

# **DIPLOMATURA EN MACHINE LEARNING CON PYTHON**

## **Algoritmos genéticos**

## Algoritmos genéticos

- Algoritmo básico en Excel
- Discusión del tipo de problemas en los que se aplica
- Discusión de otros mecanismos de optimización
- Implementación en R del método enjambre de partículas.

## Introducción

Existen muchos problemas que resultan intratables para darles una solución exacta. Un truco difundido en ese caso es considerar una familia de soluciones e ir buscando por aproximaciones sucesivas la mejor de ellas.

Un caso emblemático de este problema es el muy conocido "Recorrido del viajante". Un vendedor debe recorrer N ciudades sin pasar dos veces por ninguna de ellas minimizando la distancia a cubrir.

Otro caso típico es el problema de la mochila. Se debe elegir el orden en el que se van cargando objetos de diferentes dimensiones en la mochila de manera que se maximice el aprovechamiento del volumen.

¿Qué tienen en común ambos problemas?

Existen pedazos de la solución que son, en sí mismos, exitosos. No podemos estar seguros de que ninguno de estos pedazos forme para de la solución definitiva, pero, cada uno de ellos representa una forma bastante buena de hacer las cosas.

En el caso del viajante puede haber tres ciudades que se encuentran próximas entre sí y alejadas del resto. Conviene entonces recorrerlas en secuencia y luego volver al resto. En el caso de la mochila el equivalente sería un grupo de objetos que encajen bien entre sí.

Encontrar estos "trozos exitosos de solución" es la clave de los algoritmos genéticos.

Los algoritmos genéticos son definidos como una heurística de búsqueda, es decir una técnica utilizada para solucionar problemas cuando los métodos tradicionales son lentos o imposibles, como observamos en el problema del viajante.

Su área de aplicación es la optimización y los problemas de búsqueda. Su historia se remonta a los estudios de Turing (1954) sobre aprendizaje automático y a las formalizaciones de biólogos y genetistas cuantitativos de los años 60.

De ahí que conserve una analogía con la evolución natural. Para ello implementa técnicas inspiradas en la herencia, mutación, selección y cruzamiento.

Suele clasificarse a algoritmos genéticos dentro del conjunto de las redes neuronales. Específicamente pertenecen al campo denominado computación evolutiva. Al no haber una variable de la cual pueda aprender las soluciones a los problemas planteados se lo considera una técnica no supervisada.

El problema del viajante

Vamos a desarrollar, para fijar ideas, el problema del viajante.

Tenemos:

C: conjunto de las ciudades a visitar:

$$C \equiv \{(x_i, y_i) \mid 1 \leq i \leq N\}$$

D: distancia recorrida:

$$D = \sqrt{\sum_{i=1}^{N-1} [(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2]}$$

Notar que no es exactamente la definición de la distancia euclídea ya que esta última debiera introducir la raíz cuadrada antes de sumar y no después. Sin embargo podemos usarla sin problemas para la minimización.

Lo que estamos buscando es encontrar el orden en el cual recorreremos las ciudades tal que D es mínimo.

La forma más general de solucionar este problema es tomar todos los órdenes posibles de ordenar N ciudades, recorrerlos todos y elegir el mínimo. ¿Parece fácil?

Para un matemático es fácil. Con dos renglones despachó el problema. Cuando tratamos de hacer efectivamente las cuentas la cosa se complica. ¿Por qué?

Pensemos en un pequeño conjunto de 20 ciudades. Algo no difícil para cualquier viajante. Cuando tratamos de elegir la ciudad por la que comenzar nuestro recorrido tenemos 20 posibilidades.

Una vez que hemos elegido una ciudad tenemos 19 posibilidades para la segunda, luego 18 para la tercera y así siguiendo adelante. Cuando nos quede una ciudad ya está determinado a cual debemos ir.

Esto significa que la cantidad de posibilidades es:

$$\text{Posibilidades} = 20 \times 19 \times 18 \times \dots \times 1$$

Esto es lo que los matemáticos suelen denominar el factorial, o sea:

Posibilidades = 20!

¿Cuánto vale 20!? Realmente es un número grande se parece a  $2,43 \times 10^{18}$  o sea 2.430.000.000.000.000.000 lo que leeríamos como dos trillones, cuatrocientos treinta mil billones.

Si nuestra computadora tardara una millonésima de segundo en evaluar ese número igual le llevaría algo más de 77.416 años obtener la respuesta.

La verdad es que para entonces nuestro viajante se habrá probablemente jubilado. Debemos por lo tanto buscar otra aproximación a este problema.

Cada posible solución de este problema es un vector de 20 posiciones conteniendo el número de ciudad a visitar:

(3, 17, 5, 2, 6, 1, ..., 4)

Comenzamos generando 10 posibles soluciones ordenando de 10 formas diferentes, siempre al azar, los números del 1 al 20.

Calculamos para cada una de las 10 posibles soluciones la distancia total a recorrer. Identificamos los dos mínimos y el máximo.

Armado de la nueva generación:

La siguiente generación de soluciones estará dada por las 9 mejores soluciones (todas menos el máximo) y una nueva solución formada por una permutación entre dos ciudades tomadas al azar en la mejor solución.

Una vez que tenemos un nuevo conjunto de 10 soluciones distintas volvemos a calcular la distancia que recorreremos con cada una, identificando los dos nuevos mínimos y el nuevo máximo.

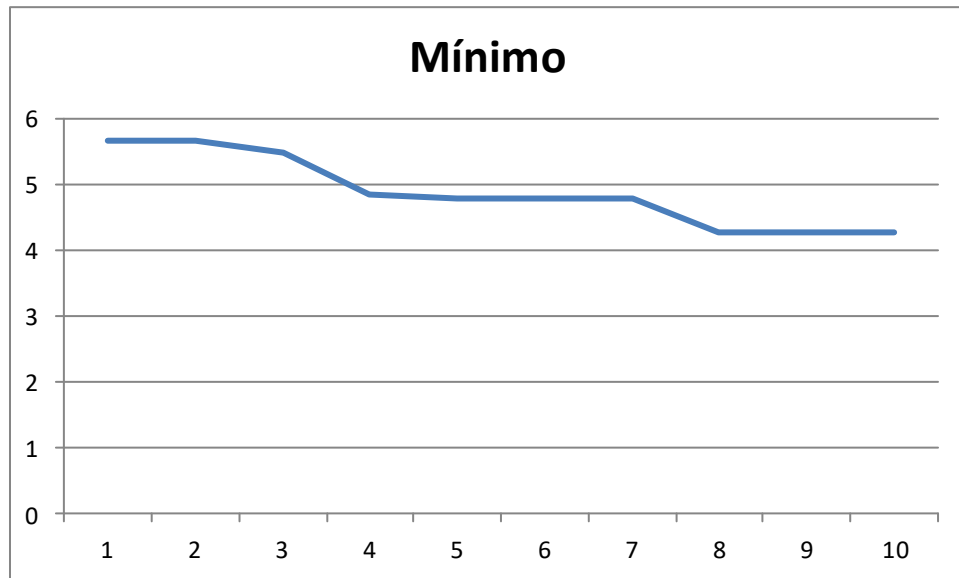
Volvemos a repetir el armado de la nueva generación hasta que la disminución de la distancia recorrida en el mínimo resulte menor a un factor pre-establecido durante una dada cantidad de rondas.

¿Estamos seguros que se trata de la solución óptima?

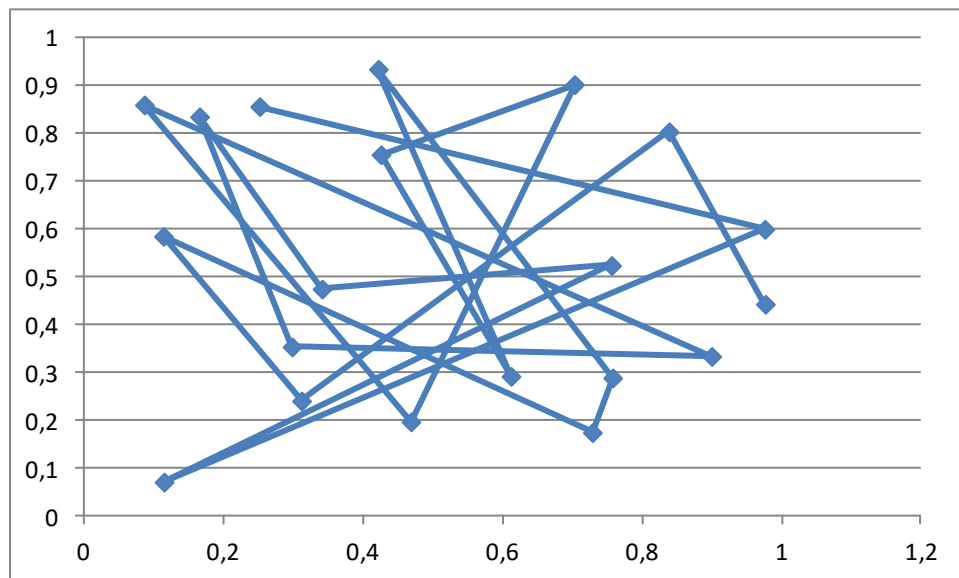
De ninguna manera.

¿Cómo podemos ganar en seguridad?

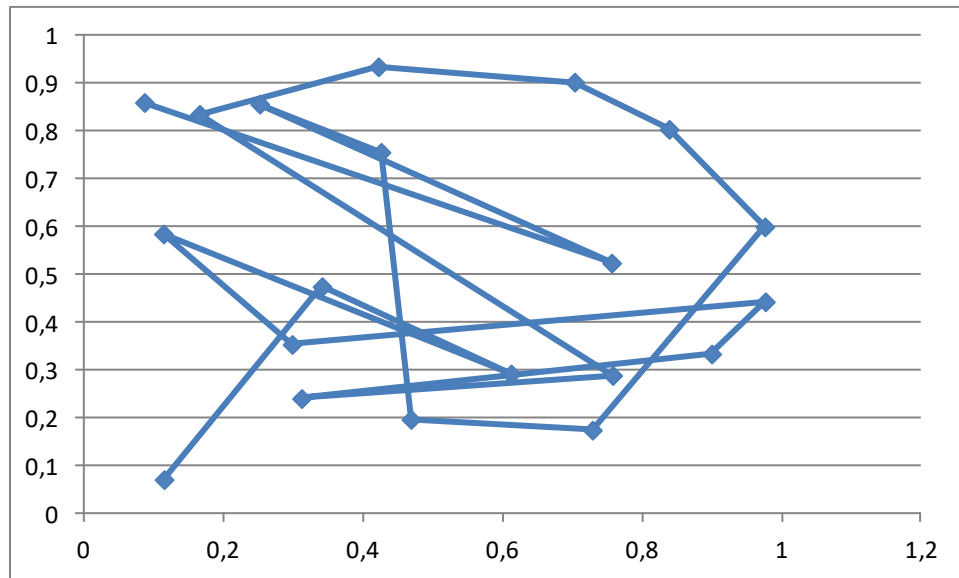
Partiendo de diferentes conjuntos y viendo que llegamos a la misma solución. Uno nunca está totalmente seguro pero repitiendo el proceso varias veces va ganando en confianza. Vemos un ejemplo sobre cómo va disminuyendo la distancia recorrida



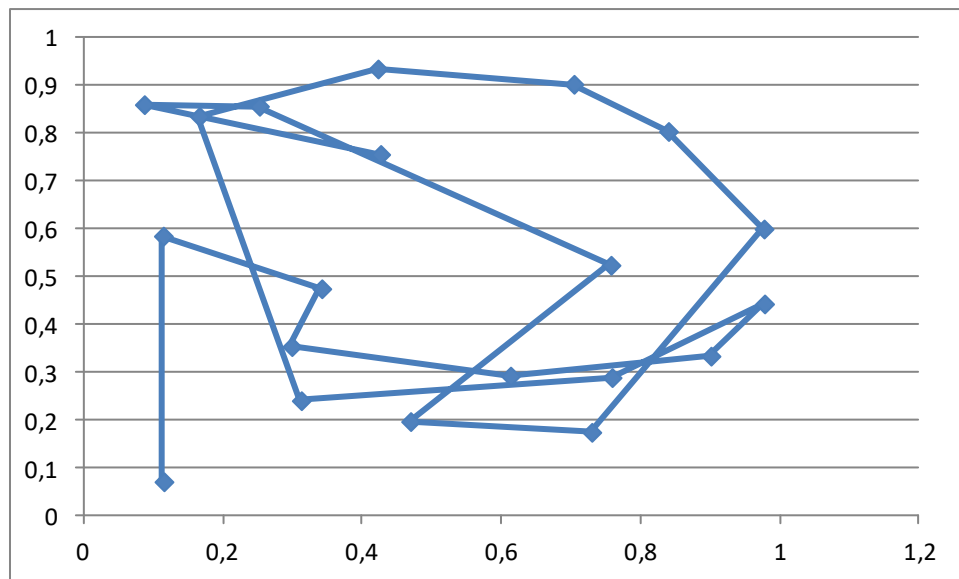
El viaje original era:



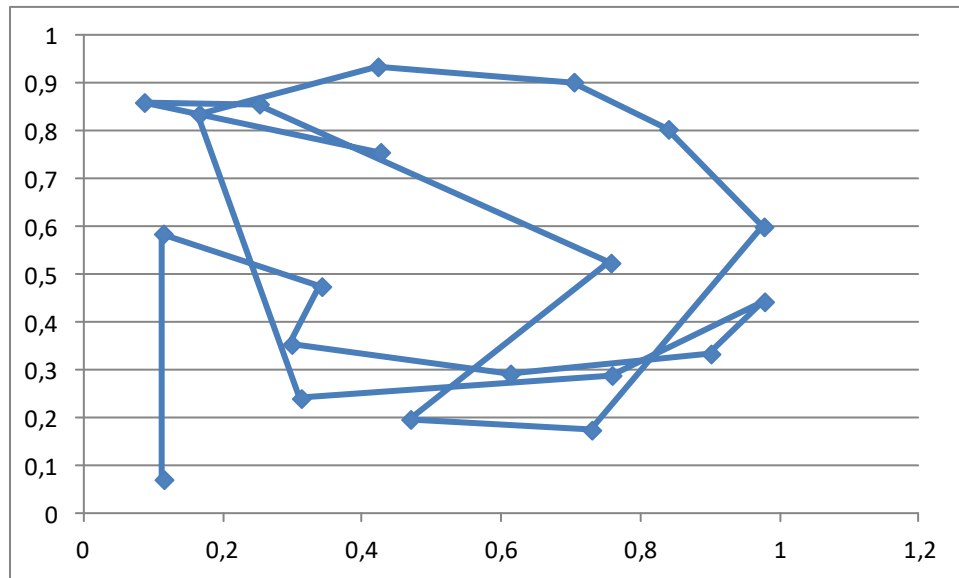
El viaje tras 10 rondas es:



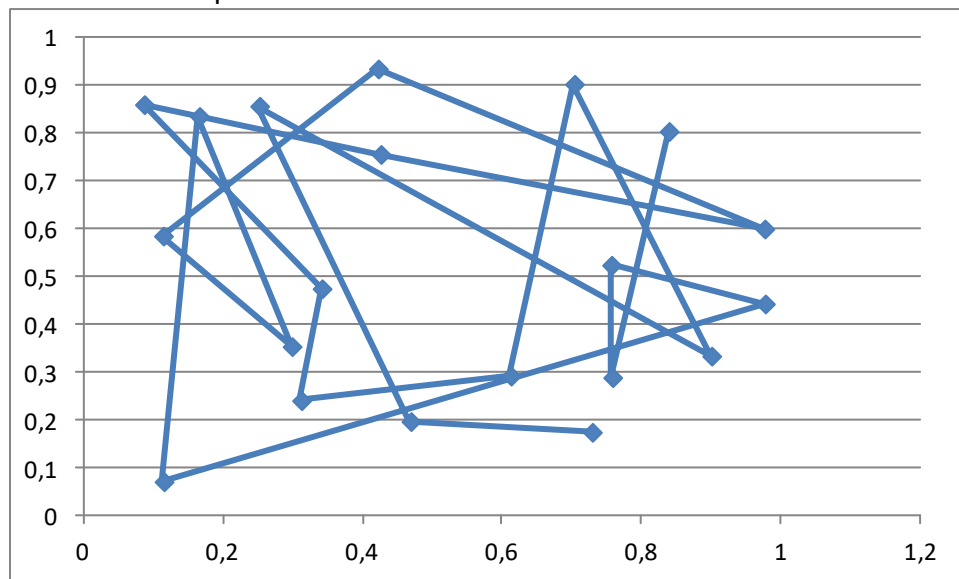
Tras unas 30 iteraciones:



Y tras 100 iteraciones sigue sin cambios:

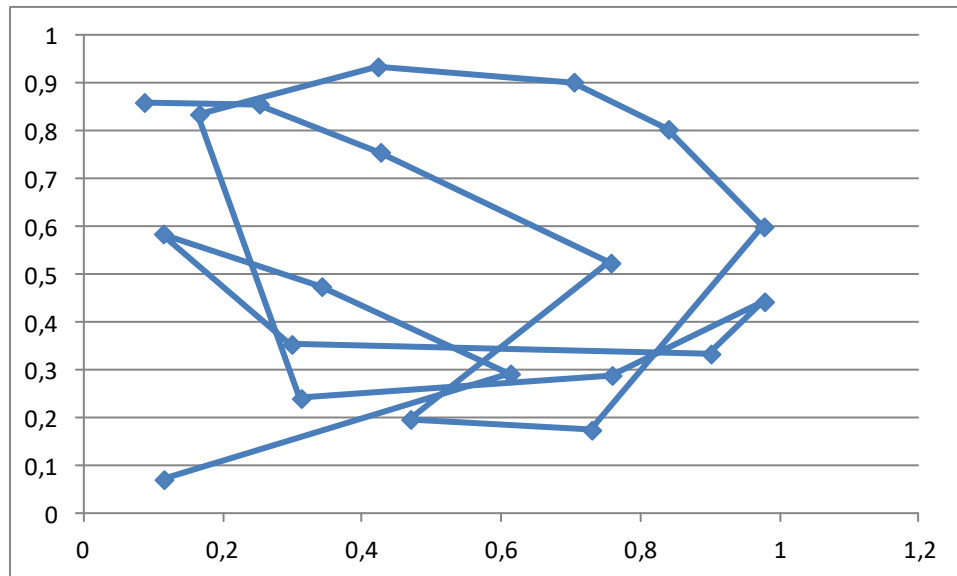


Al comenzar desde otro punto:



Tras cien iteraciones llego a:





Es posible optimizar el algoritmo de muchas maneras. Conviene tomar diferentes puntos de partida, avanzar una cantidad de pasos y combinar luego los resultados.

¿Cuándo conviene usar algoritmos genéticos?

1. Cuando no exista una solución directa.
2. Cuando las partes de la solución óptima tengan un sentido en si mismas. (Por ejemplo un tramo vinculando a varias ciudades próximas)
3. Cuando la combinación de soluciones posibles sea inexplorable en su totalidad con el poder de cómputo disponible pero la evaluación de una solución puntual sea relativamente rápida.

¿Cuándo se alcanza una solución óptima?

El entrenamiento de algoritmos genéticos puede ser imprevisible en cuanto a la duración (unos pocos minutos hasta días y semanas). Existen formas de detener las sucesivas generaciones o iteraciones, mediante uno o varios de los siguientes parámetros:

- Cuando estamos ante una situación que satisface un criterio mínimo.
- Cuando alcanza un número de generaciones fijado de antemano.
- Cuando se sobrepasa un cierto límite, por ejemplo computacional.
- Se establece que cuando en un cierto número de generaciones no se alcanza un grado de ajuste estipulado, se toma la generación mejor rankeada.
- Por inspección manual.

Problemas de los AG

- Para encontrar la solución óptima a problemas de alta dimensionalidad o multimodales se requiere realizar evaluaciones de funciones de ajuste que son computacionalmente costosas. Las simulaciones pueden durar entre horas y días.
- Los algoritmos genéticos no escalan bien con la complejidad. Cuando un gran número de elementos se expone a mutación existe un incremento exponencial del espacio de búsqueda.
- La mejor solución solamente lo es en relación a otras que son una fracción de las posibles existentes. El criterio elegido para detener el entrenamiento no es claro.
- Como las redes neuronales, existe una tendencia a converger en mínimos locales o aún en puntos arbitrarios más que en el óptimo global del problema planteado.
- Es difícil operar sobre conjuntos dinámicos como el genoma, por ejemplo.
- No soluciona problemas basados en decisiones como problemas de verdadero / falso.
- Existe una gama de algoritmos que poseen mejor rendimiento en cuanto a velocidad para problemas de optimización.

## Implementación en Python

Vamos a utilizar las clases desarrolladas por Joaquín Amat Rodrigo brillantemente descritas en:

[https://www.cienciadatos.net/documentos/py01\\_optimizacion\\_ga](https://www.cienciadatos.net/documentos/py01_optimizacion_ga)

Las clases que vamos a usar son:

- Individuo
- Población

### **Individuo:**

Esta clase representa un individuo con unas características iniciales definidas por una combinación de valores numéricos aleatorios. El rango de posibles valores para cada variable puede estar acotado.

Parámetros:

n\_variables: entero

Es el número de variables que definen al individuo.

limites\_inf: lista o numpy.ndarray

Se trata de un parámetro optativo

Provee un límite inferior para cada variable. Si sólo se quieren predefinir los límites inferiores de algunas variables, se debe emplear ``None`` para las demás. Los ``None`` serán remplazados por el valor  $(-10^{**3})$ . (El valor por defecto es ``None``)

limites\_sup: lista o numpy.ndarray

Se trata de un parámetro optativo

Provee un límite superior para cada variable. Si sólo se quieren predefinir los límites inferiores de algunas variables, se debe emplear ``None`` para las demás. Los ``None`` serán remplazados por el valor (10\*\*3). (El valor por defecto es ``None``)

verbose: lógico

Se trata de un parámetro optativo. Se utiliza para mostrar la información del individuo creado. El valor por defecto es False.

Atributos:

n\_variables: entero

Es el número de variables que definen al individuo.

limites\_inf: lista o numpy.ndarray

Se trata de un parámetro optativo

Provee un límite inferior para cada variable. Si sólo se quieren predefinir los límites inferiores de algunas variables, se debe emplear ``None`` para las demás. Los ``None`` serán remplazados por el valor (-10\*\*3). (El valor por defecto es ``None``)

limites\_sup: lista o numpy.ndarray

Se trata de un parámetro optativo

Provee un límite superior para cada variable. Si sólo se quieren predefinir los límites inferiores de algunas variables, se debe emplear ``None`` para las demás. Los ``None`` serán remplazados por el valor (10\*\*3). (El valor por defecto es ``None``)

valor\_variables: `numpy.ndarray`

Es un array con el valor de cada una de las variables que definen al individuo.

fitness: numérico flotante.

Guarda el resultado de calcular la función de fitness para los valores de las variables que definen al individuo.

valor\_funcion: numérico flotante.

Guarda el valor de la función objetivo para el individuo.

Dependiendo de si se trata de un problema de maximización o minimización, la relación del fitness con la función objetivo puede ser:

- Maximización: el individuo tiene mayor fitness cuanto mayor es el valor de la función objetivo.
- Minimización: el individuo tiene mayor fitness cuanto menor es el valor de la función objetivo, o lo que es lo mismo, cuanto mayor es el valor de la función objetivo, menor el fitness.

El algoritmo genético selecciona los individuos de mayor fitness, por lo que, para problemas de minimización, el fitness puede calcularse como  $-f(\text{individuo})$  o también  $1/(1+f(\text{individuo}))$ .

### **Poblacion:**

Esta clase almacena una población de n individuos.

Parámetros

n individuos: entero

Guarda el número de individuos que almacena la población.

n variables: entero

Guarda el número de variables que definen a cada individuo.

limites\_inf: lista o numpy.ndarray

Se trata de un parámetro optativo

Provee un límite inferior para cada variable. Si sólo se quieren predefinir los límites inferiores de algunas variables, se debe emplear ``None`` para las demás. Los ``None`` serán remplazados por el valor  $(-10^{**3})$ . (El valor por defecto es ``None``)

limites\_sup: lista o numpy.ndarray

Se trata de un parámetro optativo

Provee un límite superior para cada variable. Si sólo se quieren predefinir los límites inferiores de algunas variables, se debe emplear ``None`` para las demás. Los ``None`` serán remplazados por el valor  $(10^{**3})$ . (El valor por defecto es ``None``)

verbose: lógico

Se trata de un parámetro optativo. Se utiliza para mostrar la información del individuo creado. El valor por defecto es False.

Atributos

individuos: lista

Es una lista donde guarda todos los individuos de la población en su estado actual.

n individuos: entero

Guarda el número de individuos de la población.

n variables: entero

Guarda el número de variables que definen a cada individuo.

limites\_inf: lista o numpy.ndarray

Se trata de un parámetro optativo

Provee un límite inferior para cada variable. Si sólo se quieren predefinir los límites inferiores de algunas variables, se debe emplear ``None`` para las demás. Los ``None`` serán remplazados por el valor  $(-10^{**3})$ . (El valor por defecto es ``None``)

limites\_sup: lista o numpy.ndarray

Se trata de un parámetro optativo

Provee un límite superior para cada variable. Si sólo se quieren predefinir los límites inferiores de algunas variables, se debe emplear ``None`` para las demás. Los ``None`` serán remplazados por el valor  $(10^{**3})$ . (El valor por defecto es ``None``)

mejor\_individuo: individuo

Guarda al mejor individuo de la población en su estado actual.

mejor\_fitness: numérico flotante

Guarda el valor del fitness del mejor individuo de la población en su estado actual.

mejor\_valor\_funcion: numérico flotante

Guarda el valor de la función objetivo del mejor individuo de la población en su estado actual.

mejor\_individuo\_variables: numpy.ndarray

Guarda el valor de las variables del mejor individuo de la población en su estado actual.

historico\_individuos: lista

Guarda en una lista todos los individuos de todas las generaciones utilizadas en el cálculo.

historico\_mejor\_individuo\_variables: lista

Guarda en una lista el valor de las variables del mejor individuo en cada una de las generaciones que ha tenido la población.

historico\_mejor\_fitness: lista

Guarda en una lista con el mejor valor de fitness en cada una de las generaciones que ha tenido la población.

historico\_mejor\_valor\_funcion: lista

Guarda en una lista el valor de la función objetivo del mejor individuo en cada una de las generaciones que ha tenido la población.

diferencia\_abs: lista

Guarda en una lista la diferencia absoluta entre el mejor fitness de generaciones consecutivas.

resultados\_df: pandas.core.frame.DataFrame

Guarda en un dataframe la información del mejor fitness y valor de las variables encontrado en cada generación, así como la diferencia respecto a la generación anterior.

fitness\_optimo: numérico flotante

Es el mejor fitness encontrado tras el proceso de optimización.

valor\_funcion\_optimo: numérico flotante

Es el valor de la función objetivo encontrado tras el proceso de optimización.

valor\_variables\_optimo: numpy.ndarray

Es el valor de las variables del individuo con el que se ha conseguido el mejor fitness tras el proceso de optimización.

optimizado: lógico

Informa si la población ha sido optimizada o no.

iter\_optimizacion: entero

Es el número de iteraciones de optimización (generaciones).

#### **Parámetros del método optimizar:**

funcion\_objetivo: función

Es la función que se quiere optimizar.

optimizacion: {"maximizar" o "minimizar"}

Informa sobre si se desea maximizar o minimizar la función.

n\_generaciones: entero

Se trata de un parámetro opcional.

Informa el número de generaciones de optimización.

El valor por defecto es 50.

metodo\_seleccion: {"ruleta", "rank", "tournament"}

Es un parámetro opcional.

Informa el método de selección de selección admite los valores ruleta, rank y tournament.

El valor por defecto es tournament.

Método de ruleta: la probabilidad de que un individuo sea seleccionado es proporcional a su fitness relativo, es decir, a su fitness dividido por la suma del fitness de todos los individuos de la población. Si el fitness de un individuo es el doble que el de otro, también lo será la probabilidad de que sea seleccionado. Este método presenta problemas si el fitness de unos pocos individuos es muy superior (varios órdenes de magnitud) al resto, ya que estos serán seleccionados de forma repetida y casi todos los individuos de la siguiente generación serán “hijos” de los mismos “padres” (poca variación).

Método rank: la probabilidad de selección de un individuo es inversamente proporcional a la posición que ocupa tras ordenar todos los individuos de mayor a menor fitness. Este

método es menos agresivo que el método ruleta cuando la diferencia entre los mayores fitness es varios órdenes de magnitud superior al resto.

Selección competitiva (tournament): se seleccionan aleatoriamente dos parejas de individuos de la población (todos con la misma probabilidad). De cada pareja se selecciona el que tenga mayor fitness. Finalmente, se comparan los dos finalistas y se selecciona el de mayor fitness. Este método tiende a generar una distribución de la probabilidad de selección más equilibrada que las dos anteriores.

elitismo: numérico flotante.

Es un parámetro opcional.

Es el porcentaje de mejores individuos de la población actual que pasan directamente a la siguiente población. De esta forma, se asegura que, la siguiente generación, no sea nunca peor.

El valor por defecto es 0.1

prob\_mut : numérico flotante.

Es un parámetro opcional

Guarda la probabilidad que tiene cada posición del individuo de mutar.

El valor por defecto es 0.01

distribucion: {"normal", "uniforme", "aleatoria"}

Es un parámetro opcional.

Es la distribución que se utilizará para obtener el factor de mutación.

El valor por defecto es uniforme

media\_distribucion: numérico flotante

Es un parámetro opcional.

Informa la media de la distribución si se selecciona `distribucion = "normal"`

El valor por defecto es 1.

sd\_distribucion: numérico flotante

Es un parámetro opcional.

Informa el desvío estándar de la distribución si se selecciona `distribucion = "normal"`.

El valor por defecto es 1.

min\_distribucion: numérico flotante

Es un parámetro opcional

Informa el mínimo de la distribución si se selecciona `distribucion = "uniforme"`.

El valor por defecto es -1.

max\_distribucion: numérico flotante

Es un parámetro opcional

Informa el máximo de la distribución si se selecciona `distribucion = "uniforme"`.

El valor por defecto es 1.

parada\_temprana: lógico

Es un parámetro opcional.

Se utiliza para adelantar el final del algoritmo. Se aplica si durante las últimas rondas\_parada generaciones la diferencia absoluta entre mejores individuos no es superior al valor de tolerancia\_parada, se detiene el algoritmo y no se crean nuevas generaciones. El valor por defecto es False.

rondas\_parada: entero

Es un parámetro opcional.

Informa el número de generaciones consecutivas sin mejora mínima para que se active la parada temprana.

El valor por defecto es None.

tolerancia\_parada: numérico flotante o entero

Es un parámetro opcional.

Informa el valor mínimo que debe tener la diferencia de generaciones consecutivas para considerar que hay cambio.

El valor por defecto es None.

verbose: lógico

Es un parámetro opcional.

Informa si el algoritmo debe mostrar información del proceso por pantalla.

El valor por defecto es False.

verbose\_nueva\_generacion: Lógico

Es un parámetro opcional.

Informa si el algoritmo debe mostrar información de cada nueva generación por pantalla.

El valor por defecto es False.

verbose\_seleccion: lógico

Es un parámetro opcional.

Informa si el algoritmo debe mostrar información de cada selección por pantalla.

El valor por defecto es False.

verbose\_cruce: lógico.

Informa si el algoritmo debe mostrar información de cada cruce por pantalla.

El valor por defecto es False.

verbose\_mutacion: lógico.

Informa si el algoritmo debe mostrar información de cada mutación por pantalla.

El valor por defecto es False.

### **Ejemplo de aplicación:**

Paso 1: Definición de la función objetivo



```
D:\Dropbox\Instituto para el uso ético de la ciencia de datos\Cursos\Diplomatura en Python aplicado a la ciencia de datos\Eje...
File Edit Selection Find View Goto Tools Project Preferences Help
dataframes.py x ConexiónSQLServer.py x EjemplosCSV.py x EjemploPyODBC.py x gráficos01.py x p x
1566
1567 def funcion_objetivo(x_0, x_1):
1568     """
1569     Para la región acotada entre  $-10 \leq x_0 \leq 0$  y  $-6.5 \leq x_1 \leq 0$ 
1570     múltiples mínimos locales y un único mínimo global que
1571      $f(-3.1302468, -1.5821422) = -106.7645367$ 
1572     """
1573     f = np.sin(x_1)*np.exp(1-np.cos(x_0))**2 \
1574         + np.cos(x_0)*np.exp(1-np.sin(x_1))**2 \
1575         + (x_0-x_1)**2
1576     return(f)
1577
```

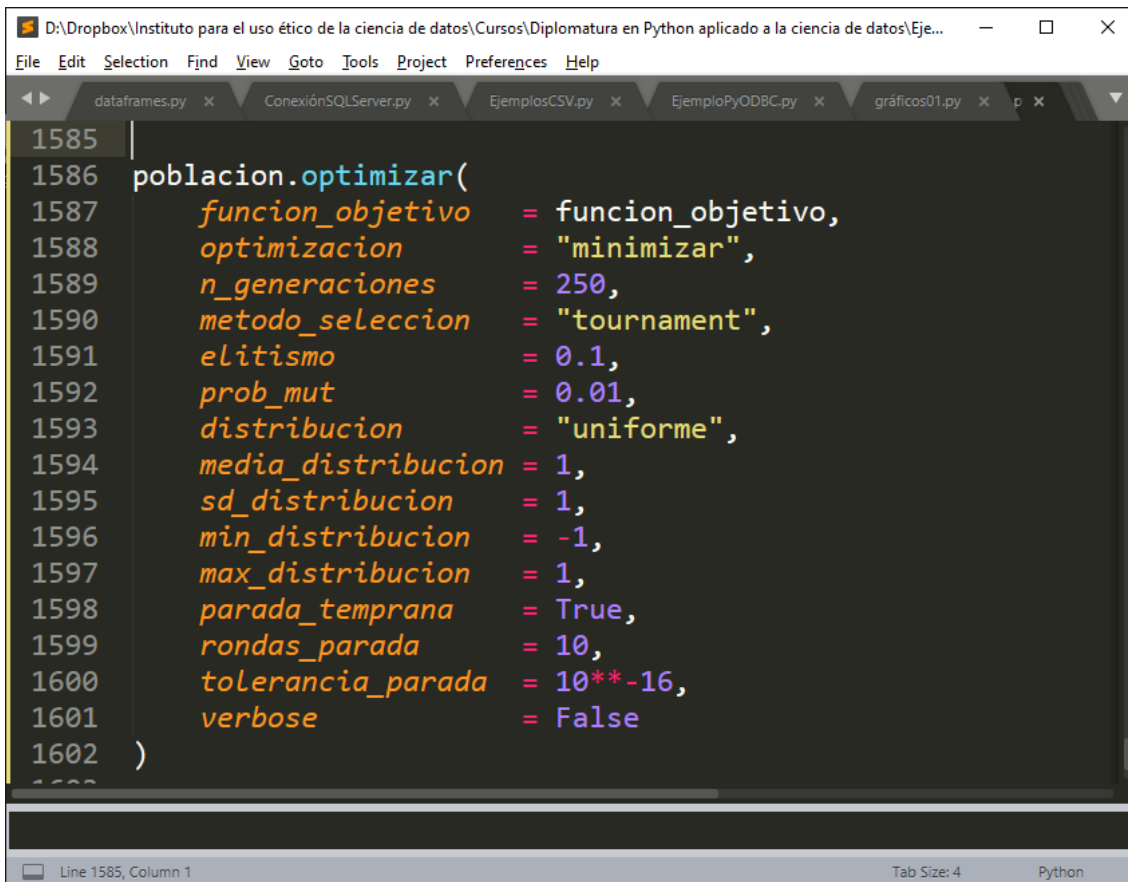
Line 564, Column 1 Tab Size: 4 Python

### Paso 2: Definición de la población

```
D:\Dropbox\Instituto para el uso ético de la ciencia de datos\Cursos\Diplomatura en Python aplicado a la ciencia de datos\Eje...
File Edit Selection Find View Goto Tools Project Preferences Help
dataframes.py x ConexiónSQLServer.py x EjemplosCSV.py x EjemploPyODBC.py x gráficos01.py x p x
1578 poblacion = Poblacion(
1579     n_individuos = 50,
1580     n_variables = 2,
1581     limites_inf = [-10, -6.5],
1582     limites_sup = [0, 0],
1583     verbose = False
1584 )
1585
```

Line 564, Column 1 Tab Size: 4 Python

### Paso 3: Optimización

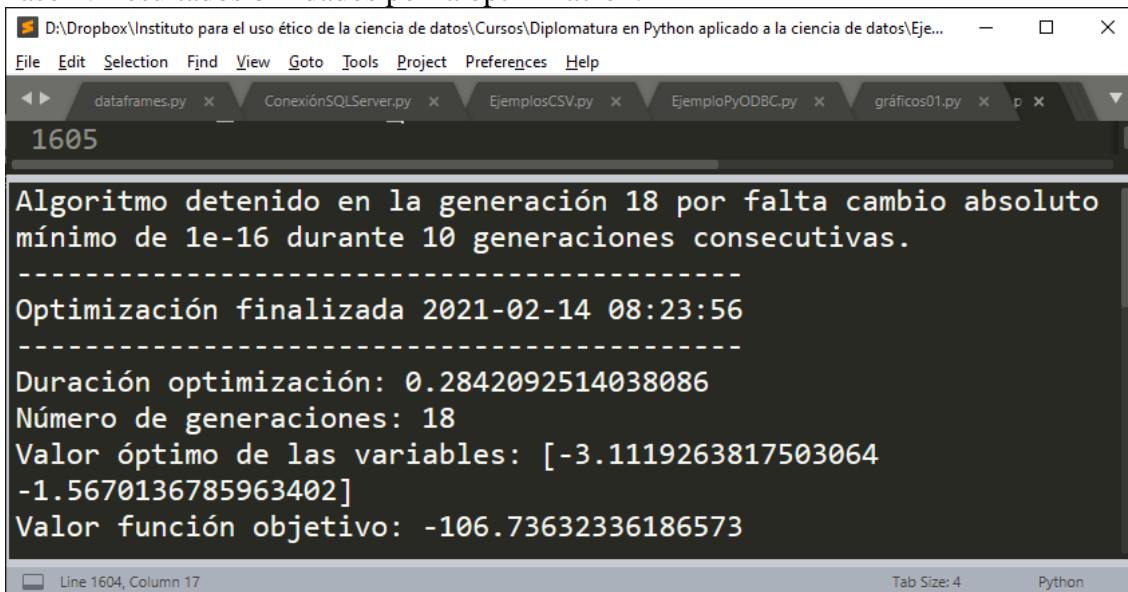


```

1585
1586 poblacion.optimizar(
1587     funcion_objetivo = funcion_objetivo,
1588     optimizacion      = "minimizar",
1589     n_generaciones    = 250,
1590     metodo_seleccion = "tournament",
1591     elitismo          = 0.1,
1592     prob_mut          = 0.01,
1593     distribucion      = "uniforme",
1594     media_distribucion = 1,
1595     sd_distribucion   = 1,
1596     min_distribucion  = -1,
1597     max_distribucion  = 1,
1598     parada_temprana  = True,
1599     rondas_parada    = 10,
1600     tolerancia_parada = 10**-16,
1601     verbose          = False
1602 )

```

Paso 4: Resultados brindados por la optimización:

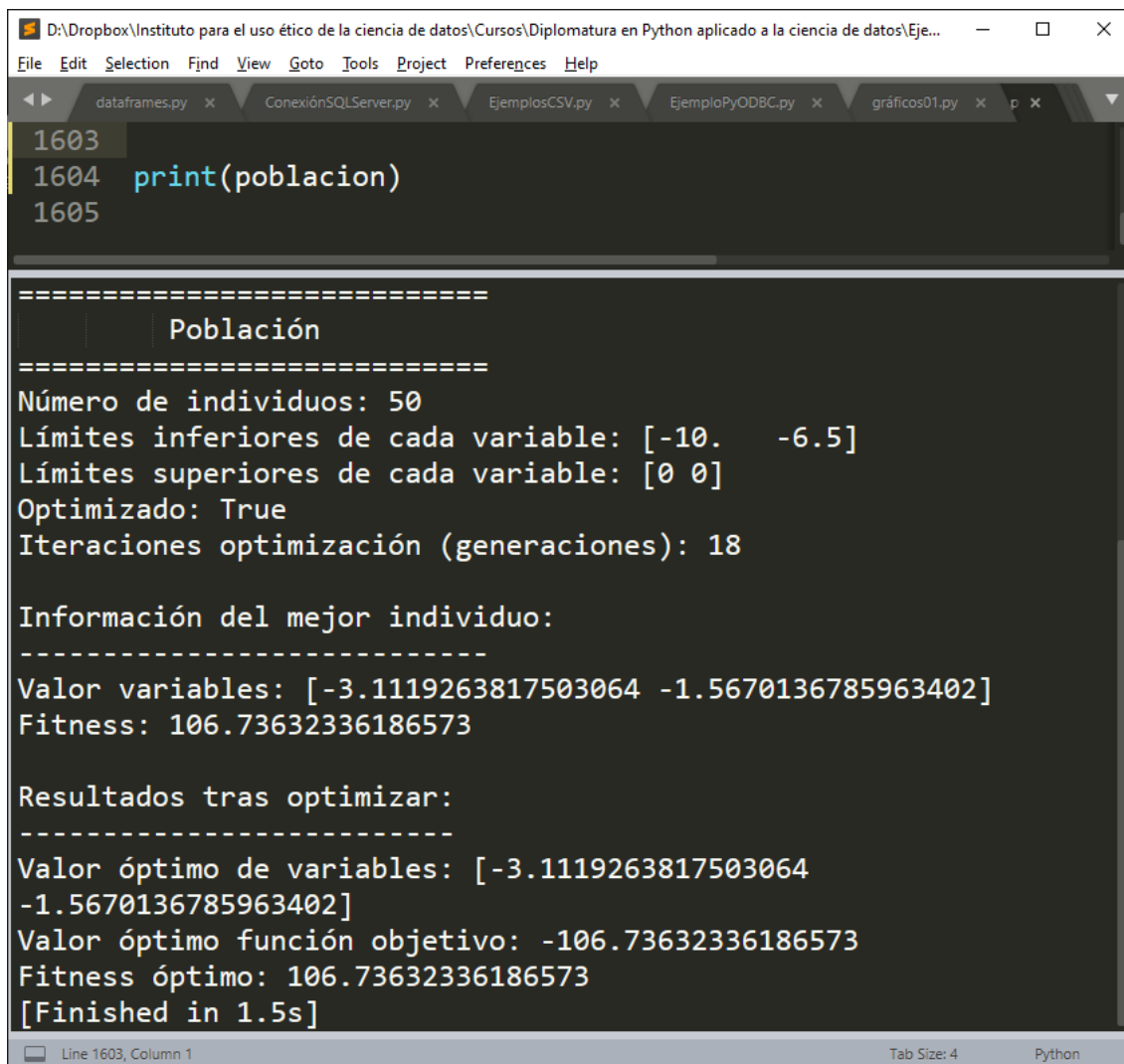


```

1605
Algoritmo detenido en la generación 18 por falta cambio absoluto
mínimo de 1e-16 durante 10 generaciones consecutivas.
-----
Optimización finalizada 2021-02-14 08:23:56
-----
Duración optimización: 0.2842092514038086
Número de generaciones: 18
Valor óptimo de las variables: [-3.1119263817503064
-1.5670136785963402]
Valor función objetivo: -106.73632336186573

```

Paso 5: Extraer más resultados



```
D:\Dropbox\Instituto para el uso ético de la ciencia de datos\Cursos\Diplomatura en Python aplicado a la ciencia de datos\Eje...
File Edit Selection Find View Goto Tools Project Preferences Help
dataframes.py x ConexiónSQLServer.py x EjemplosCSV.py x EjemploPyODBC.py x gráficos01.py x p x
1603
1604 print(poblacion)
1605

=====
Población
=====
Número de individuos: 50
Límites inferiores de cada variable: [-10. -6.5]
Límites superiores de cada variable: [0 0]
Optimizado: True
Iteraciones optimización (generaciones): 18

Información del mejor individuo:
-----
Valor variables: [-3.1119263817503064 -1.5670136785963402]
Fitness: 106.73632336186573

Resultados tras optimizar:
-----
Valor óptimo de variables: [-3.1119263817503064
-1.5670136785963402]
Valor óptimo función objetivo: -106.73632336186573
Fitness óptimo: 106.73632336186573
[Finished in 1.5s]
```