Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

9 de marzo de 2023

VERSIÓN PRELIMINAR

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro

ISBN-13 (versión electrónica): ???-?-????-????-?

Primera Edición (versión preliminar)
Escrito con XeLaTeX.

Licencia: Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0)
Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2021
Web: https://github.com/facundobatista/libro-pyciencia

10 9 8 7 6 5 4 3 2 1

VERSIÓN PRELIMINAR

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científiques, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos"que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene dos grandes partes. La primera habla de Python, algunas bibliotecas importantes y otros temas que son fundamentales. En la segunda, mostramos cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En ambos casos este libro esquiva la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de les científiques para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

Índice general

Parte I
Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos y técnicas fundamentales que serán utilizados en la Parte II al abordar temas de aplicaciones específicas. Se sugiere, a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

Parte II
Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte I.

1 | Modelado estadístico

1.1. Introducción

Los modelos estadísticos, construidos a partir de un conjunto de datos observados, son utilizados en las diferentes disciplinas científicas con dos propósitos: explicación causal y predicción empírica. Ambos enfoques no son, naturalmente, mutuamente excluyentes.

Estos modelos generalmente contienen uno o varios parámetros, que pueden determinarse a partir de los datos a través de un procedimiento de ajuste para obtener los valores que mejor expliquen los datos observados. Posteriormente se pueden utilizar para predecir los valores de nuevas observaciones dados los valores de las variables independientes del modelo.

Módulo	Versión
Matplotlib	3.5.3
Pandas	1.4.4
SciPy	1.9.1
statsmodels	0.13.2

Código disponible

Una vez establecido el modelo, se pueden realizar comparaciones estadísticas entre los valores observados y los generados por el modelo para determinar cuán bien el modelo explica los datos, y qué parámetros contribuyen más al poder predictivo.

Abordaremos el siguiente problema: dado un conjunto de variables dependientes Y (o respuesta), y variables independientes X, queremos determinar una relación matemática (o *modelo*) como función $Y = f(X)$.

Si no conocemos la función f pero tenemos acceso a datos provenientes de observaciones como pares $\{x_i, y_i\}$, podemos parametrizar la función y ajustar los valores de dichos parámetros utilizando los datos. Por ejemplo, podríamos tener el modelo lineal $f(X) = \beta_0 + \beta_1 X$ cuyos parámetros son β_0 y β_1. Por lo general tenemos más datos que parámetros en el modelo, y en tales casos podemos realizar un ajuste de mínimos cuadrados que minimiza la norma del residuo $r = Y - f(X)$.

Hemos presentado el modelo matemático representado por la función f, pero el ingrediente que hace al modelo *estadístico* lo constituye cierta incertidumbre contenida en los datos $\{x_i, y_i\}$ en virtud de, por ejemplo, errores en las mediciones u otros factores fuera de control. Esta incerteza en los datos puede ser descripta en términos de variables aleatorias, por ejemplo, $Y = f(X) + \varepsilon$, donde ε es una variable estocástica. Dependiendo de cómo aparece la variable aleatoria en el modelo, y de la distribución que la describe, podemos obtener distintos tipos de modelos estadísticos, los cuales requieren diferentes abordajes para sus análisis.

Una situación de uso habitual de los modelos estadísticos consiste en describir un conjunto y_i de valores experimentales que se obtienen registrando en cada observación x_i variables de control. Un elemento x_i puede o no ser relevante para predecir (o explicar) la respuesta y_i, por lo que un aspecto importante del modelado estadístico consiste en determinar cuáles son las variables independientes relevantes. Por supuesto, también puede suceder que haya factores relevantes que no hayan sido incluidos en el conjunto de variables independientes x_i, pero que aporten a la

determinación de y_i. En este caso puede no ser posible explicar con precisión los datos obtenidos con el modelo. La determinación de cuán precisa es la explicación de los datos con el modelo es otro aspecto importante del modelado estadístico.

1.2. Coeficiente de correlación

Al explorar conjuntos de datos de dos variables es natural preguntarse si existe alguna relación entre ellas. Por ejemplo, querríamos saber si al cambiar una de las variables, la otra también lo hace. Cuando detectamos que existe correlación entre las variables, es posible obtener una relación predictiva que puede resultar útil. Por ejemplo, la correlación existente entre la temperatura ambiente y el consumo de energía eléctrica permite anticipar que habrá un pico de consumo en días de calor extremo.

Si ambas variables están distribuidas normalmente, la medida mas común de dependencia entre estas cantidades es el coeficiente de correlación producto-momento de Pearson, o simplemente "*coeficiente de correlación de Pearson*". Para el caso de muestras representadas por pares $\{x_i, y_i\}$, este coeficiente se define como:

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

Si definimos la covariancia de la muestra como

$$s_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{n-1}$$

y s_x y s_y como las desviaciones estándar muestrales de x y y, respectivamente, el coeficiente de correlación r_{xy} se puede escribir como

$$r_{xy} = \frac{s_{xy}}{s_x \cdot s_y}$$

El coeficiente de correlación de Pearson resulta entonces una cantidad adimensional que puede tomar cualquier valor entre -1 y $+1$. Un coeficiente de correlación nulo indica que no existe una relación lineal entre las variables, mientras que los valores extremos indican correlación lineal perfecta. Si la correlación es un número positivo, las variables están directamente relacionadas (esto es, el aumento de una variable implica el aumento de la otra), mientras que si el número es negativo, las variables están inversamente relacionadas (cuando una aumenta la otra disminuye su valor). La figura 1.1 muestra varios conjuntos de puntos y el valor del coeficiente de correlación correspondiente.

Veremos un ejemplo de cálculo de coeficiente de correlación de Pearson analizando un conjunto de datos de atletas olímpicos recolectados desde los juegos de Atenas de 1896 hasta Río de Janeiro en 2016, con el propósito de obtener una estimación cuantitativa de las correlaciones entre el peso de los atletas y sus alturas y edades. Estos datos pueden descargarse libremente de kaggle[1], y analizaremos esta información utilizando las funcionalidades de Pandas.

En la celda 1 importamos los módulos requeridos para este ejemplo, y en la 2 generamos el *dataset* a partir del archivo de datos descargados.

[1] https://www.kaggle.com/datasets/heesoo37/120-years-of-olympic-history-athletes-and-results/download

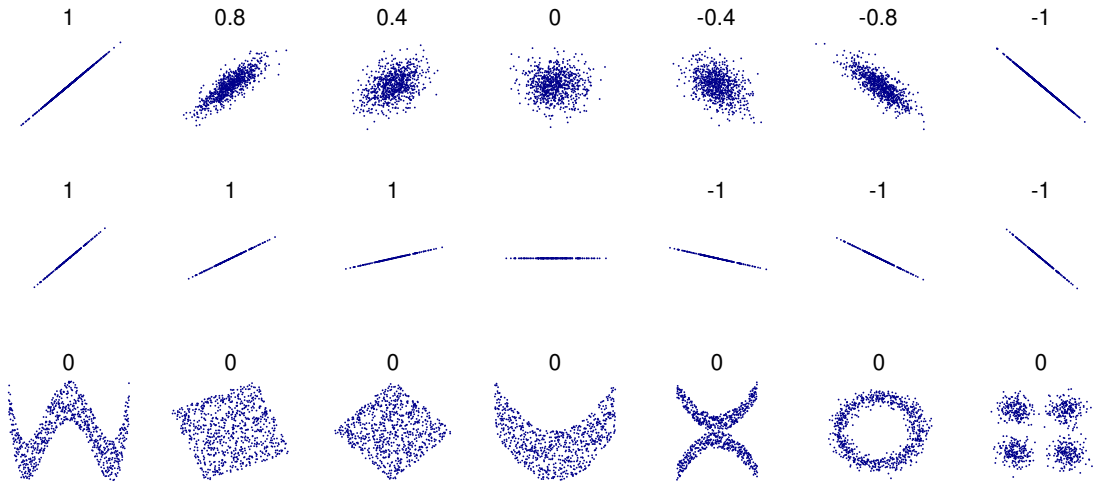


FIGURA 1.1: Distintos conjuntos de valores $\{x, y\}$ con el coeficiente de Pearson para cada caso. En la primera file puede notarse que el coeficiente refleja tanto la dirección de una relación lineal como el apartamiento de la misma, pero no la pendiente de la relación lineal (como se aprecia en la línea media, no tampoco relaciones no lineales (línea inferior). Notar que la figura en el centro tiene pendiente nula, pero el coeficiente de correlación es indefinido. Fuente: Wikipedia (accedida el 20.06.2022).

CELL 01

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy.stats import rankdata
```

CELL 02

```
olymp = pd.read_csv('athlete_events.csv', usecols=['Sex', 'Age', 'Height', 'Weight', 'Year',
                                                   'Sport', 'NOC']).dropna()
olymp.head()
```

```
  Sex   Age  Height  Weight  NOC  Year          Sport
0   M  24.0   180.0    80.0  CHN  1992     Basketball
1   M  23.0   170.0    60.0  CHN  2012           Judo
4   F  21.0   185.0    82.0  NED  1988  Speed Skating
5   F  21.0   185.0    82.0  NED  1988  Speed Skating
6   F  25.0   185.0    82.0  NED  1992  Speed Skating
```

En la celda 3 exploramos la estructura del *dataset*, y vemos que disponemos de 206.165 registros para analizar.

CELL 03

```
olymp.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 206165 entries, 0 to 271115
Data columns (total 7 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   Sex     206165 non-null  object
 1   Age     206165 non-null  float64
 2   Height  206165 non-null  float64
 3   Weight  206165 non-null  float64
 4   NOC     206165 non-null  object
 5   Year    206165 non-null  int64
 6   Sport   206165 non-null  object
dtypes: float64(3), int64(1), object(3)
memory usage: 12.6+ MB
```

Para inspeccionar la forma en que se agrupan los puntos tomando los pares altura-peso y edad-peso, los graficamos en forma de puntos:

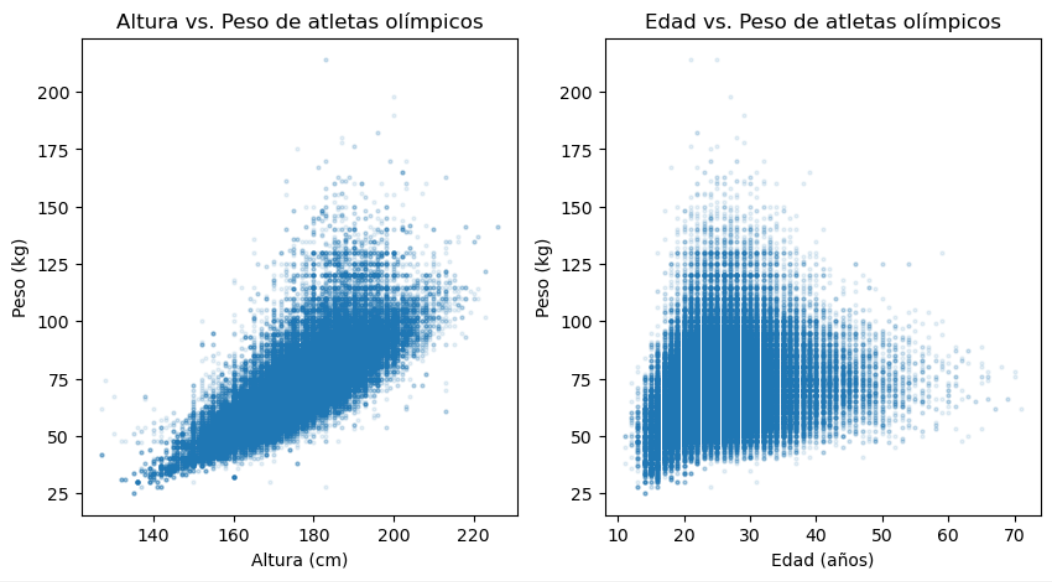
CELL 04

```
# Creamos los objetos figura y ejes
fig, ax = plt.subplots(1, 2,figsize=(10,5))

# Creamos los scatterplots
ax[0].plot(olymp['Height'], olymp['Weight'], marker='o', linestyle='',
        alpha=0.1, markersize=2)
ax[1].plot(olymp['Age'], olymp['Weight'], marker='o', linestyle='',
        alpha=0.1, markersize=2)

# Títulos y ejes
ax[0].set(title='Altura vs. Peso de atletas olímpicos',
      xlabel='Altura (cm)', ylabel='Peso (kg)')
ax[1].set(title='Edad vs. Peso de atletas olímpicos',
      xlabel='Edad (años)', ylabel='Peso (kg)')

plt.show()
```



Obtenemos el coeficiente de correlación de Pearson entre las variables peso, altura y edad

simplemente utilizando el método `corr()` de los *dataset* de Pandas:

CELL 05

```
olymp[['Weight', 'Height', 'Age']].corr()
```

```
           Weight    Height       Age
Weight   1.000000  0.796573  0.212041
Height   0.796573  1.000000  0.141684
Age      0.212041  0.141684  1.000000
```

Naturalmente, la correlación de una variable con si misma es perfecta, por lo que los elementos diagonales toman el valor uno. Tal como anticipa la figura de la celda 4, existe una mayor correlación entre la altura y peso de los atletas que entre la edad y el peso. Esta tabla muestra también una menor correlación aún entre la edad y la altura de los atletas.

El coeficiente de correlación de Pearson puede verse afectado por la existencia de valores extremos, que pueden exagerar o atenuar la intensidad de la relación lineal, y por lo tanto es inapropiado su uso cuando una o ambas variables no están normalmente distribuidas. Una de las formas más usadas para evaluar la correlación entre dos variables, tanto continuas como discretas, que obedecen a distribuciones sesgadas es la ρ de Spearman. Esta métrica utiliza el *orden* (o *ranking*) en que se presentan los datos, en vez de sus valores en sí. Por ejemplo, podemos crear un *dataframe* reducido del original conservando solo los primeros diez registros de las columnas peso y altura, y luego agregar tres columnas más conteniendo el orden de los valores peso y altura (generados con el método `rankdata()` de `scipy.stats`, y la diferencia d entre ambos valores para el mismo registro:

CELL 06

```
oly_10 = olymp.loc[:10, ['Weight', 'Height']]
oly_10['r_Weight'] = rankdata(oly_10['Weight'])
oly_10['r_Height'] = rankdata(oly_10['Height'])
oly_10['d'] = oly_10['r_Weight'] - oly_10['r_Height']
oly_10
```

```
    Weight  Height  r_Weight  r_Height    d
0     80.0   180.0       3.0       2.0  1.0
1     60.0   170.0       1.0       1.0  0.0
4     82.0   185.0       6.5       5.5  1.0
5     82.0   185.0       6.5       5.5  1.0
6     82.0   185.0       6.5       5.5  1.0
7     82.0   185.0       6.5       5.5  1.0
8     82.0   185.0       6.5       5.5  1.0
9     82.0   185.0       6.5       5.5  1.0
10    75.0   188.0       2.0       9.0 -7.0
```

Lo que muestra la celda 6 es que, para el registro 1, tanto el peso como la altura tienen el orden más bajo (1). Cuando los valores se repiten se toma el valor promedio del orden (en este caso, el valor de altura 185 ocupa los órdenes desde el 3 hasta el 8: $(3+4+5+6+7+8)/6 = 5{,}5$). Con esta diferencia de orden d se calcula el coeficiente de correlación de Spearman como:

$$\rho = -\frac{6\sum_{i=1}^{n} d_i^2}{n(n^2-1)}$$

donde d_i es la diferencia de orden entre los valores de x y y. Podemos obtener el coeficiente de correlación de Spearman para el *dataset* completo de los atletas olímpicos invocando nuevamente el método `corr()` de Pandas, pero esta vez indicando que el método que queremos utilizar para el cálculo es `'spearman'`:

CELL 07

```
olymp[['Weight', 'Height', 'Age']].corr(method='spearman')
```

```
           Weight    Height       Age
Weight   1.000000  0.827500  0.216952
Height   0.827500  1.000000  0.148001
Age      0.216952  0.148001  1.000000
```

El coeficiente de correlación de Spearman es más robusto ante la presencia de valores extremos, y es una medida del grado de correlación entre funciones monótonas (más que de la correlación lineal como el coeficiente de Pearson). También toma los valores extremos $+1$ y -1 cuando la correlación es perfecta (directa o inversa, respectivamente).

1.3. Definición de modelos estadísticos con *patsy*

Al realizar un modelado estadístico es necesario especificar la relación matemática entre las variables dependientes Y y las variables independientes X. Por ejemplo, en el caso de los modelos lineales, la variable Y puede expresarse como una combinación lineal de las variables independientes X, o de funciones de estas variables. Por ejemplo, podríamos proponer que esta relación es:

$$Y = \beta_1 X + \beta_2 X^3 + \cos(X)$$

Es importante notar que no necesariamente Y debe ser función lineal de X en los modelos lineales, sino que debe ser lineal respecto de los coeficientes desconocidos β_i. Por otro lado, el modelo $Y = \exp(\beta_0 + \beta_1 X)$ es un ejemplo de modelo no lineal, ya que en este caso Y no es una función lineal de β_0 y β_1. Sin embargo, podemos transformar este modelo en uno lineal tomando logaritmos en ambos miembros lo que conduce a $\widetilde{Y} = \beta_0 + \beta_1 X$ con $\widetilde{Y} = \log Y$. La clase de problemas que pueden transformarse en modelos lineales pueden ser tratados con el modelo lineal generalizado.

El paquete *patsy*[2] de Python provee el acceso a un mini lenguaje que permite describir fórmulas para representar relaciones entre variables dependientes e independientes, y está basado en la notación introducida por Wilkinson y Rogers en 1973[2]. Con este paquete, podemos escribir

$$Y \sim X$$

para representar la relación directa entre la variable Y y X. Podemos representar una relación más compleja en la que Y depende de las variables X, a y b de la siguiente manera:

$$Y \sim X + a + b + a : b$$

Los símbolos que se pueden utilizar para construir estas expresiones pueden verse en la Tabla 1.1. La descripción detallada y completa del lenguaje se puede ver en la documentación del paquete.

Una vez definida la relación funcional entre las variables que constituyen el modelo, el paso siguiente es construir las llamadas matrices de diseño. Para un modelo lineal simple $Y = X\beta + \epsilon$ son necesarias dos matrices: una con los valores de la variable dependendiente Y y otra con las variables dependientes del segundo miembro (más una columna opcional de unos que representan la ordenada al origen)[3]. Los elementos X_{ij} de la matriz de diseño X son los valores de funciones de las variables independientes correspondientes a cada coeficiente β_i y observación

[2] https://patsy.readthedocs.io/.

[3] Se suelen nombrar también *vector observación* y y dejar específicamente *matriz de diseño* para X. Usaremos estas notaciones indistintamente.

TABLA 1.1: Elementos principales de la sintaxis de fórmulas con *patsy*.

Operador	Significado
$\sim$	Separa el lado derecho del izquierdo en la expresión. Si se omite se asume que solo existe el lado derecho.
$+$	Combina términos en ambos lados de la expresión (unión de conjuntos).
$-$	Elimina términos en el lado derecho del conjunto de términos del lado izquierdo (diferencia de conjuntos).
:	Término de interacción (por ejemplo $a : b$ denota $a \cdot b$
$*$	$a * b$ es una forma abreviada de $a + b + a : b$
$/$	a/b es una forma abreviada de $a + a : b$
$**$	Toma un conjunto de términos en el lado izquierdo y un entero n en el derecho, y calcula el resultado de $*$ sobre el conjunto de términos con sí mismo n veces.
`f(x)`	Función arbitraria de Python (o NumPy) utilizada para transformar términos en la expresión (por ejemplo: `np.log(x)`
`C(x)`	Trata a la variable `x` como categórica

y_i. Por ejemplo, si los valores observados son $y = [1, 2, 3, 4, 5]$ para dos variables independientes con los valores correspondientes $x_1 = [6, 7, 8, 9, 10]$ y $x_2 = [13, 14, 15, 16, 17]$, y si el modelo en consideración es $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2$, la matriz de diseño para el lado derecho es $X = [\mathbf{1}, x_1, x_2, x_1 x_2]$. En las celdas del siguiente *notebook* podemos ver de qué forma construimos el modelo con *patsy* y obtenemos las matrices de diseño.

CELL 01

```
import numpy as np
import pandas as pd
from patsy import dmatrices, dmatrix
```

CELL 02

```
y = [1, 2, 3, 4, 5]
x1 = [6, 7, 8, 9, 10]
x2 = [13, 14, 15, 16, 17]
data = {'y': np.array(y), 'x1': np.array(x1), 'x2': np.array(x2)}
```

CELL 03

```
dmatrices("y ~ x1 + x2 + x1:x2", data)
```

```
(DesignMatrix with shape (5, 1)
   y
   1
   2
   3
   4
   5
   Terms:
     'y' (column 0),
 DesignMatrix with shape (5, 4)
   Intercept  x1  x2  x1:x2
           1   6  13    78
           1   7  14    98
           1   8  15   120
           1   9  16   144
           1  10  17   170
   Terms:
     'Intercept' (column 0)
     'x1' (column 1)
     'x2' (column 2)
     'x1:x2' (column 3))
```

En las celdas 1 y 2 importamos los módulos necesarios y definimos los valores de nuestras variables dependiente e independientes, respectivamente. En la celda 3 utilizamos la función `dmatrices()` para obtener las matrices de diseño correspondientes al modelo definido como argumento de dicha función, que se construyen utilizando `data` ingresado como segundo argumento. Si solo queremos la matriz de diseño sin los valores de y, podemos usar `dmatrix()` como se muestra en la celda siguiente:

CELL 04

```
X = dmatrix("x1 + x2 + x1:x2")
X
```

```
DesignMatrix with shape (5, 4)
  Intercept  x1  x2  x1:x2
          1   6  13    78
          1   7  14    98
          1   8  15   120
          1   9  16   144
          1  10  17   170
  Terms:
    'Intercept' (column 0)
    'x1' (column 1)
    'x2' (column 2)
    'x1:x2' (column 3)
```

La definición del modelo con *patsy* consiste en definir una cadena de caracteres que representa la relación matemática entre las variables (nótese que no figuran los coeficientes β_i en la fórmula ya que se asume implícitamente que cada témino de la fórmula tiene un parámetro del modelo como coeficiente), mientras que los datos utilizan una representación con diccionarios.

La matriz de diseño obtenida con `dmatrix` es un objeto del tipo `DesignMatrix` de *patsy*. Este objeto es completamente compatible con los arrays de NumPy, e incluso se muestra como tal como podemos ver a continuación:

CELL 05

```
print(type(X))
print(X)
```

```
<class 'patsy.design_info.DesignMatrix'>
[[  1.   6.  13.  78.]
 [  1.   7.  14.  98.]
 [  1.   8.  15. 120.]
 [  1.   9.  16. 144.]
 [  1.  10.  17. 170.]]
```

En forma alternativa, y dada la popularidad de Pandas para la representación de conjuntos de datos, podemos establecer el argumento `return_type` de `dmatrices` como `"dataframe"`. Además, dado que los objetos `DataFrame` se comportan como diccionarios, los podemos utilizar para definir los datos como segundo argumento de la función `dmatrices`:

CELL 06

```
df_data = pd.DataFrame(data)
y, X = dmatrices("y ~x1 + x2 + x1:x2", df_data, return_type="dataframe")
X
```

```
   Intercept    x1    x2  x1:x2
0        1.0   6.0  13.0   78.0
1        1.0   7.0  14.0   98.0
2        1.0   8.0  15.0  120.0
3        1.0   9.0  16.0  144.0
4        1.0  10.0  17.0  170.0
```

Una vez obtenidas las matrices de diseño con *patsy*, requeridas para resolver un modelo estadístico, podemos utilizar un método de mínimos cuadrados como el de NumPy `np.linalg.lstsq`:

CELL 07

```
beta, res, rank, sval = np.linalg.lstsq(X, y, rcond=None)
beta
```

```
array([[-3.33333333e-01],
       [ 1.66666667e+00],
       [-6.66666667e-01],
       [-4.44089210e-16]])
```

O también usar cualquiera de los muchos modelos estadísticos provistos por la biblioteca statsmodels.

1.4. Regresión lineal

El modelado estadístico suele involucrar un análisis interactivo de los datos, comenzando con una inspección visual, buscando correlaciones y relaciones funcionales. A partir de esta primera inspección, se propone un modelo estadístico que podría describir los datos. En los casos más simples, las relaciones entre los datos se puede describir con un modelo lineal:

$$Y = kX + d$$

A continuación se procede a determinar los parámetros del modelo (en este caso, k y d). Luego es necesario determinar la calidad del modelo, y finalmente se analizan los residuos (diferencias entre los valores observados y los predichos) para verificar si el modelo propuesto falla en describir algunas características de los datos.

Si los residuos presentan valores altos u *outliers*, o si sugieren una relación funcional diferente a la propuesta, es necesario modificar el modelo. Este procedimiento se repite hasta que los resultados sean satisfactorios.

La biblioteca statsmodels [3] provee diversos modelos estadísticos que comparten el mismo modo de uso, lo que facilita cambiar entre diferentes modelos. Estos modelos estadísticos están representados por clases, que pueden ser inicializadas usando tanto las matrices de diseño como expresiones en patsy, y un *dataframe* u otro objeto de tipo diccionario conteniendo los datos. Los pasos para establecer y analizar un modelo son los siguientes (asumiendo que importamos `statsmodels.api` como `sm` y `statsmodels.formula.api` como `smf`):

1. Crear una instancia de la clase modelo, por ejemplo, `modelo = sm.MODEL(Y, X)` o `modelo = smf.model(formula, data)`, utilizando las mayúsculas cuando usamos las matrices de diseño y las minúsculas para expresiones de patsy. `MODEL` o `model` denotan el nombre de un modelo particular como OLS (mínimos cuadrados ordinarios), GLS (mínimos cuadrados generalizados), etc.

2. Instanciar un modelo no realiza ningún cálculo. Para ajustar el modelo a los datos se utiliza el método `fit`, por ejemplo, `resultados = modelo.fit()`, que realiza el ajuste y devuelve un objeto que contiene atributos y métodos para análisis posteriores.

3. Inspeccionar el resumen de las estadísticas contenidas en el objeto resultado. Este objeto varía levemente entre diferentes modelos, pero la mayoría implementa el método `summary` que produce una salida que describe el resultado del ajuste.

4. Postprocesamiento de los resultados. Además del método `summary`, el objeto resultado tiene métodos y atributos para obtener los parámetros ajustados (`params`), los residuos (`resid`), los valores ajustados (`fittedvalues`) y un método para predecir los valores de las variables dependientes para nuevos valores de las variables independientes (`predict`).

5. Opcionalmente, suele ser útil visualizar el resultado del ajuste. Para ello se puede utilizar Matplotlib o algunas de las rutinas gráficas provistas por la bilbioteca statsmodels.

Recorreremos estos pasos utilizando un ejemplo en el cual conocemos la relación entre la variable dependiente y las independientes, y ajustaremos un modelo que genere una predicción cercana a estos datos a los que agregaremos ruido distribuido normalmente.

Vamos a considerar que la relación matemática que vincula las variables independientes x_1 y x_2 con la variable dependiente y está dada por

$$y = 4 + x_1 - 2x_2 + 3x_1x_2 \tag{1.1}$$

Para ello generaremos un conjunto de números aleatorios (estableciendo previamente un valor de la semilla del generador para poder reproducir los valores muestreados) y construiremos a partir de estos valores los correspondientes a la variable dependiente según la ecuación (1.1). En las siguientes dos celdas del *notebook* importamos primero los módulos necesarios y construimos un *dataframe* de Pandas conteniendo los valores de las variables dependientes x_1 y x_2 y la variable dependiente y:

CELL 01

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy import stats
import statsmodels.api as sm
import statsmodels.formula.api as smf
import statsmodels.graphics.api as smg
plt.rcParams.update({"text.usetex": True})
```

CELL 02

```
N = 100  # Cantidad de datos
np.random.seed(13)  # Semilla del generador de números aleatorios
x1 = np.random.randn(N)
x2 = np.random.randn(N)
data = pd.DataFrame({"x1": x1, "x2": x2})
def y(x1, x2):
    return 4 + x1 - 2 * x2 + 3 * x1 * x2
data["y_exacto"] = y(x1, x2)
```

En la celda siguiente generamos un array de números aleatorios (`epsilon`) que sumaremos a la variable dependiente, de modo de simular la presencia de "ruido" en los datos que queremos ajustar:

CELL 03

```
epsilon = 0.5 * np.random.randn(N)
data["y"] = data["y_exacto"] + epsilon
```

Construimos ahora un primer modelo en el que suponemos que la relación entre las variables independientes y la dependiente es solo la suma de las primeras, omitiendo el término que contiene el producto $x_1 x_2$ de la ecuación (1.1), es decir, $Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2$ que representamos con la fórmula de patsy `"y ~ x1 + x2"` en la que hacemos uso de los datos "con ruido" de la variable dependiente. Utilizaremos el método de mínimos cuadrados ordinarios (u OLS por su sigla en inglés: *Ordinary Least Squares*) para ajustar el modelo, para lo cual instanciaremos un objeto `modelo_1` de la clase `smf.ols`, e invocaremos al método `fit()` de ese objeto. Para ver el resultado del ajuste, invocaremos el método `summary()`:

CELL 04

```
modelo_1 = smf.ols("y ~ x1 + x2", data)
resultado_1 = modelo_1.fit()
print(resultado_1.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.094
Model:                            OLS   Adj. R-squared:                  0.075
Method:                 Least Squares   F-statistic:                     5.041
Date:                Wed, 08 Mar 2023   Prob (F-statistic):            0.00827
Time:                        12:01:15   Log-Likelihood:                -252.20
No. Observations:                 100   AIC:                             510.4
Df Residuals:                      97   BIC:                             518.2
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      4.1240      0.311     13.240      0.000       3.506       4.742
x1             0.5549      0.328      1.690      0.094      -0.097       1.207
x2            -1.0186      0.378     -2.693      0.008      -1.769      -0.268
==============================================================================
Omnibus:                       26.557   Durbin-Watson:                   2.067
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               86.445
Skew:                           0.811   Prob(JB):                     1.69e-19
Kurtosis:                       7.256   Cond. No.                         1.32
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

La descripción detallada de la salida de `summary()` excede el alcance de este libro, pero señalaremos algunos indicadores de la calidad del ajuste en la primera tabla. Entre ellos, `R-squared` es un estadístico que indica la proporción de variación en la variable dependiente debida a la variación en las variables independientes, y comprende el rango entre 0 y 1, tomando el valor 1 cuando el ajuste es perfecto. Además de estar informado en la salida de `summary()`, está contenido en el atributo `rsquared` de `resultado_1`, tal como se ve en la celda siguiente:

CELL 05

```
resultado_1.rsquared
```

```
0.09414676944969591
```

En este caso vemos que el ajuste es muy pobre, ya que apenas llega a $0{,}094$. El valor ajustado de R^2 (`Adj. R-squared`) es una corrección de R^2 que penaliza el número de parámetros en el modelo. Otros indicadores útiles para comparar la calidad de ajustes entre modelos son `AIC` (*Akaike Information Criterion*) y `BIC` (*Bayesian Information Criterion*). Estos dos indicadores no nos dicen qué tan bueno es un ajuste individual, pero son útiles para decidir entre diferentes modelos ya que resulta mejor el que tiene valores menores de estos indicadores.

La segunda tabla de `summary()` contiene los coeficientes β_i obtenidos por el ajuste del modelo. Estos coeficientes (o pesos) de la regresión lineal están contenidos en el atributo `params` de `resultado_1` y se devuelve como un objeto `Series` de pandas:

CELL 06

```
resultado_1.params
```

```
Intercept    4.124015
x1           0.554879
x2          -1.018580
dtype: float64
```

Asumiendo que los residuos están normalmente distribuidos, la columna `std err` ofrece una estimación de los errores estándar de los coeficientes del modelo, así como el estadístico t y su correspondiente p-valor para el test estadístico según el cual la hipótesis nula es que el correspondiente coeficiente es cero. Esta columna entonces (`P>|t|`) nos permite juzgar cuáles variables independientes tienen coeficientes con probabilidad de ser diferente de cero, lo que significa que tienen un poder predictivo significativo. La hipótesis alternativa es que el predictor contribuye a la respuesta. Como es usual, establecemos un umbral $\alpha = 0{,}05$ o $0{,}001$, y si el p-valor es menor que el umbral ($P(T \geq |t|) < \alpha$), podemos rechazar la hipótesis nula. El *t-test* nos permite evaluar la importancia de los diferentes predictores siempre que los residuos del modelo tengan una distribución normal alrededor de cero. Si esto no es el caso, el *t-test* sugiere la existencia de alguna no linealidad entre las variables y por lo tanto no se puede utilizar para estimar la importancia de los predictores individuales.

Para tener una idea más precisa sobre el supuesto de que los errores están normalmente distribuidos, es necesario analizar los residuos del modelo que ajustamos a los datos. Los errores estándar de los coeficientes son las raíces cuadradas de los elementos diagonales de la matriz de covarianza que resulta de

$$C = cov(\beta) = \sigma^2(\boldsymbol{X}\boldsymbol{X}^T)^{-1}$$

donde $\boldsymbol{X}$ es la matriz de diseño y σ^2 es la varianza, o error cuadrático medio, de los residuos.

Podemos analizar los residuos para justificar la suposición de distribución normal. Éstos son accesibles mediante el atributo `resid` del objeto `resultado_1`:

CELL 07

```
resultado_1.resid.head()
```

```
0    2.772770
1   -1.074751
2    0.563161
3   -0.057005
4   -0.949921
dtype: float64
```

Podemos invocar la función `normaltest()` de `scipy.stats` para verificar la hipótesis que los residuos constituyen una muestra de una variable aleatoria con distribución normal (hipótesis nula). Si establecemos un $\alpha = 0{,}001$, y calculamos el p-valor:

CELL 08

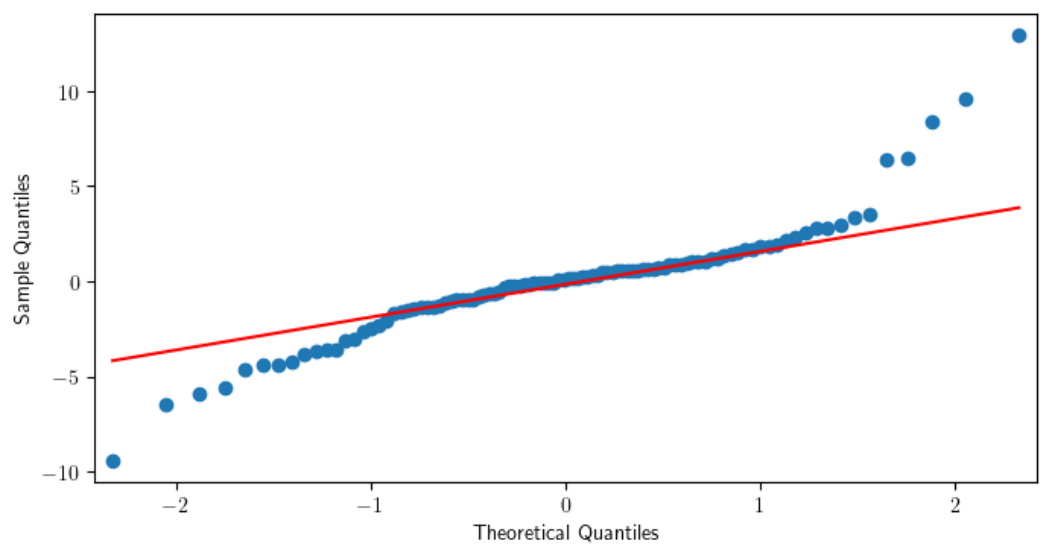
```
z, p = stats.normaltest(resultado_1.resid)
p
```

```
1.7109127008978512e-06
```

vemos que $p < \alpha$, con lo que podemos rechazar la hipótesis nula (es decir, podemos concluir que la suposición de que los residuos están normalmente distribuidos no se cumple). Una forma visual de verificar la normalidad de los residuos es a través de un gráfico Q-Q (*Quantile-Quantile plot*), que compara los cuantiles empíricos de la muestra con los valores teóricos, y que si los valores muestrales están normalmente distribuidos la figura debería aproximarse a una línea recta. Para nuestro caso, podemos generar el gráfico Q-Q mediante `smg.qqplot`:

CELL 09

```
fig, ax = plt.subplots(figsize=(8, 4))
smg.qqplot(resultado_1.resid, ax=ax, line='q');
```



Como se puede ver en la salida de la celda 9, los puntos se desvían notablemente de una recta, sugiriendo que es poco probable que los residuos observados sean una muestra de una variable aleatoria distribuida normalmente.

Por último, mencionamos que en la segunda tabla de la salida de `summary()` se muestran los intervalos de confianza para cada predictor. Un intervalo de confianza pequeño indica que el ajuste permite establecer con mayor precisión el valor de estos predictores, mientras que inter-

valos más grandes indican una mayor incerteza o varianza en estos parámetros. Al igual que los valores de `AIC` y `BIC`, los intervalos de confianza resultan útiles a la hora de comparar el grado de ajuste entre diferentes modelos.

Los resultados previos sugieren que el modelo propuesto no es suficiente y que necesitamos mejorarlo. Podemos entonces incluir el término de interacción en la fórmula de patsy y repetir los análisis:

CELL 10

```
modelo_2 = smf.ols("y ~ x1 + x2 + x1 : x2", data)
resultado_2 = modelo_2.fit()
print(resultado_2.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.972
Model:                            OLS   Adj. R-squared:                  0.971
Method:                 Least Squares   F-statistic:                     1119.
Date:                Wed, 08 Mar 2023   Prob (F-statistic):           1.57e-74
Time:                        12:01:15   Log-Likelihood:                -77.990
No. Observations:                 100   AIC:                             164.0
Df Residuals:                      96   BIC:                             174.4
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      3.9800      0.055     72.494      0.000       3.871       4.089
x1             0.9716      0.058     16.666      0.000       0.856       1.087
x2            -1.9678      0.069    -28.610      0.000      -2.104      -1.831
x1:x2          2.9938      0.054     55.075      0.000       2.886       3.102
==============================================================================
Omnibus:                        1.217   Durbin-Watson:                   2.304
Prob(Omnibus):                  0.544   Jarque-Bera (JB):                0.713
Skew:                           0.134   Prob(JB):                        0.700
Kurtosis:                       3.316   Cond. No.                         1.51
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

Ahora vemos que varios indicadores muestran un mejor ajuste del modelo. Para empezar, R^2 está más cerca de la unidad:

CELL 11

```
resultado_2.rsquared
```

```
0.972209883186191
```

También los valores de `AIC` y `BIC` son menores que en el caso anterior, indicando una mejora en la calidad del modelo. Si realizamos un *t-test* sobre los residuos obtenemos:

CELL 12

```
z, p = stats.normaltest(resultado_2.resid)
p
```

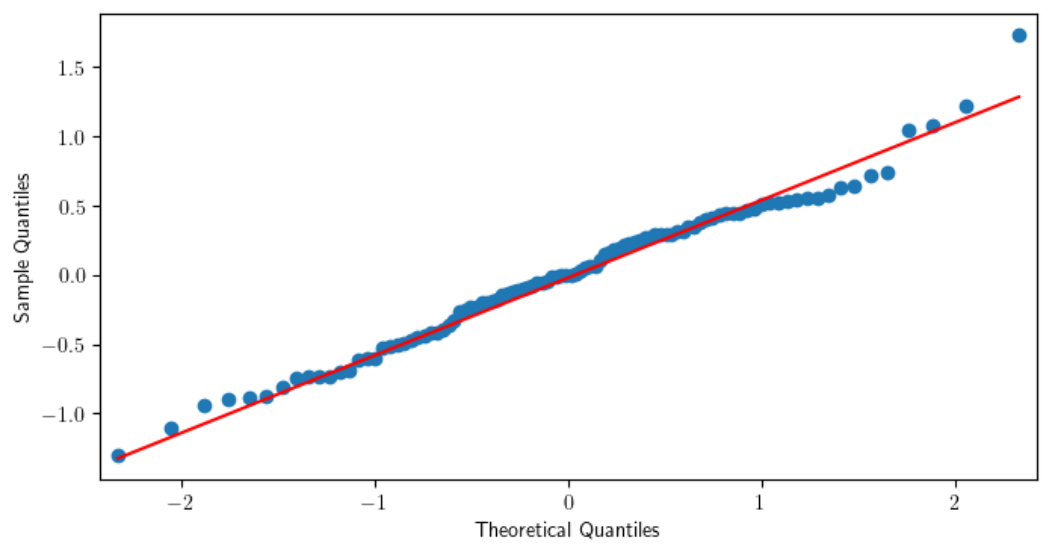
```
0.5442003027882543
```

y ahora el p-valor es mayor que nuestro umbral, con lo que no podemos rechazar la hipótesis nula que indica la normalidad de la distribución de residuos. Esto se visualiza con el gráfico Q-Q,

que se aproxima mejor a una recta:

CELL 13

```
fig, ax = plt.subplots(figsize=(8, 4))
smg.qqplot(resultado_2.resid, ax=ax, line='q');
```



Los parámetros de ajustes del modelo mejorado son:

CELL 14

```
resultado_2.params
```

```
Intercept    3.979993
x1           0.971593
x2          -1.967837
x1:x2        2.993805
dtype: float64
```

que son bastante cercanos a los modelos "verdaderos" con los cuales generamos los datos (sin el ruido agregado). Finalmente, podemos comparar los valores que se obtienen mediante la función correcta y los predichos por el modelo en forma gráfica:

CELL 15

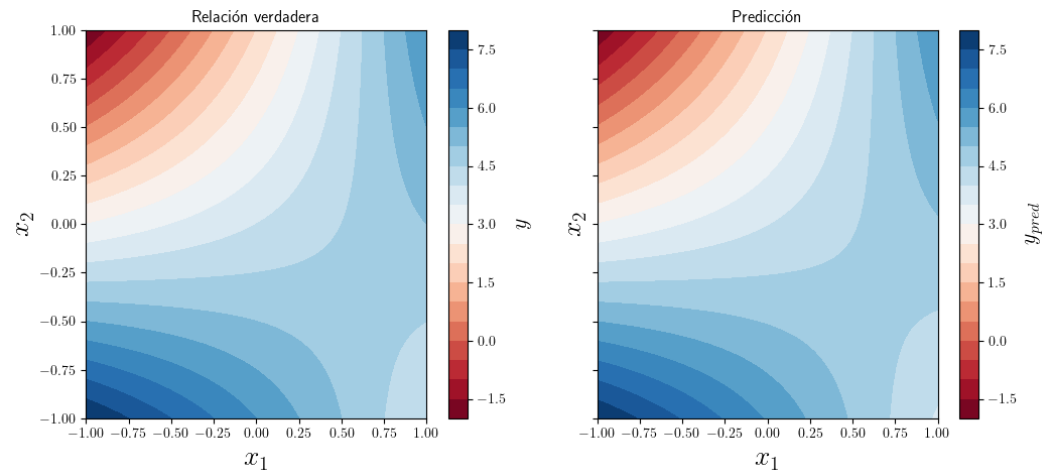
```
N_new = 100
x = np.linspace(-1, 1, N_new)
X1, X2 = np.meshgrid(x, x)
datos_nuevos = pd.DataFrame({"x1": X1.ravel(), "x2": X2.ravel()})
y_pred = resultado_2.predict(datos_nuevos)
```

CELL 16

```
y_pred = y_pred.values.reshape(N_new, N_new)
fig, axes = plt.subplots(1, 2, figsize=(12, 5), sharey = True)

def plot_y_contour(ax, Y, title, bar_title):
    c = ax.contourf(X1, X2, Y, 20, cmap=plt.cm.RdBu)
    ax.set_xlabel(r"$x_1$", fontsize=20)
    ax.set_ylabel(r"$x_2$", fontsize=20)
    ax.set_title(title)
    cb = fig.colorbar(c, ax=ax)
    cb.set_label(bar_title, fontsize=16)

plot_y_contour(axes[0], y(X1, X2), "Relación verdadera", r"$y$")
plot_y_contour(axes[1], y_pred, "Predicción", r"$y_{pred}$")
```



En el panel de la izquierda mostramos los valores obtenidos utilizando la relación inicial verdadera entre las variables independientes x_1 y x_2, mientras que en el panel de la derecha visualizamos la predicción. De este modo podemos cuantificar en forma visual el grado de precisión de nuestro modelo.

1.5. Regresión discreta

Los modelos de regresión discreta son aquellos en los que la variable dependiente toma valores discretos. El más simple de los casos es el que la variable dependiente y es binaria (solo puede tomar dos valores, que por conveniencia y sin perder generalidad, denotamos por 0 y 1). Por ejemplo, podemos definir que $y = 1$ si un individuo es adulto y $y = 0$ si no lo es; 1 si hace deportes y 0 en caso contrario; 1 si un vehículo tiene motor eléctrico y 0 si no; etc.

Cuando la variable dependiente toma más de dos valores, podemos clasificar los diferentes casos en variables categóricas y no categóricas. Como ejemplo del primer caso, podemos categorizar individuos según su rango etario:

- $y = 1$ si la edad es menor a 20 años,
- $y = 2$ si la edad es entre 20 y 60 años,
- $y = 3$ si la edad es mayor a 60 años,

Como ejemplo de una variable no categórica podemos mencionar el número de unidades vendidas por una empresa durante un día. En este caso y puede tomar los valores $0, 1, 2, \ldots$, es decir, valores discretos sin ser una variable categórica.

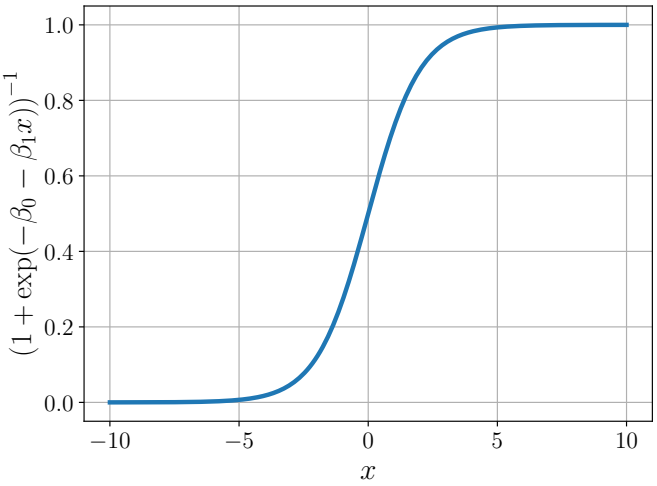


FIGURA 1.2: Representación gráfica de la función logística con los valores $\beta_0 = 0$ y $\beta_1 = 1$.

Los métodos usuales de regresión lineal que vimos hasta ahora no son suficientes, dado que en esos casos la variable de respuesta obtenida es continua, y no puede ser utilizada directamente para modelar una variable que solo puede tomar valores discretos. Sin embargo, es posible mapear un predictor lineal mediante una transformación adecuada a un intervalo que se puede interpretar como una probabilidad para diferentes resultados discretos. Para el caso de variables de respuesta binaria, una transformación muy utilizada es la función logística:

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x$$

o

$$p = (1 + \exp(-\beta_0 - \beta_1 x))^{-1}$$

que mapea $x \in (-\infty, \infty)$ a $p \in (0, 1)$, y cuya representación gráfica se muestra en la figura 1.2. Es decir, que la variable independiente (continua o discreta) x es mapeada a través de los parámetros del modelo lineal β_0 y β_1, y posteriormente utilizando la función logística transformamos la salida del modelo lineal en una probabilidad p. Si $p < 0{,}5$ podemos decir que $y = 0$, y si $p \geq 0{,}5$, la predicción es que $y = 1$.

La biblioteca statsmodels ofrece varios métodos para regresiones discretas, incluyendo la clase `Logit` para regresiones binarias, la clase para regresión logística multinomial `MNLogit` (para más de dos categorías), y la clase `Poisson` para variables no categóricas con distribución de Poisson (enteros positivos). Vamos a desarrollar un ejemplo de regresión discreta binaria utilizando el famoso conjunto de datos de flor Iris de Fisher[4]. Este conjunto contiene 50 muestras de cada una de las tres especies de Iris: setosa, virginica e versicolor. En cada muestra se midieron el largo y el ancho del sépalo y del pétalo (en centímetros). En el ejemplo, tomaremos un subconjunto de los datos pertenecientes solo a dos especies e intentaremos construir un modelo que prediga, en términos de los valores de las dimensiones del sépalo y del pétalo, a cuál de las dos especies corresponden esos valores.

Como es habitual, en la primera celda del *jupyter-notebook* importamos los módulos necesarios. A continuación, utilizamos el submódulo `datasets` de statsmodels cuya función `get_rdataset` provee acceso a conjuntos de datos utilizados en el popular lenguaje R[5].

[4]Se puede leer más sobre este conjunto de datos en la entrada de Wikipedia.

[5]Los conjuntos de datos están disponibles en `https://vincentarelbundock.github.io/Rdatasets/`.

CELL 01

```
import matplotlib.pyplot as plt
%config InlineBackend.figure_formats = ['svg']
import numpy as np
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf
from scipy import stats
plt.rcParams.update({"text.usetex": True})
```

CELL 02

```
df = sm.datasets.get_rdataset("iris").data
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   Sepal.Length  150 non-null    float64
 1   Sepal.Width   150 non-null    float64
 2   Petal.Length  150 non-null    float64
 3   Petal.Width   150 non-null    float64
 4   Species       150 non-null    object
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
```

Extrayendo los valores únicos en la columna `Species` de nuestro *dataset*, vemos que están las tres especies de Iris:

CELL 03

```
df.Species.unique()
```

```
array(['setosa', 'versicolor', 'virginica'], dtype=object)
```

Construimos ahora un nuevo *dataframe* que constituye un subconjunto del original, incluyendo solo las especies versicolor y setosa, y reasignamos los valores de la columna `Species` a los valores `0` o `1` para versicolor y virginica respectivamente. Finalmente, antes de proceder con la construcción del modelo, es necesario renombrar las columnas del *dataframe* reemplazando los puntos por otro carácter de modo que Python pueda interpretar correctamente su nombre. Si pasamos como nombre de variable en expresiones Patsy `"Sepal.Length"`, Python intentará encontrar una variable llamada `Sepal` que tiene un atributo `Length`. Evitamos esto reemplazando los puntos por, por ejemplo, el carácter `"_"`:

CELL 04

```
df_subconjunto = df[df.Species.isin(["versicolor", "virginica"])].copy()
df_subconjunto.Species = df_subconjunto.Species.map({"versicolor": 0, "virginica": 1})
df_subconjunto.columns = [c.replace('.', '_') for c in df_subconjunto.columns]
df_subconjunto.head()
```

	Sepal_Length	Sepal_Width	Petal_Length	Petal_Width	Species
50	7.0	3.2	4.7	1.4	0
51	6.4	3.2	4.5	1.5	0
52	6.9	3.1	4.9	1.5	0
53	5.5	2.3	4.0	1.3	0
54	6.5	2.8	4.6	1.5	0

Podemos construir ahora el modelo utilizando la clase `Logit`, y realizamos el ajuste con su

método `fit()` utilizando como variables independientes las medidas del pétalo (largo y ancho):

CELL 05

```
modelo = smf.logit("Species ~ Petal_Length + Petal_Width", data=df_subconjunto)
resultado = modelo.fit()
```

```
Optimization terminated successfully.
         Current function value: 0.102818
         Iterations 10
```

Como ya vimos, los resultados del ajuste se pueden mostrar con el método `summary()`:

CELL 06

```
print(resultado.summary())
```

```
                           Logit Regression Results
==============================================================================
Dep. Variable:                Species   No. Observations:                  100
Model:                          Logit   Df Residuals:                       97
Method:                           MLE   Df Model:                            2
Date:                Wed, 22 Feb 2023   Pseudo R-squ.:                  0.8517
Time:                        17:34:37   Log-Likelihood:                -10.282
converged:                       True   LL-Null:                       -69.315
Covariance Type:            nonrobust   LLR p-value:                 2.303e-26
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept    -45.2723     13.612     -3.326      0.001     -71.951     -18.594
Petal_Length   5.7545      2.306      2.496      0.013       1.235      10.274
Petal_Width   10.4467      3.756      2.782      0.005       3.086      17.808
==============================================================================

Possibly complete quasi-separation: A fraction 0.34 of observations can be
perfectly predicted. This might indicate that there is complete
quasi-separation. In this case some parameters will not be identified.
```

Al igual que en los ejemplos precedentes, no haremos un análisis exhaustivo de toda la información que muestra `summary()` sino que solo centraremos nuestra atención en los parámetros del ajuste (`Intercept`, `Petal_Length` y `Petal_Width`) ya que éstos nos permitirán realizar predicciones sobre nuevos valores de mediciones del largo y ancho del pétalo. Podemos simular nuevas mediciones generando valores con variables aleatorias dentro de los rangos de variación de estas magnitudes, los cuales registramos en un nuevo *dataframe* `df_nuevo`. Utilizando el ajuste del modelo podemos predecir a qué especie pertenece cada "nuevo" valor, tomando el criterio de asignar el valor `1` o `0` según la predicción sea mayor o menor a 0,5:

CELL 07

```
n_datos = 20
amplitud = 1.2
np.random.seed(1729)
df_nuevo = pd.DataFrame({"Petal_Length": np.random.randn(n_datos) * amplitud + 5,
                         "Petal_Width": np.random.rand(n_datos) * amplitud + 1.1})
df_nuevo["Species_pred"] = resultado.predict(df_nuevo)
print(df_nuevo["Species_pred"].head(3))
df_nuevo["Species_pred"] = (df_nuevo["Species_pred"] > 0.5).astype(int)
df_nuevo["Species_pred"].head(3)
```

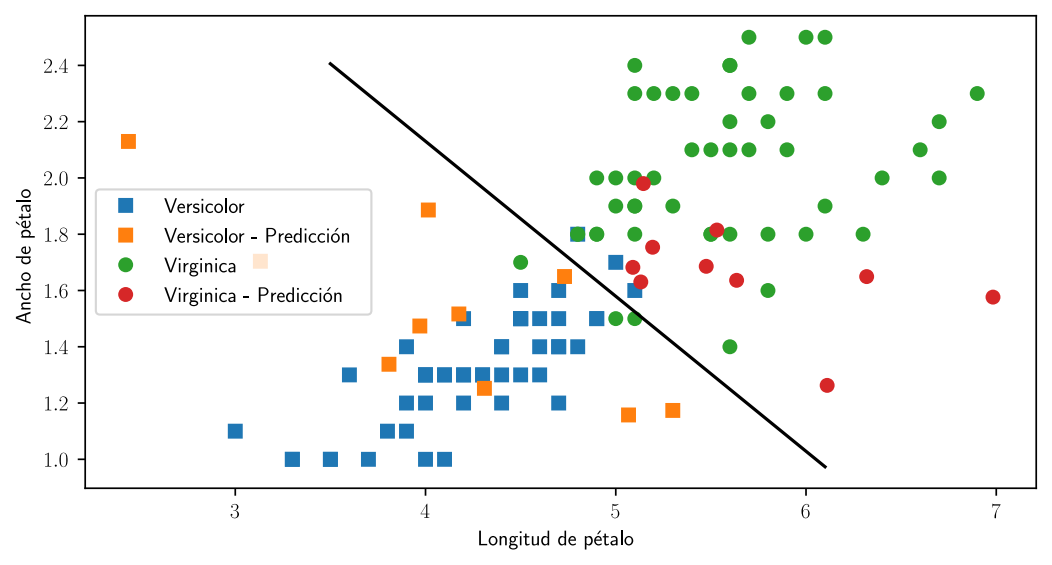
```
0    0.004483
1    0.078261
2    0.999989
Name: Species_pred, dtype: float64

0    0
1    0
2    1
Name: Species_pred, dtype: int64
```

Para tener una apreciación visual del desempeño del modelo en la clasificación de las especies y en la predicción de los resultados, podemos realizar un gráfico que muestre los valores originales (de los cuales conocemos a qué especie pertenecen), la recta que separa ambas clasificaciones, y las predicciones sobre los nuevos valores obtenidos aleatoriamente:

CELL 08

```
parametros = resultado.params
# Parámetros de la recta que define la separación de especies
a = -parametros['Intercept']/parametros['Petal_Width']
b = -parametros['Petal_Length']/parametros['Petal_Width']
fig, ax = plt.subplots(1, 1, figsize=(8, 4))
ax.plot(df_subconjunto[df_subconjunto.Species == 0].Petal_Length.values,
        df_subconjunto[df_subconjunto.Species == 0].Petal_Width.values,
        's', label='Versicolor')
ax.plot(df_nuevo[df_nuevo.Species_pred == 0].Petal_Length.values,
        df_nuevo[df_nuevo.Species_pred == 0].Petal_Width.values,
        's', label='Versicolor - Predicción')
ax.plot(df_subconjunto[df_subconjunto.Species == 1].Petal_Length.values,
        df_subconjunto[df_subconjunto.Species == 1].Petal_Width.values,
        'o', label='Virginica')
ax.plot(df_nuevo[df_nuevo.Species_pred == 1].Petal_Length.values,
        df_nuevo[df_nuevo.Species_pred == 1].Petal_Width.values,
        'o', label='Virginica - Predicción')
x = np.array([3.5, 6.1])
ax.plot(x, a + b * x, 'k')
ax.set_xlabel('Longitud de pétalo')
ax.set_ylabel('Ancho de pétalo')
plt.legend()
plt.show()
```



Vemos que algunos valores originales próximos a la recta de separación se encuentran en el lado erróneo de la clasificación, pero las predicciones sobre los valores generados aleatoriamente son muy buenas.

Como último ejemplo de modelado sobre variables discretas, intentaremos ahora realizar un ajuste sobre un conjunto de datos que registra el número de visitas al médico en las últimas dos semanas para una muestra de 5.190 adultos de la Encuesta de Salud Australiana 1977-78[6]. Este y otros conjuntos de datos acerca del uso de servicios de salud fueron objeto de análisis por A. Cameron y P. Trivedi en el contexto de un modelo económico [4].

El modelo discreto más simple que podemos considerar para modelar datos que representan conteos es el de Poisson, que expresa la probabilidad de un número dado de eventos que ocurren en un intervalo de tiempo (o espacio) fijo, en el caso en que estos eventos ocurran con una fre-

[6]El conjunto de datos se puede descargar desde este enlace.

cuencia media constante e independientemente del intervalo de tiempo desde el último evento. Si suponemos que los eventos ocurren aleatoriamente en el tiempo de modo que se cumplan las siguientes condiciones:

- La probabilidad de ocurrencia de al menos un evento en un intervalo dado es proporcional a la longitud del intervalo.
- La probabilidad de dos o más ocurrencias de eventos en un intervalo muy pequeño es despreciable.
- El número de ocurrencias de eventos en intervalos de tiempos disjuntos son mutuamente independientes.

entonces la distribución de probabilidad del número de ocurrencias de eventos en un intervalo de tiempo fijo es una distribución de Poisson con media $\mu = \lambda\, t$, donde λ es la frecuencia de ocurrencia del evento por unidad de tiempo y t es la longitud del intervalo de tiempo. Un proceso que satisface las tres condiciones mencionadas se denomina proceso de Poisson. Ejemplos modelados como procesos de Poisson que podemos mencionar son: la cantidad de goles anotados en un partido de fútbol, potenciales de acción emitidos por una neurona o la cantidad de vehículos que pasan por una calle.

Formalmente, si Y_i denota el número de ocurrencias para el i-ésimo de N individuos, de un evento de interés en un intervalo dado de tiempo, la densidad de probabilidad está dada por:

$$P(Y_i = y_i) = \frac{\lambda_i^{y_i} e^{-\lambda_i}}{y_i!}, \quad y_i = 0, 1, 2, \ldots, \quad i = 1, 2, \ldots, N$$

donde y_i es el valor obtenido de la variable aleatoria. Esta es una distribución con un solo parámetro en la cual el valor medio y varianza de Y_i son ambos iguales a λ_i. Para incoporar un conjunto de variables independientes $X_{ij}(j = 1, \ldots, K)$, incluyendo un término constante, el parámetro λ resulta:

$$\lambda_i = \exp(\boldsymbol{X}_i \boldsymbol{\beta})$$

Volviendo al ejemplo, vemos en la celda 9 que leemos el archivo `DoctorVisits.csv` y almacenamos sus datos en el *dataframe* `df`. Invocando el método `info()` podemos ver los nombres de las columnas, la cantidad de valores registrados y los tipos de datos. En la celda siguiente podemos ver que la columna `'visits'` contiene 10 valores y la cantidad de veces que aparece cada uno de ellos en los datos.

CELL 09

```
df = pd.read_csv('DoctorVisits.csv', header=0, infer_datetime_format=True, parse_dates=[0],
                 index_col=[0])
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5190 entries, 1 to 5190
Data columns (total 12 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   visits     5190 non-null   int64
 1   gender     5190 non-null   object
 2   age        5190 non-null   float64
 3   income     5190 non-null   float64
 4   illness    5190 non-null   int64
 5   reduced    5190 non-null   int64
 6   health     5190 non-null   int64
 7   private    5190 non-null   object
 8   freepoor   5190 non-null   object
 9   freerepat  5190 non-null   object
 10  nchronic   5190 non-null   object
 11  lchronic   5190 non-null   object
dtypes: float64(2), int64(4), object(6)
memory usage: 527.1+ KB
```

CELL 10

```
df['visits'].value_counts()
```

```
0    4141
1     782
2     174
3      30
4      24
7      12
6      12
5       9
8       5
9       1
Name: visits, dtype: int64
```

En la celda 11 instanciamos el objeto `modelo` especificando que es un modelo de Poisson en el cual la columna `'visits'` del *dataframe* constituye nuestra variable dependiente, a la que queremos explicar solo a través de una constante, sin relacionarla aún con otros campos del *dataframe*. A continuación realizamos el ajuste del modelo con el método `fit()` asignándolo al objeto `resultado`, y mostramos los resultados del dicho ajuste mediante el método `summary()`:

Cell 11

```
modelo = smf.poisson("visits ~ 1", data=df)
resultado = modelo.fit()
resultado.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.767475
         Iterations 1

<class 'statsmodels.iolib.summary.Summary'>
"""
                          Poisson Regression Results
==============================================================================
Dep. Variable:                 visits   No. Observations:                 5190
Model:                         Poisson   Df Residuals:                     5189
Method:                            MLE   Df Model:                            0
Date:                Wed, 22 Feb 2023   Pseudo R-squ.:              -2.220e-16
Time:                        17:34:40   Log-Likelihood:                -3983.2
converged:                       True   LL-Null:                       -3983.2
Covariance Type:            nonrobust   LLR p-value:                       nan
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -1.1982      0.025    -47.416      0.000      -1.248      -1.149
==============================================================================
"""
```

El resultado que estamos buscando aquí es el valor de `Intercept`, que es el que podemos vincular con el parámetro λ de la distribución. En la celda siguiente calculamos este parámetro y generamos tres distribuciones de Poisson: una con el parámetro recién obtenido, y dos utilizando los valores extremos del intervalo de confianza del 95 % de `Intercept` informados en la celda 11:

Cell 12

```
lam = np.exp(resultado.params)
print(lam)
X = stats.poisson(lam)
X_ci_l = stats.poisson(np.exp(resultado.conf_int().values)[0,0])
X_ci_u = stats.poisson(np.exp(resultado.conf_int().values)[0,1])
```

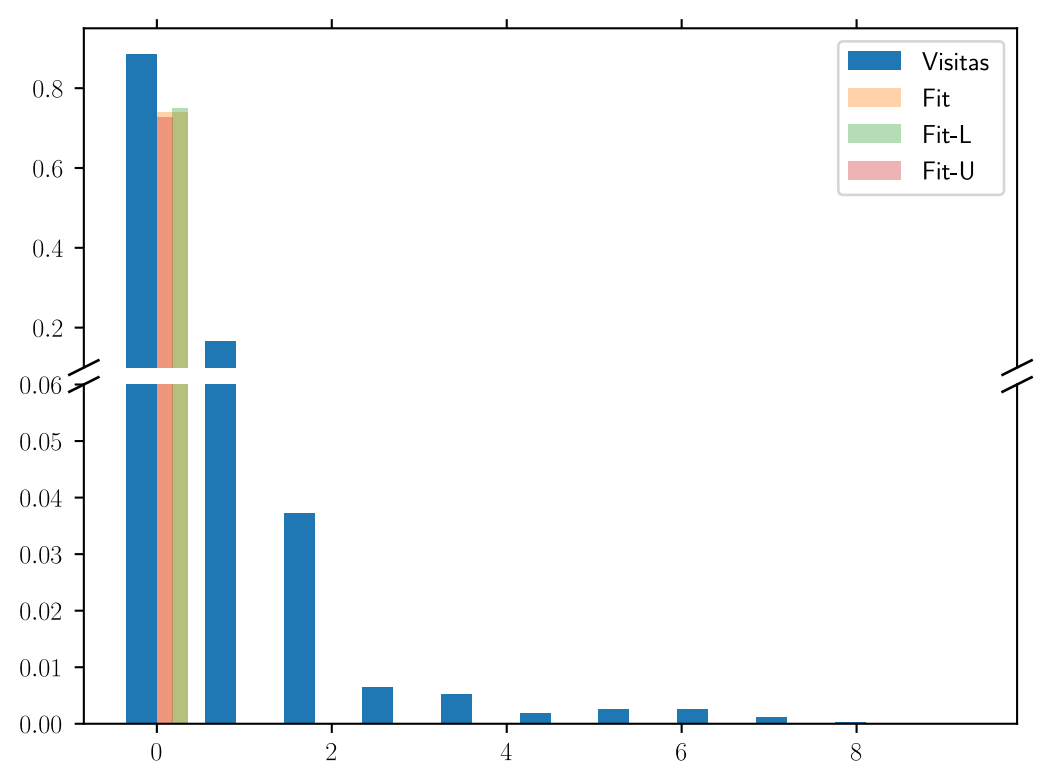
```
Intercept    0.301734
dtype: float64
```

Utilizando estas tres distribuciones, compararemos el histograma de visitas de los datos (normalizado con el número total de visitas) con las funciones de masa de probabilidad de las distribuciones, obtenidas mediante el método `pmf()`[7]. Generamos esta comparación en forma gráfica, como se ve en la celda a continuación:

[7] La función de masa de probabilidad devuelve la probabilidad de que una variable aleatoria discreta sea exactamente igual a un valor dado.

CELL 13

```
v, k = np.histogram(df.visits, density=True)
fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
fig.subplots_adjust(hspace=0.05)
width = 0.35
for ax in (ax1, ax2):
    ax.bar(k[:-1]-width/2, v, width, align='center', label='Visitas',alpha=1)
    ax.bar(k+width/2, X.pmf(k), width, align='center', label='Fit',alpha=0.35)
    ax.bar(k+width/2, X_ci_l.pmf(k), width/2, align='edge', label='Fit-L',alpha=0.35)
    ax.bar(k+width/2, X_ci_u.pmf(k), -width/2, align='edge', label='Fit-U',alpha=0.35)
ax1.set_ylim(0.1, 0.95)
ax2.set_ylim(0, 0.06)
ax1.spines.bottom.set_visible(False)
ax2.spines.top.set_visible(False)
ax1.xaxis.tick_top()
ax1.tick_params(labeltop=False)
ax2.xaxis.tick_bottom()
d = .5  # proporción de extensión vertical a horizontal de la marca de separación
kwargs = dict(marker=[(-1, -d), (1, d)], markersize=12,
              linestyle="none", color='k', mec='k', mew=1, clip_on=False)
ax1.plot([0, 1], [0, 0], transform=ax1.transAxes, **kwargs)
ax2.plot([0, 1], [1, 1], transform=ax2.transAxes, **kwargs)
ax1.legend()
plt.show();
```



Podemos ver que los valores del histograma no están comprendidos entre las distribuciones con los límites superior e inferior del intervalo de confianza del parámetro ajustado, por lo que debemos rechazar la hipótesis de que el número de visitas al médico en las últimas dos semanas es un proceso de Poisson. No tener éxito en el ajuste de un modelo a un conjunto de datos es habitual durante el proceso de modelado estadístico, y tal vez podríamos obtener mejores resultados si intentamos explicar nuestra variable dependiente con algunas de las columnas del *dataframe*. Por ejemplo, podríamos reemplazar la expresión del modelo en la celda 11 por

`"visits ~ 1 + age + illness * income"`. Dejamos aquí al lector la tarea de explorar estas posibilidades, que resultan bastante más complejas que el caso que usamos como ejemplo.

1.6. Series temporales

Una serie temporal consiste en el registro de datos sucesivos en el tiempo. Las técnicas utilizadas para analizar series temporales tratan de identificar patrones de los datos en el pasado e intentan anticipar (o predecir) los valores futuros. Estos datos se encuentran ordenados cronológicamente:

$$y_0, y_1, \ldots, y_{t-2}, y_{t-1}, y_t$$

El análisis de series temporales presenta una característica distintiva respecto de los análisis realizados hasta aquí: no se puede considerar que un conjunto de observaciones de una serie temporal representen una muestra de valores aleatorios de una población, ya que existe una fuerte correlación entre observaciones que están cercanas en el tiempo. Por otra parte, se pueden identificar los valores futuros de una serie temporal como las variables dependientes, que son predichas por los valores pasados de la serie (variables independientes). Esto se refuerza cuando existen autocorrelaciones como ciclos diarios o anuales, o fuertes tendencias que se mantienen en el tiempo. Ejemplos de series temporales son las observaciones climáticas, orográficas, poblacionales, precios de acciones, etc.

El modelo más simple que podemos considerar es el autoregresivo AR (*"autoregressive model"*), que utiliza una combinación lineal de p valores pasados para estimar el presente. La expresión matemática para el modelo AR(p) es:

$$y_t = \phi_0 + \sum_{n=1}^{p} \phi_n y_{t-n} + \varepsilon_t$$

Aquí, p es el orden del modelo y ε_t representa ruido blanco sin correlación. Los modelos autoregresivos se aplican a procesos estacionarios, es decir, aquellos en los cuales sus propiedades estadísticas no varían en el tiempo. Esta condición impone restricciones sobre los valores que pueden tener los coeficientes ϕ_i.

Otro algoritmo estadístico simple para modelar series temporales es el de la media móvil MA (*"moving average"*), que utiliza la dependencia entre una observación dada y los residuos calculados a partir de observaciones previas. Estos residuos consisten en la diferencia entre los valores registrados en el instante t y el promedio (o media móvil) de un subconjunto precedente de q datos. Matemáticamente el modelo MA(q) se expresa como:

$$y_t = \mu + \sum_{n=1}^{q} \theta_n \varepsilon_{t-n} + \varepsilon_t$$

donde θ_i son los parámetros del modelo, μ es la media de la serie y ε_i los residuos.

Estos abordajes pueden combinarse dando origen al modelo ARMA(p, q), pero tal como mencionamos, solo puede aplicarse a series que sean estacionarias. Vamos a intentar modelar ahora una serie temporal que registra mediciones atmosféricas de CO_2 desde 1958 en el Observatorio Mauna Loa [5], en partes por millón (ppm). Para ello utilizaremos las herramientas disponibles en el módulo statsmodels.

Como es usual, iniciamos el *notebook* importando los módulos necesarios:

CELL 01

```
import pandas as pd
import matplotlib.pyplot as plt
plt.rcParams.update({"text.usetex": True})
%config InlineBackend.figure_formats = ['svg']
import statsmodels.api as sm
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.api import ARIMA, SARIMAX
import warnings
warnings.filterwarnings("ignore")
import itertools
```

En la celda 2 leemos los datos desde los *datasets* disponibles en statsmodels, podemos ver que obtenemos un *dataframe* de pandas con 2284 registros, ordenados por fecha:

CELL 02

```
data = sm.datasets.co2.load_pandas()
co2 = data.data
print(co2.info())
print(co2.head())
```

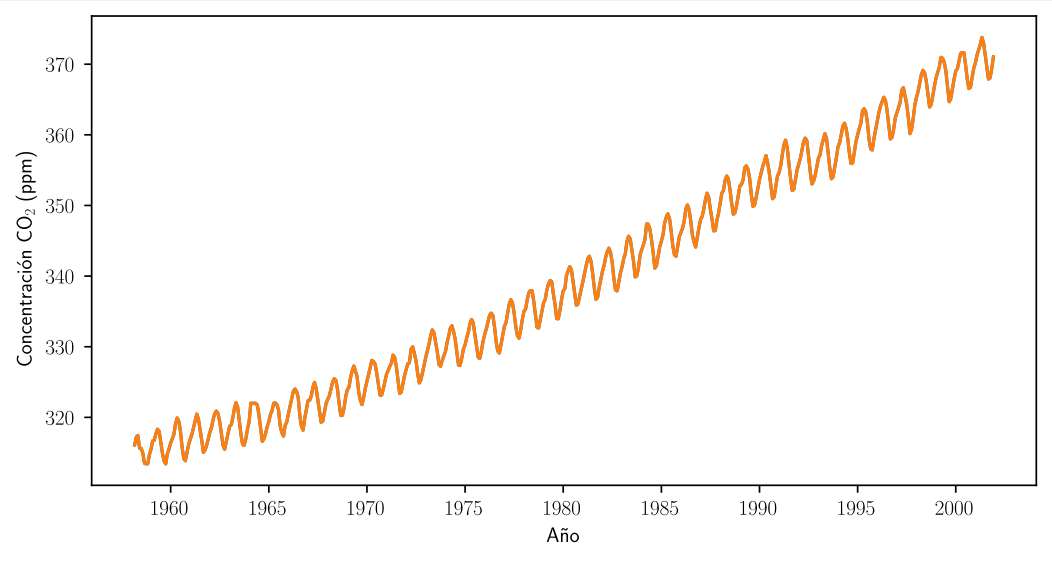
```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2284 entries, 1958-03-29 to 2001-12-29
Freq: W-SAT
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   co2     2225 non-null   float64
dtypes: float64(1)
memory usage: 35.7 KB
None
              co2
1958-03-29  316.1
1958-04-05  317.3
1958-04-12  317.6
1958-04-19  317.5
1958-04-26  316.4
```

Antes de comenzar el análisis, vamos a transformar nuestra serie temporal original en otra cuyos índices temporales sean el primer día de cada mes conteniendo el valor medio de los datos del mes correspondiente. Además buscamos fechas sin registros (encontramos 5) y completamos esos registros artificialmente con el valor precedente. Finalmente graficamos la serie temporal:

CELL 03

```
y = co2['co2'].resample('MS').mean()
print(y.isnull().sum())
y = y.fillna(method='backfill')
fig, ax = plt.subplots(1, 1, figsize=(8, 4))
ax.plot(y)
plt.xlabel('Año')
plt.ylabel(r'Concentración CO$_2$ (ppm)')
plt.plot(y);
```

```
5
```



De este modo, finalizamos con una serie temporal con 526 valores. En la figura se observa claramente que existe una tendencia creciente de la concentración de CO_2 con el tiempo, a la que se le superpone una oscilación estacional. Además de la inspección visual, es útil disponer de una herramienta cuantitativa para determinar la estacionalidad de una serie temporal. En nuestro caso, haremos uso del *test* de Dickey-Fuller aumentado[8] cuya hipótesis nula consiste en que la serie no es estacionaria, por lo que al obtener el p-valor del *test* por encima del valor $0{,}05$ no podemos asegurar que la serie no es estacionaria con el 95 % de confianza. En la celda siguiente definimos una función que calcular este p-valor y nos informa para distintos valores críticos:

CELL 04

```
def check_stationary(values):
    dftest = adfuller(values, autolag='AIC')
    print("Test statistic = {:.3f}".format(dftest[0]))
    print("P-value = {:.3f}".format(dftest[1]))
    print("Critical values :")
    mensaje = "\t{}: {} - Los datos {} estacionarios con {}% de confianza"
    for k, v in dftest[4].items():
        print(mensaje.format(k, v, "no son" if v<dftest[0] else "son", 100-int(k[:-1])))
```

Aplicamos esta función con nuestros datos, y obtenemos obviamente lo que vimos en la figura:

[8] Ver la siguiente entrada en Wikipedia.

CELL 05

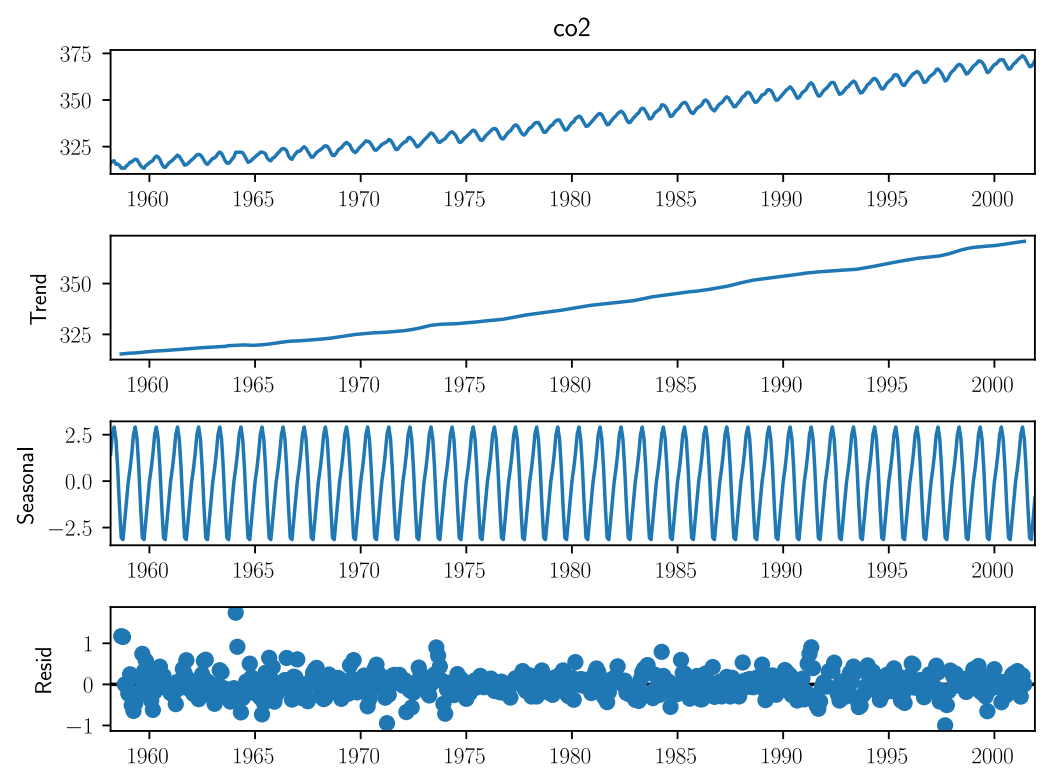
```
check_stationary(y)
```

```
Test statistic = 2.360
P-value = 0.999
Critical values :
1%: -3.4432119442564324 - Los datos no son estacionarios con 99% de confianza
5%: -2.8672126791646955 - Los datos no son estacionarios con 95% de confianza
10%: -2.569791324979607 - Los datos no son estacionarios con 90% de confianza
```

Una herramienta práctica para descomponer nuestra serie original en términos de tendencia, estacionales y residuos, es la función `seasonal_decompose()` de `statsmodels.tsa.seasonal`. Usando nuestros datos `y` como argumento, y utilizando un método aditivo para la descomposición, obtenemos:

CELL 06

```
result = seasonal_decompose(y, model='additive', period=12)
result.plot();
```



Podemos entonces repetir nuestro *test* sobre la parte estacional para ver si es estacionario, lo que se confirma claramente:

CELL 07

```
check_stationary(result.seasonal)
```

```
Test statistic = -190285817600000.906
P-value = 0.000
Critical values :
1%: -3.443161545965353 - Los datos son estacionarios con 99% de confianza
5%: -2.8671904981615706 - Los datos son estacionarios con 95% de confianza
10%: -2.5697795041589244 - Los datos son estacionarios con 90% de confianza
```

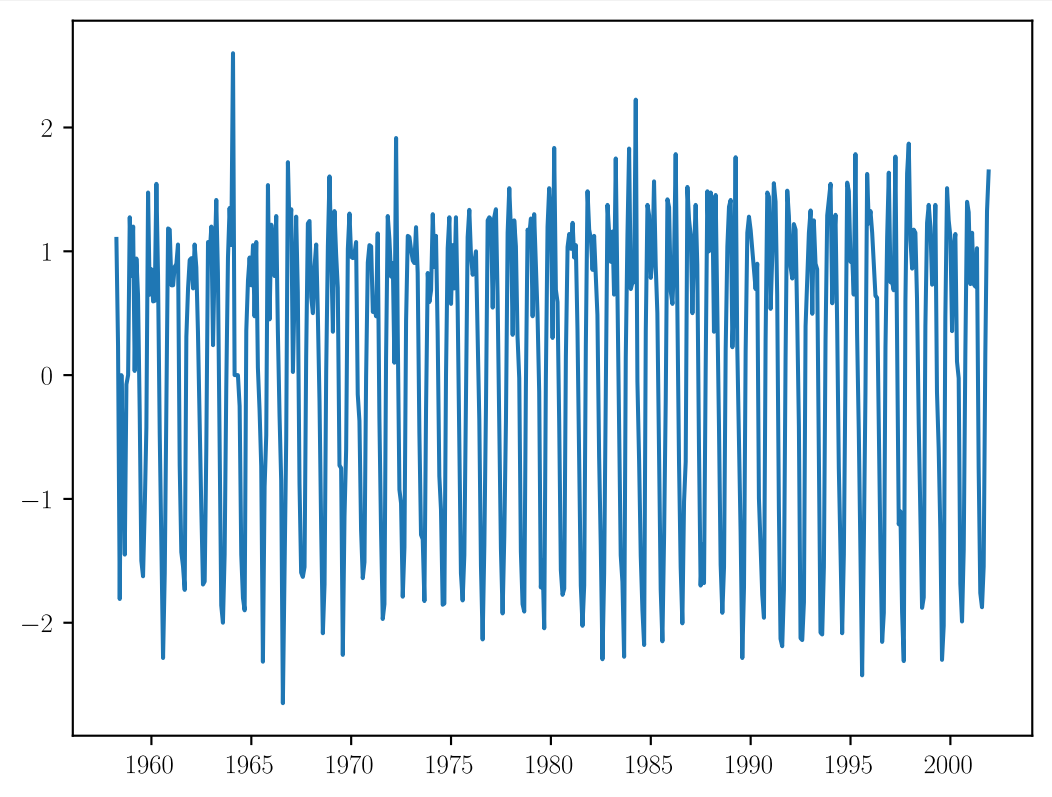
Es posible transformar una serie temporal no estacionaria en una que sí lo es a través de calcular diferencias entre los sucesivos valores. Si, por ejemplo, la línea de tendencia es lineal, generar una nueva serie temporal a través de la primera diferencia $\bar{y}_t = y_t - y_{t-1}$ produce una serie estacionaria, a la cual le podemos aplicar un modelo ARMA. Aplicando esto a nuestra serie temporal obtenemos:

CELL 08

```
y_1 = y.diff()
check_stationary(y_1[1:])
plt.plot(y_1)
```

```
Test statistic = -5.063
P-value = 0.000
Critical values :
1%: -3.4432119442564324 - Los datos son estacionarios con 99% de confianza
5%: -2.8672126791646955 - Los datos son estacionarios con 95% de confianza
10%: -2.569791324979607 - Los datos son estacionarios con 90% de confianza

[<matplotlib.lines.Line2D at 0x7fa97ac24730>]
```



Podemos generar un modelo que contemple la "desestacionalización" mediante diferencias de orden d con un modelo ARIMA(p, d, q), donde I(d) es el preprocesamiento que realiza el cálculo de las diferencias.

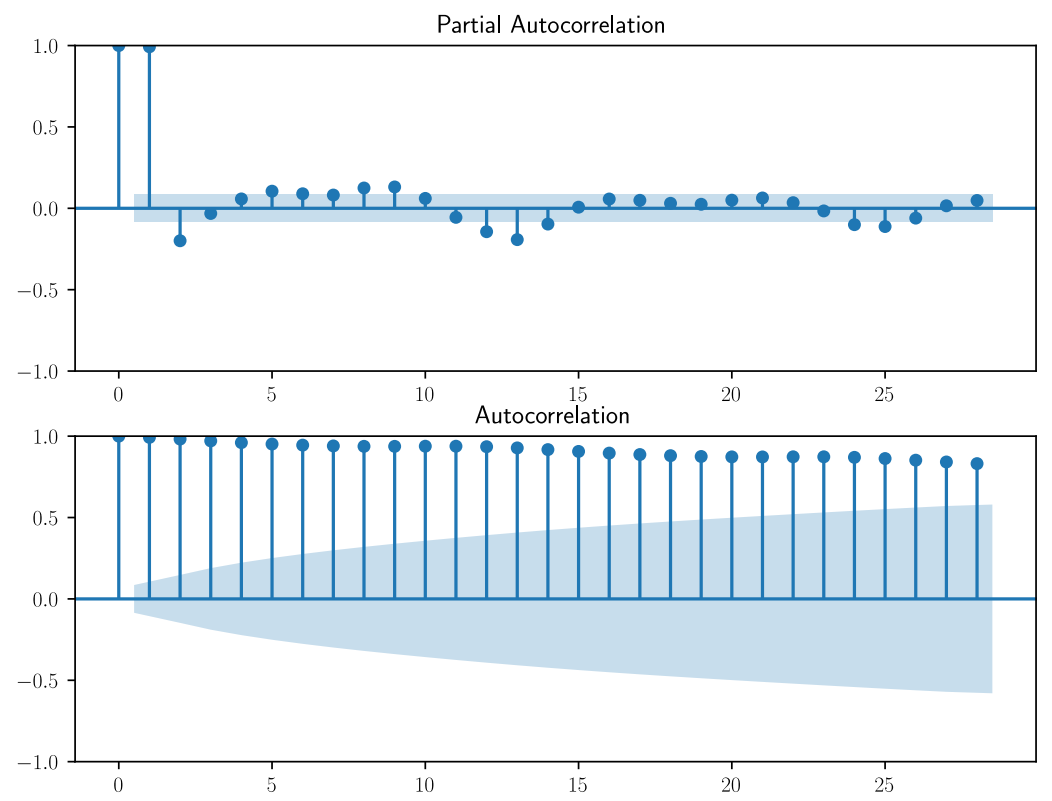
El tratamiento de series temporales con componentes estacionales se puede realizar directamente con un modelo SARIMAX, con considera además de la no estacionalidad, la posibilidad de incoporar otras variables exógenas al modelo. Intentaremos entonces trabajar sobre nuestra serie original ajustándola con un modelo SARIMAX que requiere, además de los parámetros p, d y q, parámetros adicionales para tratar con la componente estacional: P, Q, D y m, siendo este último parámetro el número de pasos temporales que comprende un período estacional.

Una forma de estimar estos parámetros es utilizando los gráficos de las funciones de autoco-

rrelación (ACF) y autocorrelación parcial (PACF). La ACF es la correlación de una serie temporal con sí misma en función del retraso o *lag*, mientras que la PACF es la correlación entre la serie temporal y una versión retrasada de si misma, sustrayendo el efecto de la correlación de *lags* menores. En el caso de nuestro ejemplo, podemos graficar estas funciones con las herramientas provistas por statsmodels:

CELL 09

```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 6))
plot_pacf(y, method='ywm', ax=ax1);
plot_acf(y, ax=ax2);
```



En estas figuras, las zonas sombreadas representan el intervalo de confianza del 95 %, por lo que los valores que quedan dentro de estas regiones podrían considerarse nulos. Se puede ver que para la serie analizada existen fuertes correlaciones de período 12 (que representan un año en nuestros datos) en la PACF, y un decaimiento muy suave para la dfunción de autocorrelación. Se suele utilizar como parámetro p el último valor del *lag* de la PACF antes del primero que se anula, y el correspondiente *lag* de la ACF para determinar q. La figura de la PACF sugiere entonces usar un valor de $p = 2$, pero no es concluyente para determinar d debido a que el decaimiento de la función es muy lento.

Entonces podemos recurrir a otros indicadores para determinar los parámetros del modelo. Al realizar un ajuste SARIMAX$(p, d, q)(P, D, Q)_m$, obtenemos los valores de AIC (*Akaike Information Criterion*) y BIC (*Bayesian Information Criterion*). En ambos casos, menores valores de estos indicadores sugieren mejores modelos. Tanto AIC como BIC penalizan modelos con muchos parámetros, aunque difieren en la forma en que penalizan la complejidad del mismo. BIC penaliza órdenes adicionales del modelo más que AIC, por lo que el criterio BIC suele sugerir modelos más simples que AIC. En muchas ocasiones, ambos sugieren el mismo modelo, pero cuando no lo hacen la elección depende de las prioridades: si queremos identificar un mejor modelo predictivo, AIC es una mejor elección, mientras que si lo que buscamos identificar un buen

modelo explicativo, entonces podemos basar nuestra decisión en el que produzca el menor valor de BIC.

Vamos a realizar entonces una búsqueda de los parámetros en una grilla, avaluando los mismos para los valores 0, 1 y 2, lo que nos da un total de $3^6 = 729$ modelos. Almacenaremos los valores de los parámetros, junto con los resultados obtenidos para AIC y BIC en una lista, y luego examinaremos las combinaciones de parámetros con menores valores de AIC y BIC.

Previamente, dividiremos nuestra serie temporal en un conjunto de datos para el ajuste del model (`y_train`) conteniendo el primer 85 % del total de datos, y reservaremos el restante 15 % (`y_valid`) para contrastar las predicciones obtenidas por nuestro mejor modelo con los datos registrados, de modo de tener una medida de cuán buenas son las predicciones:

CELL 10

```
n_train = int(0.85 * y.size)
n_samples = y.size
n_valid = n_samples - n_train
y_train, y_valid = y[:n_train], y[n_train:]
print(f'Número de muestras: {n_samples}, entrenamiento: {n_train}, validación: {n_valid}')
```

```
Número de muestras: 526, entrenamiento: 447, validación: 79
```

A continuación generamos la búsqueda, iterando sobre todas las tuplas (p, d, q) y (P, D, Q) con valores entre 0 y 2 (dejamos fijo $m = 12$). Los resultados de cada ajuste los guardamos en la lista `model_evals`.

CELL 11

```
model_evals = []
p = d = q = range(0,3)
pdq = list(itertools.product(p, d, q))
seasonal_PDQ = [(x[0], x[1], x[2], 12) for x in list(itertools.product(p, d, q))]
for param in pdq:
    for param_seasonal in seasonal_PDQ:
        try:
            mod = SARIMAX(y_train, order=param, seasonal_order=param_seasonal,
                                            enforce_stationary=False,
                                            enforce_invertibility=False)
            results = mod.fit(disp=0)
            model_evals.append([param, param_seasonal, results.aic, results.bic])
        except:
            continue
```

Mostramos ahora los parámetros obtenidos para los modelos con AIC y BIC menores a 300, siendo este valor sugerido por ajustes exploratorios para algunos pocas tuplas:

CELL 12

```
for p in model_evals:
    if (p[2] < 300) or (p[3] < 300):
        print(f"AIC: {p[2]}, BIC: {p[3]}, (p, d, q)={p[0]}, (P, D, Q, 12)={p[1]}")
```

```
AIC: 10.0, BIC: 30.512792973067846, (p, d, q)=(0, 0, 0), (P, D, Q, 12)=(2, 0, 2, 12)
AIC: 10.0, BIC: 30.512792973067846, (p, d, q)=(0, 0, 1), (P, D, Q, 12)=(2, 0, 1, 12)
AIC: 12.0, BIC: 36.25579700209428, (p, d, q)=(0, 2, 2), (P, D, Q, 12)=(1, 2, 2, 12)
AIC: 14.0, BIC: 42.717910162294984, (p, d, q)=(1, 0, 2), (P, D, Q, 12)=(1, 0, 2, 12)
AIC: 296.9064313262613, BIC: 325.41774306496416, (p, d, q)=(1, 1, 1), (P, D, Q, 12)=(2, 1, 2, 12)
AIC: 299.38141802518373, BIC: 331.96577429798697, (p, d, q)=(1, 1, 2), (P, D, Q, 12)=(2, 1, 2, 12)
AIC: 8.0, BIC: 24.410234378454277, (p, d, q)=(2, 0, 0), (P, D, Q, 12)=(0, 0, 1, 12)
AIC: 10.0, BIC: 30.512792973067846, (p, d, q)=(2, 0, 0), (P, D, Q, 12)=(1, 0, 1, 12)
AIC: 10.0, BIC: 30.512792973067846, (p, d, q)=(2, 0, 1), (P, D, Q, 12)=(0, 0, 1, 12)
AIC: 299.73667810970073, BIC: 332.32103438250397, (p, d, q)=(2, 1, 1), (P, D, Q, 12)=(2, 1, 2, 12)
```

Podemos ver que algunas combinaciones de parámetros generan valores muy bajos de AIC y BIC. Sin embargo, dichos modelos no ajustan bien a la serie temporal (esto se muestra en los *warnings* que desactivamos en la celda 1 para evitar salidas muy largas, que informan sobre problemas de convergencia del método de ajuste o inestabilidades numéricas). Descartando estos valores, el conjunto $(p, d, q) = (1, 1, 1)$ y $(P, D, Q, 12) = (2, 1, 2, 12)$ coinciden en bajos valores tanto de AIC como BIC, por lo que construiremos un modelo SARIMAX basado en estos parámetros:

CELL 13

```
model = SARIMAX(y_train, order=(1,1,1), seasonal_order=(2, 1, 2, 12),
                enforce_stationary=True, enforce_invertibility=False)
model_fit = model.fit(disp=0, maxiter=200)
print(model_fit.summary())
```

```
                                     SARIMAX Results
==========================================================================================
Dep. Variable:                                 co2   No. Observations:                  447
Model:             SARIMAX(1, 1, 1)x(2, 1, [1, 2], 12)   Log Likelihood                -141.453
Date:                                 Mon, 09 Jan 2023   AIC                            296.906
Time:                                         18:50:12   BIC                            325.418
Sample:                                     03-01-1958   HQIC                           308.160
                                          - 05-01-1995
Covariance Type:                                   opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.4175      0.083      5.020      0.000       0.254       0.580
ma.L1         -0.7347      0.065    -11.335      0.000      -0.862      -0.608
ar.S.L12       1.0033      0.086     11.680      0.000       0.835       1.172
ar.S.L24      -0.1418      0.059     -2.398      0.016      -0.258      -0.026
ma.S.L12      -1.8858      0.106    -17.736      0.000      -2.094      -1.677
ma.S.L24       0.9195      0.096      9.539      0.000       0.731       1.108
sigma2         0.1049      0.007     15.680      0.000       0.092       0.118
===================================================================================
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):               129.06
Prob(Q):                              0.95   Prob(JB):                         0.00
Heteroskedasticity (H):               0.57   Skew:                             0.48
Prob(H) (two-sided):                  0.00   Kurtosis:                         5.50
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

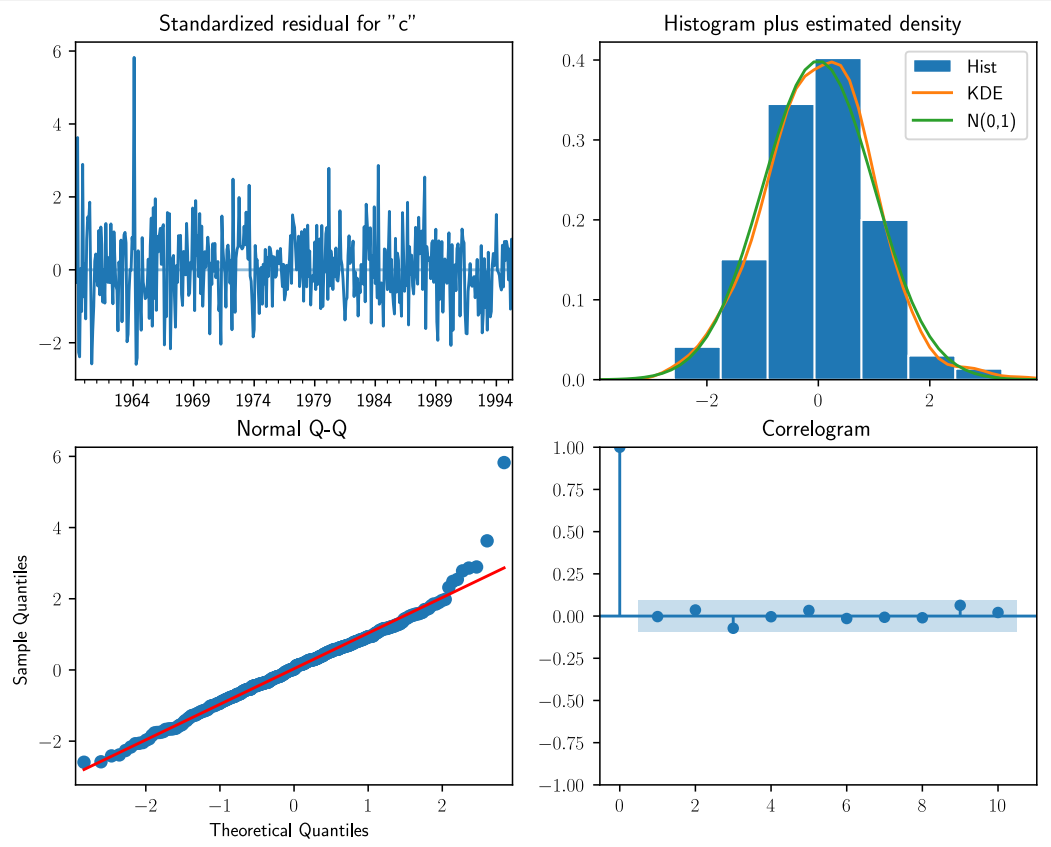
Con el método `summary()` del objeto `model_fit` podemos obtener los parámetros del ajuste y

sus correspondientes p-valores. En todos los casos, estos p-valores muestran que los coeficientes obtenidos son significativos.

También podemos examinar la calidad del ajuste a través del análisis de los residuos de los datos que utilizamos como ajuste (`y_train`). Estos residuos son la diferencia entre los valores generados por el modelo y los datos reales. El modelo ideal de ajuste debería tener residuos que sean ruido blanco gaussiano centrados en cero y sin correlación. El método `plot_diagnostics` genera cuatro gráficos que permiten evaluar estas características de los residuos:

CELL 14

```
model_fit.plot_diagnostics(figsize=(9,7));
```



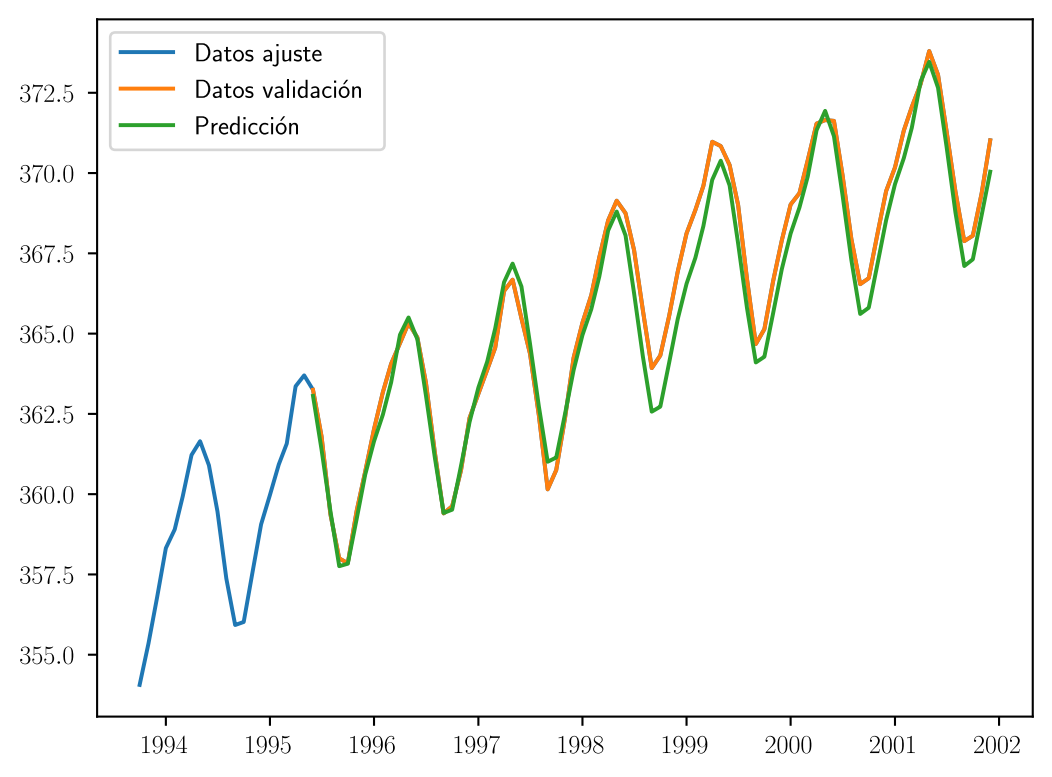
El panel superior izquierdo muestra los residuos para cada valor registrado en los datos de ajuste. Si nuestro modelo funciona adecuadamente, no se debería apreciar ningún patrón obvio en estos residuos, tal como muestra la figura. El panel superior derecho muestra la distribución de dichos residuos, tanto en forma de histograma como su KDE, y muestra también para facilitar la comparación una distribución normal con media cero y varianza 1. Se puede ver que la KDE de los residuos es muy cercana a la distribución normal.

En el panel inferior izquierdo se muestra el diagrama Q-Q de los residuos. Este diagrama compara la distribución teórica de los residuos con la observada. En el caso ideal deberíamos obtener una línea recta como la roja. Esto sucede en casi todos los valores excepto en los extremos. Finalmente, en el panel inferior derecho se muestra un correlograma que representa la ACF de los residuos. El 95 % de las correlaciones para *lags* mayores que cero no deberían ser significativos (es decir, deben estar dentro de la región sombreada). Si hubiese una correlación en los residuos, podríamos interpretar que existe información en los datos que no fue capturada por el modelo. En nuestro caso, no parecen haber correlaciones para *lags* mayores que cero, lo que indica que el ajuste es bueno.

Finalmente, podemos utilizar el modelo ajustado para predecir valores de la serie temporal posteriores a los que utilizamos para el ajuste, y compararlos con los valores que reservamos con este fin, `y_valid`:

CELL 15

```
y_valid_fore = model_fit.forecast(steps=n_valid)
plt.plot(y[n_train-20:], label='Datos ajuste')
plt.plot(y_valid, label='Datos validación')
plt.plot(y_valid_fore, label='Predicción')
plt.legend(loc=2)
plt.show()
```



Podemos ver que el modelo es capaz de predecir con buena precisión los valores posteriores a los utilizados en el ajuste del modelo, pese a que alguna diferencia es notoria.

Hemos visto un ejemplo de análisis estadístico de una serie temporal no estacionaria, ajustándola con un modelo relativamente simple. Además de statsmodels, existen otras herramientas poderosas para el ajuste de modelos, entre las que podemos destacar Darts[9] y muchas bibliotecas de redes neuronales. El proceso no es simple y hay que analizar con espíritu crítico cada resultado obtenido.

1.7. Lecturas recomendadas

- Una definición formal rigurosa de modelo estadístico puede verse en Peter McCullagh. «What is a statistical model?» En: *The Annals of Statistics* 30.5 (2002), págs. 1225 -1310. DOI: 10.1214/aos/1035844977. URL: https://doi.org/10.1214/aos/1035844977.

- Galit Shmueli realiza una detallada discusión sobre la diferencia entre modelos explicativos y modelos predictivos en Galit Shmueli. «To Explain or to Predict?» En: *Statistical Science*

[9] https://unit8co.github.io/darts/

25.3 (2010), págs. 289 -310. DOI: 10.1214/10-STS330. URL: https://doi.org/10.1214/10-STS330.

- Un libro moderno, con muchos ejemplos y código (en R), y que mantiene una versión *online* con licencia Creative Commons, es Susan Holmes y Wolfgang Huber. *Modern statistics for modern biology*. Cambridge, England: Cambridge University Press, feb. de 2019. URL: https://web.stanford.edu/class/bios221/book/introduction.html. Se puede acceder aquí[10].

- Un libro clásico sobre el uso de técnicas estadísticas aplicadas al análisis de datos experimentales: George E.P. Box, J. Stuart Hunter y William G. Hunter. *Statistics for experimenters*. 2.ª ed. Wiley Series in Probability and Statistics. Chichester, England: Wiley-Blackwell, mayo de 2005.

- El libro de Haslwanter tiene una buena introducción al modelado estadístico utilizando Python: Thomas Haslwanter. *An Introduction to Statistics with Python*. 1.ª ed. Statistics and Computing. Cham, Switzerland: Springer International Publishing, 2016. DOI: 10.1007/978-3-319-28316-6.

- Cameron y Trivedi son autores de un libro de referencia para el modelado de conjuntos de datos discretos: A. Colin Cameron y Pravin K. Trivedi. *Regression Analysis of Count Data*. 2.ª ed. Econometric Society Monographs. Cambridge University Press, 2013. DOI: 10.1017/CBO9781139013567.

- El libro de T. Mills presenta una guía práctica y muy completa para la modelización y predicción de series temporales: Terence Mills. *Applied time series analysis*. San Diego, CA: Academic Press, 2019.

- Un libro muy completo (aunque con código en R) es: George E P Box y col. *Time Series Analysis*. en. 5.ª ed. Wiley Series in Probability and Statistics. Nashville, TN: John Wiley & Sons, jun. de 2015.

[10] https://web.stanford.edu/class/bios221/book/index.html.

VERSIÓN PRELIMINAR

Parte III
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [14].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tengamos más de esos!

Bibliografía

[1] URL: https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es.

[2] G. N. Wilkinson y C. E. Rogers. «Symbolic Description of Factorial Models for Analysis of Variance». En: *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 22.3 (1973), págs. 392-399. DOI: https://doi.org/10.2307/2346786. eprint: https://rss.onlinelibrary.wiley.com/doi/pdf/10.2307/2346786. URL: https://rss.onlinelibrary.wiley.com/doi/abs/10.2307/2346786.

[3] Skipper Seabold y Josef Perktold. «statsmodels: Econometric and statistical modeling with python». En: *9th Python in Science Conference*. 2010.

[4] A. Colin Cameron y Pravin K. Trivedi. «Econometric models based on count data. Comparisons and applications of some estimators and tests». En: *Journal of Applied Econometrics* 1.1 (1986), págs. 29-53. DOI: https://doi.org/10.1002/jae.3950010104. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/jae.3950010104. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/jae.3950010104.

[5] Charles D. Keeling, Stephen C. Piper, Robert B. Bacastow, Martin Wahlen, Timothy P. Whorf, Martin Heimann y Harro A. Meijer. «Atmospheric CO2 and 13CO2 Exchange with the Terrestrial Biosphere and Oceans from 1978 to 2000: Observations and Carbon Cycle Implications». En: *A History of Atmospheric CO2 and Its Effects on Plants, Animals, and Ecosystems*. Ed. por I.T. Baldwin y col. New York, NY: Springer New York, 2005, págs. 83-113. DOI: 10.1007/0-387-27048-5_5. URL: https://doi.org/10.1007/0-387-27048-5_5.

[6] Peter McCullagh. «What is a statistical model?» En: *The Annals of Statistics* 30.5 (2002), págs. 1225 -1310. DOI: 10.1214/aos/1035844977. URL: https://doi.org/10.1214/aos/1035844977.

[7] Galit Shmueli. «To Explain or to Predict?» En: *Statistical Science* 25.3 (2010), págs. 289 -310. DOI: 10.1214/10-STS330. URL: https://doi.org/10.1214/10-STS330.

[8] Susan Holmes y Wolfgang Huber. *Modern statistics for modern biology*. Cambridge, England: Cambridge University Press, feb. de 2019. URL: https://web.stanford.edu/class/bios221/book/introduction.html.

[9] George E.P. Box, J. Stuart Hunter y William G. Hunter. *Statistics for experimenters*. 2.ª ed. Wiley Series in Probability and Statistics. Chichester, England: Wiley-Blackwell, mayo de 2005.

[10] Thomas Haslwanter. *An Introduction to Statistics with Python*. 1.ª ed. Statistics and Computing. Cham, Switzerland: Springer International Publishing, 2016. DOI: 10.1007/978-3-319-28316-6.

[11] A. Colin Cameron y Pravin K. Trivedi. *Regression Analysis of Count Data*. 2.ª ed. Econometric Society Monographs. Cambridge University Press, 2013. DOI: 10.1017/CBO9781139013567.

[12] Terence Mills. *Applied time series analysis*. San Diego, CA: Academic Press, 2019.

VERSIÓN PRELIMINAR

[13] George E P Box, Gwilym M Jenkins, Gregory C Reinsel y Greta M Ljung. *Time Series Analysis*. en. 5.ª ed. Wiley Series in Probability and Statistics. Nashville, TN: John Wiley & Sons, jun. de 2015.

[14] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: https://www.python.org/dev/peps/pep-0020/.