	46.82215	70.3125	62.37113	35.61151	314.2398
Partición 4-2	97.54386	96.83099	30.0	32.04749	77.77778	73.33333	38.88889	49.37239	71.64948	67.70833	34.53237	263.76303
Partición 5-1	97.1831	96.84211	36.66667	31.81693	79.44444	77.22222	42.22222	46.4231	73.4375	59.79381	34.17266	306.47166
Partición 5-2	98.59649	95.77465	13.33333	35.09213	77.77778	71.11111	36.66667	49.5644	71.13402	61.97917	34.89209	260.94701
Media	98.10217	95.85261	27.66667	32.68215	77.50000	74.38889	43.00000	46.57261	71.76546	64.45876	33.99281	294.73108

AM1001mej

	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	97.19298	30.0	32.70323	73.88889	70.0	42.22222	48.09855	72.39584	64.94845	33.45324	325.10031
Partición 1-2	98.59649	95.77465	26.66667	33.15238	75.0	81.11111	56.66667	43.79019	70.1031	64.0625	37.41007	271.09865
Partición 2-1	98.23943	96.14035	36.66667	31.67881	82.22222	70.55556	33.33333	49.01478	70.83334	63.91753	30.21583	340.51085
Partición 2-2	97.89474	96.47887	33.33333	31.97455	75.0	72.22222	42.22222	47.7423	70.61855	66.14583	37.76978	267.95627
Partición 3-1	97.88732	94.38596	16.66667	34.7548	75.55556	76.11111	50.0	46.07	68.75	64.43299	30.93525	340.91631
Partición 3-2	97.19299	96.47887	26.66667	33.23498	76.66666	73.88889	41.11111	48.76665	74.22681	64.0625	29.13669	306.93667
Partición 4-1	96.47887	94.03509	36.66667	32.26469	75.0	77.22222	34.44444	50.57397	72.91666	63.40206	30.57554	335.57307
Partición 4-2	98.59649	95.42254	23.33333	33.60149	78.33334	71.66667	44.44444	47.35937	70.61855	61.97917	33.45324	283.78787
Partición 5-1	97.88732	95.08772	50.0	29.4009	76.66666	77.77778	50.0	46.63732	72.91666	62.37113	29.4964	325.20679
Partición 5-2	99.29825	95.42254	36.66667	32.49137	76.11111	67.77778	53.33333	45.1124	69.58763	65.625	26.97842	309.52463
Media	98.06634	95.64196	31.66667	32.52572	76.44444	73.83333	44.77778	47.31655	71.29671	64.09472	31.94245	310.66114

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
GRASP	98.27798	95.25414	63.66667	4.05062	79.88889	70.72222	80.33333	17.57407	82.33462	71.49646	92.51799	34.47605
EGA	97.18829	95.43020	45.66667	25.29545	71.83334	73.44444	53.22222	32.77355	66.11630	64.04156	51.33094	121.72561
UXEGA	97.25772	94.90277	49.33333	25.36227	73.61111	72.61111	50.00000	32.82589	66.37350	63.89283	51.76259	121.40811
GGA	97.96219	95.57030	29.66667	28.58018	76.22222	75.38889	44.00000	33.98448	69.53125	63.36877	34.49640	146.04693
UXGGA	98.03175	96.16889	27.33333	29.21565	77.77778	72.44444	43.88889	33.93815	73.42193	64.71971	32.41007	151.89248
AM1010	98.03150	96.30900	21.00000	58.23279	77.22222	71.55556	37.55555	162.30115	71.61082	65.28673	32.58993	1705.68654
AM1001	98.10217	95.85261	27.66667	32.68215	77.50000	74.38889	43.00000	46.57261	71.76546	64.45876	33.99281	294.73108

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 metaheurísticas implementadas. Los meméticos son capaces de superar al GRASP en términos de clasificación en las bases de datos pequeñas. En *Arrhythmia* sigue ganando el GRASP con un par de puntos de ventaja.

En particular, los meméticos no obtienen una mejora sustancial en clasficación con respecto a los genéticos sin hibridar. Sólo en la base de datos más grande, *Arrhythmia*, consiguen mejorar el resultado en clasificación de sus predecesores. Esto podría ser porque la diversidad que teneran los genéticos por sí mismos es suficiente para los problemas más pequeños, pero no son capaces de explotar bien según qué zonas del espacio de búsqueda si éste es muy grande. Por esta razón, es posible que este tipo de algoritmos funcionara mejor en bases de datos más grandes todavía.

El tiempo extra que tardan estos algoritmos no compensa demasiado la ganancia que tienen, al menos en las bases de datos pequeñas. En *Arrhythmia* sí que se debe intentar rascar cualquier punto de más en clasificación puesto que de por sí la tasa de clasificación no es muy buena.

El número de individuos a mejorar con hibridación altera el equilibrio diversidad-explotación que tanto se trabaja en este campo, pero no se consiguen resultados muy dispares, ya que el equilibrio tiende más a la explotación al utilizar la búsqueda local.

En esta práctica yo me quedaría personalmente con el genético estacionario con cruce uniforme. El cruce uniforme está muy usado en la literatura por ser bastante eficiente en su cometido, ya que introduce más diversidad que el cruce en dos puntos. El estacionario hemos visto que tiene muy poca pérdida con respecto al generacional, con la ventaja de que tiene tasa de reducción de columnas notable. Un poco menos de precisión, pero el tamaño del problema se hará notablemente más pequeño.

7. Referencias

Las referencias utilizadas han sido:

- *Scikit-Learn*: La propia documentación[2] de la biblioteca.
- *SciPy*: La propia documentación[3] de la biblioteca.

[2] http://scikit-learn.org/stable/modules/classes.html
[3] http://docs.scipy.org/doc/scipy/reference/